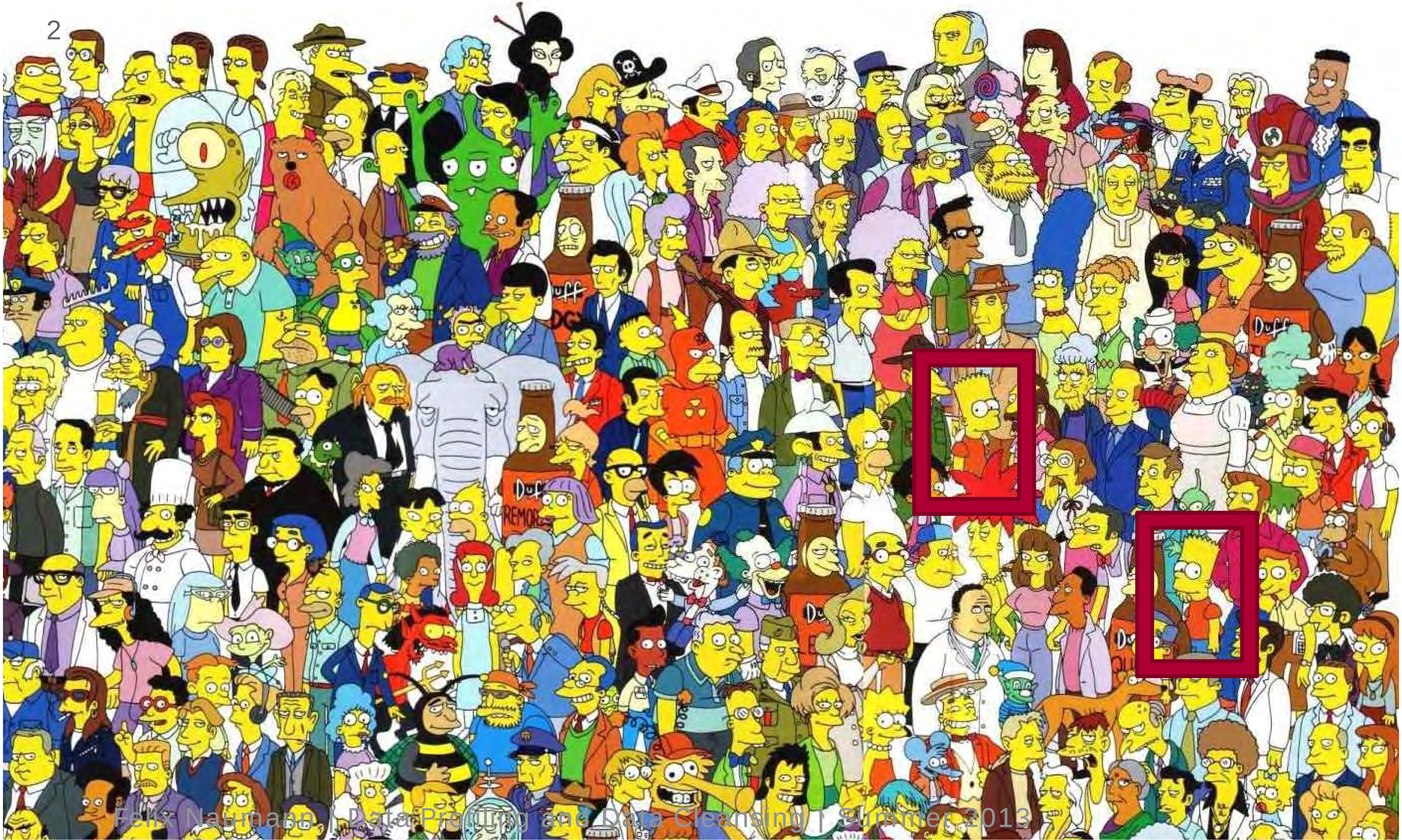
Sorted Neighborhood Methods

2.7.2013

Felix Naumann

Duplicate Detection

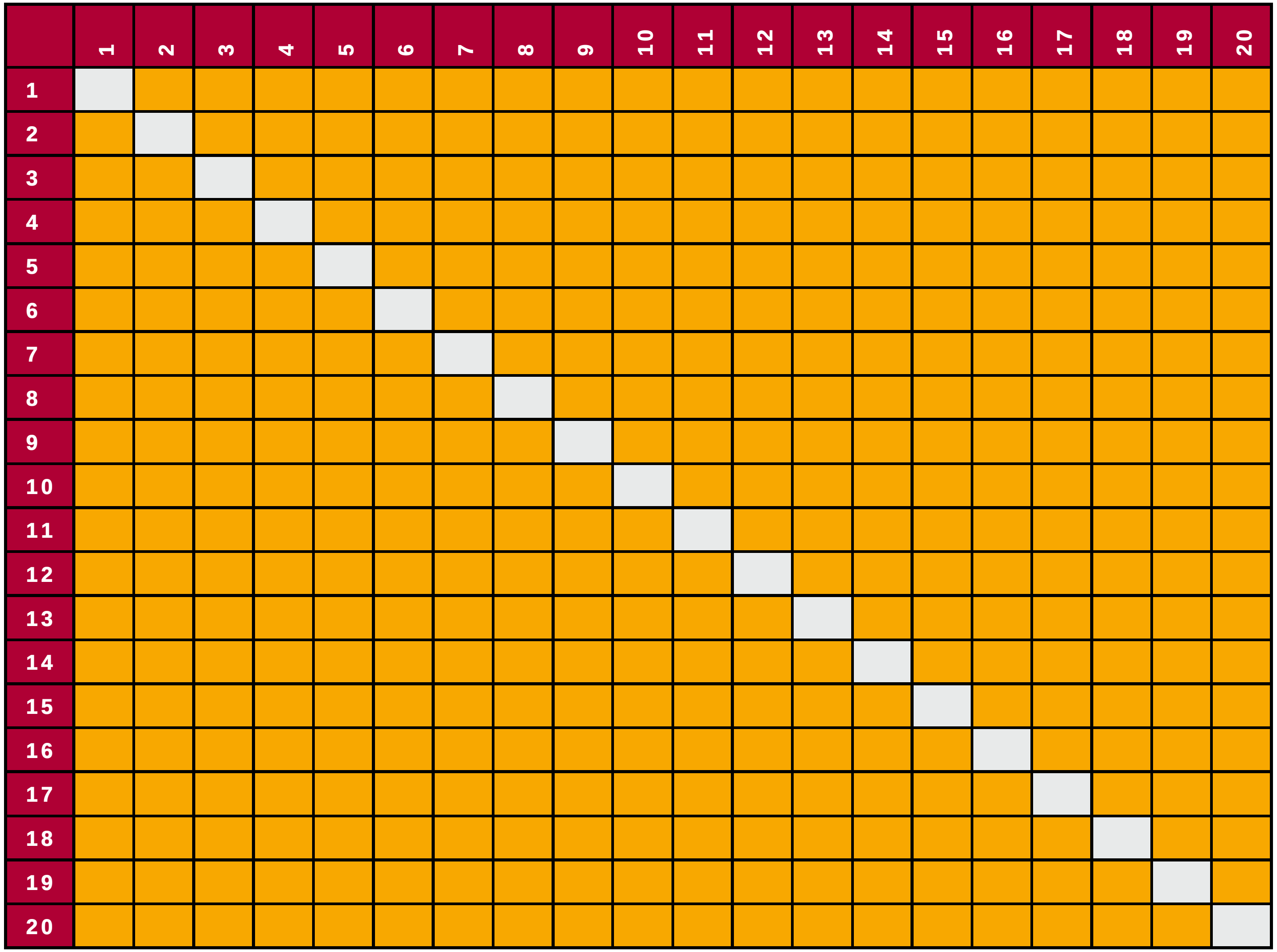


Number of comparisons: All pairs

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1																				
2																				
3																				
4																				
5																				
6																				
7																				
8																				
9																				
10																				
11																				
12																				
13																				
14																				
15																				
16																				
17																				
18																				
19																				
20																				

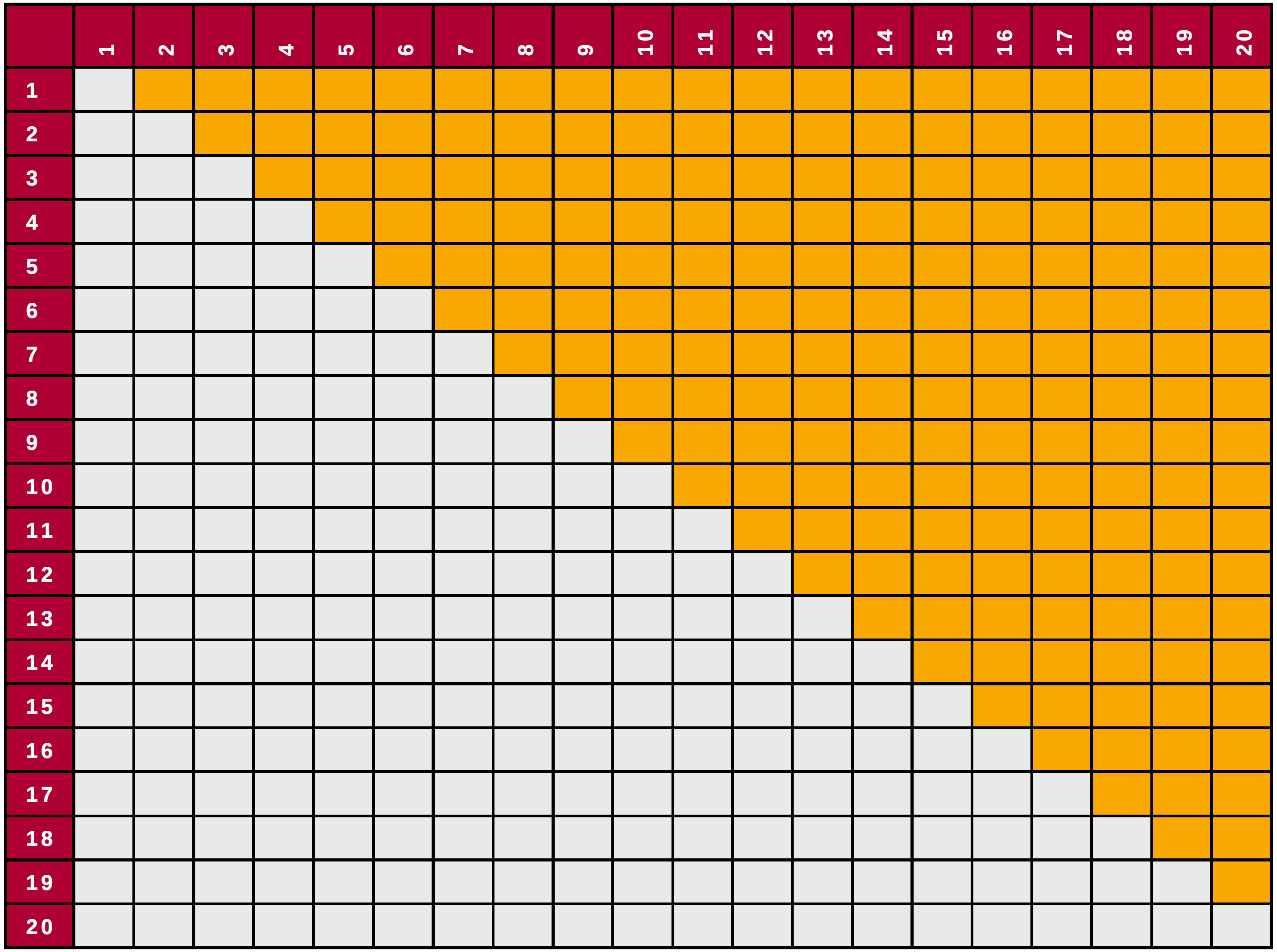
400
comparisons

Reflexivity of Similarity



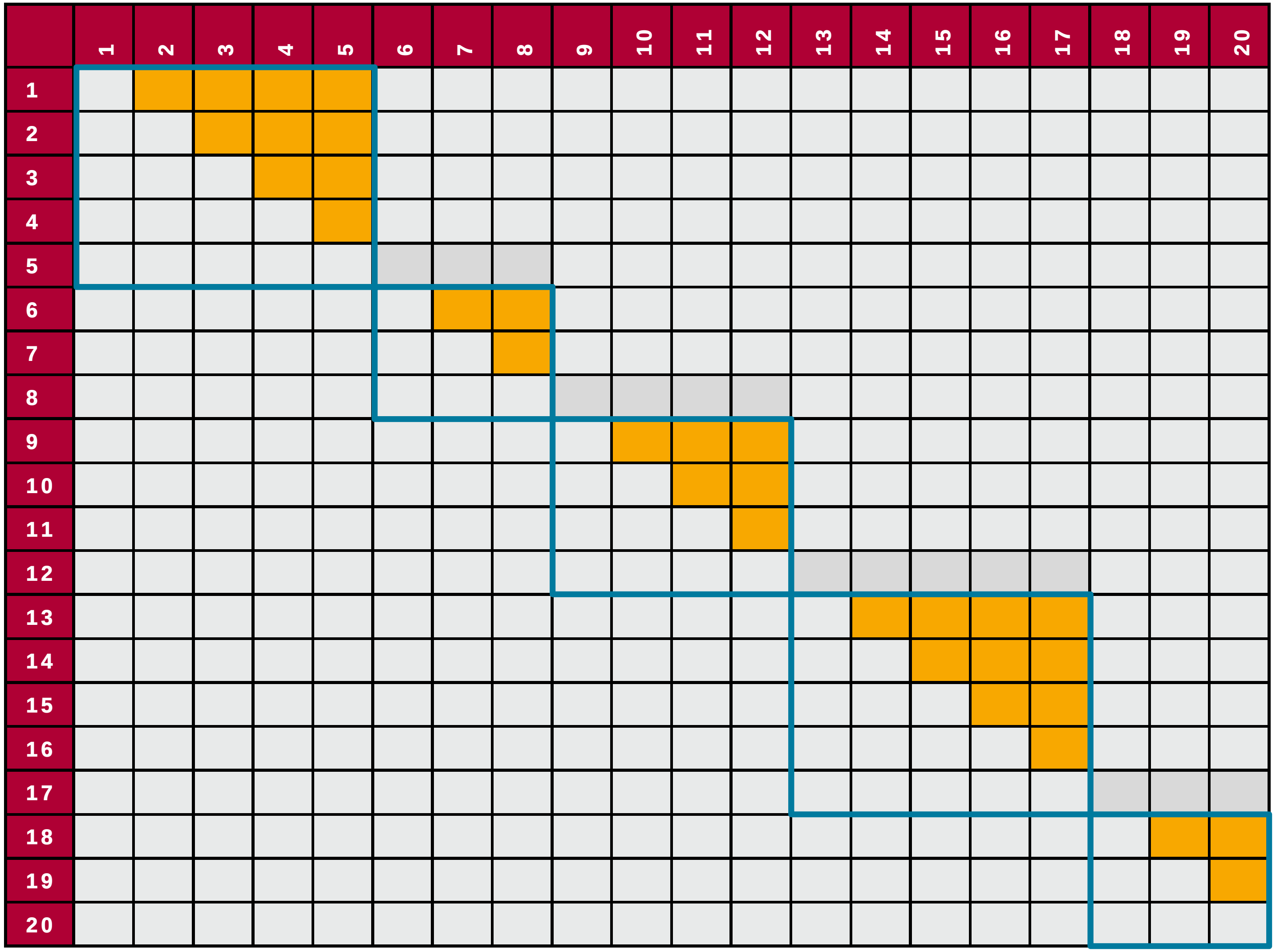
380 comparisons

Symmetry of Similarity



190
comparisons

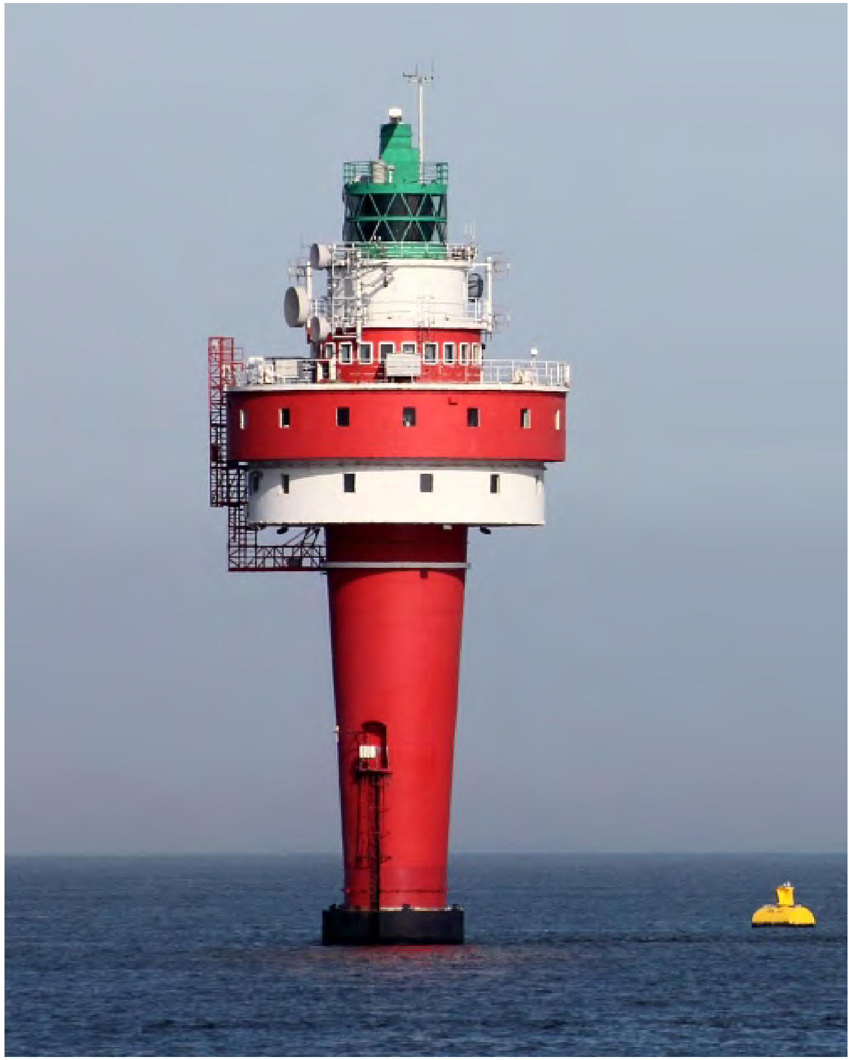
Blocking by ZIP



32 comparisons

Overview

- The Original
- Unique sorting keys
- Adaptive SNM
  - Part 1
  - Part 2
- Sorted Blocks
- Domain-independent SNM



The Sorted Neighborhood Method

- Input:
  - Table with N tuples
  - Similarity measure
- Output:
  - Classes (clusters) of equivalent tuples (duplicates)
- Problem: Many tuples
  - Comparing each tuple-pair is inefficient
  - Large table may not fit in main memory (scalability)

- Mauricio A. Hernandez and Salvatore J. Stolfo. The merge/purge problem for large databases. In Proceedings of the ACM International Conference on Management of Data (SIGMOD), 1995.
- Mauricio A. Hernandez and Salvatore J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. Data Mining and Knowledge Discovery, 2(1), 1998

Sorted Neighborhood
[Hernandez Stolfo 1998]

- Idea
  - Sort tuples so that similar tuples are close to each other.
  - Only compare tuples within a small neighborhood (window).
1. Generate key
   - E.g.: SSN+“first 3 letters of name“ + ...
   - Effectiveness strongly depends on choice of key
   - Key is only virtual and not unique (“sorting key”)
2. Sort by key
   - Similar tuples end up close to each other.
3. Slide window over sorted tuples
   - Compare all pairs of tuples within window.

- Problems
  - Choice of key
  - Choice of window size
- Complexity: At least 3 passes over data
  - Sorting!

SNM – Example

ID	Title	Year	Genre
17	Mask of Zorro	1998	Adventure
18	Addams Family	1991	Comedy
25	Rush Hour	1998	Comedy
31	Matrix	1999	Sci-Fi
52	Return of Dschafar	1994	Children
113	Adams Family	1991	Comedie
207	Return of Djaffar	1995	Children

1. Create key

ID	Key
17	MSKAD98
18	DDMCO91
25	RSHCO98
31	MTRSC99
52	RTRCH94
113	DMSCO91
207	RTRCH95

2. Sort

ID	Key
18	DDMCO91
113	DMSCO91
17	MSKAD98
31	MTRSC99
25	RSHCO98
52	RTRCH94
207	RTRCH95

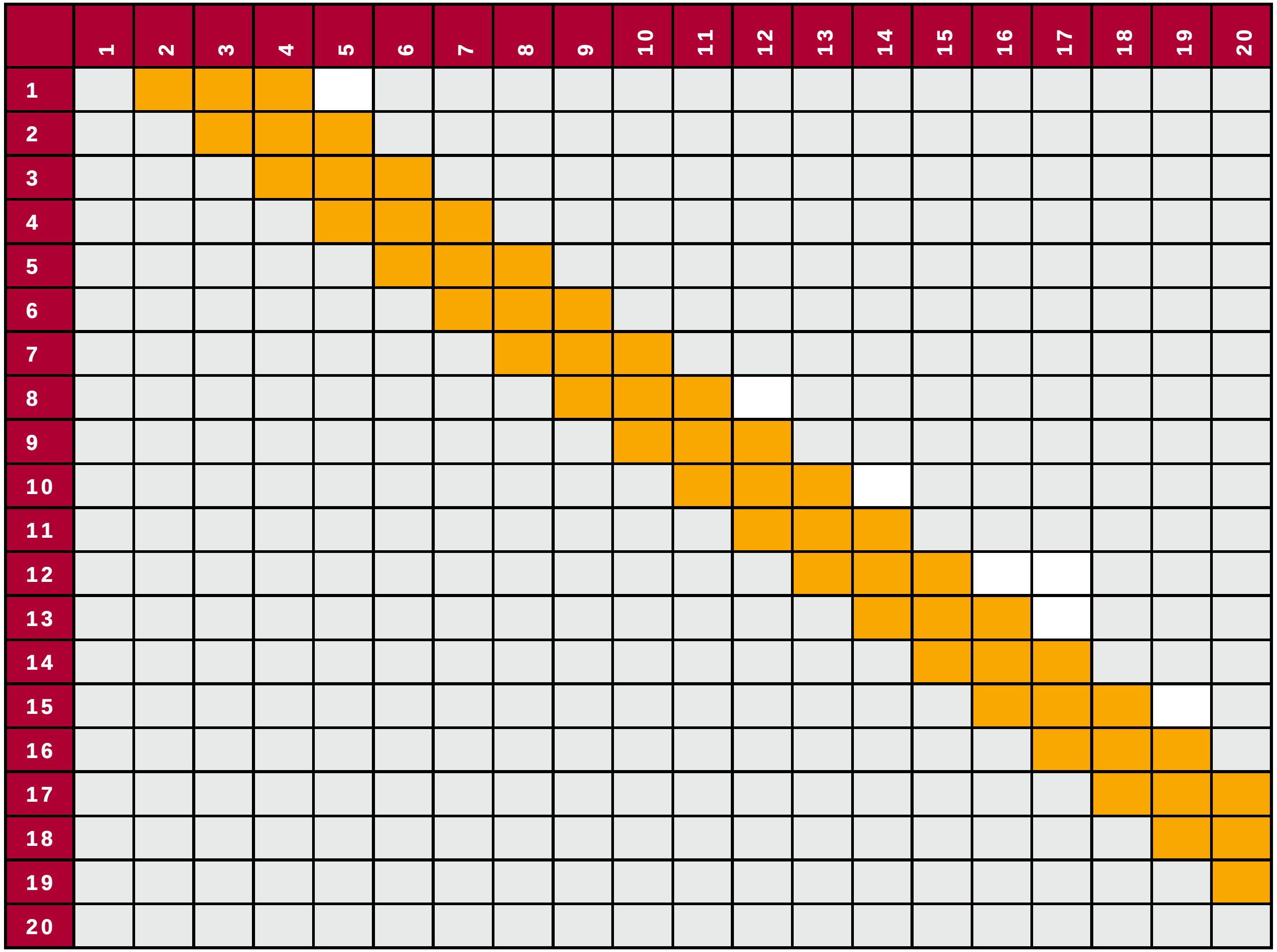
3. Merge

ID	Key
18	DDMCO91
113	DMSCO91
17	MSKAD98
31	MTRSC99
25	RSHCO98
52	RTRCH94
207	RTRCH95

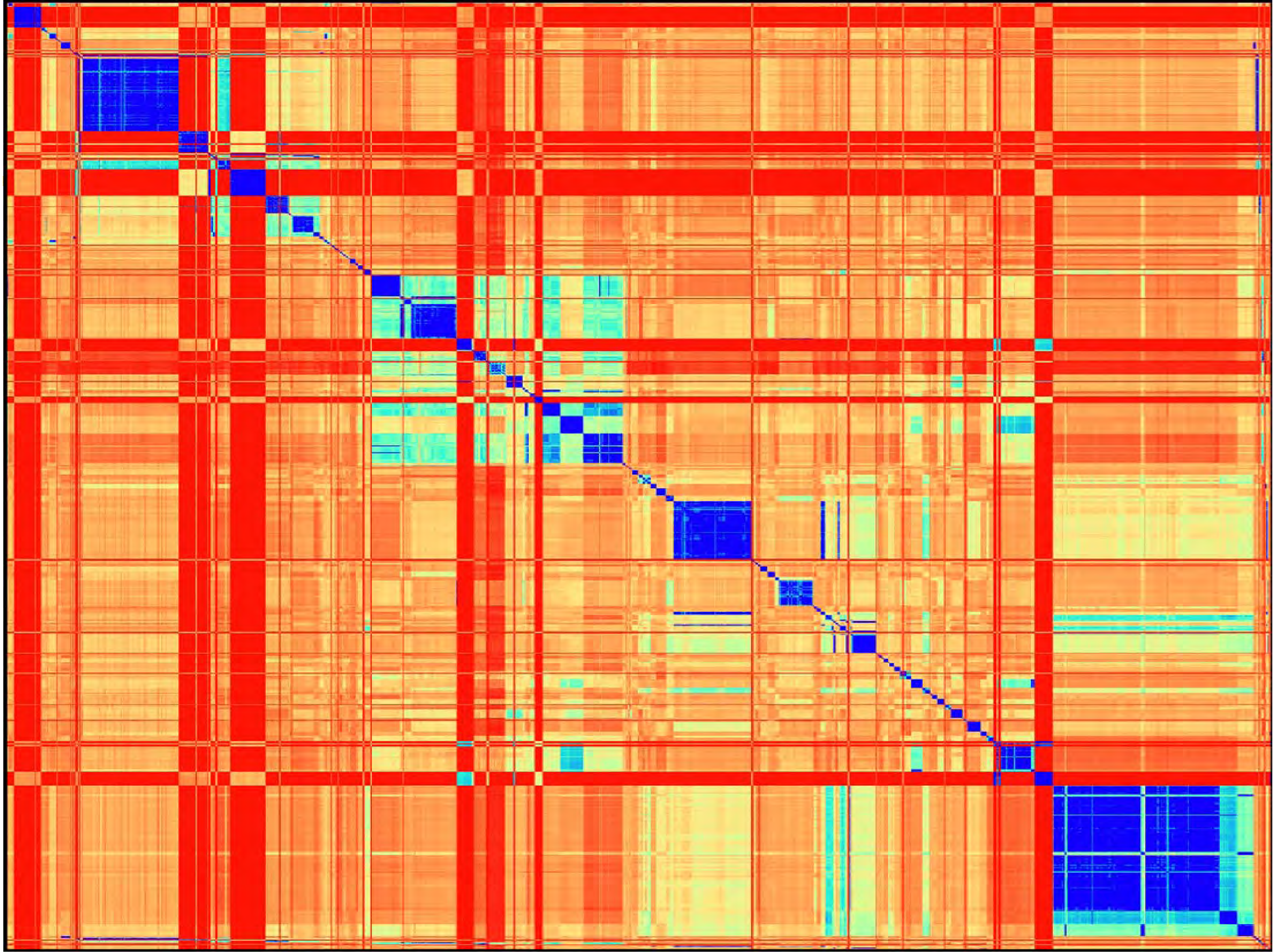
classify(18,113) → duplicates

classify(52,207) → duplicates

SNM by ZIP (window size 4)



54 comparisons



Sorted Neighborhood – Complexity

- N: Number of tuples
- w: Window size
- Computational complexity:
  - $O(N) + O(N \log N) + O(w\,N) = O(N \log N)$
    - if $w < \log N$; $O(wN)$ else
- IO complexity
  - Linear in N
  - Three passes over table on disk
    - Create key, sort, window
  - Sorting: e.g. TPMMS

Sorted Neighborhood – Configuration

- Choice of key
  - Formulierung durch Experten
  - Aufwändig
  - Schwer vergleichbare Ergebnisse
  - Für Effektivität entscheidend
- Choice of window size
  - $w = N : O(N^2) \Rightarrow$ max. accuracy & max. Zeit
  - $w = 2 : O(N) \Rightarrow$ min. accuracy & min. Zeit
- Choice of classification method / similarity measure
  - Hernandez and Stolfo suggest „equational theory“
  - Rule set

Sorted Neighborhood – Multipass Approach

- Problem in choice of key
  - Example: r_1: 193456782 und r_2: 913456782
- Solution 1:
  - Extend window size: $w \rightarrow N$
- Solution 2:
  - Multiple passes with different keys
  - Can keep *w* small
  - Transitive closure on results of each pass

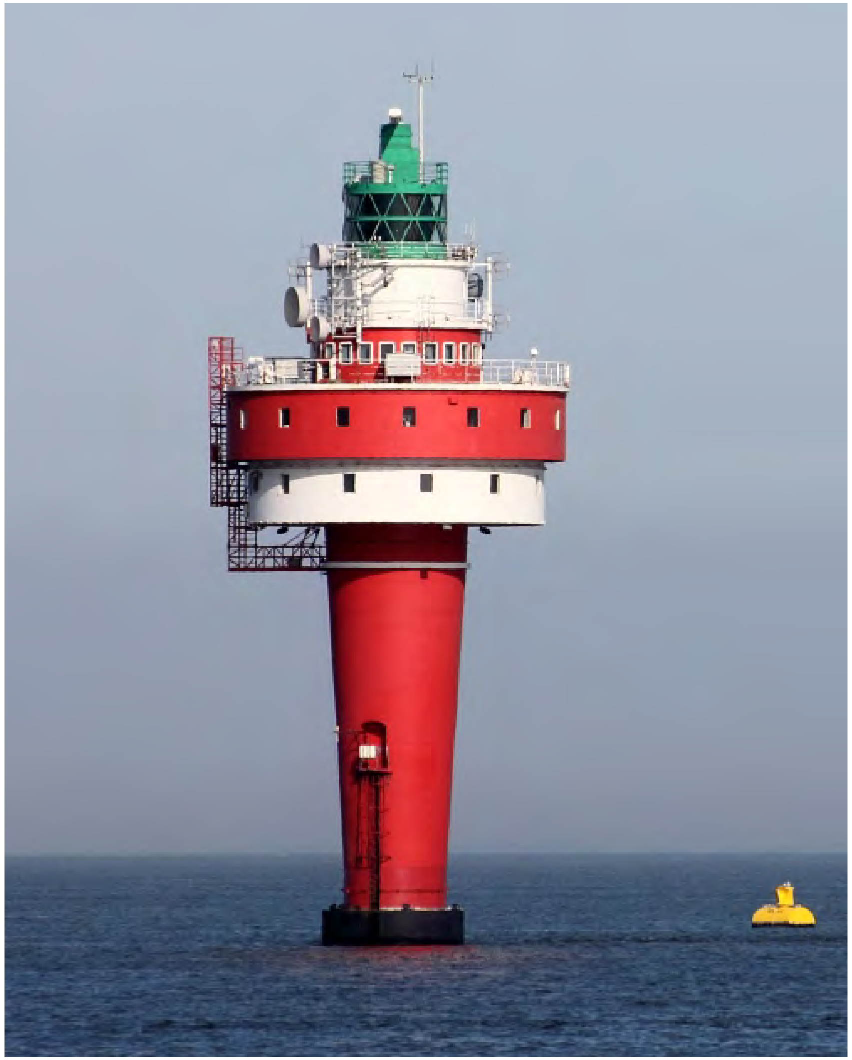
Suggested Extensions

- Incremental SNM
  - Handle inserts
  - Trivial extension

- Parallel SNM
  - Each multi-pass in parallel
  - Parallel windows
  - See also current seminar „Large Scale Duplicate Detection“
    - Final presentations: July 10, 9:15 – 12:30 in ???

Overview

- The Original
- Unique sorting keys
- Adaptive SNM
  - Part 1
  - Part 2
- Sorted Blocks
- Domain-independent SNM



Choice of sorting key(s)

- General problem: Sortation among same keys is random
- Idea:
  - Create inverted index on sorting key
  - Slide (smaller) window over index
    - w=1 => traditional blocking

- Peter Christen: A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. IEEE Trans. Knowl. Data Eng. 24(9): 1537-1555 (2012)

Example

Window positions	BKVs (Surname)	Identifiers
1	Millar	R6
2	Miller	R2, R8
3	Myler	R4
4	Peters	R3
5	Smith	R1
6	Smyth	R5, R7

Window range	Candidate record pairs
1 – 3	(R6,R2), (R6,R8), (R6,R4), (R2,R8), (R2,R4), (R8,R4)
2 – 4	(R2,R8), (R2,R4), (R2,R3), (R8,R4), (R8,R3), (R4,R3)
3 – 5	(R4,R3), (R4,R1), (R3,R1)
4 – 6	(R3,R1), (R3,R5), (R3,R7), (R1,R5), (R1,R7), (R5,R7)

Further ideas for key

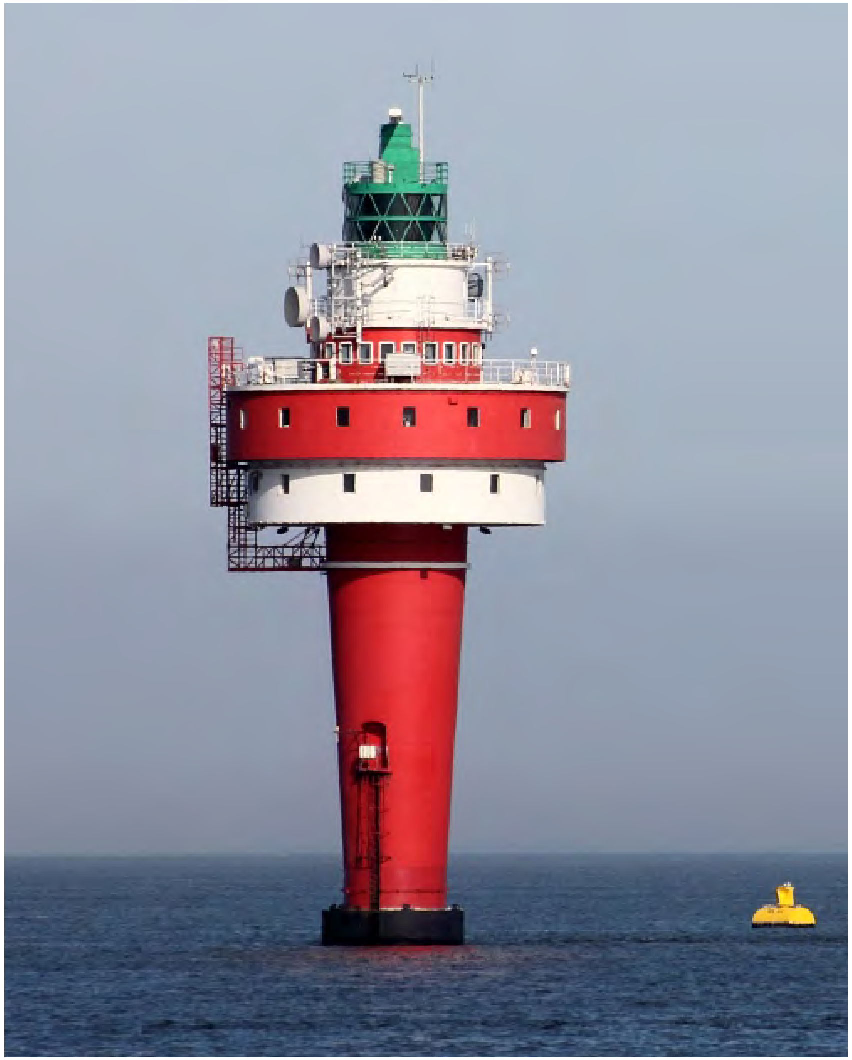
- Q-Grams

Identifiers	BKVs (Surname)	Bigram sub-lists	Index key values
R1	Smith	[sm,mi,it,th], [mi,it,th], [sm,it,th], [sm,mi,th], [sm,mi,it]	**smmiitth**, miitth, smitth, smmith, smmiit
R2	Smithy	[sm,mi,it,th,hy], [mi,it,th,hy], [sm,it,th,hy], [sm,mi,th,hy], [sm,mi,it,hy], [sm,mi,it,th]	smmiitthhy, miitthhy, smitthhy, smmithhy, smmiithy, **smmiitth**
R3	Smithe	[sm,mi,it,th,he], [mi,it,th,he], [sm,it,th,he], [sm,mi,th,he], [sm,mi,it,he], [sm,mi,it,th]	smmiitthhe, miitthhe, smitthhe, smmithhe, smmiithe, **smmiitth**

- Suffix array (up to certain length)
- Soundex and other phonetic codes
- Canopy clustering
  - Use cheap clustering approach to form blocks
- And many more

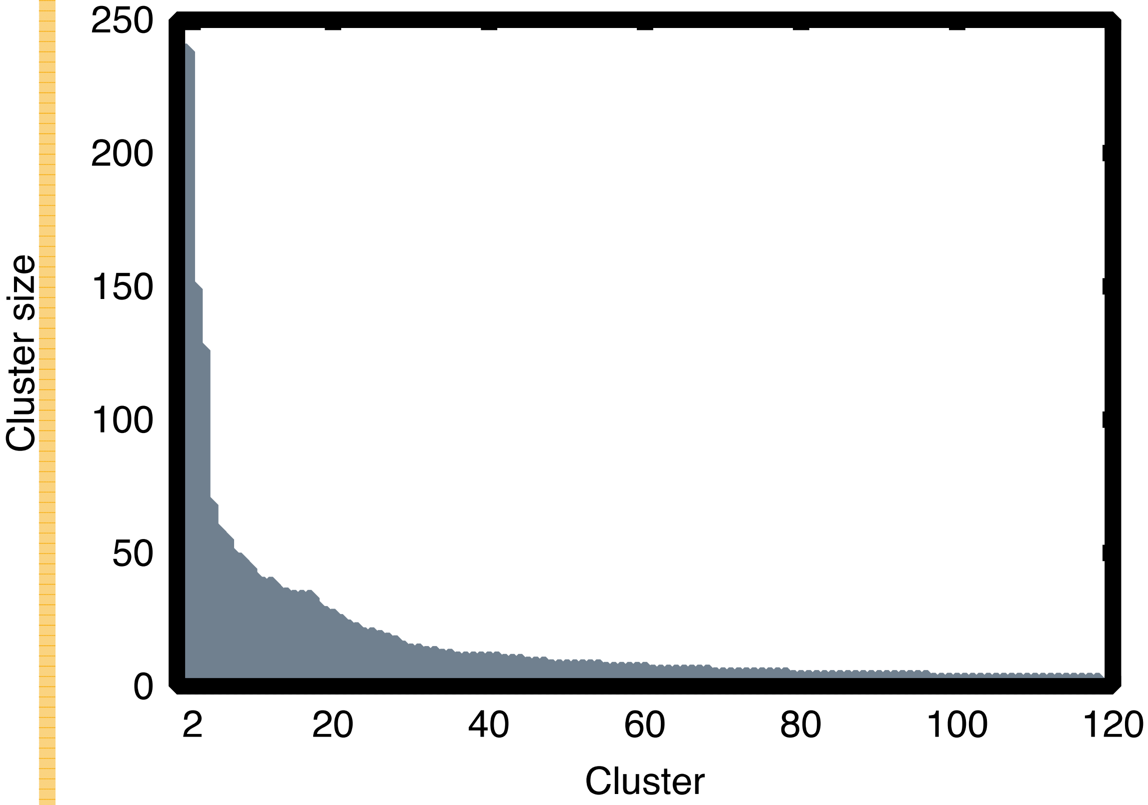
Overview

- The Original
- Unique sorting keys
- Adaptive SNM
  - Part 1
  - Part 2
- Sorted Blocks
- Domain-independent SNM



One size fits all?

- Selection of window size w
  - Too small -> some duplicates might be missed
  - Too large -> many unnecessary comparisons



Cluster sizes for the Cora Citation Matching data set (1,879 references of research papers)

Incrementally Adaptive and Accumulative Adaptive-SNM

- Yan et al. [16] discuss adaptivity of record linkage algorithms using the example of SNM. They use the window to build non-overlapping blocks that can contain different numbers of records. The pairwise record comparison then takes place within these blocks. The hypothesis is that the distance between a record and its successors in the sort sequence is monotonically increasing in a small neighborhood, although the sorting is done lexicographically and not by distance. They present two algorithms and compare them with the basic SNM.
- Incrementally Adaptive-SNM (IA-SNM) is an algorithm that incrementally increases the window size as long as the distance of the first and the last element in the current window is smaller than a specified threshold. The increase of the window size depends on the current window size.
- Accumulative Adaptive-SNM (AA-SNM) on the other hand creates windows with one overlapping record. By considering transitivity, multiple adjacent windows can then be grouped into one block, if the last record of a window is a potential duplicate of the last record in the next adjacent window. After the enlargement of the windows both algorithms have a retrenchment phase, in which the window is decreased until all records within the block are potential duplicates.
- We have implemented both IA-SNM and AA-SNM, and compare them to our work in our experimental evaluation. However, our experiments do not confirm that IA-SNM and AA-SNM perform better than SNM.
- S. Yan, D. Lee, M.-Y. Kan, and L. C. Giles, “Adaptive sorted neighborhood methods for efficient record linkage,” in Proceedings of the ACM/IEEE-CS joint conference on Digital libraries (JCDL), 2007, pp. 185–194.

Reproducability

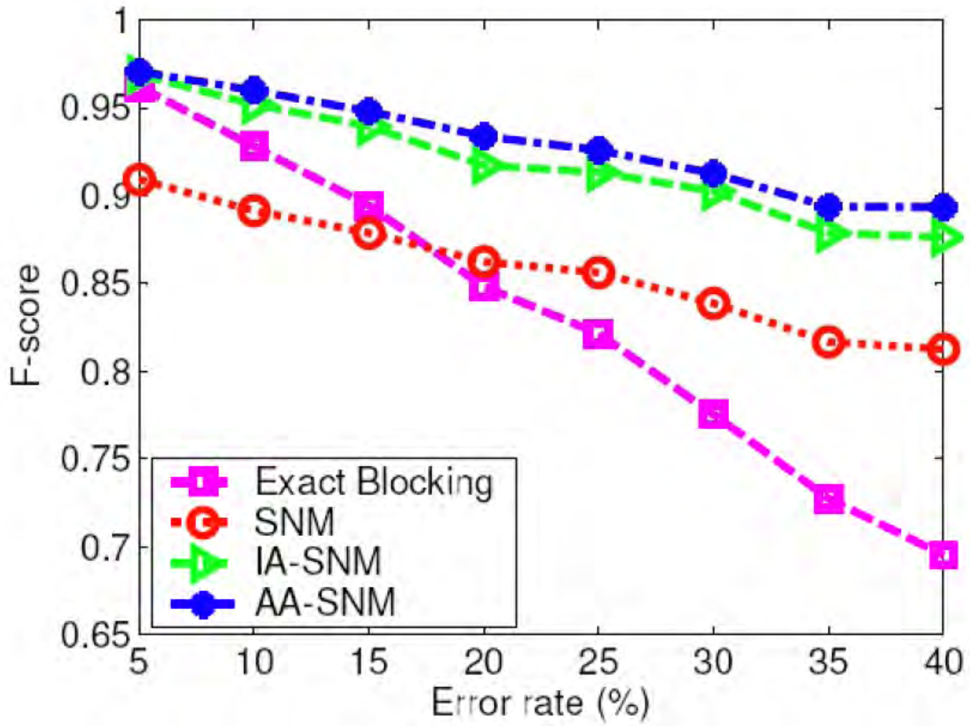


Abbildung 3: DBGen Fehlerrate aus [YLKG07]

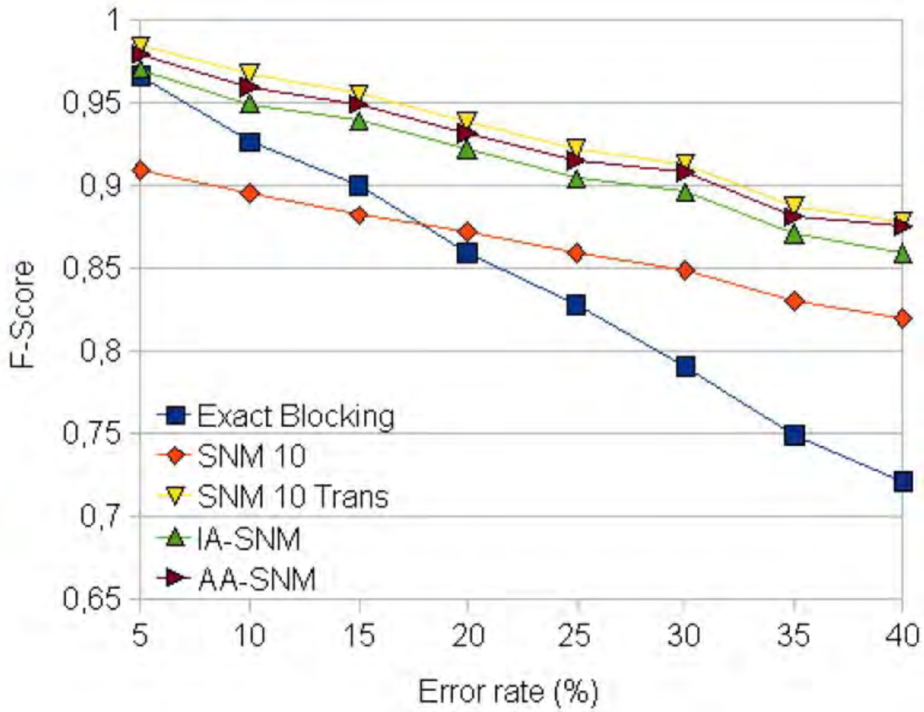
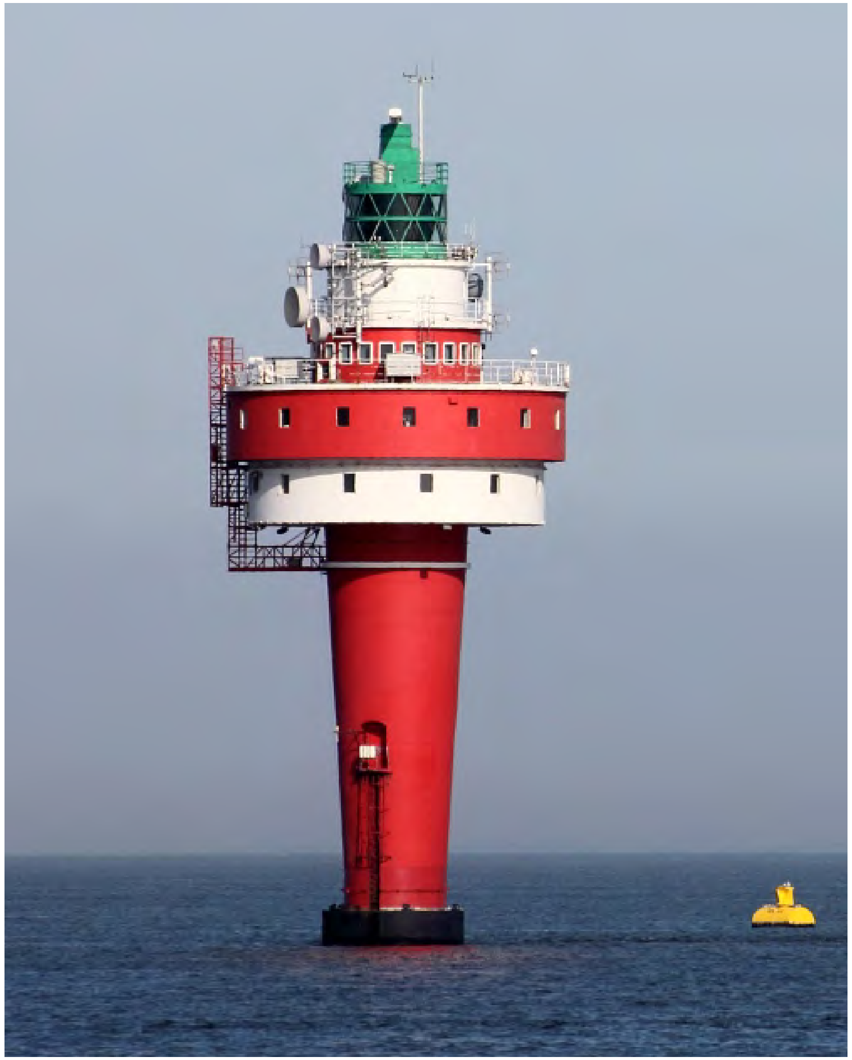


Abbildung 4: DBGen Fehlerrate Reproduziert

From: Oliver Wonneberg, *Entlarvung der Adaptive Sorted Neighborhood Method*, BTW 2009 Studierendenprogramm

Overview

- The Original
- Unique sorting keys
- Adaptive SNM
  - Part 1
  - Part 2
- Sorted Blocks
- Domain-independent SNM



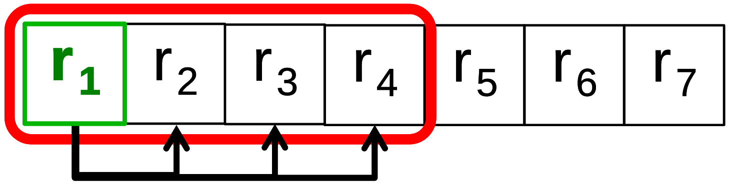
Adaptation Idea

- Vary window size based on detected duplicates
  - Adaptation can increase or reduce number of comparisons
- The more duplicates of a record are found within a window, the larger the window should be
- If no duplicate of a record within its neighborhood is found, assume that there are no duplicates or the duplicates are very far away in the sorting order.
- Each tuple t_i is once at the beginning of a window
  - Compare it with $w - 1$ successors
  - Current window: $W(i, i + w - 1)$
  - If no duplicate for t_i is found, continue as normal
  - If a duplicate is found, increase window

- Uwe Draisbach, Felix Naumann, Sascha Szott, Oliver Wonneberg. *Adaptive Windows for Duplicate Detection*. In Proceedings of the 28th International Conference on Data Engineering (ICDE), Washington, D.C., USA, 2012.

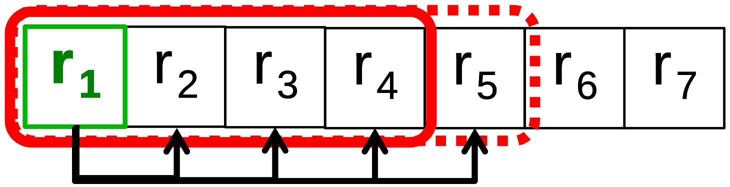
Basic Duplicate Count Strategy

1. Assign sorting key to each record and sort the records
2. Create window with initial window size *w*
3. Compare first record with all other records in the window

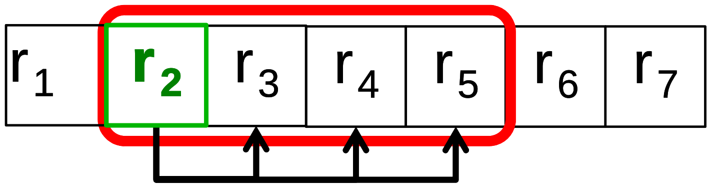


4. Increase window size while $\frac{\textit{detected duplicates}}{\textit{comparisons}} \geq \phi$

ϕ: average number of comparisons per duplicate



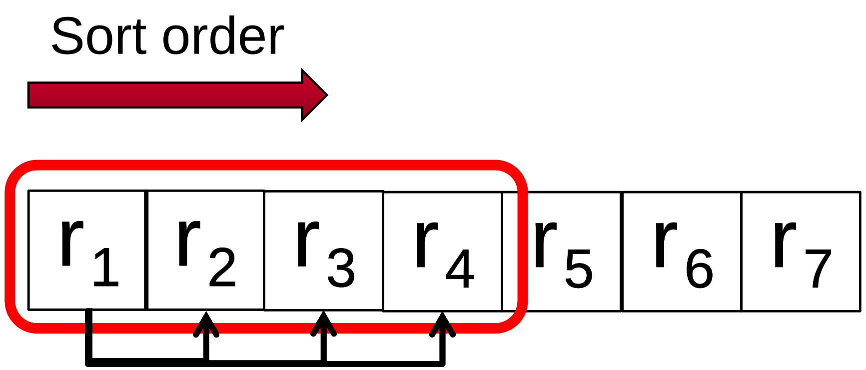
5. Slide the window (initial window size w)



6. Calculate transitive closure

Duplicate Count Strategy (*DCS*)

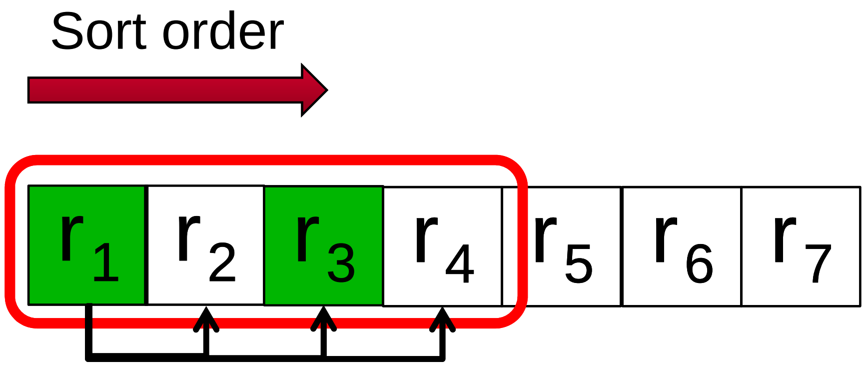
- w = initial window size
- Increase window while $\dfrac{\textit{detected duplicates}}{\textit{comparisons}} \geq \phi$



<u>Example:</u>
w = 4
ϕ = 0.30

Duplicate Count Strategy (*DCS*)

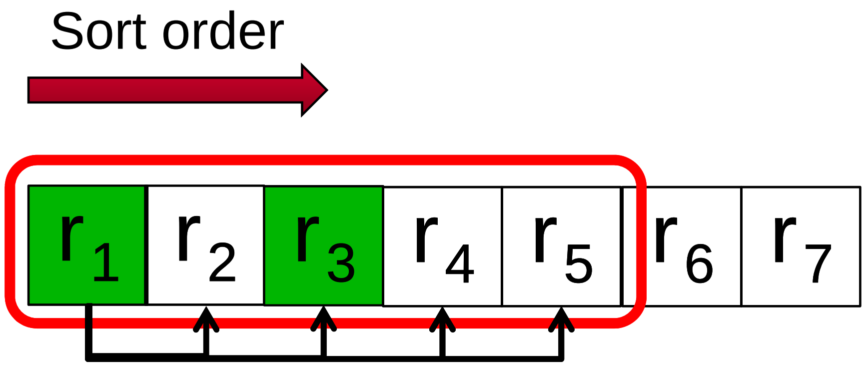
- w = initial window size
- Increase window while $\dfrac{\textit{detected duplicates}}{\textit{comparisons}} \geq \phi$



Example:
w = 4
ϕ = 0.30
d/c = 0.33

Duplicate Count Strategy (*DCS*)

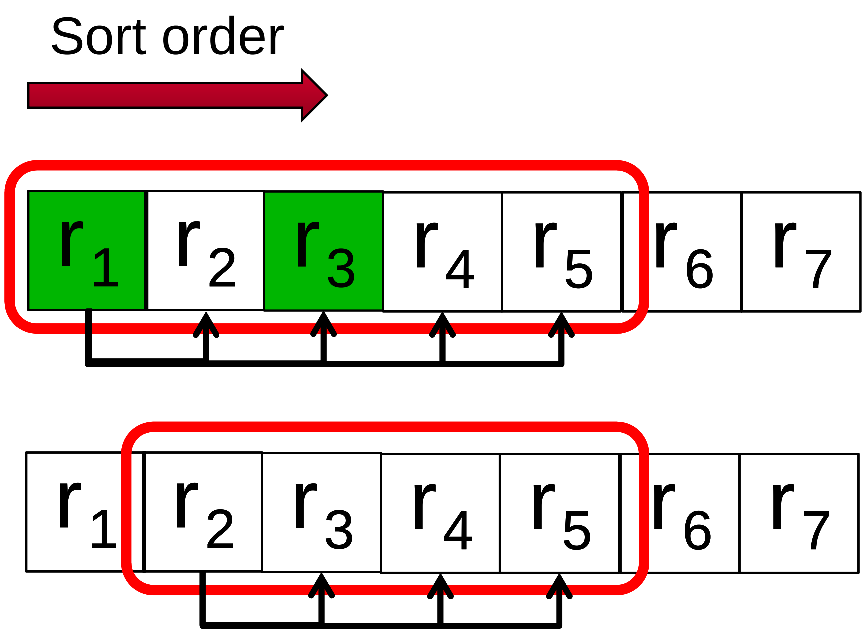
- w = initial window size
- Increase window while $\dfrac{\textit{detected duplicates}}{\textit{comparisons}} \geq \phi$



Example:
w = 4
ϕ = 0.30
d/c = 0.25

Duplicate Count Strategy (*DCS*)

- w = initial window size
- Increase window while $\dfrac{\textit{detected duplicates}}{\textit{comparisons}} \geq \phi$

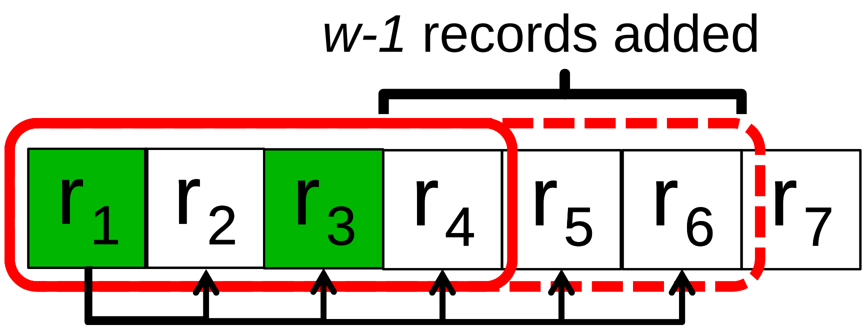


<u>Example:</u>
$w = 4$
$\phi = 0.30$
$d/c = 0.25$

$\phi = 0.30$
$d/c = 0.00$

Enhancement of *DCS: DCS++*

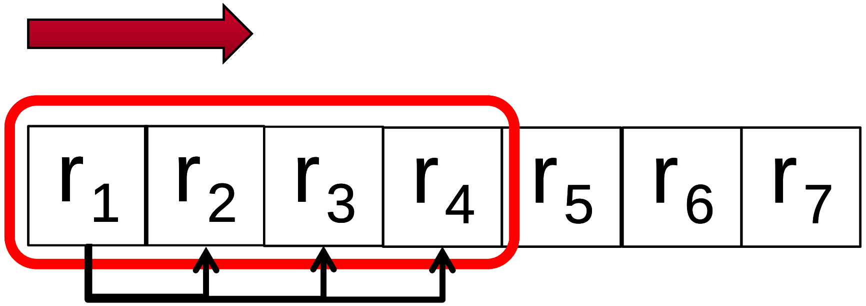
- Idea 1: In addition to *DCS*, for each detected duplicate the next *w-1* records of that duplicate are added to the window.



- Idea 2: Windows for duplicates are skipped to save comparisons
  - In example: Skip window for r_3.
  - Use the transitive closure to find additional duplicates.
  - Will not miss any, because window for r_1 covers all comparisons r_3 would have made.
  - Assumes perfect similarity measure... Can be relaxed.

DCS++

Sort order



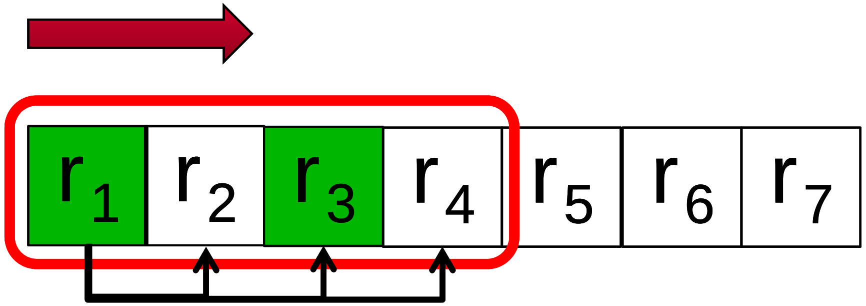
Example:

w = 4

ϕ = 0.30

DCS++

Sort order

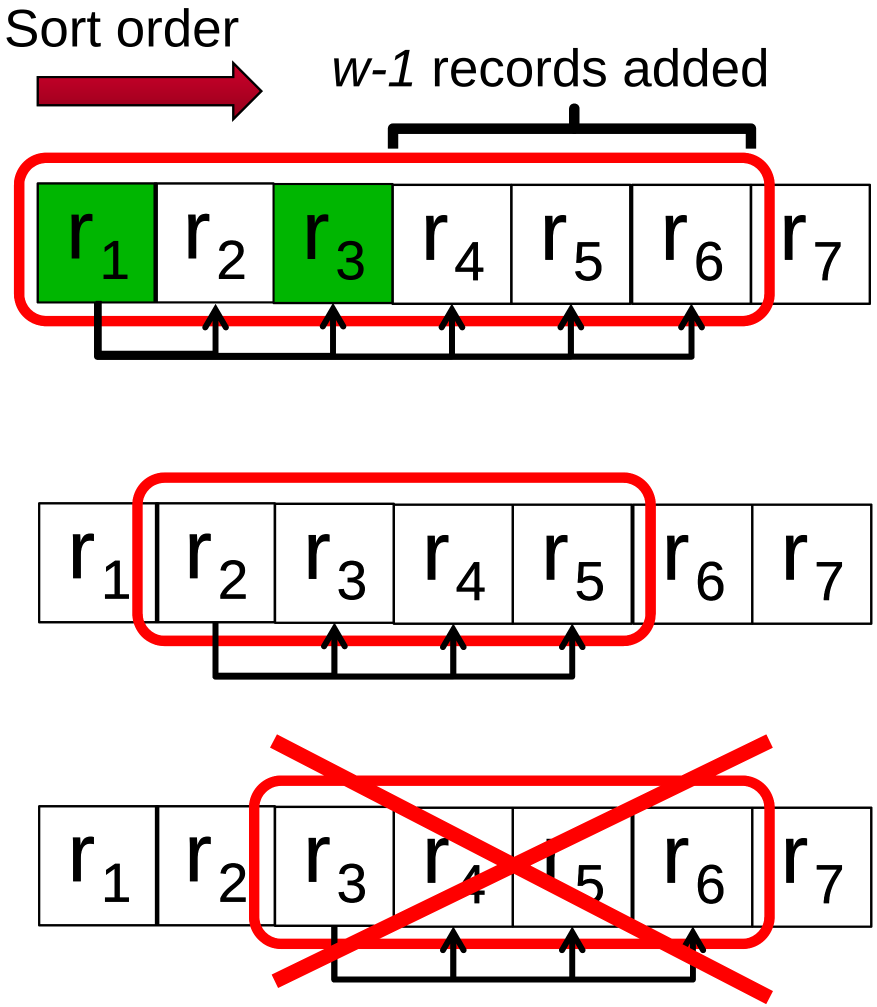


Example:

w = 4

ϕ = 0.30

DCS++



<u>Example:</u>
$w = 4$
$\phi = 0.30$
$d/c = 0.20$

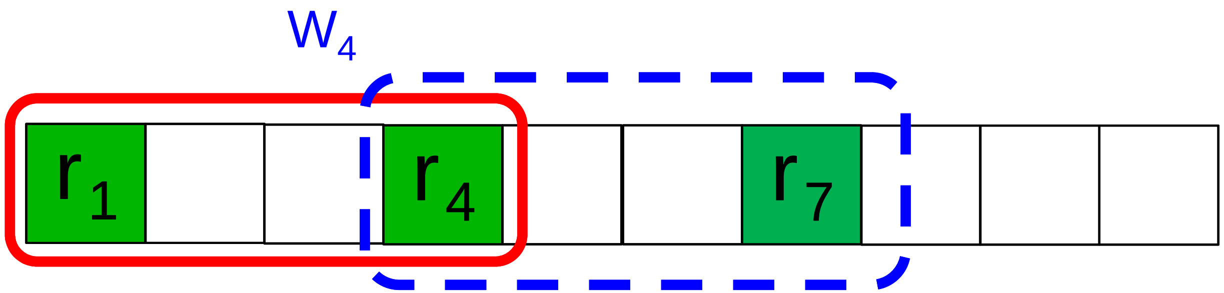
$\phi = 0.30$
$d/c = 0.00$

r_3 is duplicate of r_1

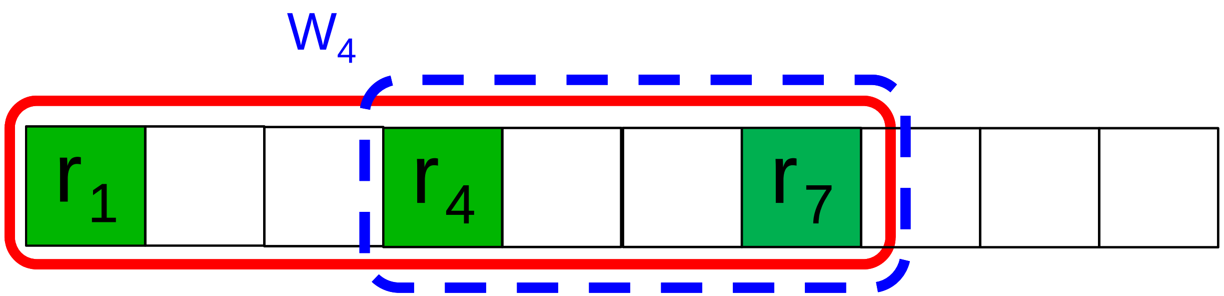
Calculation of the transitive closure will find additional duplicates of r_3

DCS++ Evaluation

- Skipping windows bears the risk to miss duplicates

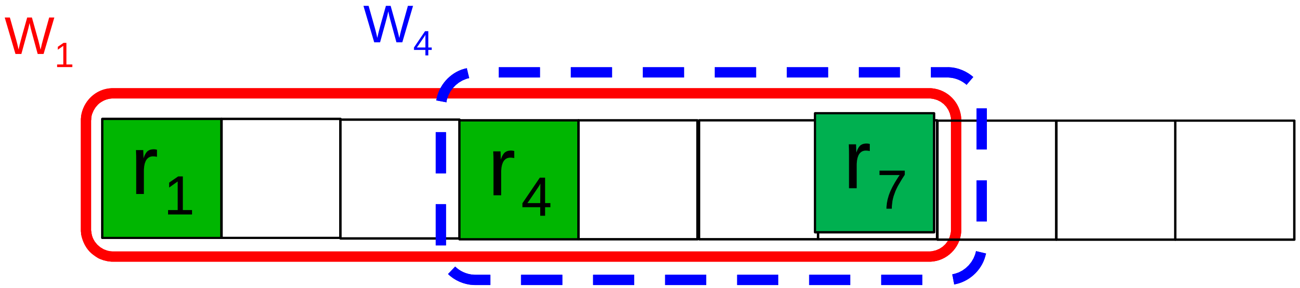


- Example: $w=4$, $\phi=1/2$
  - For w_1: $d/c = 1/3 > \phi$
  - Thus: Window is not increased, but w_4 is left out.
- Example: $w=4$, $\phi=1/3$



DCS++ Evaluation

- Skipping windows bears the risk to miss duplicates

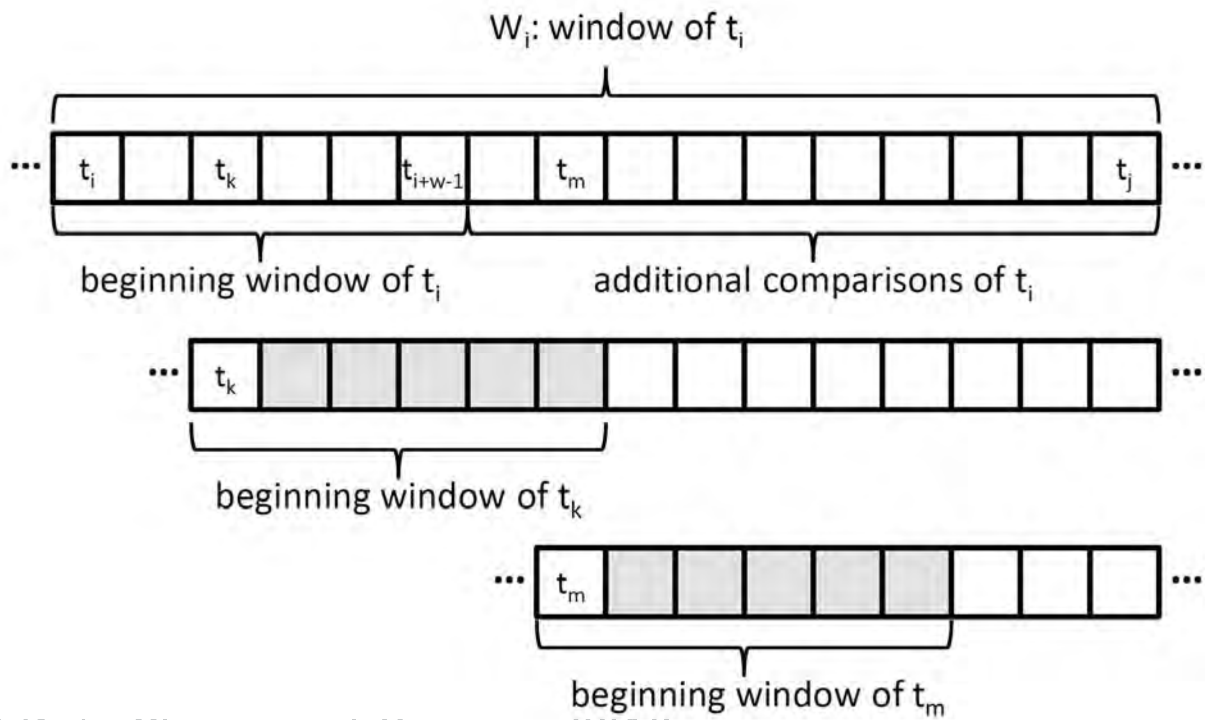


- With $\phi \leq \frac{1}{w-1}$ no duplicates will be missed due to skipping windows
- With $\phi \leq \frac{1}{w-1}$ *DCS++* is at least as efficient as *SNM* with an equivalent window size ($w_{SNM} = w_{DCS++}$)
  - Worst case: same number of comparisons
  - Best case: *DCS++* saves *w-2* comparisons per duplicate
  - Proof: Next slides

Differences in comparisons

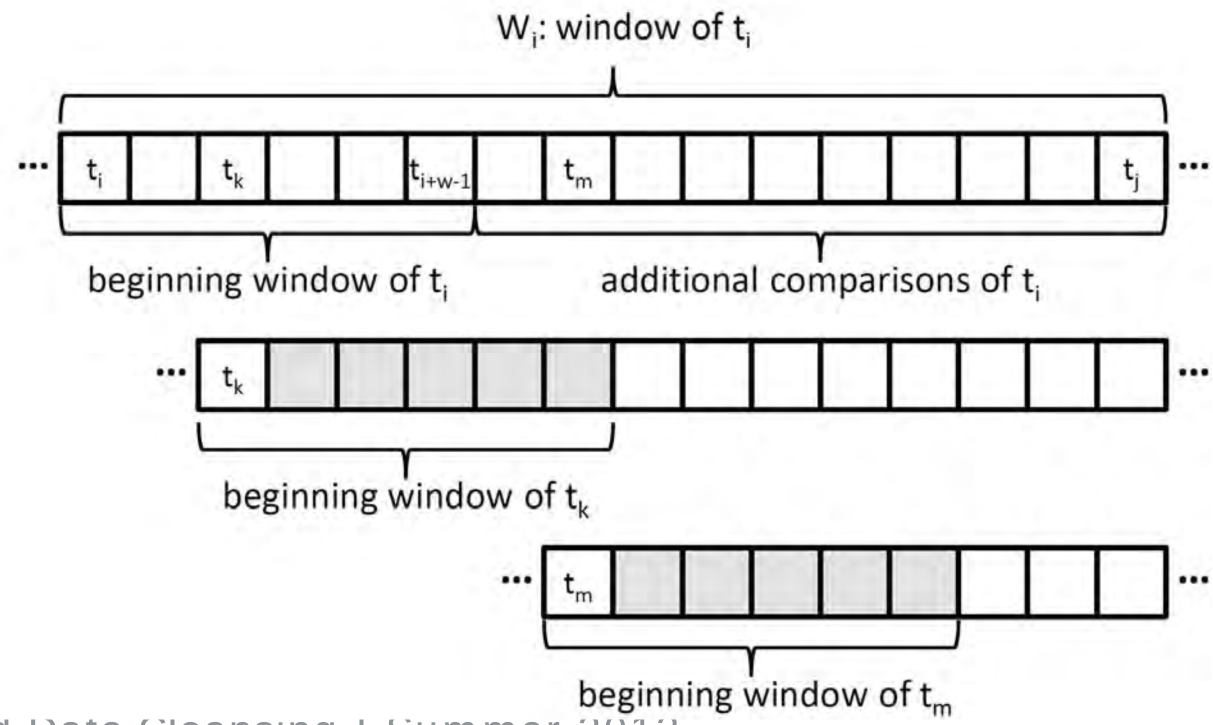
- Regard window W_i, with d detected duplicates
- Comparisons within W(i,j): c = j – i
- Additional comparisons compared to SNM: a = j – i – (w – 1)
- Saved comparisons for skipped windows: s = d (w – 1)
- We want to show: a – s ≤ 0

- Case 1: Beginning window of t_i contains no duplicate
- Case 2: Beginning window of t_i contains at least one duplicate



Differences in comparisons

- Additional comparisons: $a = j - i - (w - 1)$
- Saved comparisons: $s = d\,(w - 1)$
- Case 1: Beginning window of t_i contains no duplicate
- No duplicates => no window increase => $a = 0$
- No duplicates => no skipped windows => $s = 0$
- $a - s = 0 - 0 \leq 0$



Differences in comparisons

- Additional comparisons: $a = j - i - (w - 1)$
- Saved comparisons: $s = d\ (w - 1)$
- Case 2: Beginning window of t_i contains at least one duplicate
- $a - s = j - i - (w - 1) - d\ (w - 1)$
  $= j - i - (d + 1)\ (w - 1)$
- Window is increased until $d/c < \phi$.
- For $\phi \le {}^{1}/_{w-1}$ we need at least $c = d\ (w - 1) + 1$ comparisons to stop window increase
- *Worst case*: We find duplicate at very last comparison and increase window without any new duplicates
  - $c = d\ (w - 1) + (w - 1) \quad (= j - i)$
- $a - s = j - i \qquad - (d + 1)\ (w - 1)$
  $= d\ (w - 1) + (w - 1) - (d + 1)\ (w - 1)$
  $= (d + 1)\ (w - 1) - (d + 1)\ (w - 1)$
  $= 0$

Differences in comparisons

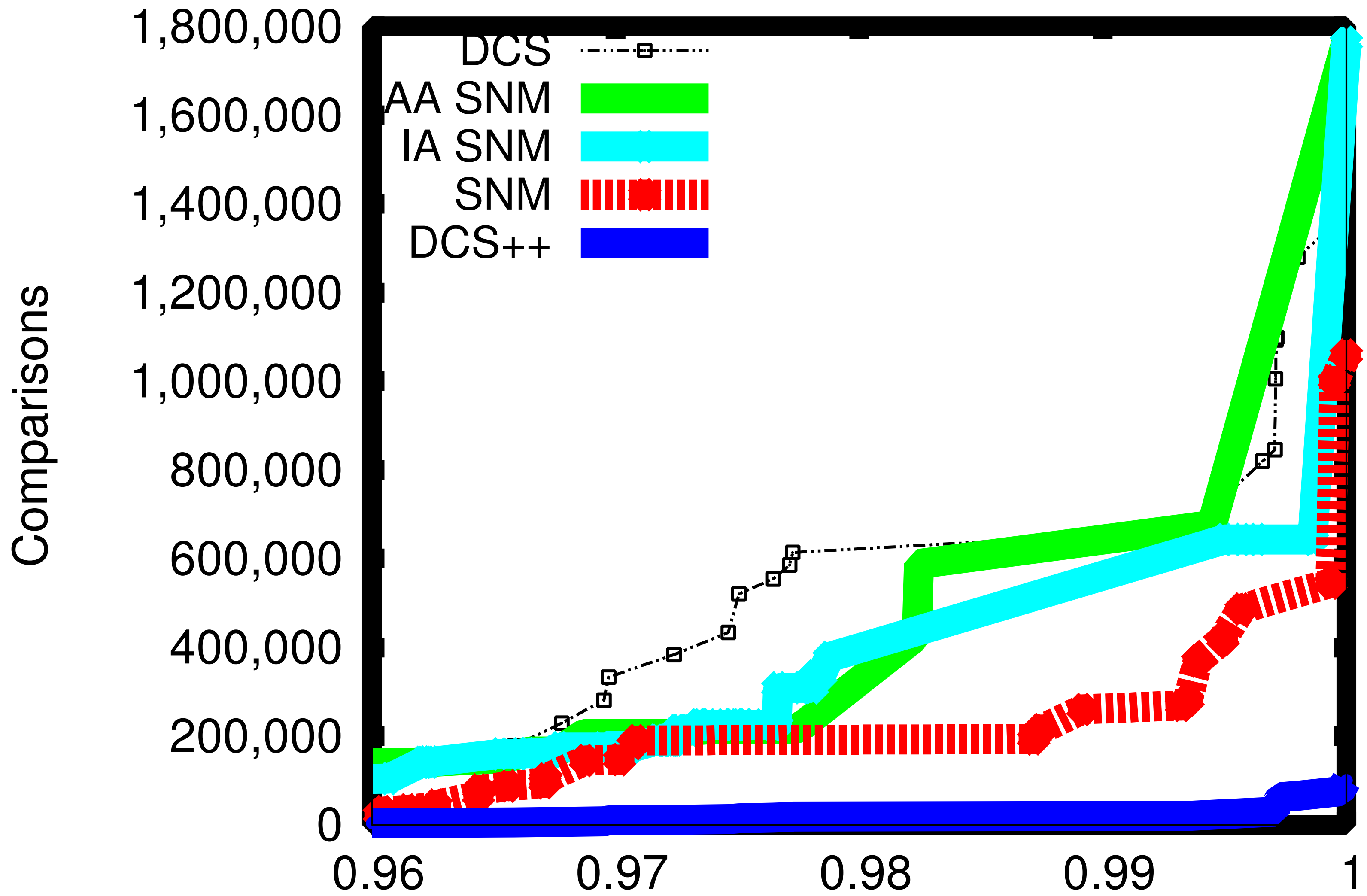
- *Worst case*: We find duplicate at very last comparison and increase window without any new duplicates
  - $c = d\,(w - 1) + (w - 1) \quad (= j - i)$
- *Best case*: We find duplicate immediately after t_i.
  - $c = d\,(w - 1) + 1 \quad (= j - i)$
- $$\begin{aligned} a - s &= j - i \qquad\qquad - (d + 1)\,(w - 1) \\ &= d\,(w - 1) + 1 - (d + 1)\,(w - 1) \\ &= 1 - (w - 1) \\ &= 2 - w \end{aligned}$$
- Can save up to 2 – w per duplicate compared to SNM

Experimental Evaluation

Data set	Provenance	# of records	# of dupl. pairs	Max. cluster size
Cora	real-world	1,879	64,578	238
Febrl	synthetic	300,009	101,153	10
Persons	synthetic	1,039,776	89,784	2

- Perfect classifier (lookup in the gold standard)
- Algorithms
  - Sorted Neighborhood Method (*SNM*)
  - Duplicate Count Strategy (*DCS* / *DCS++*)
  - Adaptive SNM (*AA SNM* / *IA SNM*) (previous slides)

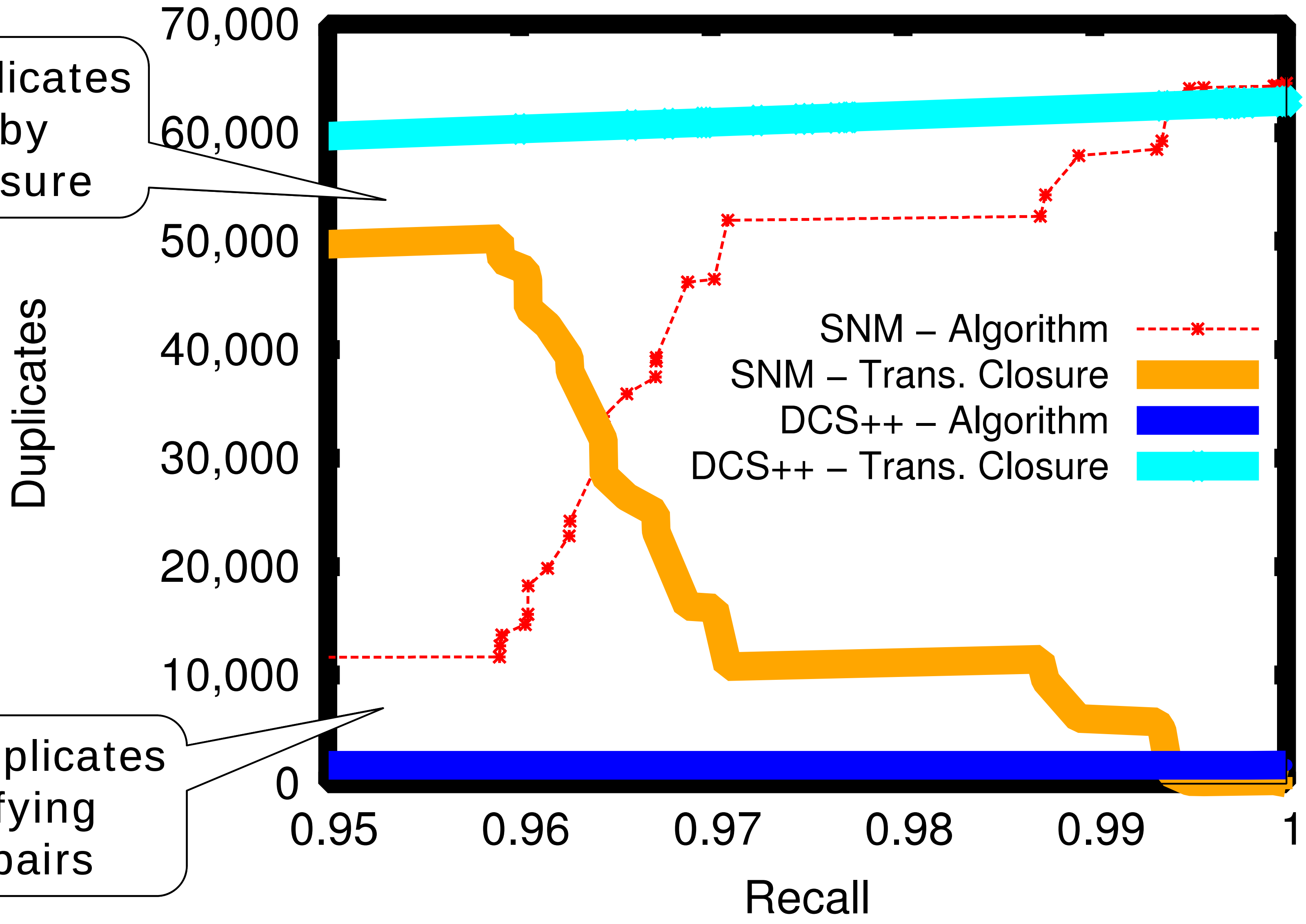
Results Cora: Comparisons



$$recall = \frac{\#\ \text{detected duplicates}}{\#\ \text{real duplicates}}$$

Results Cora: Duplicate Provenance

Additional duplicates calculated by transitive closure



Detected duplicates by classifying created pairs

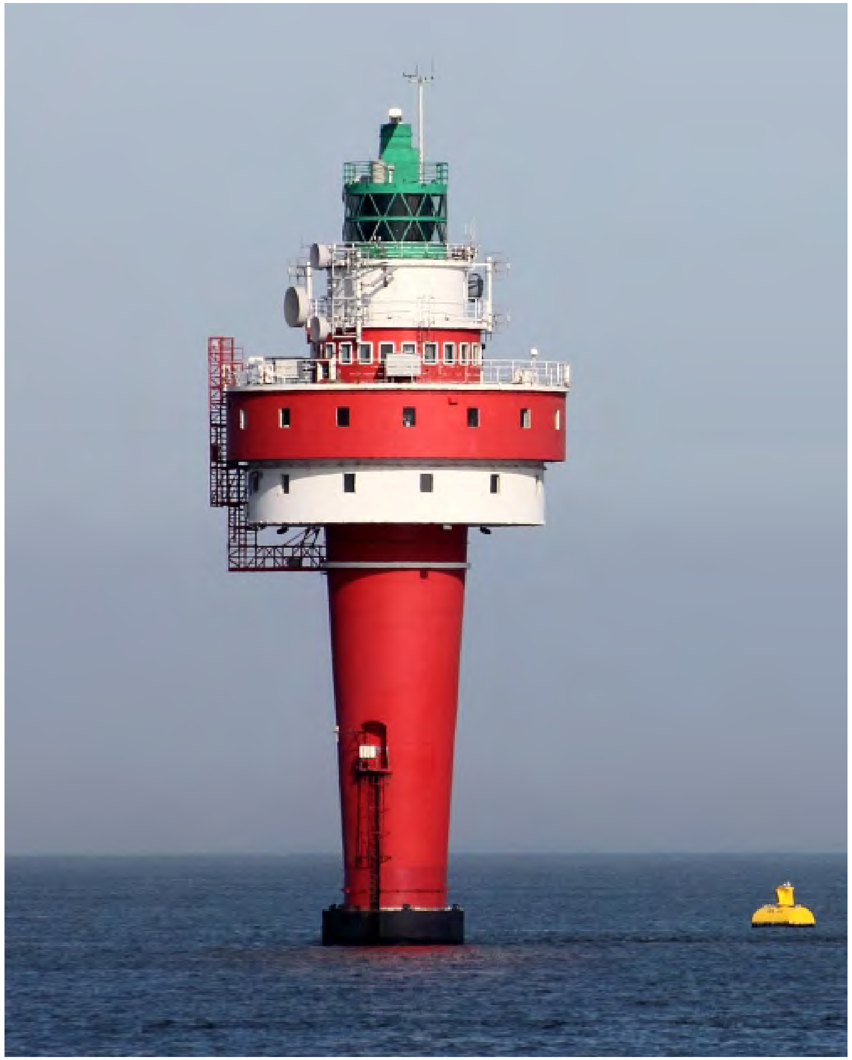
Other Variants

- From Master thesis of Oliver Wonneberg
- Sorting key strategy
  - Increase window if sorting keys are similar
  - Decrease window size for dissimilar sorting keys
  - Use different sizes of increase (depending on similarity)
- Similarity strategy
  - Same as before, but based on tuple similarity

- Difficult to calibrate

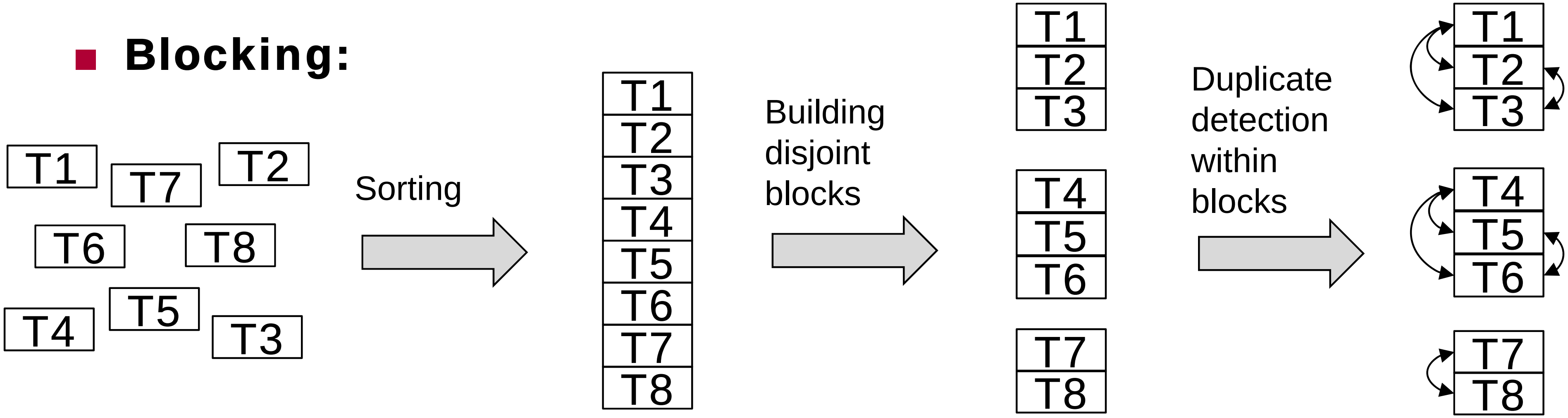
Overview

- The Original
- Unique sorting keys
- Adaptive SNM
  - Part 1
  - Part 2
- Sorted Blocks
- Domain-independent SNM

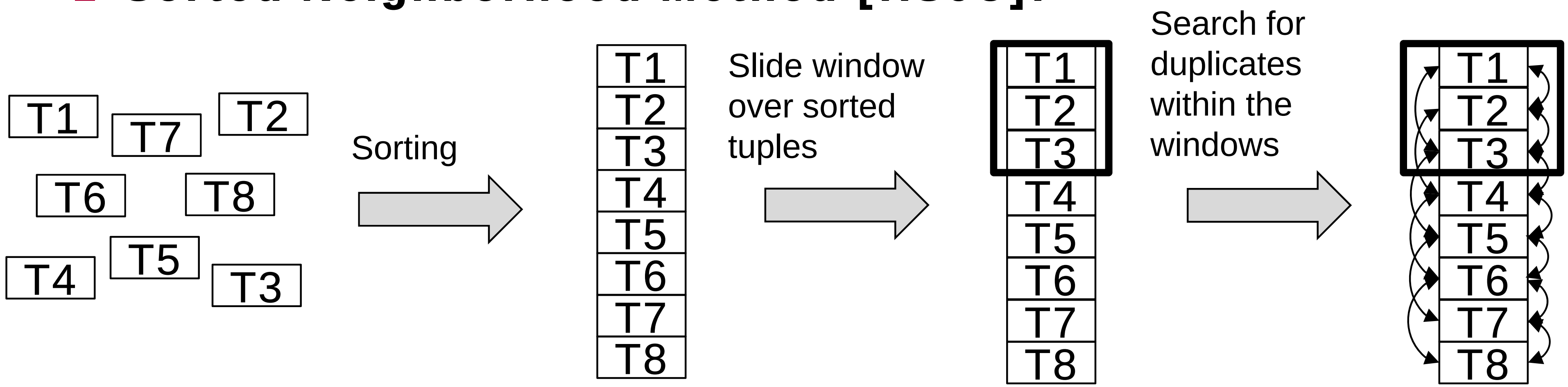


Blocking and Windowing Algorithms

- **Blocking:**



- **Sorted Neighborhood Method [HS98]:**



Comparing Blocking and Windowing

SNM

Blocking

Window size: 3

Block size: 5

Sorted tuples

Tuples 1 & 5 are only compared using Blocking

Tuples 16 & 14 are only compared using SNM

Comparing Blocking and Windowing

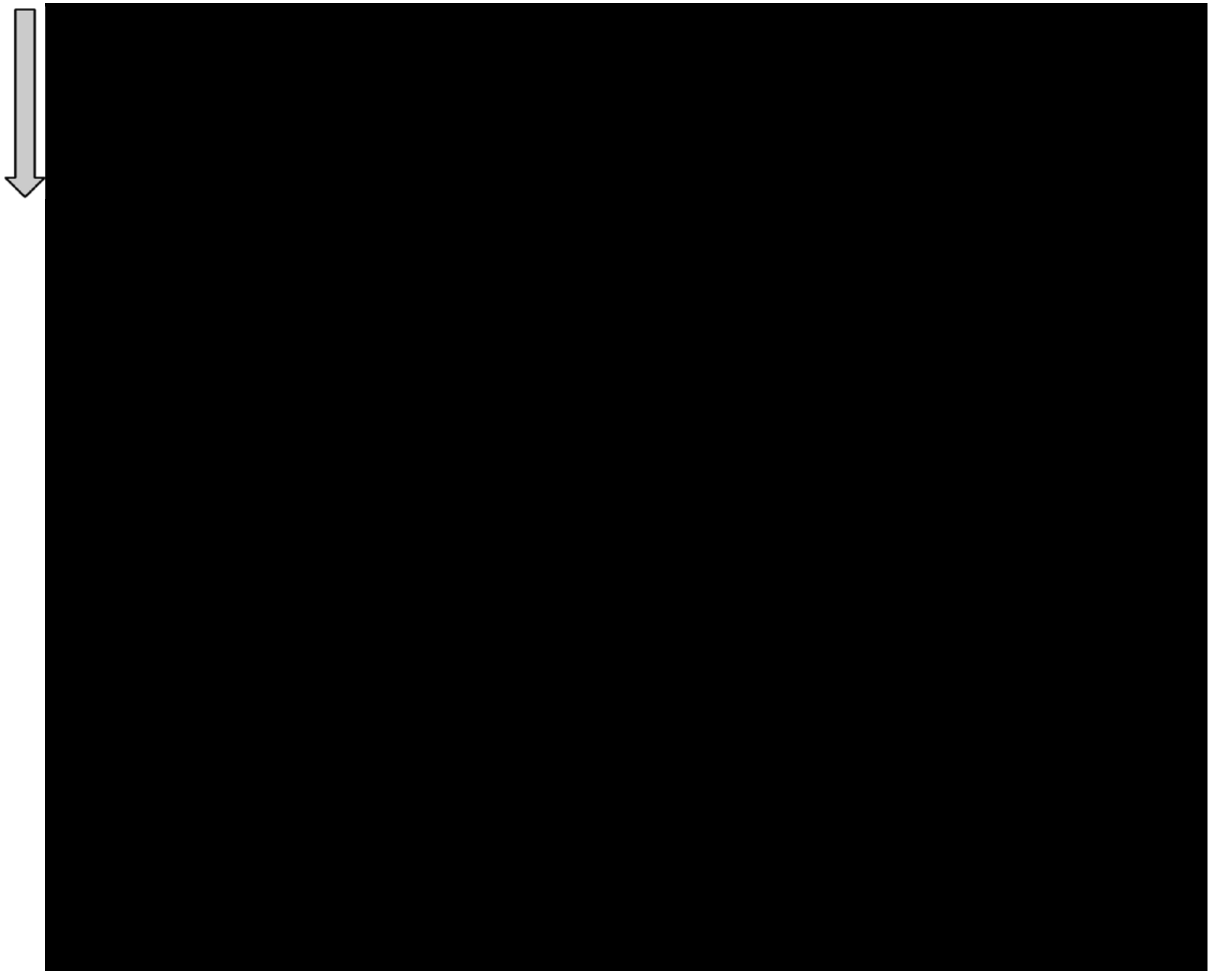
SNM

Blocking

Window size: 5

Block size: 5

Sorted tuples



Increasing window size to approximate Blocking

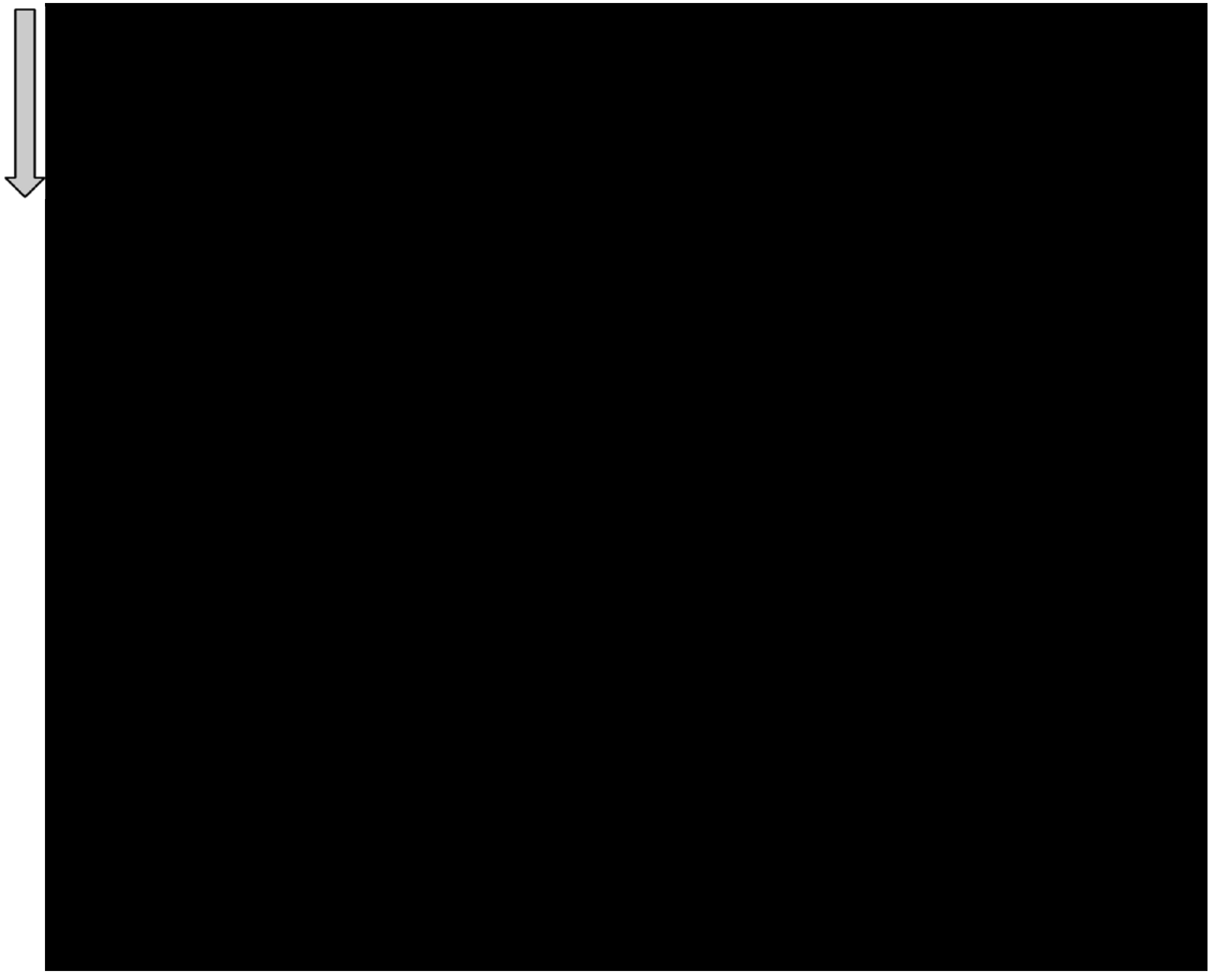
Comparing Blocking and Windowing

SNM

Blocking

Window size: 3
Block size: 5

Sorted tuples



Overlapping blocks to approximate Windowing

Sorted Blocks Method

- Generalization of blocking and windowing
- Approach

Blocking
1. Sort records and build disjoint partitions
   - Sorting key might use more attributes than the partioning predicate
2. Perform complete comparison within partitions
3. Overlap partitions and slide fixed size window across sorted records within overlap
4. Calculate transitive closure

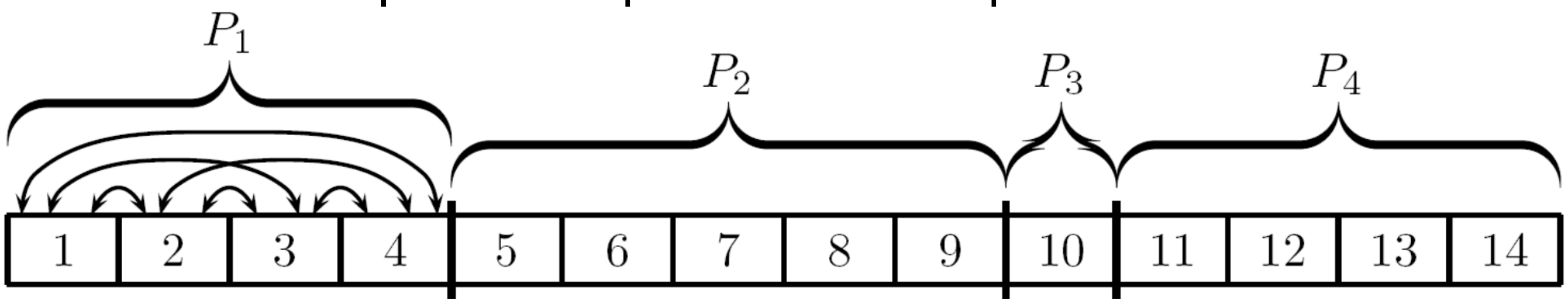
- Overlap
  - Parameter o = number of records from one partition that are part of the overlap
  - Overlap size = $2o$
  - Size of window = $o+1$

Uwe Draisbach and Felix Naumann. A Generalization of Blocking and Windowing Algorithms for Duplicate Detection. In Proceedings of the International Conference on Data and Knowledge Engineering (ICDKE), Milan, Italy, 2011.

Sorted Blocks Method

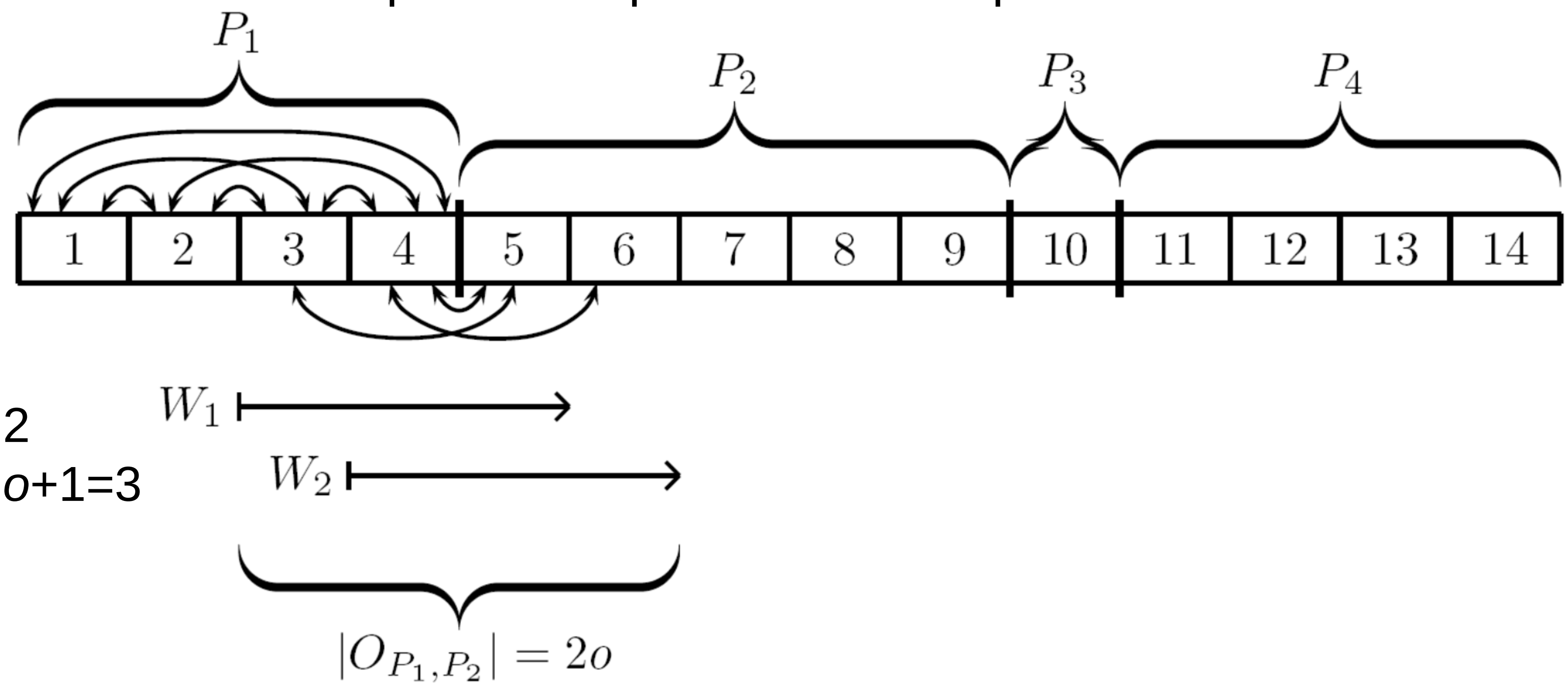
Quadratic complexity

Complete comparison within partitions



Sorted Blocks Method

Complete comparison within partitions



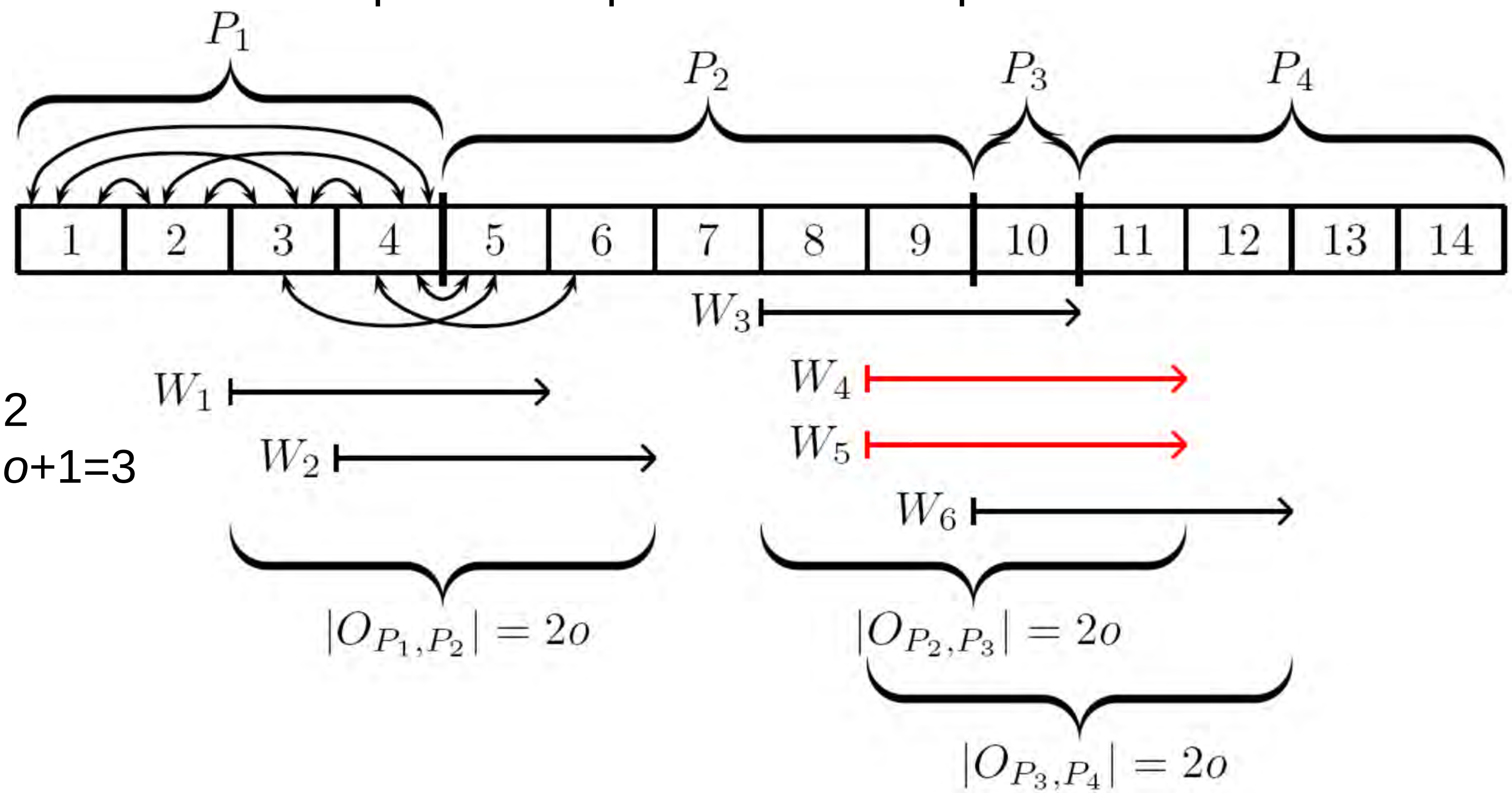
o = 2
w = o+1=3

Comparisons within overlap

Linear complexity

Sorted Blocks Method

Complete comparison within partitions



$o = 2$

$w = o+1=3$

Comparisons within overlap

Sorted Blocks Configurations

```
sort records on key
/* initialization */
listComparisonRecords ← [] // List of records that are compared with the currently processed record
windowNr ← o+1 // Number of the window in the overlapping area
i ← 1
/* iterate over all records and search for duplicates */
while i ≤ records.length do
    if records[i] is 1st element of new partition and i > 1 then
        while listComparisonRecords.length > o do
            listComparisonRecords.remove[1]
        end while
        windowNr ← 1
    else if windowNr ≤ o then
        listComparisonRecords.remove[1]
        windowNr ← windowNr + 1
    end if
    /* compare current record with all records in listComparisonRecords */
    for j = 1 to listComparisonRecords.length do
        compare records[i] with listComparisonRecords[j]
    end for
    listComparisonRecords.append(records[i])
    i ← i + 1
end while
calculate transitive closure
```

Choose o = 1 for Blocking

Choose o = w and evalute to true for SNM

Complexity Analysis

	Method			
	Blocking	**Windowing**	**Sorted Blocks** (fixed partition size)	**Full enumeration**
Key generation	$O(n)$	$O(n)$	$O(n)$	---
Sorting	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	---
Detection	$O(\frac{n^2}{2b})$	$O(wn)$	$O(\frac{nm}{2})$	$O(\frac{n^2}{2})$
Overall	$O(n(\frac{n}{2b} + \log n))$	$O(n(w + \log n))$	$O(n(\frac{m}{2} + \log n))$	$O(\frac{n^2}{2})$

- n = number of tuples
- b = number of blocks
- w = window size
- m = partition size

Sorted Blocks variants

- Overall execution time for Sorted Blocks is dominated by the largest blocks
  - E.g. partitioning by city results in large partitions for Berlin, London, etc.

- Use additional parameter: max. partition size

- 2 variants with maximum partition size:
  1. Create new partition when max. partition size is reached, independently of the partition predicate
  2. Slide window when max. partition size is reached
     - Similar to the Sorted Neighborhood Method for large partitions

Experimental Evaluation

- DuDe-toolkit for experiment execution (http://tinyurl.com/dude-toolkit)
- 8 algorithms
  - Sorted Blocks – basic
  - Sorted Blocks – fixed partition size
  - Sorted Blocks – new partition when max. size is reached
  - Sorted Blocks – slide window when max. size is reached

  - Blocking
  - Sorted Neigborhood Method

  - Incrementally-adaptive SNM (IA-SNM) [1]
  - Accumulatively-adaptive SNM (AA-SNM) [1]

[1] Yan et al. (2007), Adaptive sorted neighborhood methods for efficient record linkage

Experimental Evaluation

- 3 datasets (real-world and artificial)

Dataset	Type	Records	Duplicate pairs
CD [1]	real-world	9,763	299
Restaurant [2]	real-world	864	112
Address data	artificial	1,039,776	89,784

[1] http://www.freedb.org
[2] http://www.cs.utexas.edu/users/ml/riddle/data.html

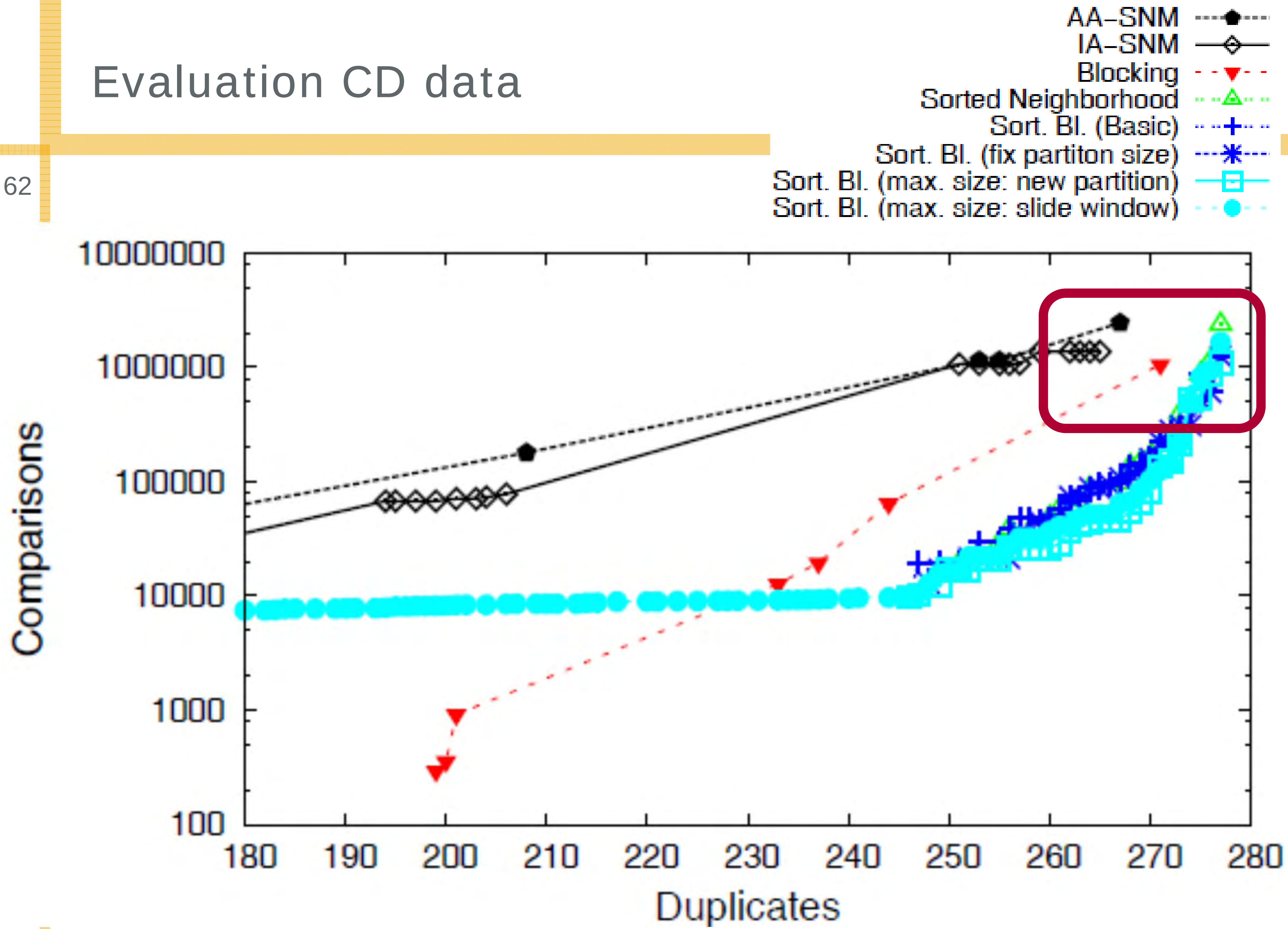
- Varying settings for
  - overlap parameter *o*
  - Partition predicate
  - Max. partition size

Evaluation CD data

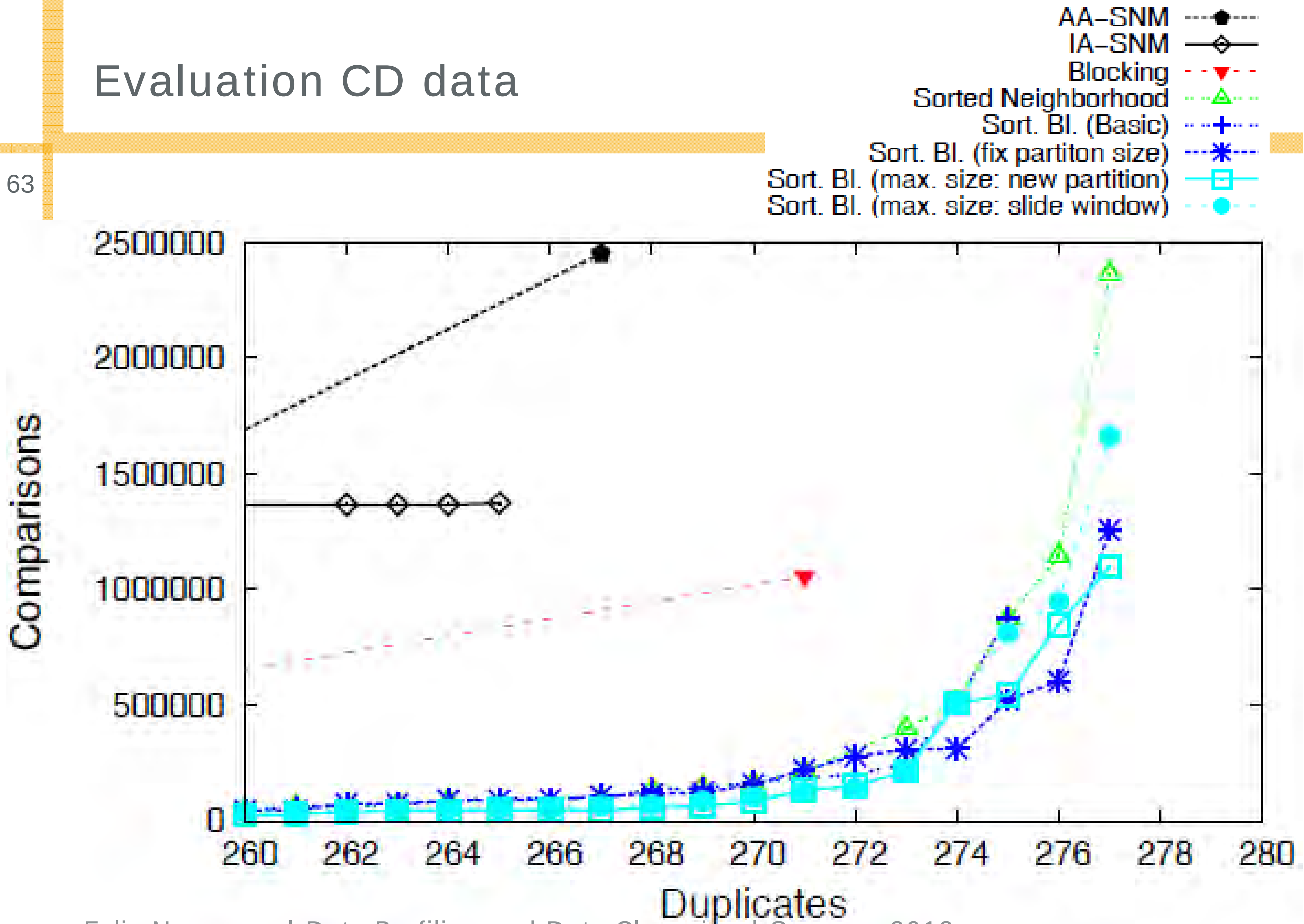
Dataset	Type	Records	Duplicate pairs
CD	real-world	9,763	299
Restaurant	real-world	864	112
Address data	artificial	1,039,776	89,784

- Sorting key: first few letters of artist, CD title, and track 01
- Partition predicate: first 1-9 letters of the sorting key
- Overlap o: 1-100
- Max. partition size: 2-1000

Evaluation CD data



Evaluation CD data

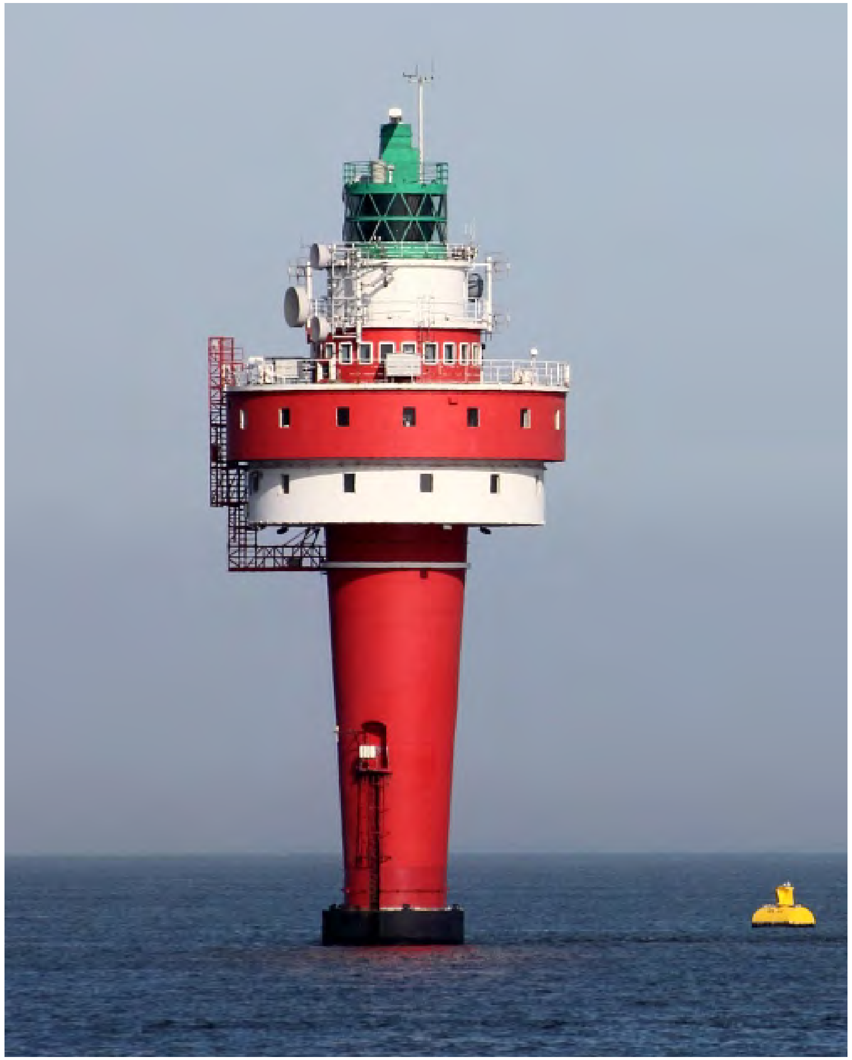


Sorted Blocks Conclusion

- Blocking and windowing are competitive approaches to reduce the number of comparisons
  - Sorted Neighborhood outperforms Blocking slightly
- Sorted Blocks is a generalization of blocking and windowing
  - Sorted Blocks outperforms Sorted Neighborhood slightly
- Experimental evaluation shows that it is superior to windowing and blocking.
- Configuration is more difficult as it has more parameters than the other 2 approaches.

Overview

- The Original
- Unique sorting keys
- Adaptive SNM
  - Part 1
  - Part 2
- Sorted Blocks
- Domain-independent SNM



Two Ideas for Domain Independence

- Domain-independent key definition
  - Define key and reverse key
  - Union-find data structure compares only representatives of each cluster
    - Relaxation of Christen-idea from before (unique sorting key)

- A. E. Monge and C. Elkan, “An efficient domain-independent algorithm for detecting approximately duplicate database records,” in Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD), 1997

Two Passes

- Regard each tuple as a single long string
  - Concatenate all attribute values
- 1st pass: Key = tuple
- 2nd pass: Key = reversed tuple

- "No" dependency on good key choice

- Similarity measure: Smith-Waterman
  - Suitable for long strings

Union-find Data Structure

- Interpret result as graph
  - Connected components represent duplicate clusters
  - Graph is transitive closure
- Compare next tuple only once with each connected component
- Union-find data structure (Robert Tarjan, Journal of the ACM, 1975)
  - Collection of disjoint updateable sets
  - Each set is identified by a representative
  - Initialized with |R| singletons
- Union(x,y)
  - Unions the sets containing tuples x and y to new set, deletes old sets
  - Chooses new prime representative
- Find(x)
  - Returns unique representative of set containing x
- For each detected duplicate <u,v>:
  - If Find(u) ≠ Find(v) then Union(u,v)
- Two nodes u and v are in same connected component ⇔ Find(u) = Find(v)

Union-find Data Structure

- Define **prime representative** for each detected duplicate group
- Compare records first to the representatives
  - avoiding comparisons that can be derived through transitivity.
- Similar to Swoosh idea, but records keep their identity
- If the similarity is high enough (some intermediate threshold), compare with other members of cluster
- Slight improvement: Allow multiple representatives
  - To represent large variety of tuples in cluster

Algorithm

- Priority Queue: Contains sets of tuples
  - Fixed size (≈ window size)
  - Sorted by recency of addition: Queue represents last few detected clusters
- Sort records by key (2 passes)
- For each record r
  - Test if r already part of a cluster in queue:
    - Improvement: Ignore step if first pass
    - Find(r) based on representatives
    - If successful: move cluster up in queue
    - If not successful: similarity comparison with all representatives
      - If similar:
        - » Union(r,x)
        - » Make r representative if not too similar
        - » break
  - Else: r is new singleton cluster at top of queue

Summary

- The Original
- Unique sorting keys
- Adaptive SNM
  - Part 1
  - Part 2
- Sorted Blocks
- Domain-independent SNM