Efficient Parallel Set-Similarity Joins Using MapReduce

Rares Vernica

Department of Computer Science
University of California, Irvine

SIGMOD 2010
Joint work with Michael J. Carey and Chen Li (UC Irvine)

Outline

1. Motivation
2. Problem Statement
3. Single Machine Algorithms
   - Inverted List Index
   - Set-Similarity Filters
4. Parallel Algorithms
5. Experimental Evaluation

Example 1: Data Cleaning/Master-Data-Management

Customer data from two departments

Sales

ID	Name	...
S10	John W Smith	...
⋮		

Returns

ID	Name	...
R20	Smith John	...
⋮		

Master customer data across two departments

Customers

ID	Name	...
C30	John W Smith	...
⋮		

Example 1: Data Cleaning/Master-Data-Management

Customer data from two departments

Sales

ID	Name	...
S10	**John W Smith**	...
⋮		

Returns

ID	Name	...
R20	**Smith John**	...
⋮		

Master customer data across two departments

Customers

ID	Name	...
C30	John W Smith	...
⋮		

Example 1: Data Cleaning/Master-Data-Management

Customer data from two departments

Sales

ID	Name	...
S10	`John W Smith`	...
⋮		

Returns

ID	Name	...
R20	`Smith John`	...
⋮		

Master customer data across two departments

Customers

ID	Name	...
C30	**`John W Smith`**	...
⋮		

Example 1: Data Cleaning/Master-Data-Management

String → Set

- S10: "`John W Smith`" → {`John, Smith, W`}
- R20: "`Smith John`" → {`John, Smith`}

Set-Similarity Metric

- Jaccard similarity/Tanimoto coefficient: $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$
- $jaccard(S10, R20) = \frac{2}{3}$

Set-Similarity Join

Set-Similarity Join "`Sales`" and "`Returns`" on "`Name`"

Example 1: Data Cleaning/Master-Data-Management

String → Set

- S10: “`John W Smith`” → {`John, Smith, W`}
- R20: “`Smith John`” → {`John, Smith`}

Set-Similarity Metric

- Jaccard similarity/Tanimoto coefficient: $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$
- $jaccard(S10, R20) = \frac{2}{3}$

Set-Similarity Join

Set-Similarity Join “`Sales`” and “`Returns`” on “`Name`”

Example 1: Data Cleaning/Master-Data-Management

String → Set

- S10: "`John W Smith`" → {`John, Smith, W`}
- R20: "`Smith John`" → {`John, Smith`}

Set-Similarity Metric

- Jaccard similarity/Tanimoto coefficient: $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$
- $jaccard(S10, R20) = \frac{2}{3}$

Set-Similarity Join

Set-Similarity Join "`Sales`" and "`Returns`" on "`Name`"

Example 1: Data Cleaning/Master-Data-Management

String → Set

- S10: "`John W Smith`" → `{John, Smith, W}`
- R20: "`Smith John`" → `{John, Smith}`

Set-Similarity Metric

- Jaccard similarity/Tanimoto coefficient: $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$
- $jaccard(S10, R20) = \frac{2}{3}$

Set-Similarity Join

Set-Similarity Join "`Sales`" and "`Returns`" on "`Name`"

Example 1: Data Cleaning/Master-Data-Management

String → Set

- S10: "`John W Smith`" → {`John, Smith, W`}
- R20: "`Smith John`" → {`John, Smith`}

Set-Similarity Metric

- Jaccard similarity/Tanimoto coefficient: $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$
- $jaccard(S10, R20) = \frac{2}{3}$

Set-Similarity Join

Set-Similarity Join "`Sales`" and "`Returns`" on "`Name`"

Example 1: Data Cleaning/Master-Data-Management

String $\rightarrow$ Set

- S10: "`John W Smith`" $\rightarrow$ `{John, Smith, W}`
- R20: "`Smith John`" $\rightarrow$ `{John, Smith}`

Set-Similarity Metric

- Jaccard similarity/Tanimoto coefficient: $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$
- $jaccard(S10, R20) = \frac{2}{3}$

Set-Similarity Join

Set-Similarity Join "`Sales`" and "`Returns`" on "`Name`"

Example 1: Data Cleaning/Master-Data-Management

String → Set

- S10: "`John W Smith`" → `{John, Smith, W}`
- R20: "`Smith John`" → `{John, Smith}`

Set-Similarity Metric

- Jaccard similarity/Tanimoto coefficient: $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$
- $jaccard(S10, R20) = \frac{2}{3}$

Set-Similarity Join

Set-Similarity Join "`Sales`" and "`Returns`" on "`Name`"

Example 2: Social Networking Recommendations

Current user's profile

Name:	**Alice**
School:	`UCI`
Emp.:	`IBM`
Hobby:	`Running`

Related user's profiles

Name:	**Bob**
School:	`Harvard`
Emp. :	`IBM`
Hobby:	`Running`

Name:	**John**
School:	`UCI`
Emp.:	`Yahoo`
Hobby:	`Running`

Name:	**Mike**
School:	`MIT`
Emp.:	`IBM`
Member:	`ACM`

Example 2: Social Networking Recommendations

Current user's profile

Name:	**Alice**
School:	`UCI`
Emp.:	`IBM`
Hobby:	`Running`

Related user's profiles

Name:	**Bob**
School:	`Harvard`
Emp. :	`IBM`
Hobby:	`Running`

Name:	**John**
School:	`UCI`
Emp.:	`Yahoo`
Hobby:	`Running`

Name:	**Mike**
School:	`MIT`
Emp.:	`IBM`
Member:	`ACM`

Example 2: Social Networking Recommendations

Profile → Set

- Alice: `{Emp.:IBM, Hobby:Running, School:UCI}`
- Bob: `{Emp.:IBM, Hobby:Running, School:Harvard}`
- John: `{Emp.:Yahoo, Hobby:Running, School:UCI}`
- Mike: `{Emp.:IBM, Member:ACM, School:MIT}`

Set-Similarity Metric (*jaccard*)

- $jaccard(Alice, Bob) = \frac{2}{4}$
- $jaccard(Alice, John) = \frac{2}{4}$
- $jaccard(Alice, Mike) = \frac{1}{5}$

Set-Similarity Self-Join

Set-Similarity Self-Join users on profile

Example 2: Social Networking Recommendations

Profile → Set

- Alice: `{Emp.:IBM, Hobby:Running, School:UCI}`
- Bob: `{Emp.:IBM, Hobby:Running, School:Harvard}`
- John: `{Emp.:Yahoo, Hobby:Running, School:UCI}`
- Mike: `{Emp.:IBM, Member:ACM, School:MIT}`

Set-Similarity Metric (*jaccard*)

- $jaccard(Alice, Bob) = \frac{2}{4}$
- $jaccard(Alice, John) = \frac{2}{4}$
- $jaccard(Alice, Mike) = \frac{1}{5}$

Set-Similarity Self-Join

Set-Similarity Self-Join users on profile

Example 2: Social Networking Recommendations

Profile → Set

- Alice: `{Emp.:IBM, Hobby:Running, School:UCI}`
- Bob: `{Emp.:IBM, Hobby:Running, School:Harvard}`
- John: `{Emp.:Yahoo, Hobby:Running, School:UCI}`
- Mike: `{Emp.:IBM, Member:ACM, School:MIT}`

Set-Similarity Metric (*jaccard*)

- $jaccard(Alice, Bob) = \frac{2}{4}$
- $jaccard(Alice, John) = \frac{2}{4}$
- $jaccard(Alice, Mike) = \frac{1}{5}$

Set-Similarity Self-Join

Set-Similarity Self-Join users on profile

Example 2: Social Networking Recommendations

Profile → Set

- Alice: `{Emp.:IBM, Hobby:Running, School:UCI}`
- Bob: `{Emp.:IBM, Hobby:Running, School:Harvard}`
- John: `{Emp.:Yahoo, Hobby:Running, School:UCI}`
- Mike: `{Emp.:IBM, Member:ACM, School:MIT}`

Set-Similarity Metric (*jaccard*)

- $jaccard(Alice, Bob) = \frac{2}{4}$
- $jaccard(Alice, John) = \frac{2}{4}$
- $jaccard(Alice, Mike) = \frac{1}{5}$

Set-Similarity Self-Join

Set-Similarity Self-Join users on profile

Example 2: Social Networking Recommendations

Profile $\rightarrow$ Set

- Alice: `{Emp.:IBM, Hobby:Running, School:UCI}`
- Bob: `{Emp.:IBM, Hobby:Running, School:Harvard}`
- John: `{Emp.:Yahoo, Hobby:Running, School:UCI}`
- Mike: `{Emp.:IBM, Member:ACM, School:MIT}`

Set-Similarity Metric (*jaccard*)

- $jaccard(Alice, Bob) = \frac{2}{4}$
- $jaccard(Alice, John) = \frac{2}{4}$
- $jaccard(Alice, Mike) = \frac{1}{5}$

Set-Similarity Self-Join

Set-Similarity Self-Join users on profile

Example 2: Social Networking Recommendations

Profile → Set

- Alice: `{Emp.:IBM, Hobby:Running, School:UCI}`
- Bob: `{Emp.:IBM, Hobby:Running, School:Harvard}`
- John: `{Emp.:Yahoo, Hobby:Running, School:UCI}`
- Mike: `{Emp.:IBM, Member:ACM, School:MIT}`

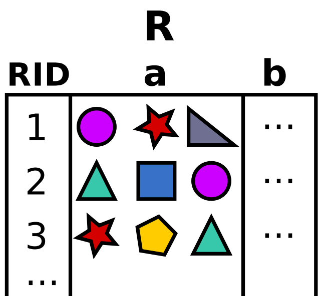
Set-Similarity Metric (*jaccard*)

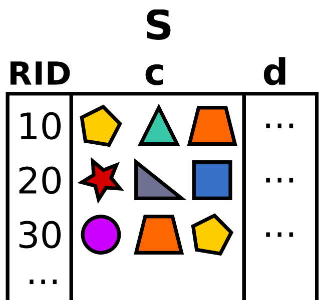
- $jaccard(Alice, Bob) = \frac{2}{4}$
- $jaccard(Alice, John) = \frac{2}{4}$
- $jaccard(Alice, Mike) = \frac{1}{5}$

Set-Similarity Self-Join

Set-Similarity Self-Join users on profile

Problem Statement: Set-Similarity Join





Problem Statement: Set-Similarity Join

R

RID	a	b
1	● ★ ◣	...
2	▲ ■ ●	...
3	★ ⬟ ▲	...
...		

$\bowtie$

R.a∼S.c
Sim = Jaccard
τ = .5

S

RID	c	d
10	⬟ ▲ ⏢	...
20	★ ◣ ■	...
30	● ⏢ ⬟	...
...		

RID_R	a	b	RID_S	c	d	Sim

Problem Statement: Set-Similarity Join

R

RID	a	b
1		...
2		...
3		...
...		

$\bowtie$

R.a∼S.c
Sim = Jaccard
τ = .5

S

RID	c	d
10		...
20		...
30		...
...		

RID_R	a	b	RID_S	c	d	Sim
1		...	20		...	.5

Problem Statement: Set-Similarity Join

R

RID	a	b
1		…
2		…
3		…
…		

S

RID	c	d
10		…
20		…
30		…
…		

$\bowtie$

R.a∼S.c

Sim = Jaccard

τ = .5

RID_R	a	b	RID_S	c	d	Sim
1		…	20		…	.5
3		…	10		…	.5
…						

Problem Statement: Set-Similarity Join

Input

- Two files of records e.g., $R(RID, a, b)$ and $S(RID, c, d)$
- A join column on each file e.g., $R.a$ and $S.c$
- A similarity function, *sim* e.g., Jaccard
- A similarity threshold, τ

Output

All pairs of records from R and S where $sim(R.a, S.c) \geq \tau$

Outline

1. Motivation
2. Problem Statement
3. Single Machine Algorithms
   - Inverted List Index
   - Set-Similarity Filters
4. Parallel Algorithms
5. Experimental Evaluation

Single Machine Set-Similarity Join

1. Nested loops
2. Inverted list index [Sarawagi and Kirpal, 2004]
   1. Indexing phase
   2. Candidate generation phase
   3. Verification phase

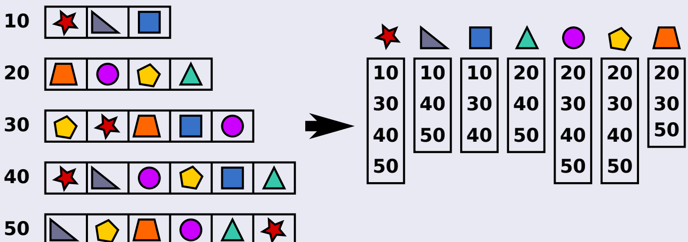
Single Machine Set-Similarity Join

1. Nested loops
2. Inverted list index [Sarawagi and Kirpal, 2004]
   1. Indexing phase
   2. Candidate generation phase
   3. Verification phase

10
20
30
40
50

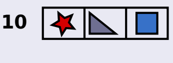
Single Machine Set-Similarity Join

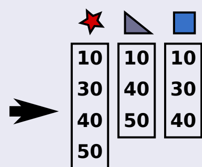
1. Nested loops
2. Inverted list index [Sarawagi and Kirpal, 2004]
   1. Indexing phase
   2. Candidate generation phase
   3. Verification phase



Single Machine Set-Similarity Join

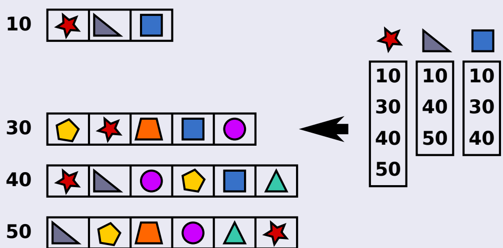
1. Nested loops
2. Inverted list index [Sarawagi and Kirpal, 2004]
    1. Indexing phase
    2. Candidate generation phase
    3. Verification phase





Single Machine Set-Similarity Join

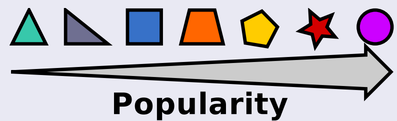
1. Nested loops
2. Inverted list index [Sarawagi and Kirpal, 2004]
   1. Indexing phase
   2. Candidate generation phase
   3. Verification phase



Set-Similarity Filtering

Prefix Filtering [Chaudhuri et al., 2006]

- Pigeonhole principle
- Global order for set elements:

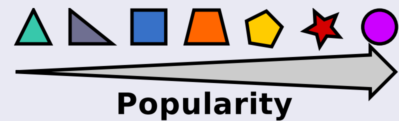


- Sort each record's token

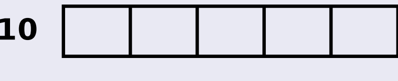
Set-Similarity Filtering

Prefix Filtering [Chaudhuri et al., 2006]

- Pigeonhole principle
- Global order for set elements:



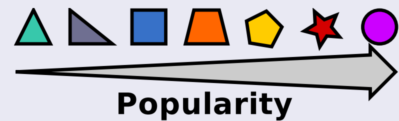
- Sort each record's token
- E.g., *sim* is overlap size, $\tau = 4$



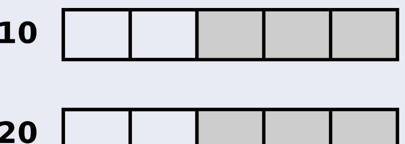
Set-Similarity Filtering

Prefix Filtering [Chaudhuri et al., 2006]

- Pigeonhole principle
- Global order for set elements:



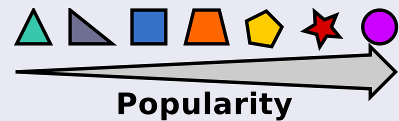
- Sort each record's token
- E.g., *sim* is overlap size, $\tau = 4$
- Prefix length is 2



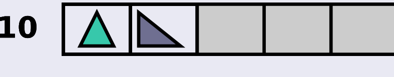
Set-Similarity Filtering

Prefix Filtering [Chaudhuri et al., 2006]

- Pigeonhole principle
- Global order for set elements:



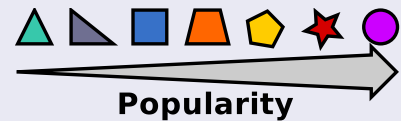
- Sort each record's token
- E.g., *sim* is overlap size, $\tau = 4$
- Prefix length is 2



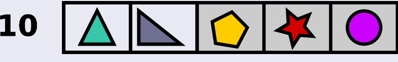
Set-Similarity Filtering

Prefix Filtering [Chaudhuri et al., 2006]

- Pigeonhole principle
- Global order for set elements:



- Sort each record's token
- E.g., *sim* is overlap size, $\tau = 4$
- Prefix length is 2



Set-Similarity Filtering

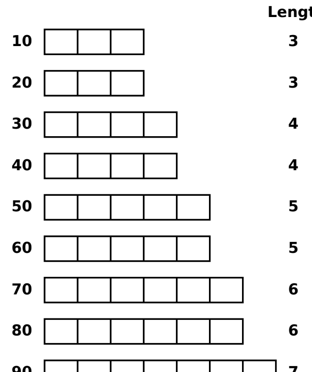
Length Filtering [Arasu et al., 2006]

- Similar records have similar lengths
- E.g.
  - *sim* is Jaccard
  - $\tau = .8$
  - Record length is 5
- Similar records have length $\in [4, 6]$

Set-Similarity Filtering

Length Filtering [Arasu et al., 2006]

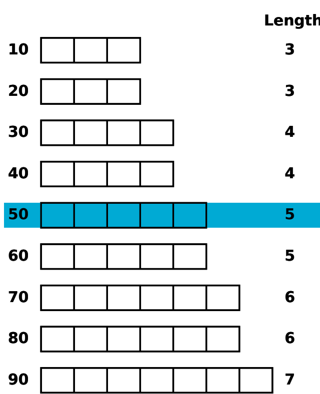
- Similar records have similar lengths
- E.g.
  - *sim* is Jaccard
  - $\tau = .8$
  - Record length is 5
- Similar records have length $\in [4, 6]$



Set-Similarity Filtering

Length Filtering [Arasu et al., 2006]

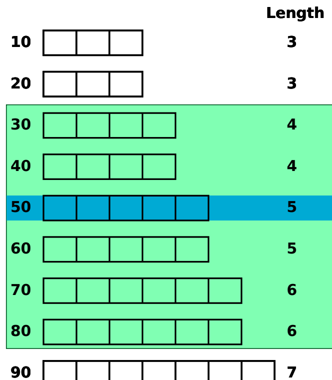
- Similar records have similar lengths
- E.g.
  - *sim* is Jaccard
  - $\tau = .8$
  - Record length is 5
- Similar records have length $\in [4, 6]$



Set-Similarity Filtering

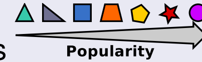
Length Filtering [Arasu et al., 2006]

- Similar records have similar lengths
- E.g.
    - *sim* is Jaccard
    - $\tau = .8$
    - Record length is 5
- Similar records have length $\in [4, 6]$

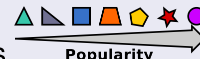


Set-Similarity Filtering

Position Filtering [Xiao et al., 2008]

- Global order for set elements 

Position Filtering [Xiao et al., 2008]

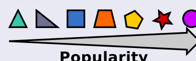


- Global order for set elements
- E.g., *sim* is Jaccard, $\tau = .8$

Set-Similarity Filtering

Position Filtering [Xiao et al., 2008]

- Global order for set elements

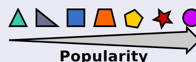


- E.g., *sim* is Jaccard, $\tau = .8$
- Prefix length is 2

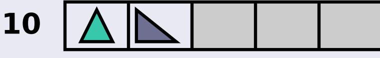
Set-Similarity Filtering

Position Filtering [Xiao et al., 2008]

- Global order for set elements



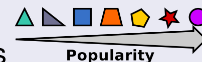
- E.g., *sim* is Jaccard, $\tau = .8$
- Prefix length is 2



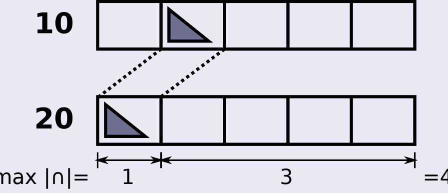
Set-Similarity Filtering

Position Filtering [Xiao et al., 2008]

- Global order for set elements



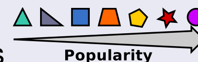
- E.g., *sim* is Jaccard, $\tau = .8$
- Prefix length is 2



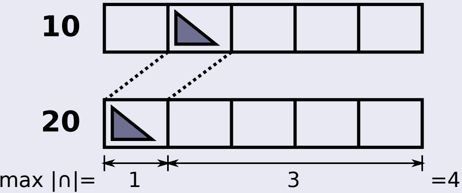
Set-Similarity Filtering

Position Filtering [Xiao et al., 2008]

- Global order for set elements



- E.g., *sim* is Jaccard, $\tau = .8$
- Prefix length is 2



- $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$
- maximum $jaccard(10, 20) = \frac{4}{6} = .66 < .8$

Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

- Global order for set elements

Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

- Global order for set elements

- E.g., *sim* is overlap size, $\tau = 7$

10

20

Set-Similarity Filtering

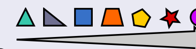
Suffix Filtering [Xiao et al., 2008]

- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2

Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

- Global order for set elements

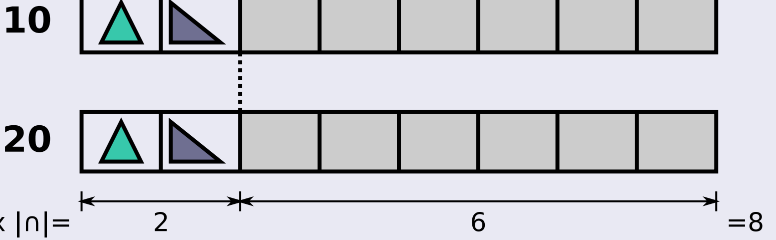


- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2

Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

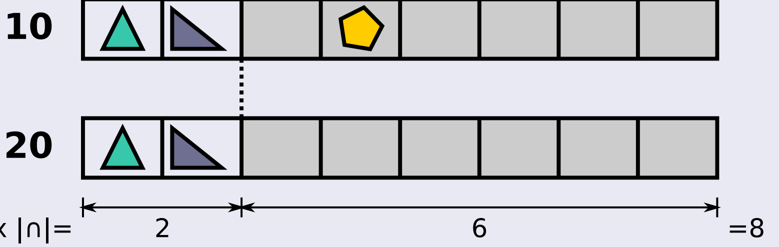
- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2



Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

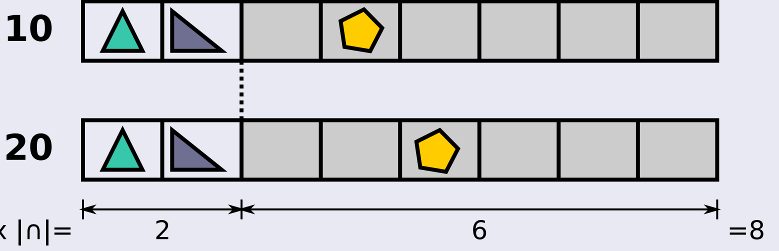
- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2



Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

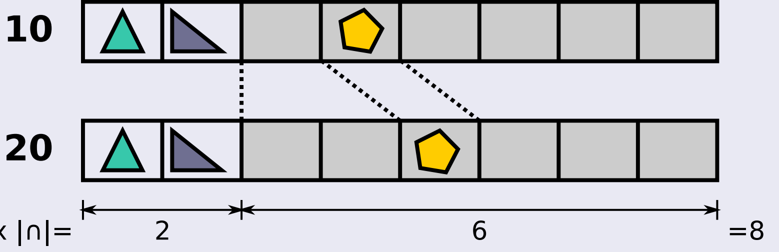
- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2



Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2

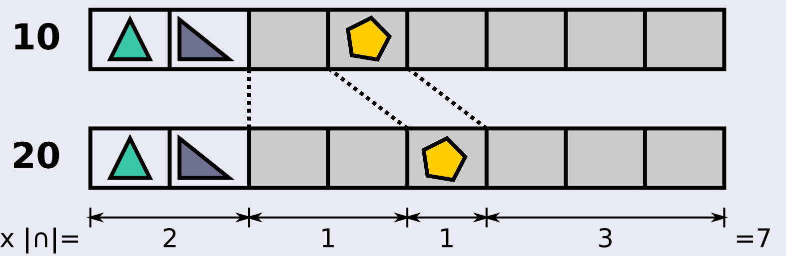


Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2

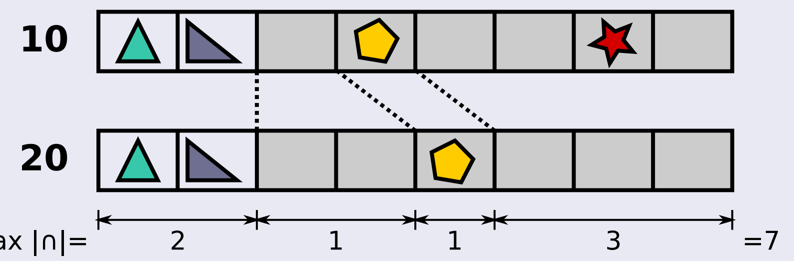
Popularity



Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

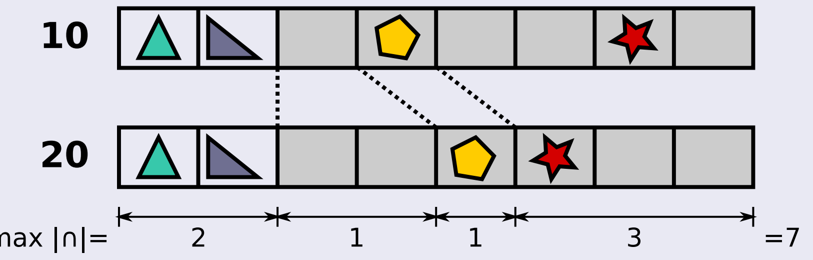
- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2



Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2

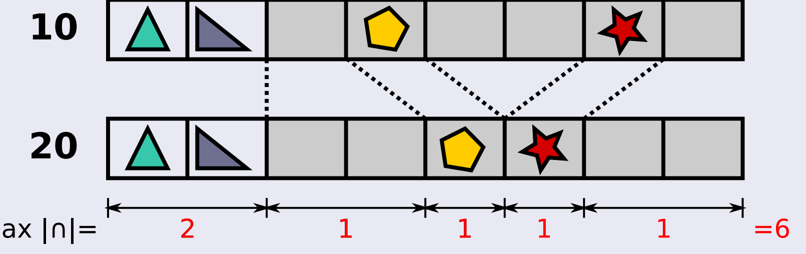


Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2

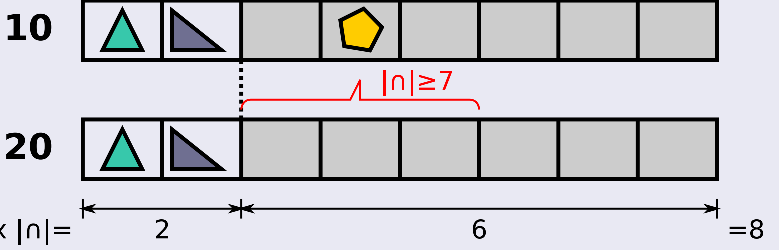
Popularity



Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

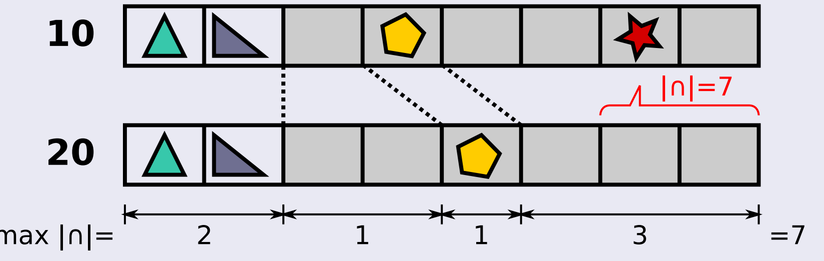
- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2



Set-Similarity Filtering

Suffix Filtering [Xiao et al., 2008]

- Global order for set elements
- E.g., *sim* is overlap size, $\tau = 7$
- Prefix length is 2



Outline

1. Motivation
2. Problem Statement
3. Single Machine Algorithms
   - Inverted List Index
   - Set-Similarity Filters
4. Parallel Algorithms
5. Experimental Evaluation

Parallelizing Set-Similarity Joins

Large amounts of data

- E.g., GeneBank: 100M, Google N-gram: 1T
- Data or processing does not fit in one machine
- Use a cluster of machines and a parallel algorithm
- **MapReduce**: shared-nothing data-processing platform

Challenges

- Partition problem for parallelism
- Solve using `Map`, `Sort`, and `Reduce`
- Compute *end-to-end* set-similarity joins
- Deal with out-of-memory situations

Parallelizing Set-Similarity Joins

Large amounts of data

- E.g., GeneBank: 100M, Google N-gram: 1T
- Data or processing does not fit in one machine
- Use a cluster of machines and a parallel algorithm
- **MapReduce**: shared-nothing data-processing platform

Challenges

- Partition problem for parallelism
- Solve using `Map`, `Sort`, and `Reduce`
- Compute *end-to-end* set-similarity joins
- Deal with out-of-memory situations

MapReduce Review

```
map     (k1,v1)          → list(k2,v2);
reduce  (k2,list(v2))   → list(k3,v3).
```

```
combine (k2,list(v2))   → list(k2,v2).
```

MapReduce Review

```
map     (k1,v1)         → list(k2,v2);
reduce  (k2,list(v2))   → list(k3,v3).
```

Input: (k1,v1)
Output: list(k2,v2)
Input: (k2, list(v2))
Output: list(k3,v3)

DFS
INPUT 1/3
INPUT 2/3
INPUT 3/3

MAP
MAP
MAP

REDUCE
REDUCE

DFS
OUTPUT 1/2
OUTPUT 2/2

MERGE
SHUFFLE

```
combine (k2,list(v2)) → list(k2,v2).
```

MapReduce Review

```
map     (k1,v1)         → list(k2,v2);
reduce  (k2,list(v2))   → list(k3,v3).
```

Input: (k1,v1)

Output: list(k2,v2)

Input: (k2, list(v2))

Output: list(k3,v3)

DFS

INPUT 1/3

INPUT 2/3

INPUT 3/3

MAP

MAP

MAP

REDUCE

REDUCE

DFS

OUTPUT 1/2

OUTPUT 2/2

MERGE

SHUFFLE

```
combine  (k2,list(v2))  → list(k2,v2).
```

Parallel Set-Similarity Joins in MapReduce

Main idea

- Hash-partition data across the network based on keys
- Join values **cannot** be directly used as keys
- Use set tokens as keys
  - e.g., "`John W Smith`" → {`John, Smith, W`}

Partition using prefix filter

- Use tokens in the prefix as keys
  - **Minimize replication**
- Global order: increasing freqnecy
  - **Reduce skew**

Parallel Set-Similarity Joins in MapReduce

Main idea

- Hash-partition data across the network based on keys
- Join values **cannot** be directly used as keys
- Use set tokens as keys
  - e.g., "`John W Smith`" → {`John, Smith, W`}

Partition using prefix filter

- Use tokens in the prefix as keys
  - **Minimize replication**
- Global order: increasing freqnecy
  - **Reduce skew**

Processing Stages and Alternatives

Stage 1: Token Ordering

- Compute the token frequencies and sort

  `Records → Token_Order`

Stage 2: Kernel (RID-Pair Generation)

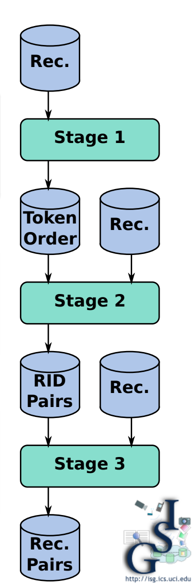
- Generate similar record-ID ("RID") pairs

  `{Records, Token_Order} → RID-pairs`

Stage 3: Record Join

- Generate pairs of joined records

  `{Records, RID-pairs} → Record-pairs`



Processing Stages and Alternatives

Stage 1: Token Ordering

- Compute the token frequencies and sort

  `Records → Token_Order`

Stage 2: Kernel (RID-Pair Generation)

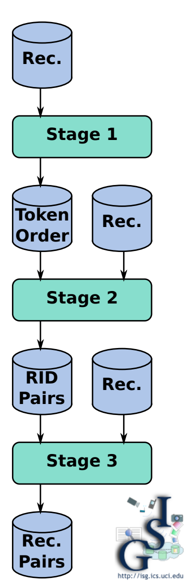
- Generate similar record-ID ("RID") pairs

  `{Records, Token_Order} → RID-pairs`

Stage 3: Record Join

- Generate pairs of joined records

  `{Records, RID-pairs} → Record-pairs`



Processing Stages and Alternatives

Stage 1: Token Ordering

- Compute the token frequencies and sort

  `Records → Token_Order`

Stage 2: Kernel (RID-Pair Generation)

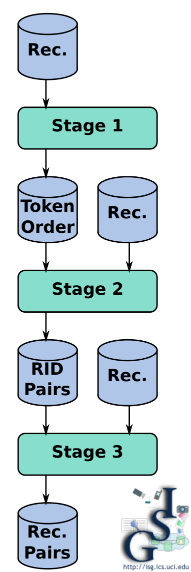
- Generate similar record-ID ("RID") pairs

  `{Records, Token_Order} → RID-pairs`

Stage 3: Record Join

- Generate pairs of joined records

  `{Records, RID-pairs} → Record-pairs`



Stage 1: Token Ordering

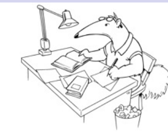
Overview

- **Input:** original records
- **Output:** token order
- Compute the token frequencies and sort

Alternatives

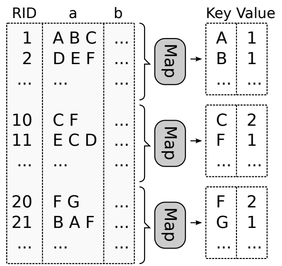
- Basic Token Ordering (BTO)
  - Two MapReduce phases: sort in MapReduce
- One Phase Token Ordering (OPTO)
  - One MapReduce phase: sort in memory

Stage 1: Basic Token Ordering (BTO)

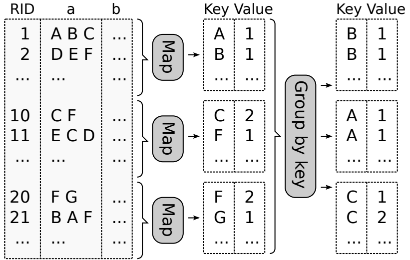


RID	a	b
1	A B C	...
2	D E F	...
...	...	...
10	C F	...
11	E C D	...
...	...	...
20	F G	...
21	B A F	...
...	...	...

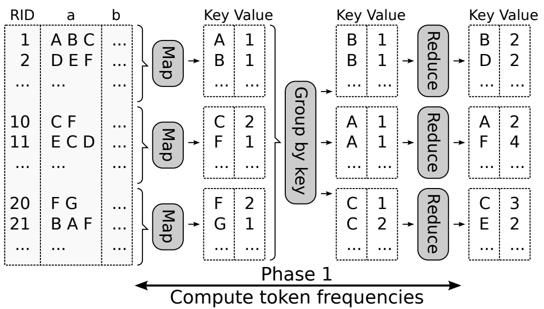
Stage 1: Basic Token Ordering (BTO)



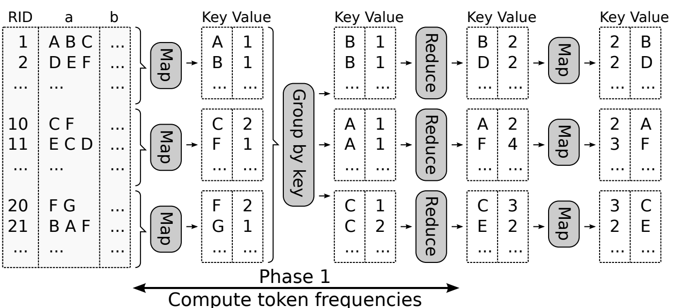
Stage 1: Basic Token Ordering (BTO)



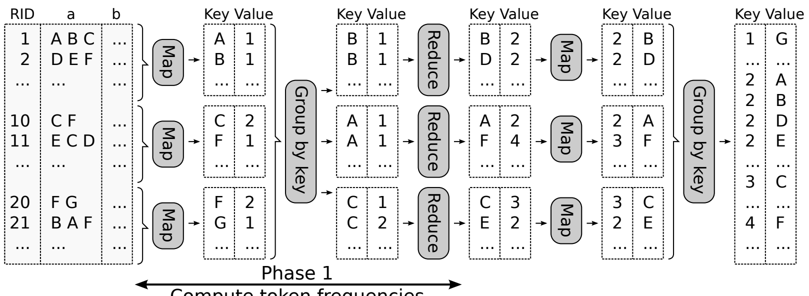
Stage 1: Basic Token Ordering (BTO)



Stage 1: Basic Token Ordering (BTO)



Stage 1: Basic Token Ordering (BTO)



Stage 1: Basic Token Ordering (BTO)

Input records:

RID	a	b
1	A B C	...
2	D E F	...
...	...	...
10	C F	...
11	E C D	...
...	...	...
20	F G	...
21	B A F	...
...	...	...

Phase 1 — Compute token frequencies

Map → (Key, Value): A 1, B 1, ... | C 2, F 1, ... | F 2, G 1, ...

Group by key → (Key, Value): B 1, B 1, ... | A 1, A 1, ... | C 1, C 2, ...

Reduce → (Key, Value): B 2, D 2, ... | A 2, F 4, ... | C 3, E 2, ...

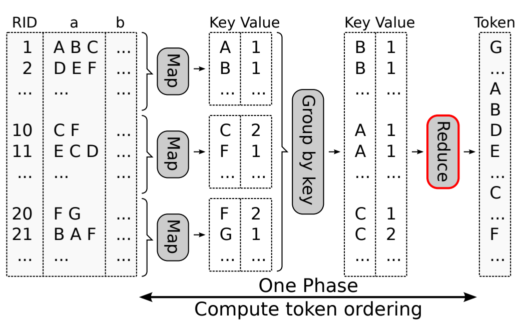
Phase 2 — Sort tokens by freqency

Map → (Key, Value): 2 B, 2 D, ... | 2 A, 3 F, ... | 3 C, 2 E, ...

Group by key → (Key, Value): 1 G, ..., 2 A, 2 B, 2 D, 2 E, ..., 3 C, ..., 4 F, ...

Reduce → Token: G, ..., A, B, D, E, ..., C, ..., F, ...

Stage 1: One Phase Token Ordering (OPTO)



Stage 2: Kernel (RID-Pair Generation)

Overview

- **Input:** original records and token order
- **Output:** list of similar-RID pairs
- Partition using prefix filter

Steps

- Load token order in memory
- Extract RIDs and join value of records
- Distribute records on prefix tokens
- Group RIDs and join values
- Cross-pair and verify candidates

Stage 2: Kernel (RID-Pair Generation)

Overview

- **Input:** original records and token order
- **Output:** list of similar-RID pairs
- Partition using prefix filter

Steps

- Load token order in memory
- Extract RIDs and join value of records
- Distribute records on prefix tokens
- Group RIDs and join values
- Cross-pair and verify candidates

Stage 2: Kernel (RID-Pair Generation)

Overview

- **Input:** original records and token order
- **Output:** list of similar-RID pairs
- Partition using prefix filter

Steps

- Load token order in memory
- Extract RIDs and join value of records

- Distribute records on prefix tokens
- Group RIDs and join values
- Cross-pair and verify candidates

Stage 2: Kernel (RID-Pair Generation)

Overview

- **Input:** original records and token order
- **Output:** list of similar-RID pairs
- Partition using prefix filter

Steps

- Load token order in memory
- Extract RIDs and join value of records
- Distribute records on prefix tokens
- Group RIDs and join values
- Cross-pair and verify candidates

Map

Shuffle

http://isg.ics.uci.edu

Stage 2: Kernel (RID-Pair Generation)

Overview

- **Input:** original records and token order
- **Output:** list of similar-RID pairs
- Partition using prefix filter

Steps

- Load token order in memory
- Extract RIDs and join value of records
- Distribute records on prefix tokens
- Group RIDs and join values
- Cross-pair and verify candidates

} Map

Shuffle

Stage 2: Kernel (RID-Pair Generation)

Overview

- **Input:** original records and token order
- **Output:** list of similar-RID pairs
- Partition using prefix filter

Steps

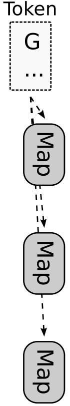
- Load token order in memory
- Extract RIDs and join value of records
- Distribute records on prefix tokens
- Group RIDs and join values
- Cross-pair and verify candidates

Map (load token order; extract RIDs and join values)

Shuffle (distribute records on prefix tokens)

Reduce (group RIDs and join values; cross-pair and verify candidates)

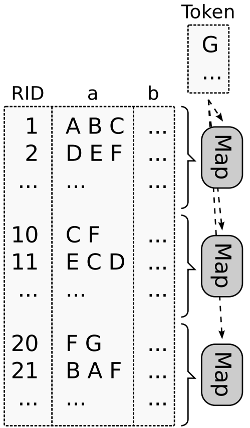
Stage 2: Partition Using Individual Tokens



Stage 2: Partition Using Individual Tokens

Token

G
...



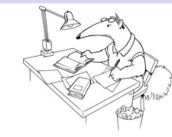
RID	a	b
1	A B C	...
2	D E F	...
...	...	...
10	C F	...
11	E C D	...
...	...	...
20	F G	...
21	B A F	...
...	...	...

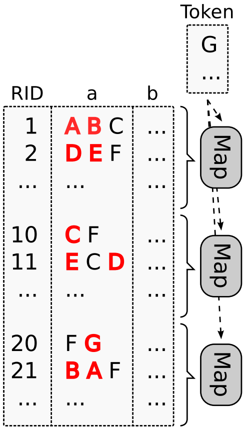
Map

Map

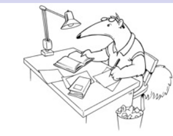
Map

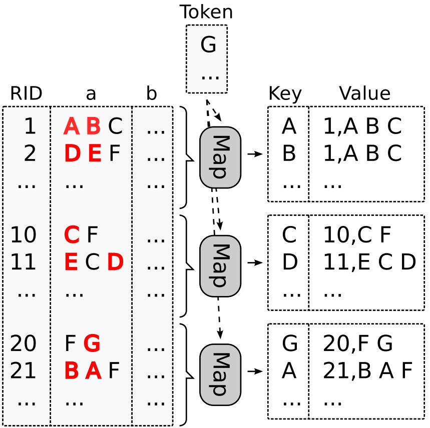
Stage 2: Partition Using Individual Tokens



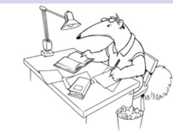


Stage 2: Partition Using Individual Tokens





Stage 2: Partition Using Individual Tokens



Token

G
...

RID	a	b
1	A B C	...
2	D E F	...
...	...	...
10	C F	...
11	E C D	...
...	...	...
20	F G	...
21	B A F	...
...	...	...

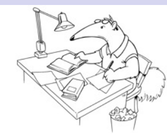
Map

Key	Value
A	1,A B C
B	1,A B C
...	...
C	10,C F
D	11,E C D
...	...
G	20,F G
A	21,B A F
...	...

Group by key

Key	Value
B	1,A B C
B	21,B A F
...	...
A	1,A B C
A	21,B A F
...	...
C	10,C F
E	2,D E F
...	...

Stage 2: Partition Using Individual Tokens



Token: G, ...

RID	a	b
1	A B C	...
2	D E F	...
...	...	...
10	C F	...
11	E C D	...
...	...	...
20	F G	...
21	B A F	...
...	...	...

Map →

Key	Value
A	1,A B C
B	1,A B C
...	...
C	10,C F
D	11,E C D
...	...
G	20,F G
A	21,B A F
...	...

Group by key →

Key	Value
B	1,A B C
B	21,B A F
...	...
A	1,A B C
A	21,B A F
...	...
C	10,C F
E	2,D E F
...	...

Reduce →

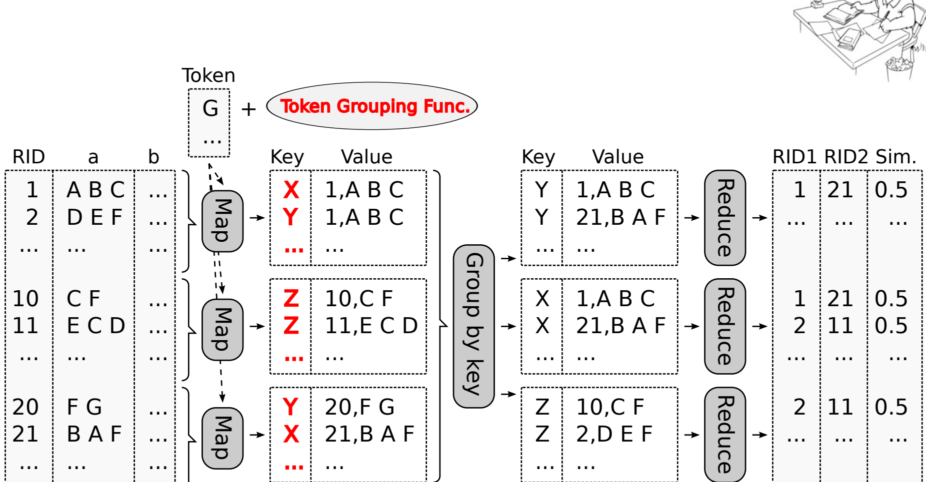
RID1	RID2	Sim.
1	21	0.5
...	...	...
1	21	0.5
2	11	0.5
...	...	...
2	11	0.5
...	...	...

Stage 2: Partition Using Individual Tokens

Alternatives

- Basic Kernel (BK): nested loops
- PPJoin+ Kernel (PK): inverted list index

Stage 2: Partition Using Grouped Tokens



Stage 3: Record Join

Overview

- **Input:** original records and similar-RID pairs
- **Output:** similar-record pairs
- Generate pairs of similar records

Alternatives

- Basic Record Join (BRJ)
  - Two MapReduce phases: reduce-side join
- One Phase Record Join (OPRJ)
  - One MapReduce phase: map-side join

Additional Contributions

- Solve R-S-join case
- Optimize memory requirements for R-S join
- Handle insufficient memory
  - Introduce additional filters
  - Use external memory

Outline

1. Motivation
2. Problem Statement
3. Single Machine Algorithms
   - Inverted List Index
   - Set-Similarity Filters
4. Parallel Algorithms
5. Experimental Evaluation

Experimental Setting

Hardware

- 10-node IBM x3650 cluster
  - Intel Xeon processor E5520 2.26GHz with four cores
  - Four 300GB hard disks
  - 12GB RAM

Software

- Ubuntu 9.06, 64-bit, server edition OS
- Java 1.6, 64-bit, server
- Hadoop 0.20.1

Experimental Setting

Datasets

- DBLP
  - Average length: 259 bytes
  - Number of records: 1.2M
  - Total size: 300MB
- CITESEERX
  - Average length: 1374 bytes
  - Number of records: 1.3M
  - Total size: 1.8GB
- Increased each up to $\times$25, preserving join properties
  - DBLP: 31M records, 8.2GB
  - CITESEERX: 32M records, 45GB

Running Time

Time (seconds)

0
200
400
600
800

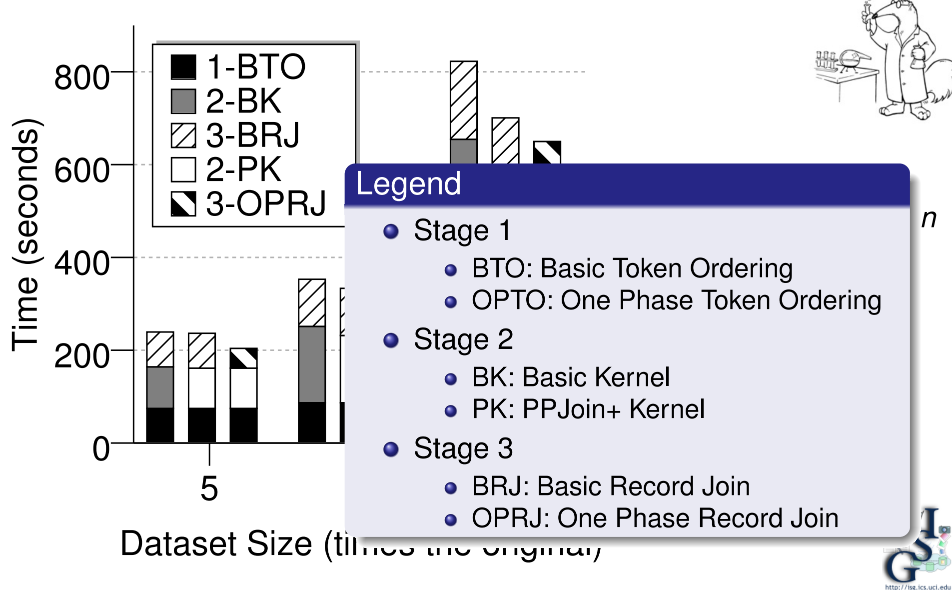
1-BTO
2-BK
3-BRJ
2-PK
3-OPRJ

5
10
25

Dataset Size (times the original)

- Self-join DBLP$\times n$
- $n \in [5, 25]$
- 10-node cluster
- Best time
- Bulk of the time

Running Time



Running Time

Time (seconds)

800
600
400
200
0

1-BTO
2-BK
3-BRJ
2-PK
3-OPRJ

5
10
25

Dataset Size (times the original)

- Self-join DBLP$\times n$
- $n \in [5, 25]$
- 10-node cluster
- **Best time**
- Bulk of the time

Running Time

Time (seconds)

800
600
400
200
0

1-BTO
2-BK
3-BRJ
2-PK
3-OPRJ

5
10
25

Dataset Size (times the original)

- Self-join DBLP$\times n$
- $n \in [5, 25]$
- 10-node cluster
- Best time
- **Bulk of the time**

Speedup

Speedup

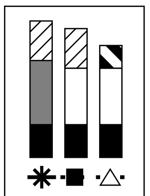
5
4
3
2
1

BTO-BK-BRJ
BTO-PK-BRJ
BTO-PK-OPRJ
Ideal

2 3 4 5 6 7 8 9 10

Nodes

- Relative running time
- Self-join DBLP×10
- Different cluster sizes



Speedup Breakdown

Stage 1

Speedup

BTO
OPTO
Ideal

Nodes

Stage 2

Speedup

BK
PK
Ideal

Nodes

Stage 3

Speedup

BRJ
OPRJ
Ideal

Nodes

- Relative running time
- Self-join DBLP$\times$10
- Different cluster sizes

Scaleup

Time (seconds)

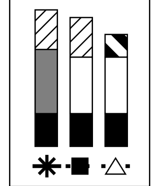
1000
800
600
400
200
0

2 3 4 5 6 7 8 9 10

Nodes and Dataset Size

BTO-BK-BRJ
BTO-PK-BRJ
BTO-PK-OPRJ
Ideal

- Running time
- Self-joining DBLP $\times n$
- $n \in [5, 25]$
- Proportional cluster



Scaleup Breakdown

Stage 1

Time (seconds)

160
120
80
40
0

2 3 4 5 6 7 8 9 10

Nodes and Dataset Size

BTO
OPTO
Ideal

Stage 2

Time (seconds)

600
500
400
300
200
100
0

2 3 4 5 6 7 8 9 10

Nodes and Dataset Size

BK
PK
Ideal

Stage 3

Time (seconds)

200
150
100
50
0

2 3 4 5 6 7 8 9 10

Nodes and Dataset Size

BRJ
OPRJ
Ideal

- Running time
- Self-joining DBLP$\times n$, $n \in [5, 25]$
- Proportional cluster

Summary

- Set-similarity joins in MapReduce
  - Three-stage approach
  - Balance workload and minimize replication
- *End-to-end* algorithms
  - Self-join
  - R-S join
- Memory issues
  - Optimize memory requirements
  - Handle insufficient memory
- Experiments
  - Speedup and scaleup
  - 40 cores, 40 disks cluster

More Info

The paper, the source-code and the datasets:

http://asterix.ics.uci.edu/fuzzyjoin-mapreduce/

This work is part of the
ASTERIX
http://asterix.ics.uci.edu/
and
Flamingo
http://flamingo.ics.uci.edu/
projects at UC Irvine.

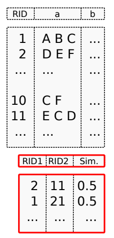
Arasu, A., Ganti, V., and Kaushik, R. (2006).
Efficient exact set-similarity joins.
In *VLDB*, pages 918–929.

Chaudhuri, S., Ganti, V., and Kaushik, R. (2006).
A primitive operator for similarity joins in data cleaning.
In *ICDE*, page 5.

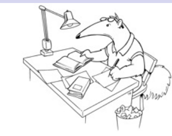
Sarawagi, S. and Kirpal, A. (2004).
Efficient set joins on similarity predicates.
In *SIGMOD Conference*, pages 743–754.

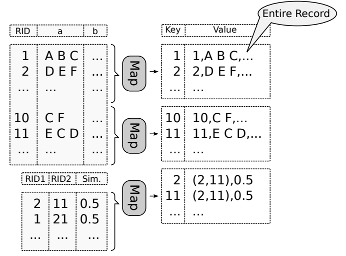
Xiao, C., Wang, W., Lin, X., and Yu, J. X. (2008).
Efficient similarity joins for near duplicate detection.
In *WWW*, pages 131–140.

Stage 3: Basic Record Join (BRJ)

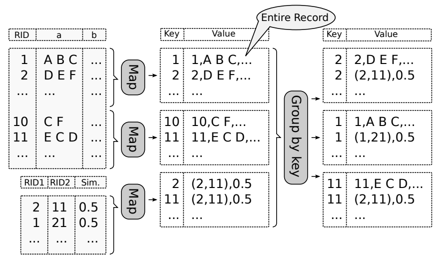


Stage 3: Basic Record Join (BRJ)

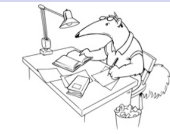


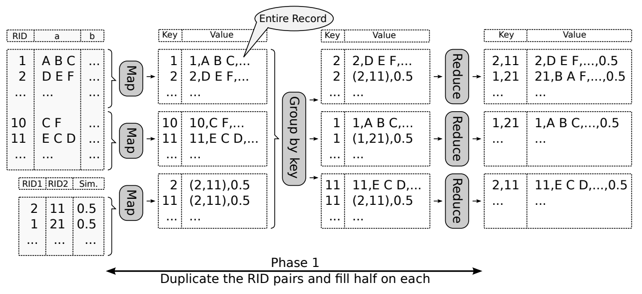


Stage 3: Basic Record Join (BRJ)

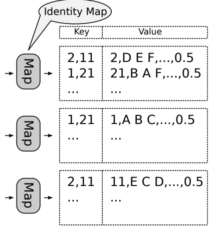


Stage 3: Basic Record Join (BRJ)

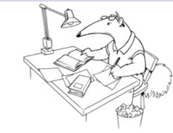


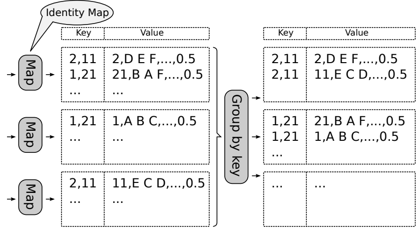


Stage 3: Basic Record Join (BRJ)

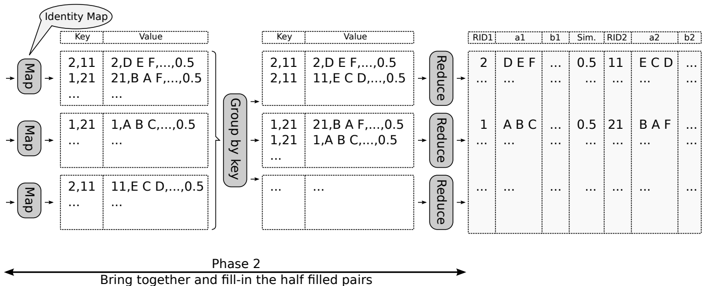


Stage 3: Basic Record Join (BRJ)

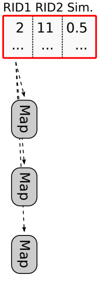




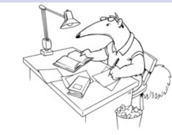
Stage 3: Basic Record Join (BRJ)

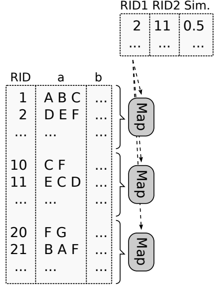


Stage 3: One-Phase Record Join (OPRJ)

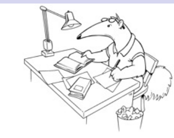


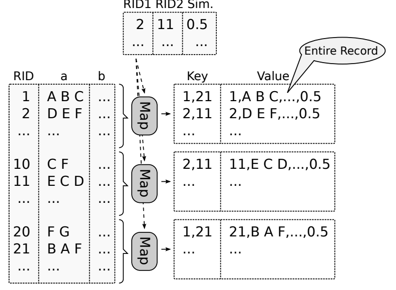
Stage 3: One-Phase Record Join (OPRJ)



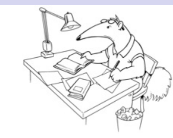


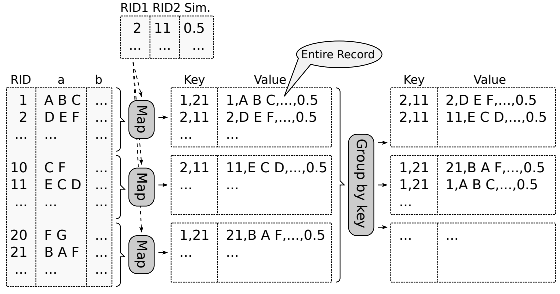
Stage 3: One-Phase Record Join (OPRJ)



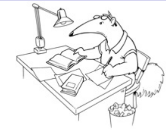


Stage 3: One-Phase Record Join (OPRJ)





Stage 3: One-Phase Record Join (OPRJ)



RID1	RID2	Sim.
2	11	0.5
...	...	...

RID	a	b
1	A B C	...
2	D E F	...
...	...	...
10	C F	...
11	E C D	...
...	...	...
20	F G	...
21	B A F	...
...	...	...

Map

Key	Value
1,21	1,A B C,...,0.5
2,11	2,D E F,...,0.5
...	...
2,11	11,E C D,...,0.5
...	...
1,21	21,B A F,...,0.5
...	...

Entire Record

Group by key

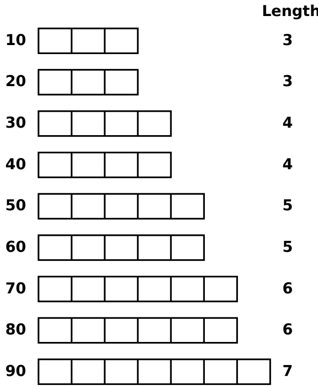
Key	Value
2,11	2,D E F,...,0.5
2,11	11,E C D,...,0.5
1,21	21,B A F,...,0.5
1,21	1,A B C,...,0.5
...	
...	...

Reduce

RID1	a1	b1	Sim.	RID2	a2	b2
2	D E F	...	0.5	11	E C D	...
...	...	...	...	...	...	...
1	A B C	...	0.5	21	B A F	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...

Memory Requirement Optimizations (self-join)

- Group by *Token* and sort by (*Token*, *Length*)
- Keep in memory only the records within the length range



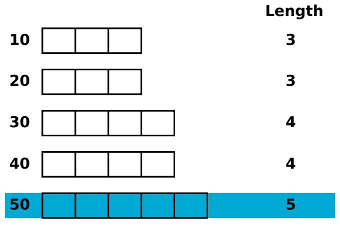
Memory Requirement Optimizations (self-join)

- Group by *Token* and sort by (*Token*, *Length*)
- Keep in memory only the records within the length range

	Length
10	3
20	3
30	4
40	4

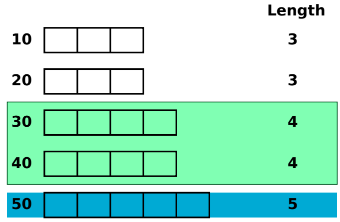
Memory Requirement Optimizations (self-join)

- Group by *Token* and sort by (*Token*, *Length*)
- Keep in memory only the records within the length range



Memory Requirement Optimizations (self-join)

- Group by *Token* and sort by (*Token*, *Length*)
- Keep in memory only the records within the length range



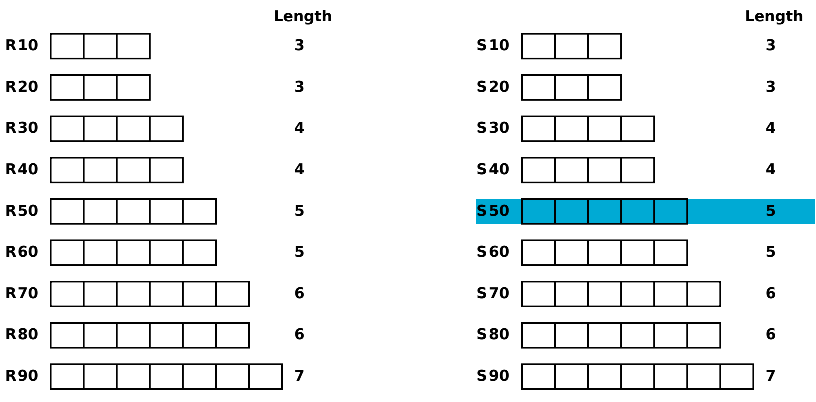
Memory Requirement Optimizations (R-S join)

- Group by *Token* and sort by (*Token*, *Length*, *Relation*)
- For *R* sort by (*Token*, *Relation*, *Length'*)

	Length		Length
R10	3	S10	3
R20	3	S20	3
R30	4	S30	4
R40	4	S40	4
R50	5	S50	5
R60	5	S60	5
R70	6	S70	6
R80	6	S80	6
R90	7	S90	7

Memory Requirement Optimizations (R-S join)

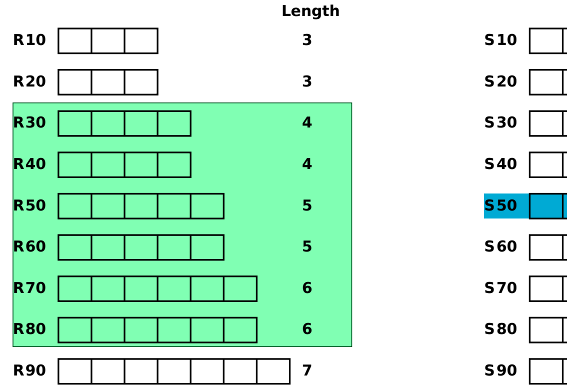
- Group by *Token* and sort by (*Token*, *Length*, *Relation*)
- For *R* sort by (*Token*, *Relation*, *Length'*)

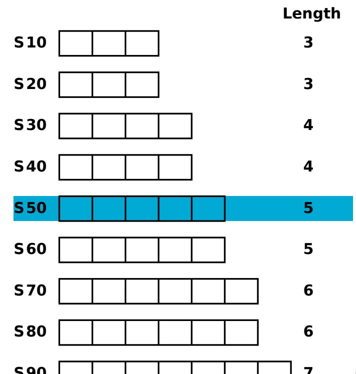


http://isg.ics.uci.edu

Memory Requirement Optimizations (R-S join)

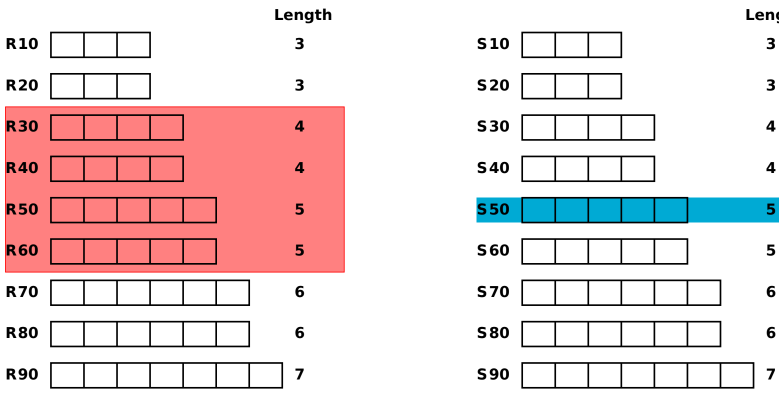
- Group by *Token* and sort by (*Token*, *Length*, *Relation*)
- For *R* sort by (*Token*, *Relation*, *Length′*)





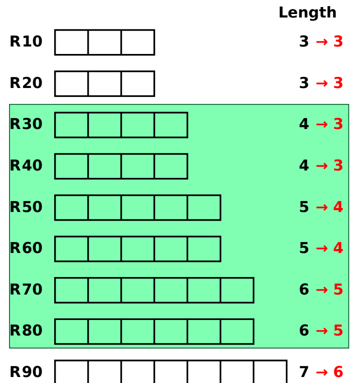
Memory Requirement Optimizations (R-S join)

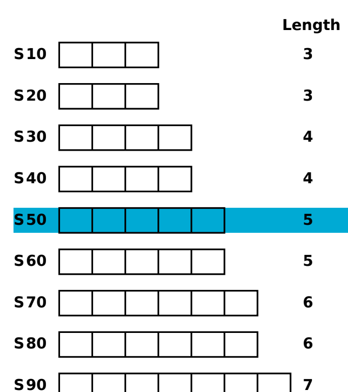
- Group by *Token* and sort by (*Token*, *Length*, *Relation*)
- For *R* sort by (*Token*, *Relation*, *Length*')



Memory Requirement Optimizations (R-S join)

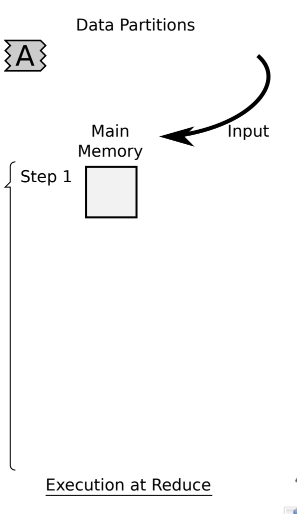
- Group by *Token* and sort by (*Token*, *Length*, *Relation*)
- For *R* sort by (*Token*, *Relation*, *Length′*)





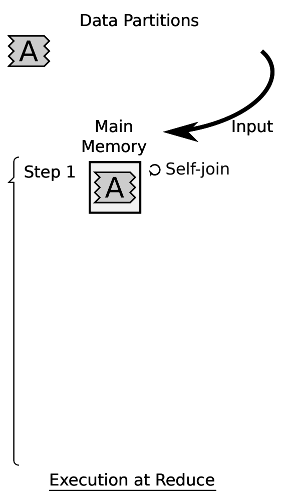
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Replicate data over network**



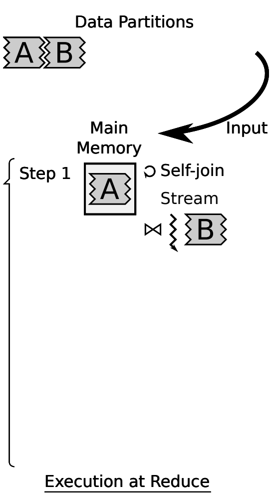
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Replicate data over network**



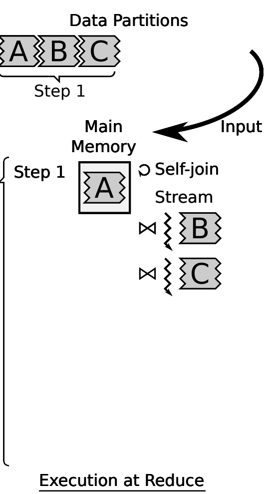
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Replicate data over network**



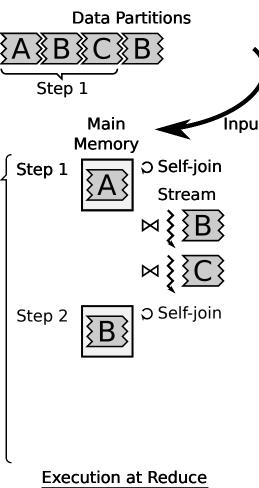
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Replicate data over network**



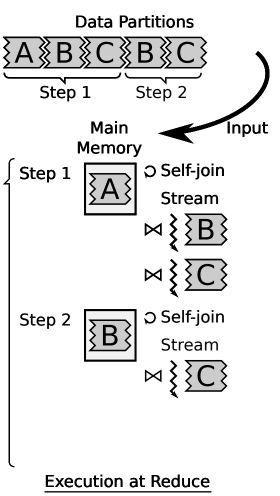
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Replicate data over network**



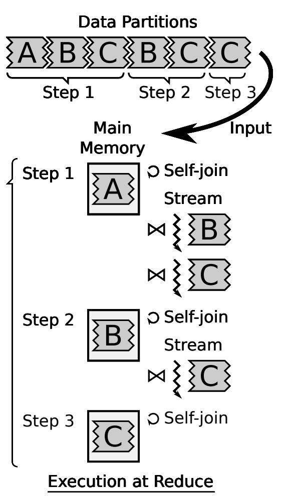
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks *A*, *B*, and *C*
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Replicate data over network**



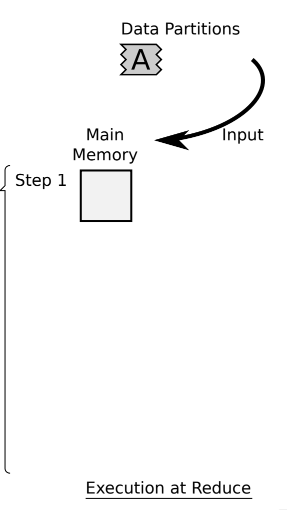
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Replicate data over network**



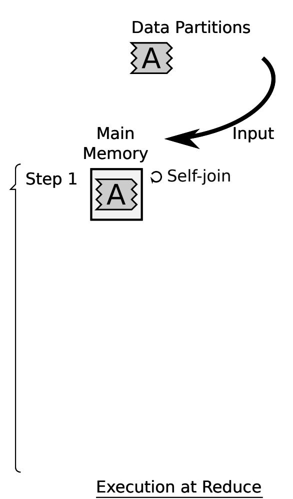
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Spill data to disk**



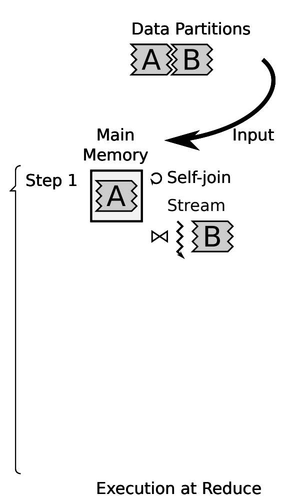
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Spill data to disk**



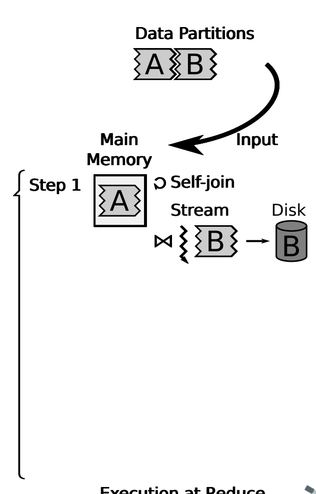
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Spill data to disk**



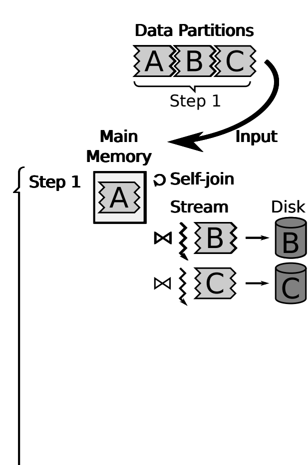
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Spill data to disk**



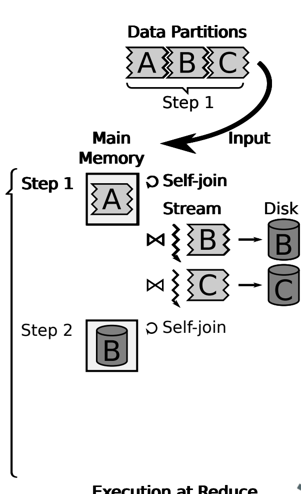
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Spill data to disk**



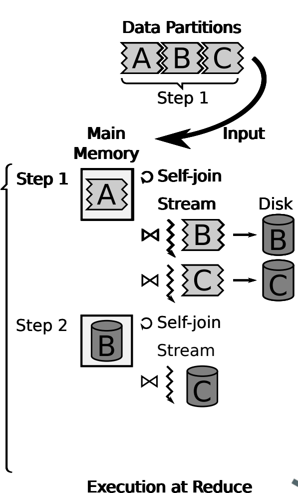
Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Spill data to disk**



Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks A, B, and C
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Spill data to disk**



Using External Memory (self join)

- Records that need to be cross-verified do not fit in memory
- Example: split data in memory-size blocks *A*, *B*, and *C*
- Need to compute:
  - $A \bowtie A$
  - $A \bowtie B$
  - $A \bowtie C$
  - $B \bowtie B$
  - $B \bowtie C$
  - $C \bowtie C$
- **Spill data to disk**