Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Data Profiling and Data Cleansing – Assignment 2

Inclusion Dependencies

Group:

Christoph Oehlke christoph.oehlke@student.hpi.uni-potsdam.de

Markus Hinsche markus.hinsche@student.hpi.uni-potsdam.de

Pipeline architecture

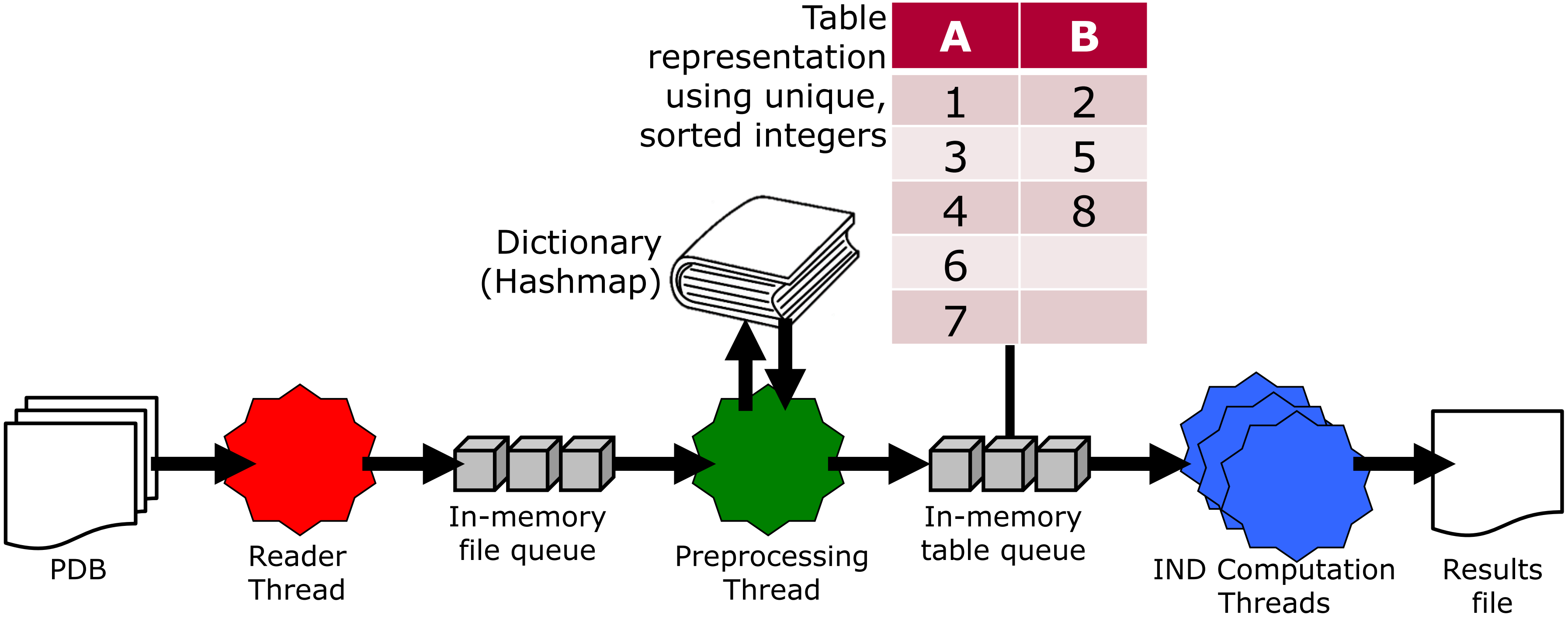
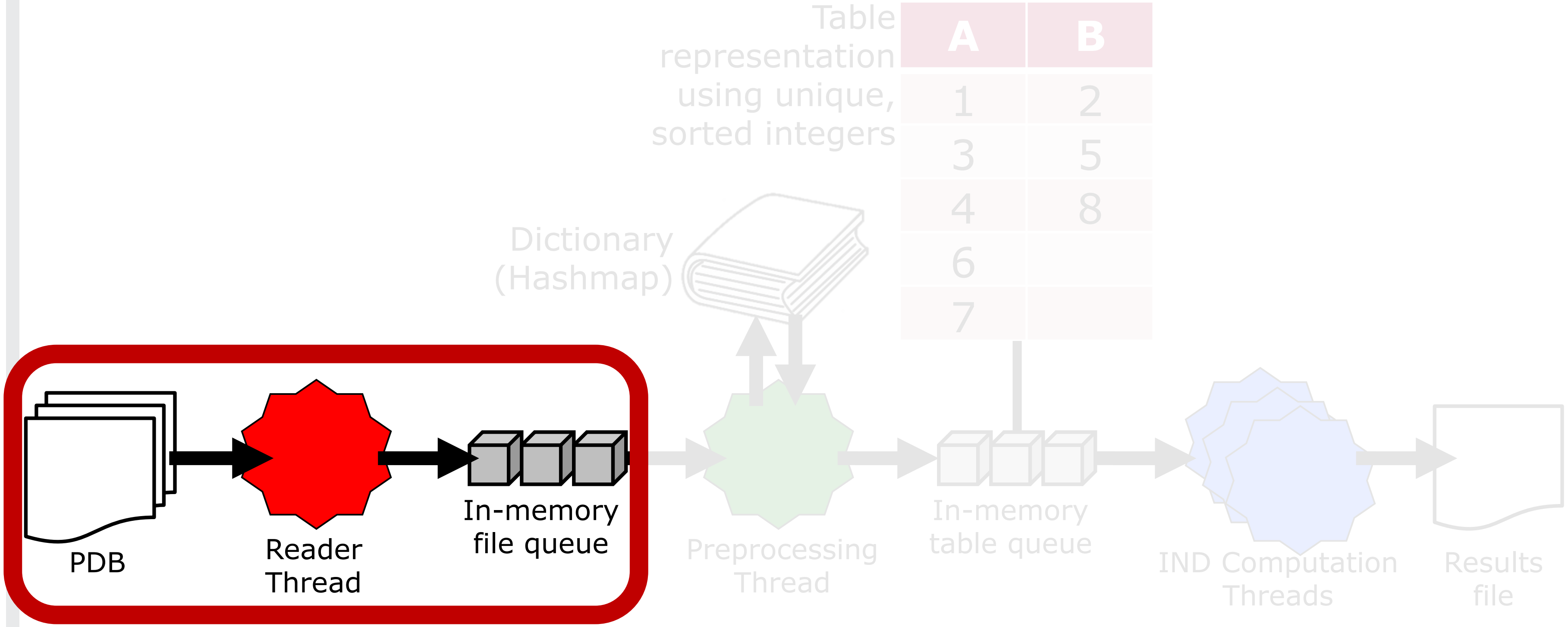


Table representation using unique, sorted integers

A	B
1	2
3	5
4	8
6	
7	

Pipeline architecture

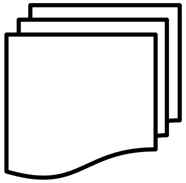


A	B
1	2
3	5
4	8
6	
7	

Reading data from disc

Input: PDB tsv file names

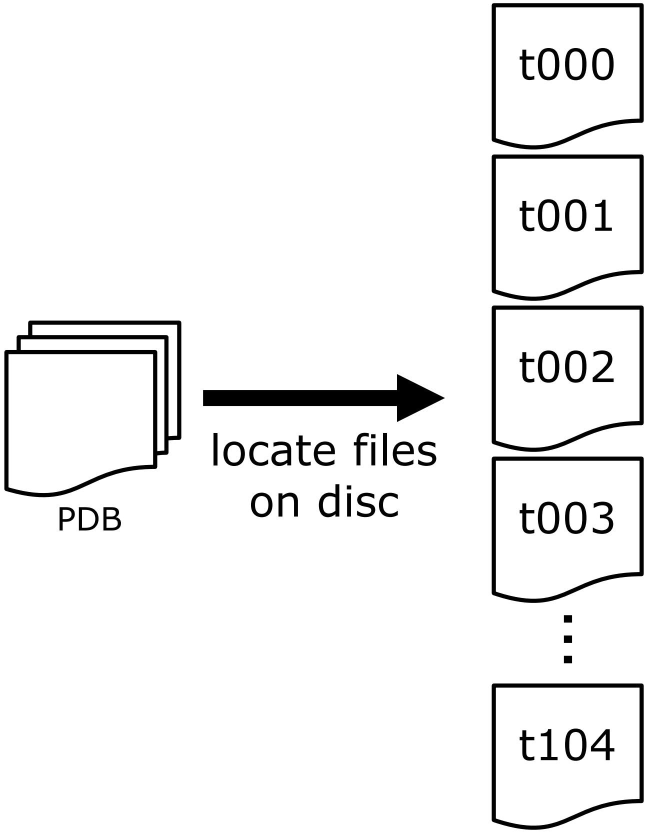
Output: raw tsv data in memory



Reading data from disc

Input: PDB tsv file names

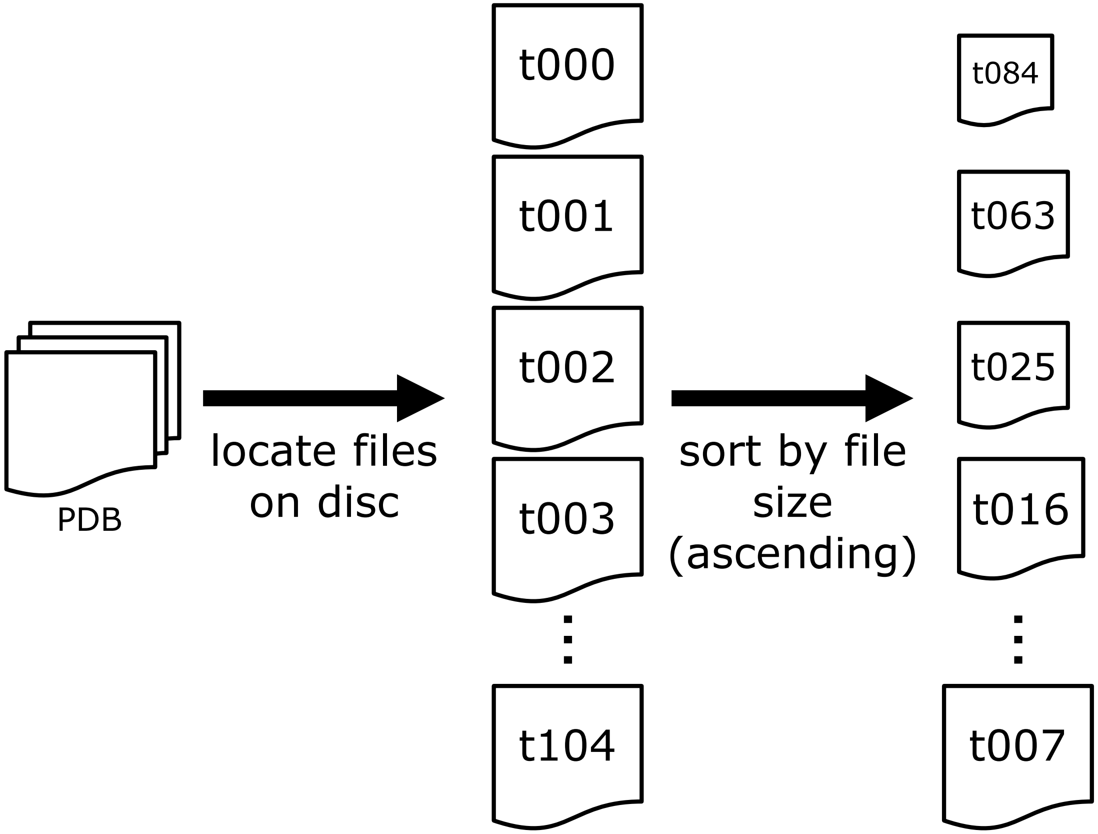
Output: raw tsv data in memory



Reading data from disc

Input: PDB tsv file names

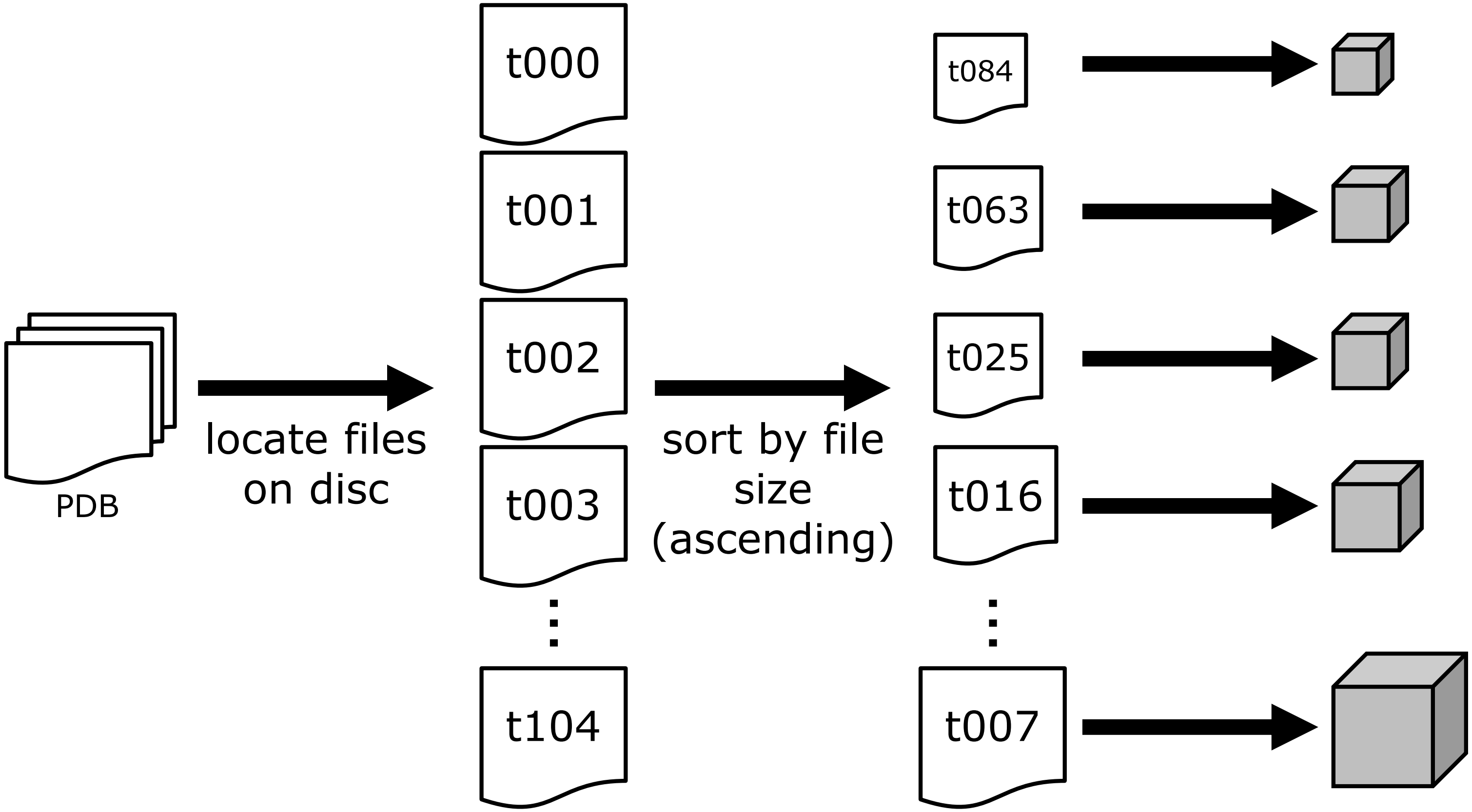
Output: raw tsv data in memory



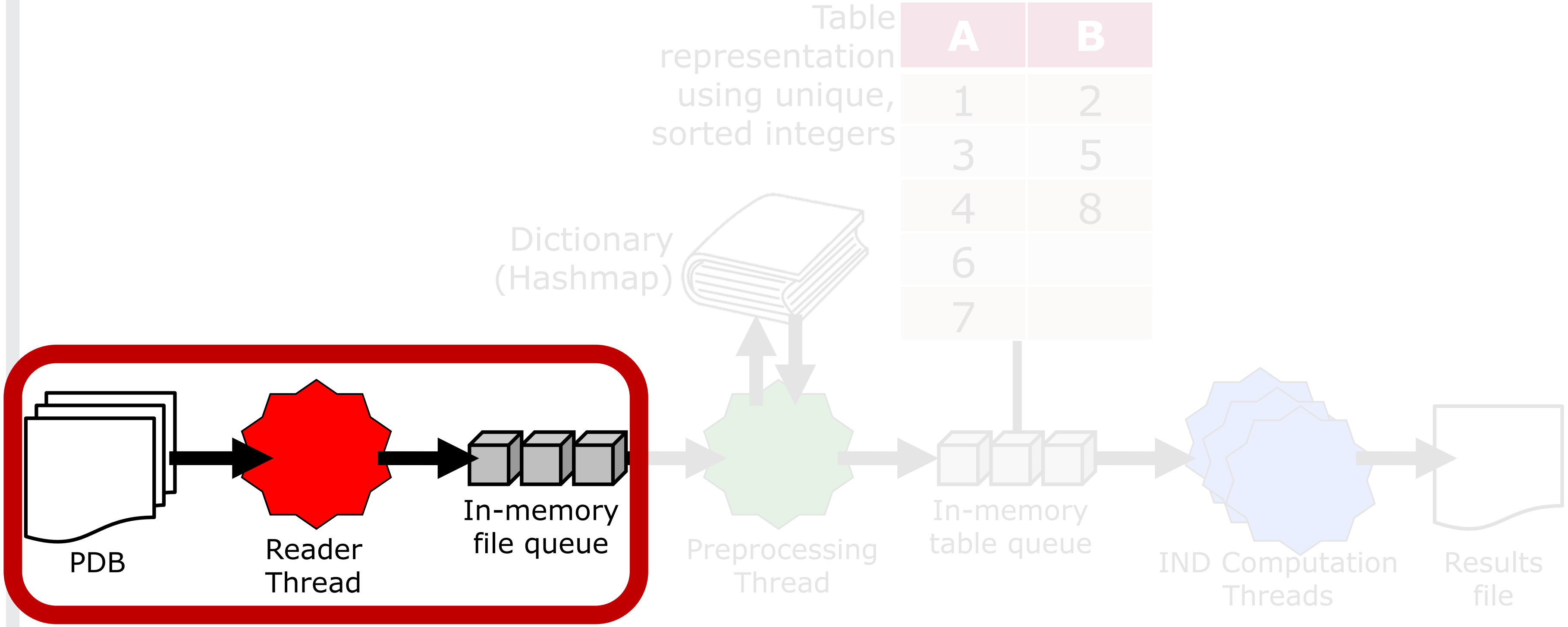
Reading data from disc

Input: PDB tsv file names

Output: raw tsv data in memory

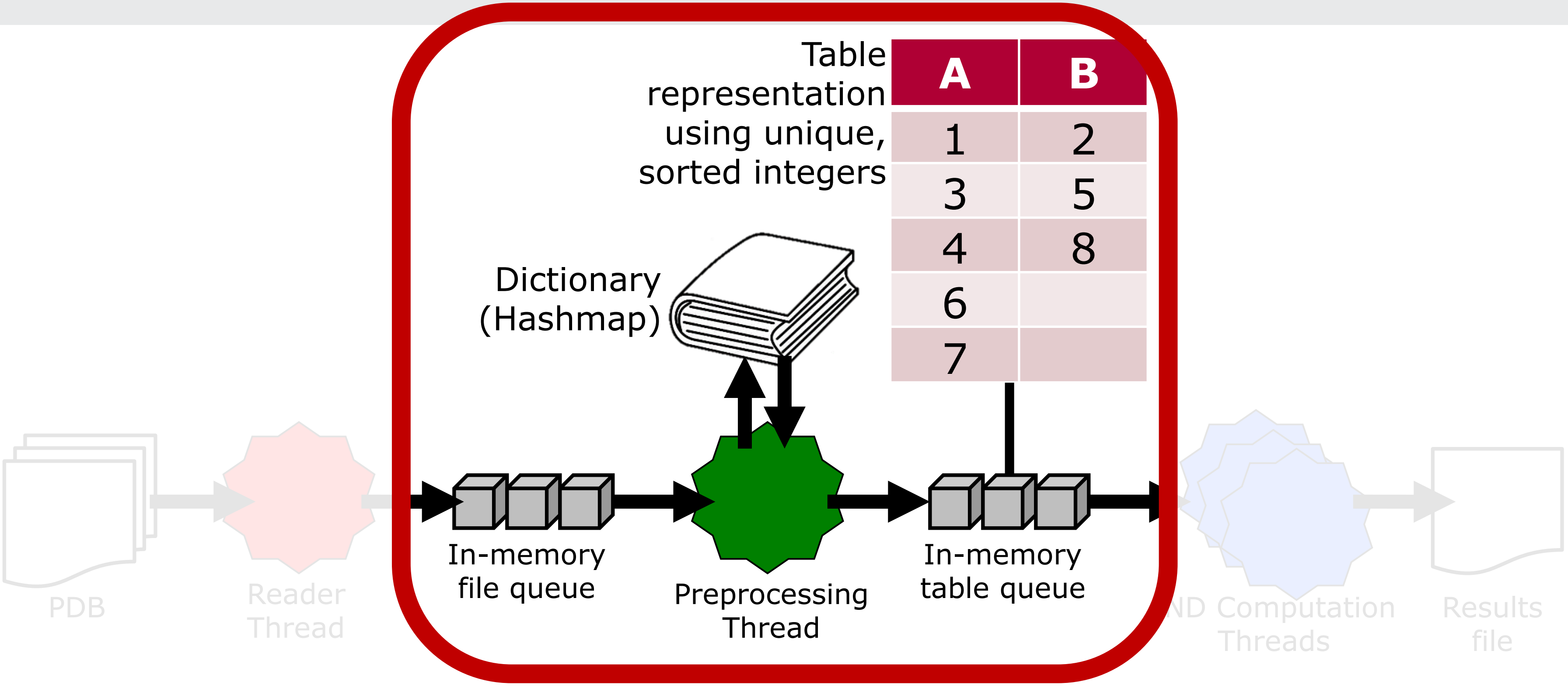


Pipeline architecture



A	B
1	2
3	5
4	8
6	
7	

Pipeline architecture



A	B
1	2
3	5
4	8
6	
7	

Parsing and preprocessing tsv data

Input: raw tsv data in memory

Output: in-memory representation optimized for IND computation

A	B
Iris	Computing
Susi	Computing
Reuven	
Eli	
Naomi	Math

Parsing and preprocessing tsv data

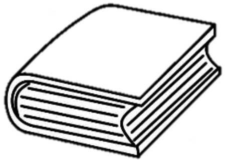
Input: raw tsv data in memory

Output: in-memory representation optimized for IND computation

A	B
Iris	Computing
Susi	Computing
Reuven	
Eli	
Naomi	Math

A	B
1	2
3	2
4	5
6	5
7	8

Dictionary (Hashmap)



Maps actual values to distinct integers

Key	Value
Iris	1
Computing	2
Susi	3
Reuven	4
	5
Eli	6
…	…

Parsing and preprocessing tsv data

Input: raw tsv data in memory

Output: in-memory representation optimized for IND computation

A	B
Iris	Computing
Susi	Computing
Reuven	
Eli	
Naomi	Math

→

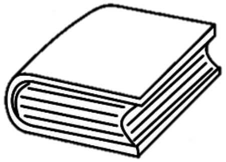
A	B
1	2
3	2
4	5
6	5
7	8

→

A	B
1	2
3	5
4	8
6	
7	

Table representation using unique, sorted integers

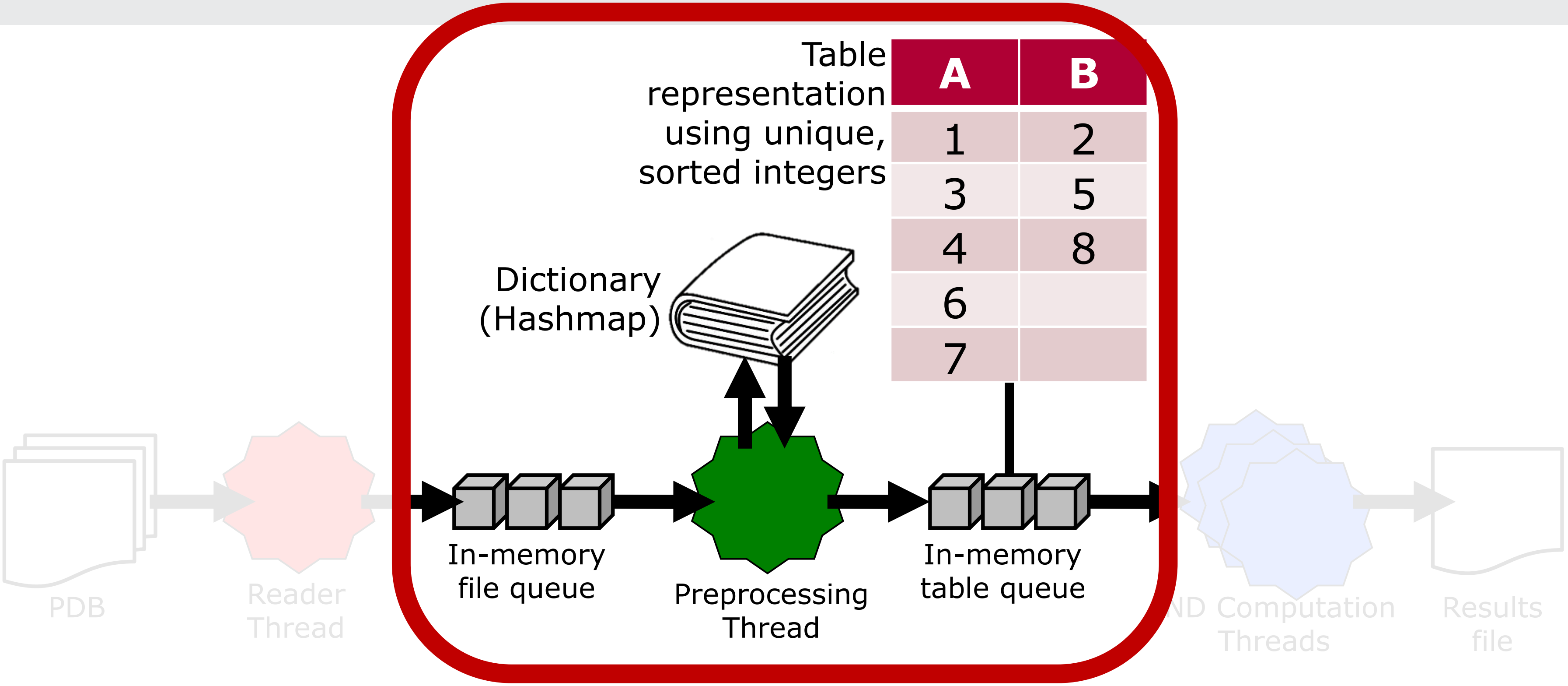
Dictionary (Hashmap)



Maps actual values to distinct integers

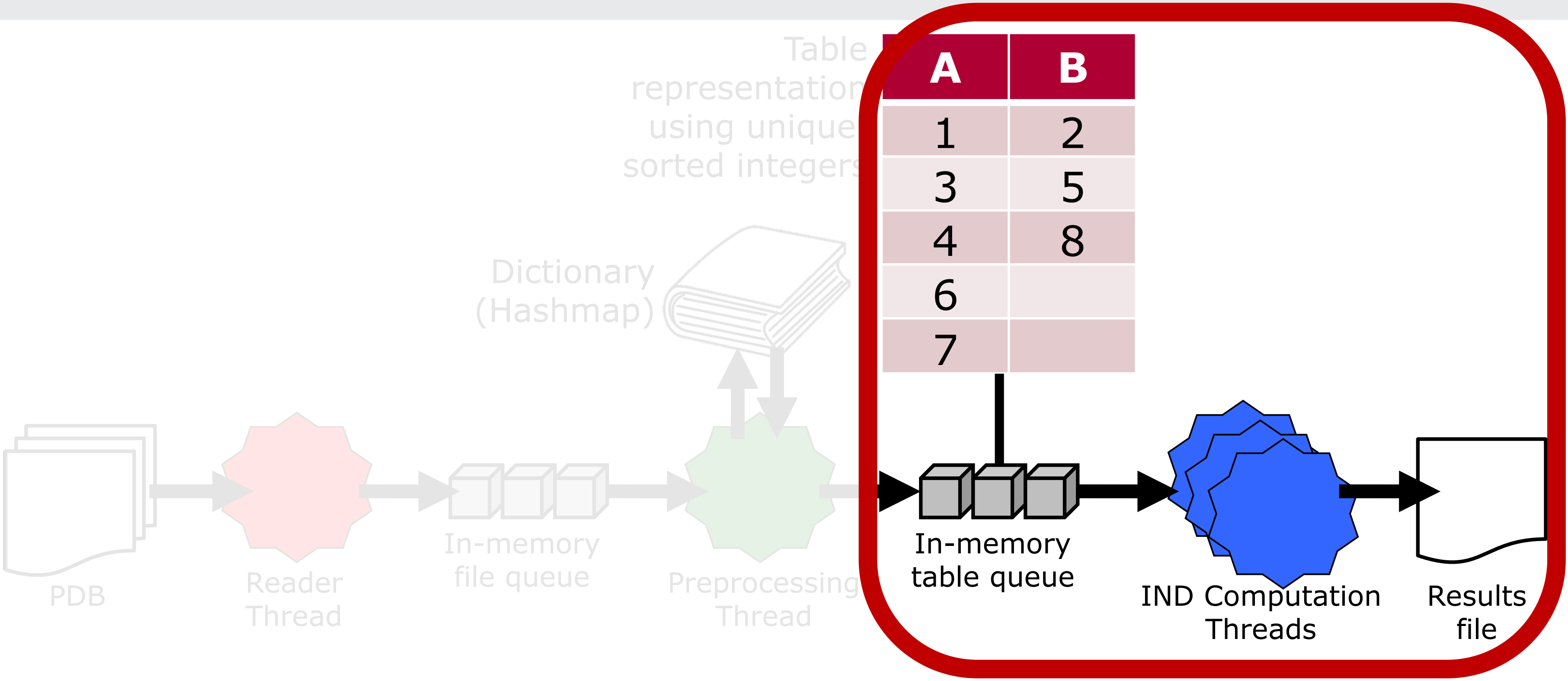
Key	Value
Iris	1
Computing	2
Susi	3
Reuven	4
	5
Eli	6
…	…

Pipeline architecture



A	B
1	2
3	5
4	8
6	
7	

Pipeline architecture



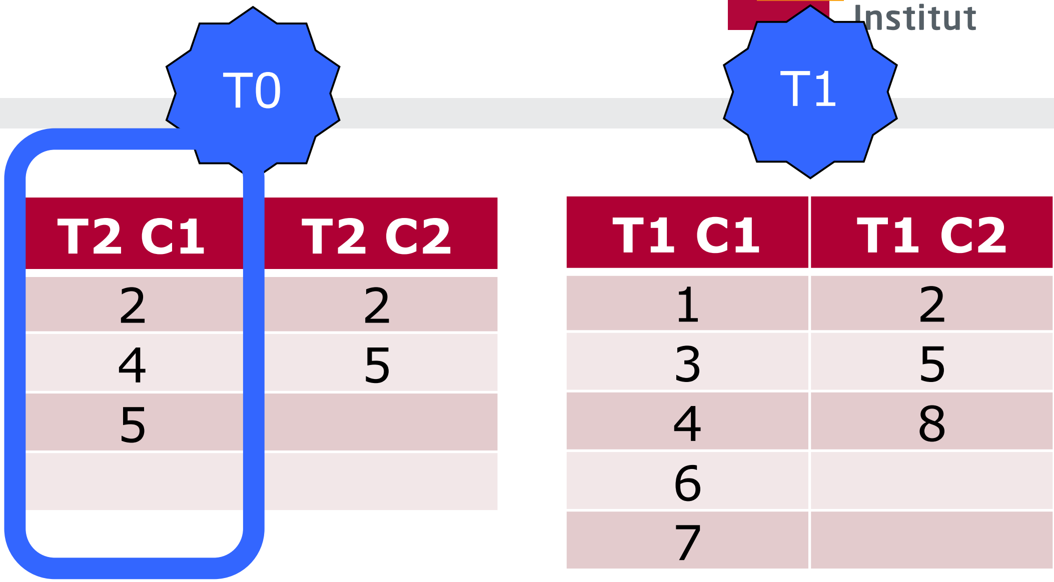
A	B
1	2
3	5
4	8
6	
7	

INDs Computation

- Each thread takes a table
- Checks combinations
  - with all other tables and their columns
  - inside itself

- Compares to all tables with smaller ID are taken care of

- Whenever it finds an IND, it writes it into the .tsv-file



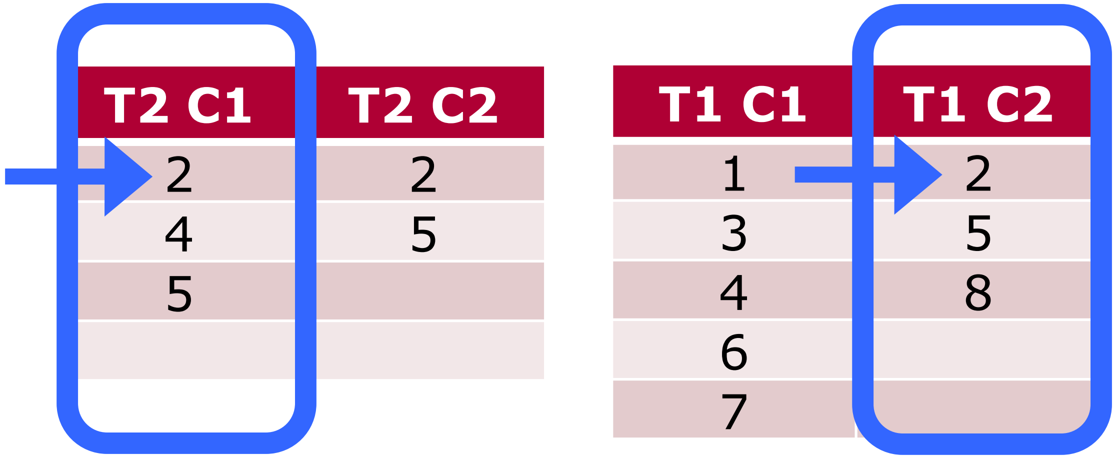
T2 C1	T2 C2
2	2
4	5
5	

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

INDs core algorithm

- Given two columns
- Assumption:
    - sorted
    - unique
    - integer
- using iterators

INDs core algorithm – case 1 – no IND

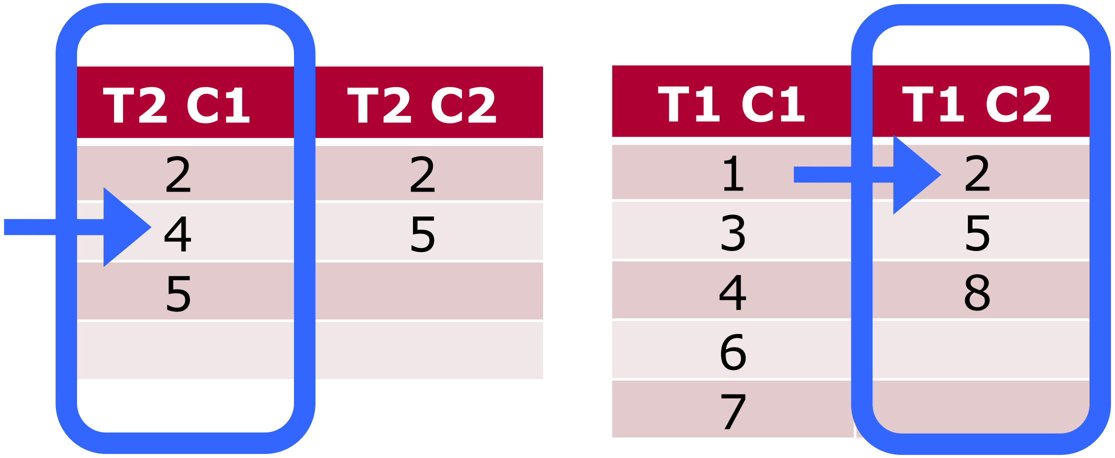


T2 C1	T2 C2
2	2
4	5
5	

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
  - case 1: (left number == right number)
    - left iterator increased (algorithm proceeds)
  - case 2: (right number > left number)
    - no IND (algorithm stops)

INDs core algorithm – case 1 – no IND

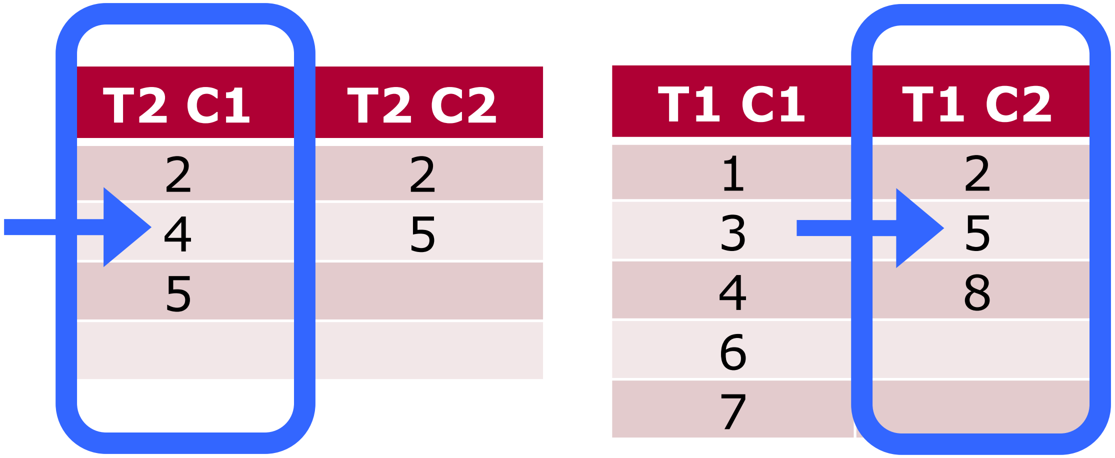


T2 C1	T2 C2
2	2
4	5
5	

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
  - case 1: (left number == right number)
    - left iterator increased (algorithm proceeds)
  - case 2: (right number > left number)
    - no IND (algorithm stops)

INDs core algorithm – case 1 – no IND

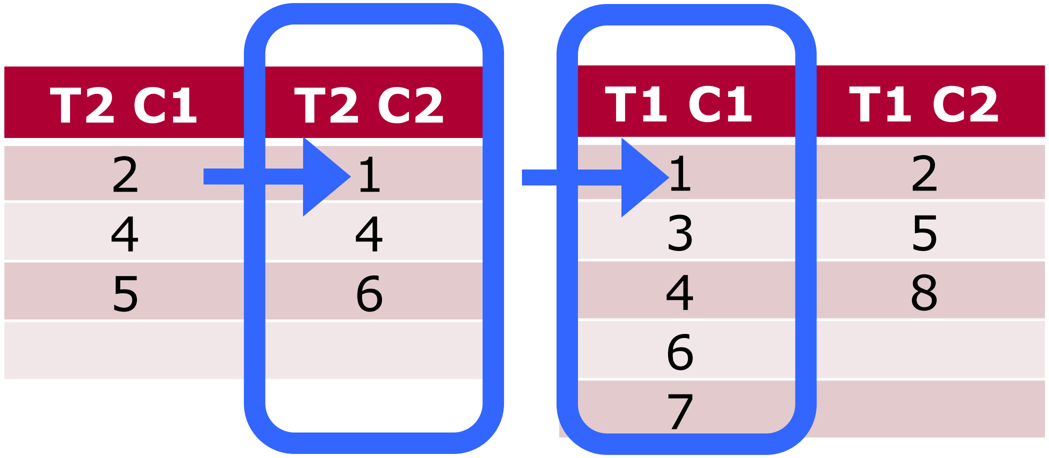


T2 C1	T2 C2
2	2
4	5
5	

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
  - case 1: (left number == right number)
    - left iterator increased (algorithm proceeds)
  - case 2: (right number > left number)
    - no IND (algorithm stops)

INDs core algorithm – case 2 – IND

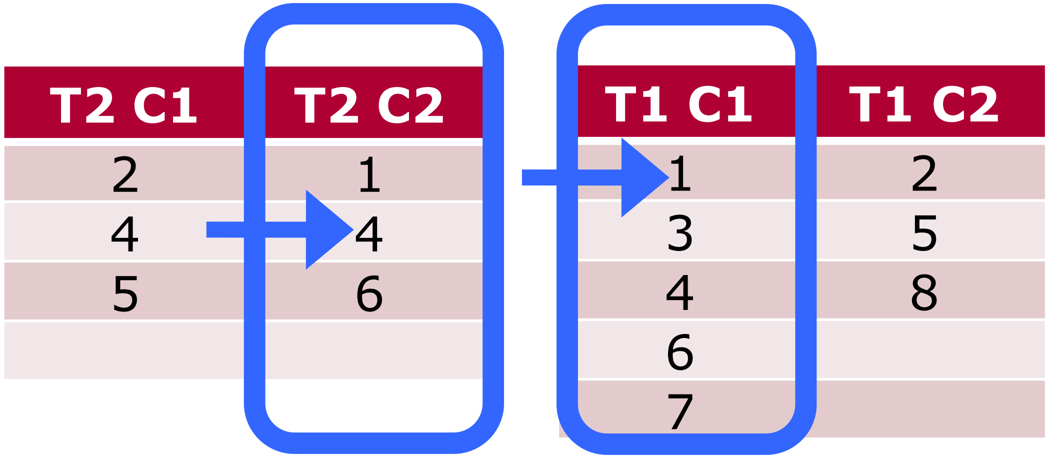


T2 C1	T2 C2
2	1
4	4
5	6

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
  - case 1: (left number == right number)
    - left iterator increased (algorithm proceeds)
  - case 2: (right number > left number)
    - no IND (algorithm stops)

INDs core algorithm – case 2 – IND

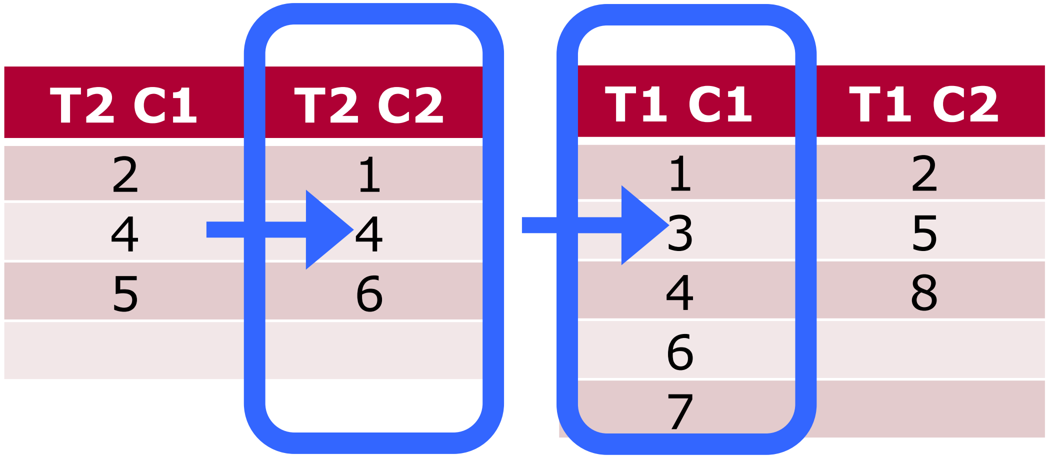


T2 C1	T2 C2
2	1
4	4
5	6

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
  - case 1: (left number == right number)
    - left iterator increased (algorithm proceeds)
  - case 2: (right number > left number)
    - no IND (algorithm stops)

INDs core algorithm – case 2 – IND

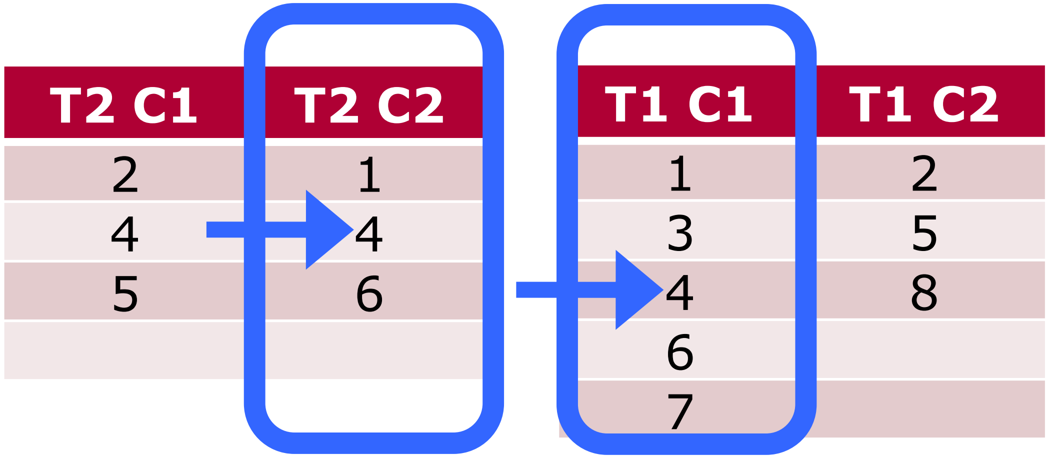


T2 C1	T2 C2
2	1
4	4
5	6

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
  - case 1: (left number == right number)
    - left iterator increased (algorithm proceeds)
  - case 2: (right number > left number)
    - no IND (algorithm stops)

INDs core algorithm – case 2 – IND

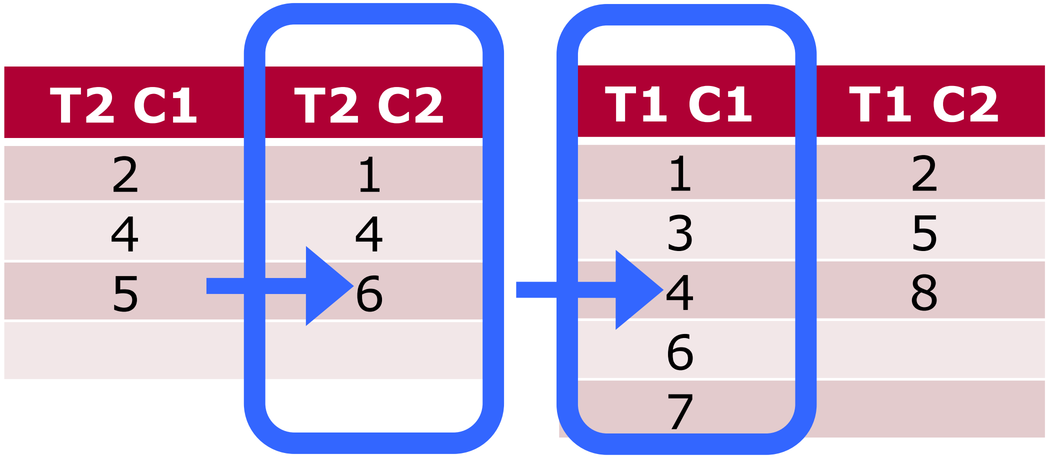


T2 C1	T2 C2
2	1
4	4
5	6

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
  - case 1: (left number == right number)
    - left iterator increased (algorithm proceeds)
  - case 2: (right number > left number)
    - no IND (algorithm stops)

INDs core algorithm – case 2 – IND

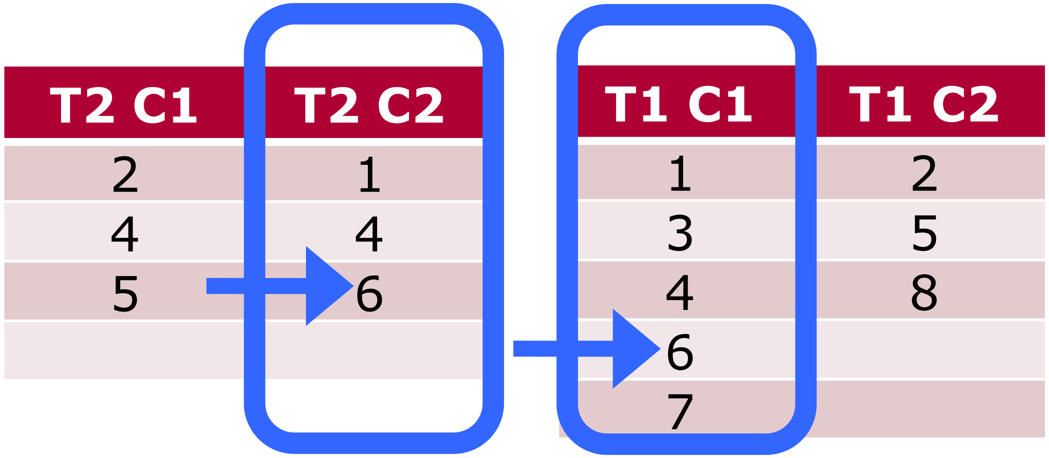


T2 C1	T2 C2
2	1
4	4
5	6

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
  - case 1: (left number == right number)
    - left iterator increased (algorithm proceeds)
  - case 2: (right number > left number)
    - no IND (algorithm stops)

INDs core algorithm – case 2 – IND

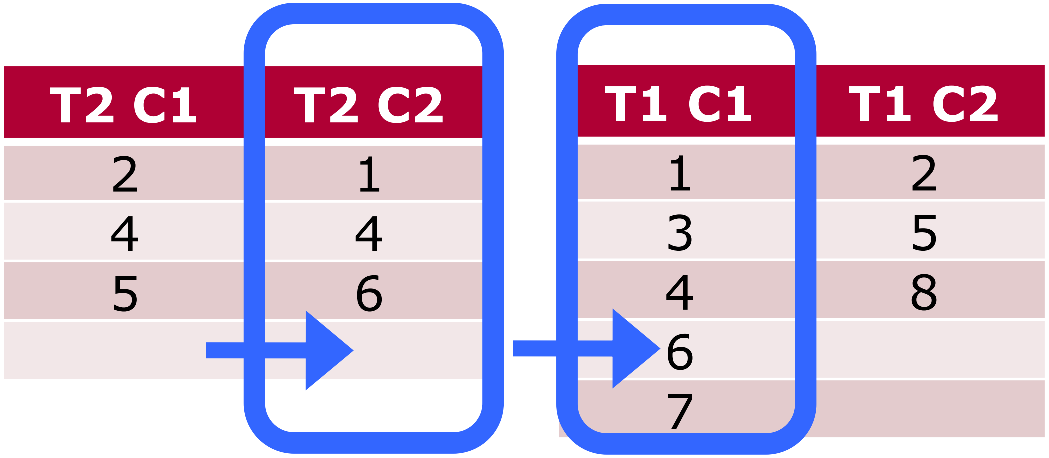


T2 C1	T2 C2
2	1
4	4
5	6

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
    - case 1: (left number == right number)
        - left iterator increased (algorithm proceeds)
    - case 2: (right number > left number)
        - no IND (algorithm stops)

INDs core algorithm – case 2 – IND



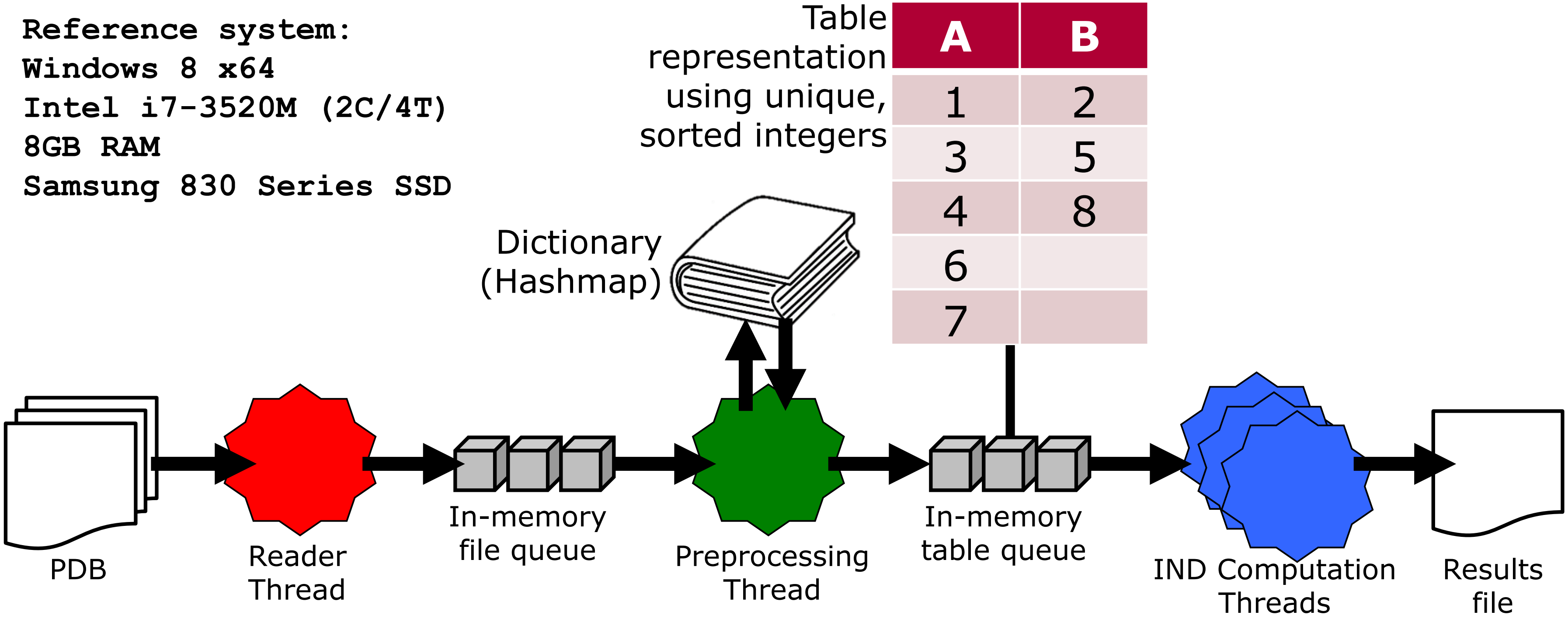
T2 C1	T2 C2
2	1
4	4
5	6

T1 C1	T1 C2
1	2
3	5
4	8
6	
7	

- iterator on the right goes down until
  - case 1: (left number == right number)
    - left iterator increased (algorithm proceeds)
  - case 2: (right number > left number)
    - no IND (algorithm stops)

Performance

```
Reference system:
Windows 8 x64
Intel i7-3520M (2C/4T)
8GB RAM
Samsung 830 Series SSD
```



A	B
1	2
3	5
4	8
6	
7	

~50s **~155s** **~55s**

- Estimated single core performance: 50s + 155s + 55s = 4m 20s
- Actual performance thanks to pipelining: 165s = **2m 45s**

Conclusion & Discussion

- In the end, we found **27493** unary inclusion dependencies

➕
- Reading small files first to feed the pipeline
  - 80% of all INDs are found in the first 20% of runtime

➕
- Sequential reading of entire files instead of “row by row”

➕
- Pipelining approach reduces CPU idle time
  - Start processing data when the first file is loaded

➖
- Preprocessing is still a bottleneck
  - Lots of dictionary reads/writes

➖
- Memory usage increases linearly with data size
  - Especially when containing lots of unique values