# Caching for Data Analysis

**Ken Birman**, Theo Gkountouvas

# Data Analysis

Data processing is growing very fast compared to the hardware acceleration.

1. Volume
2. Complexity

# Spark RDDs

- Spark uses Resilient Distributed Datasets (RDDs) as a core structure.
- Word Count Example (Scala):

```scala
val textRDD = sc.textFile("hdfs://...")
val flatMapRDD = textRDD.flatMap(line => line.split(" "))
val mapRDD = flatMapRDD.map(word => (word, 1))
val counts = mapRDD.reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

<u>textRDD</u>
Input RDD(s): -
Operation: readFile

<u>mapRDD</u>
Input RDD(s): flatMapRDD
Operation: map

# Lineage of RDDs and Lazy Execution

textRDD
Input RDD(s): -
Operation: readFile

flatMapRDD
Input RDD(s): textRDD
Operation: flatMap

mapRDD
Input RDD(s): flatMapRDD
Operation: map

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

<u>textRDD</u>
Input RDD(s): -
Operation: readFile

<u>flatMapRDD</u>
Input RDD(s): textRDD
Operation: flatMap

<u>mapRDD</u>
Input RDD(s): flatMapRDD
Operation: map

Triggers execution

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

<u>textRDD</u>
Input RDD(s): -
Operation: readFile

<u>flatMapRDD</u>
Input RDD(s): textRDD
Operation: flatMap

<u>mapRDD</u>
Input RDD(s): flatMapRDD
Operation: map

Needs results of operation

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

<u>textRDD</u>
Input RDD(s): -
Operation: readFile

<u>flatMapRDD</u>
Input RDD(s): textRDD
Operation: flatMap

<u>mapRDD</u>
Input RDD(s): flatMapRDD
Operation: map

Has input RDDs

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

<u>textRDD</u>
Input RDD(s): -
Operation: readFile

<u>flatMapRDD</u>
Input RDD(s): textRDD
Operation: flatMap

Needs result of operation

<u>mapRDD</u>
Input RDD(s): flatMapRDD
Operation: map

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

textRDD
Input RDD(s): -
Operation: readFile

flatMapRDD
Input RDD(s): textRDD
Operation: flatMap

mapRDD
Input RDD(s): flatMapRDD
Operation: map

No input RDDs

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

<u>textRDD</u>
Input RDD(s): -
Operation: readFile

<u>flatMapRDD</u>
Input RDD(s): textRDD
Operation: flatMap

<u>mapRDD</u>
Input RDD(s): flatMapRDD
Operation: map

Execute operation

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

"Hello World!"
"Hello Ithaca"

textRDD
Input RDD(s): -
Operation: readFile

flatMapRDD
Input RDD(s): textRDD
Operation: flatMap

mapRDD
Input RDD(s): flatMapRDD
Operation: map

Provide results

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

<u>textRDD</u>
Input RDD(s): -
Operation: readFile

<u>flatMapRDD</u>
Input RDD(s): textRDD
Operation: flatMap

[“Hello”,”World”,
“Hello”, “Ithaca”]

<u>mapRDD</u>
Input RDD(s): flatMapRDD
Operation: map

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

<u>textRDD</u>
Input RDD(s): -
Operation: readFile

<u>flatMapRDD</u>
Input RDD(s): textRDD
Operation: flatMap

<u>mapRDD</u>
Input RDD(s): flatMapRDD
Operation: map

[(“Hello”,1),(”World”,1),
(“Hello”,1),(“Ithaca”,1)]

```
val counts = mapRDD.reduceByKey(_ + _)
```

# Lineage of RDDs and Lazy Execution

<u>textRDD</u>
Input RDD(s): -
Operation: readFile

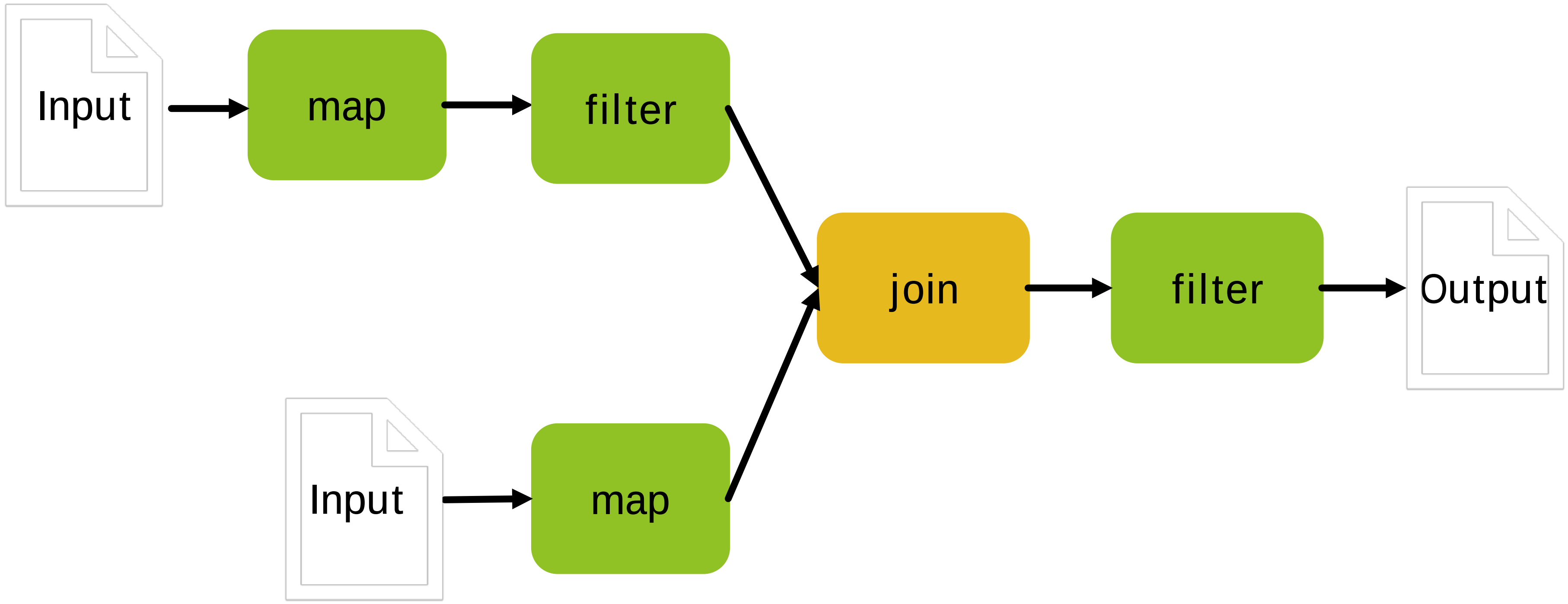
<u>flatMapRDD</u>
Input RDD(s): textRDD
Operation: flatMap

<u>mapRDD</u>
Input RDD(s): flatMapRDD
Operation: map

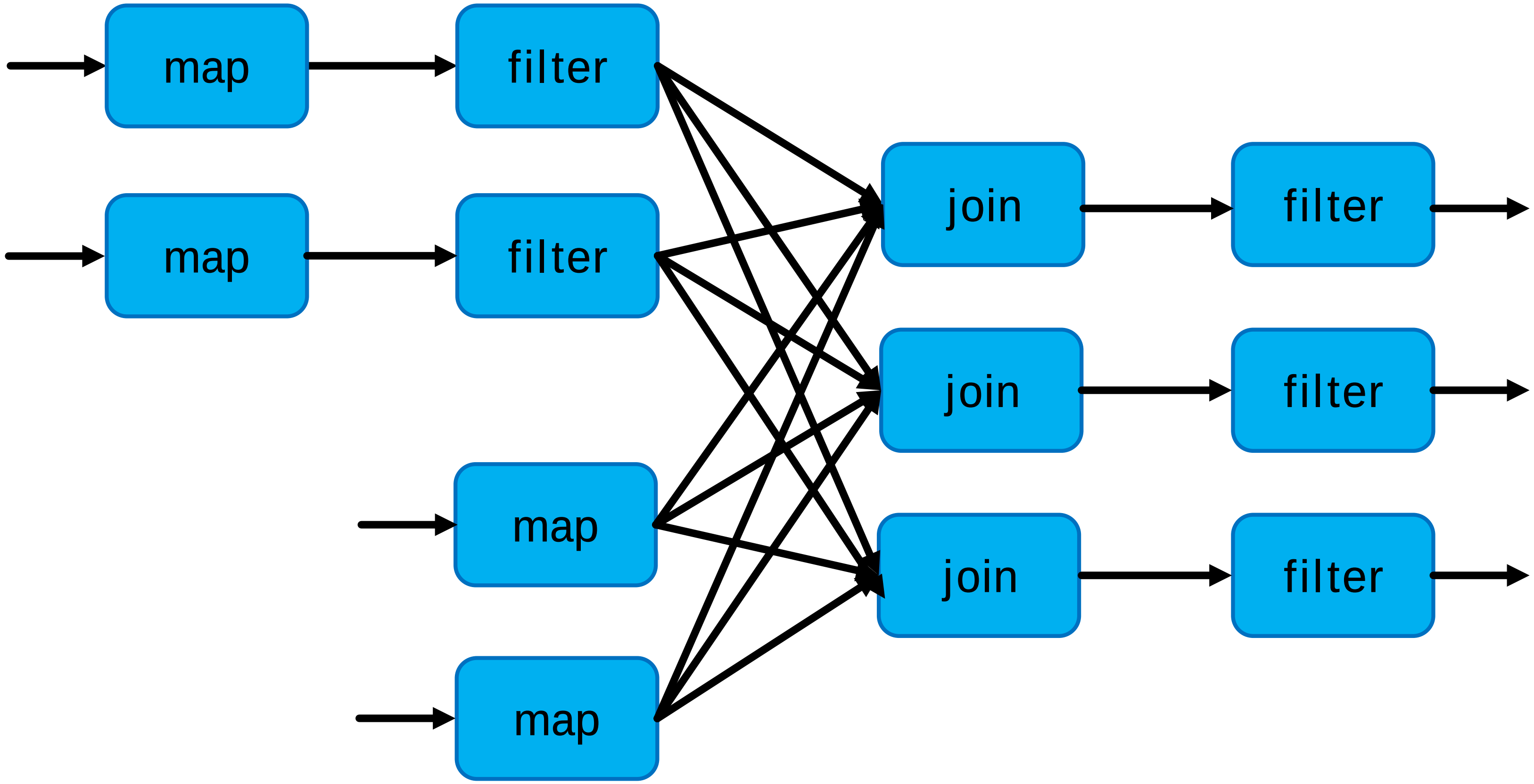
{“Hello”:2,”World”:1,
“Ithaca”:1}

```
val counts = mapRDD.reduceByKey(_ + _)
```

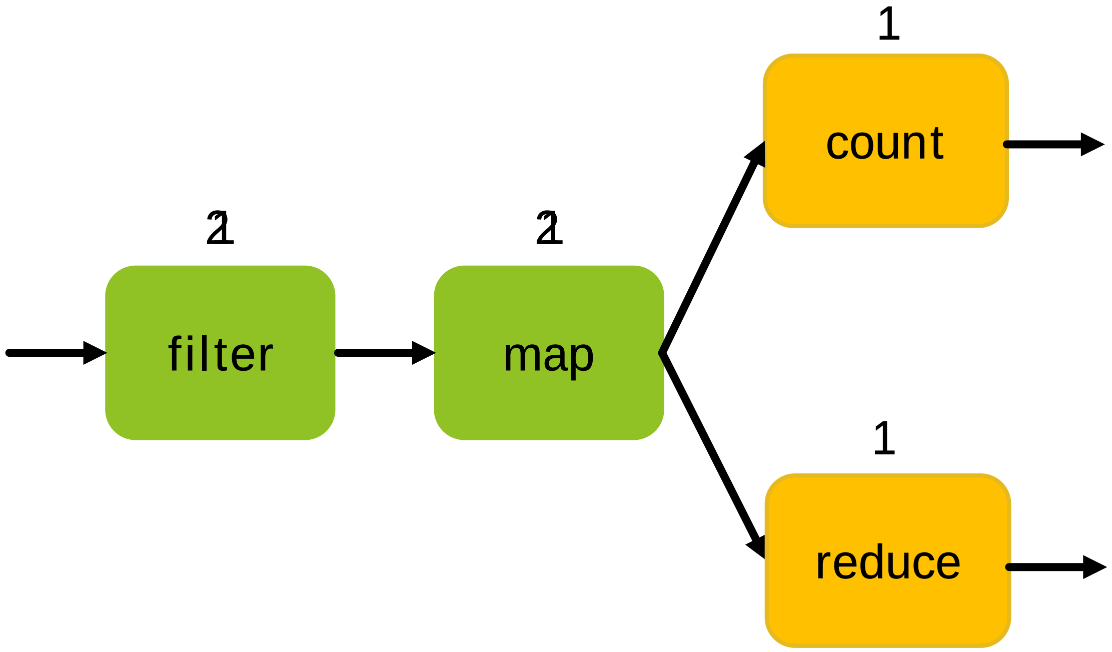
# Dataflow – Logical Plan (Operations)



# Dataflow – Execution Plan (Tasks)



# Why caching in Spark is essential?




1. Cache intermediate results
2. Avoid re-execution of operations.
3. Save mostly CPU-cycles instead of I/O.

# Multiple choices for of caching

- NONE (Default)
- MEMORY_ONLY
- MEMORY_ONLY_SER
- MEMORY_AND_DISK
- MEMORY_AND_DISK_SER
- DISK_ONLY
- ...

# User decides what to cache in Spark

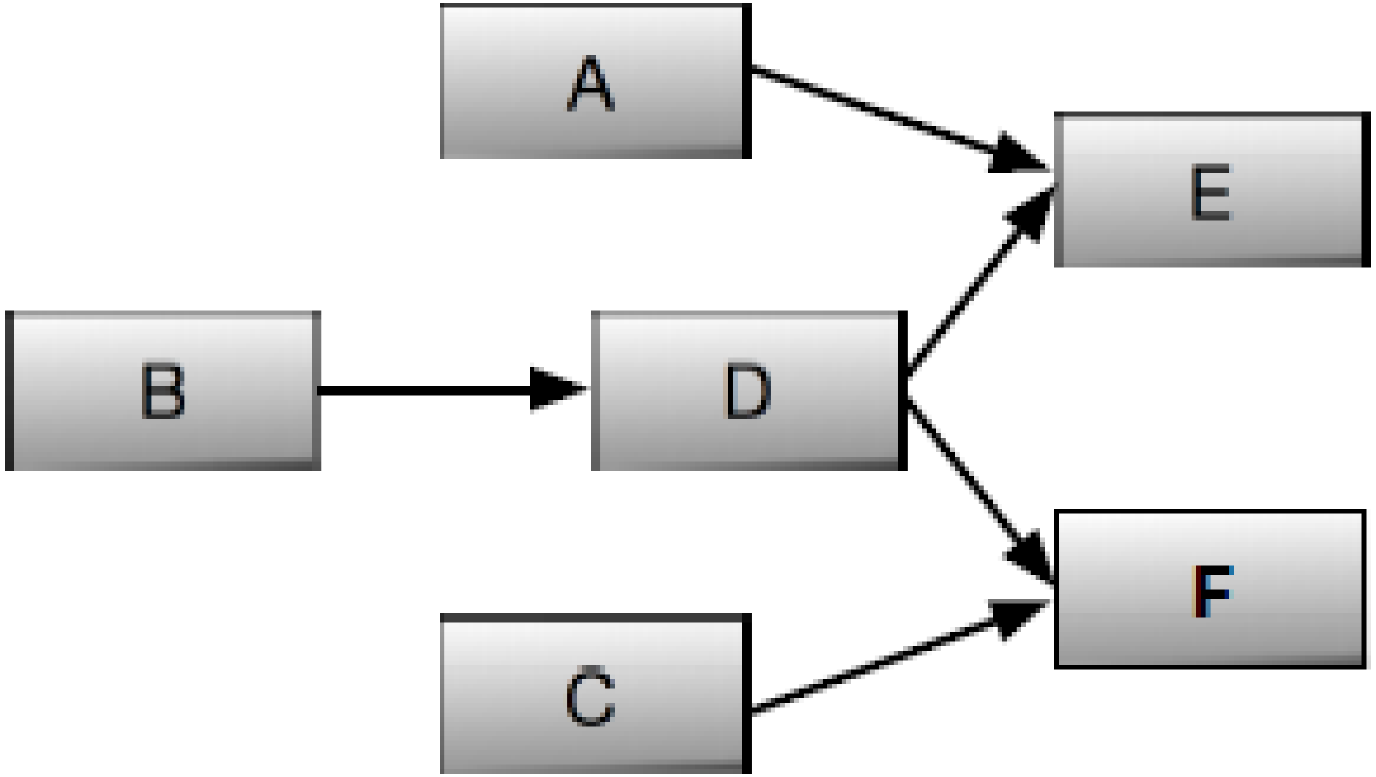
Users have to define what they want to cache by using *cache()* or *persist()* keywords after RDD.

1. Static analysis for what to cache is harder than traditional cases. Instead of caching only initial data, Spark has the ability to cache intermediate results, too.
2. Multiple choices about where to cache complicate things (Memory, SSD, Disk, etc.).
3. Caching might lead to worse results than simply re-executing (especially with SSD, Disks, Serialization).

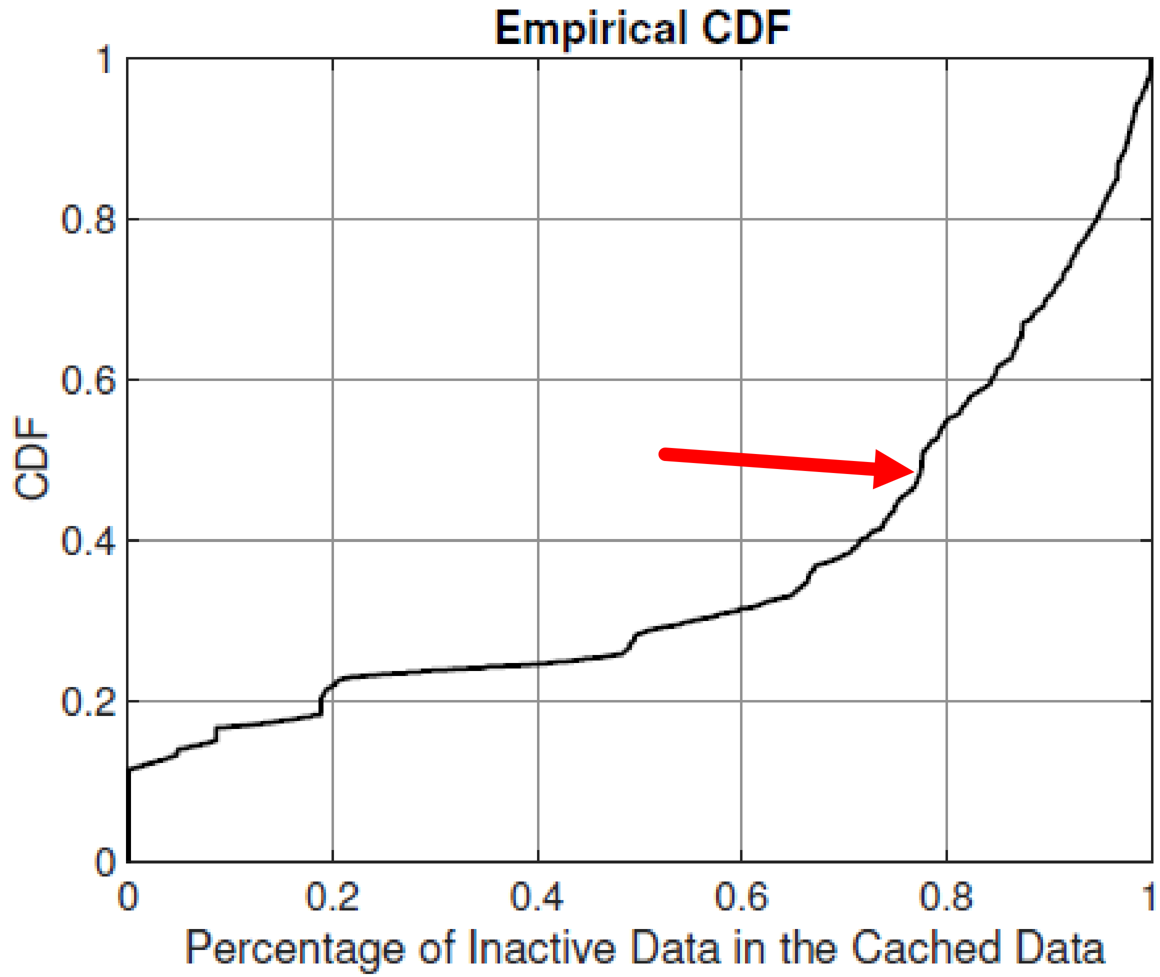
# Eviction Policy

- Spark uses LRU for default eviction policy. Unlike selection about what to cache, eviction is automatic.
- However, classic eviction policies do not exploit structure of the graph.

# Why LRU is not so good?



# Experimental Study on Spark Bench (15 jobs)



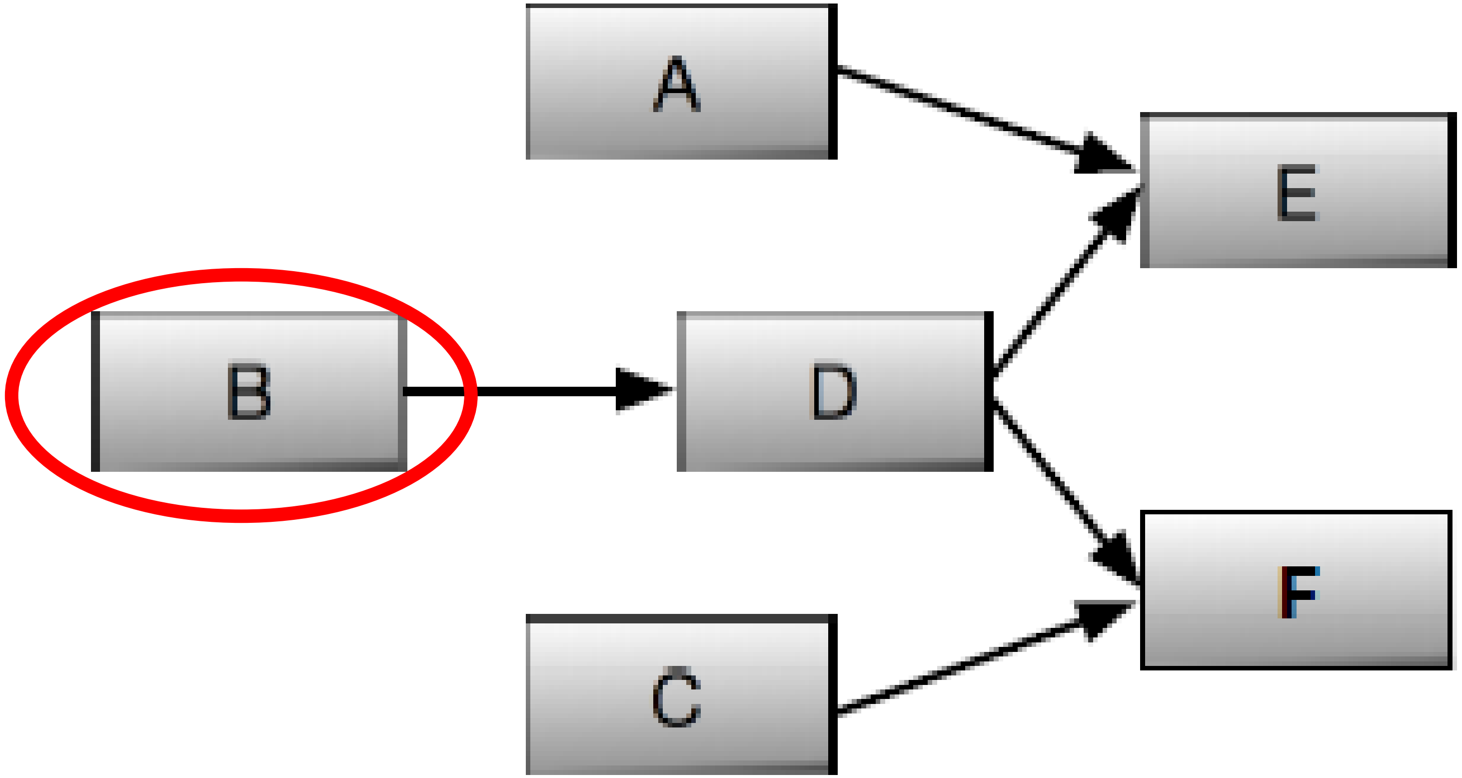
# LRC: Dependency-Aware Cache Management for Data Analytics Clusters

Yinghao Yu, Wei Wang, Jun Zhang, Khaled Ben Letaief

**Definition** (*Reference Count*):

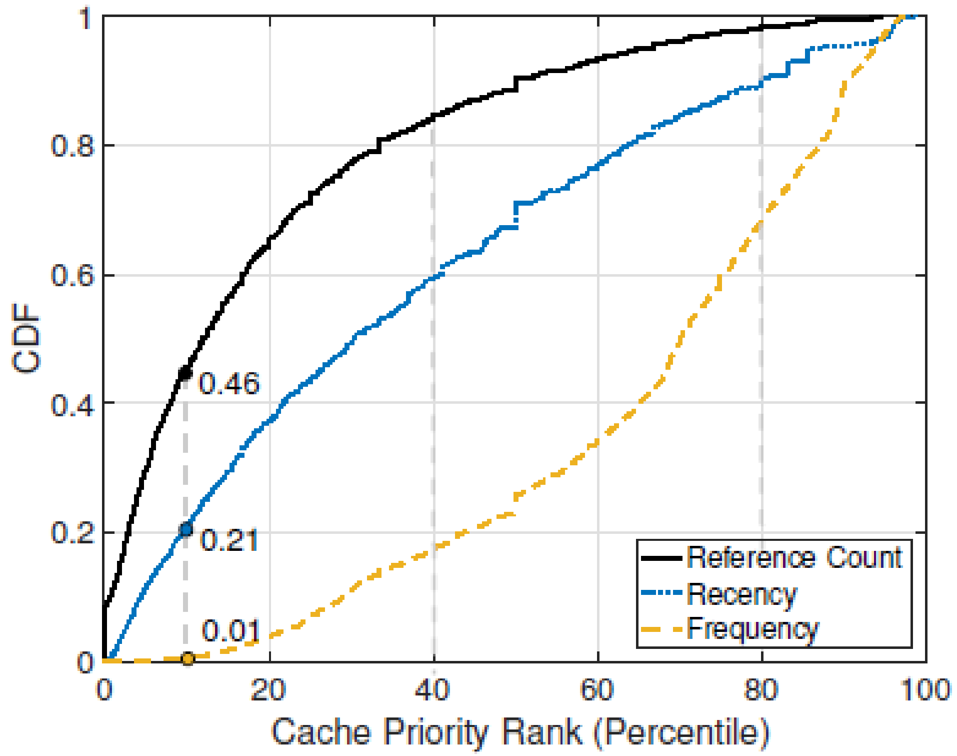
For each data block $b$, the reference count is define as the number of child blocks that are derived from $b$, but have not yet been computed.

# LRC: Least Reference Count

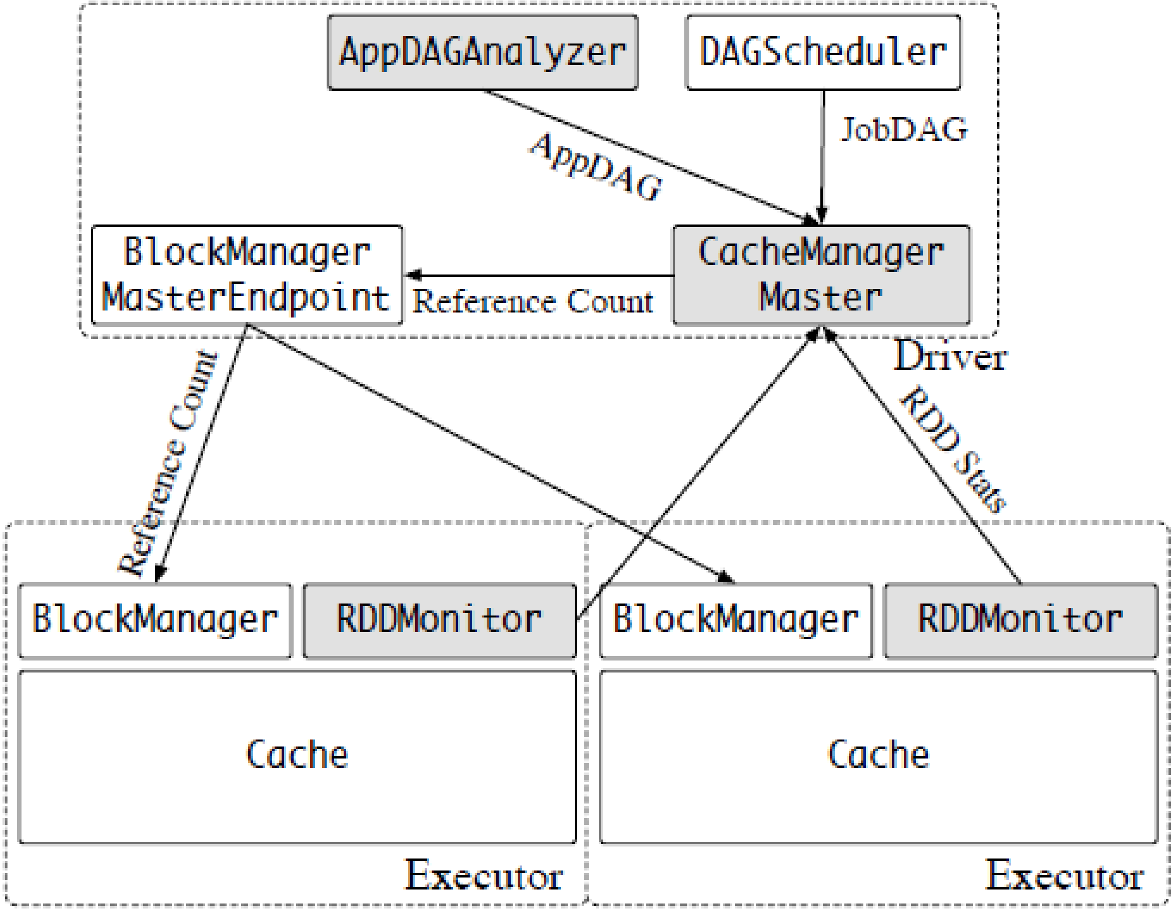


# LRC: Least Reference Count

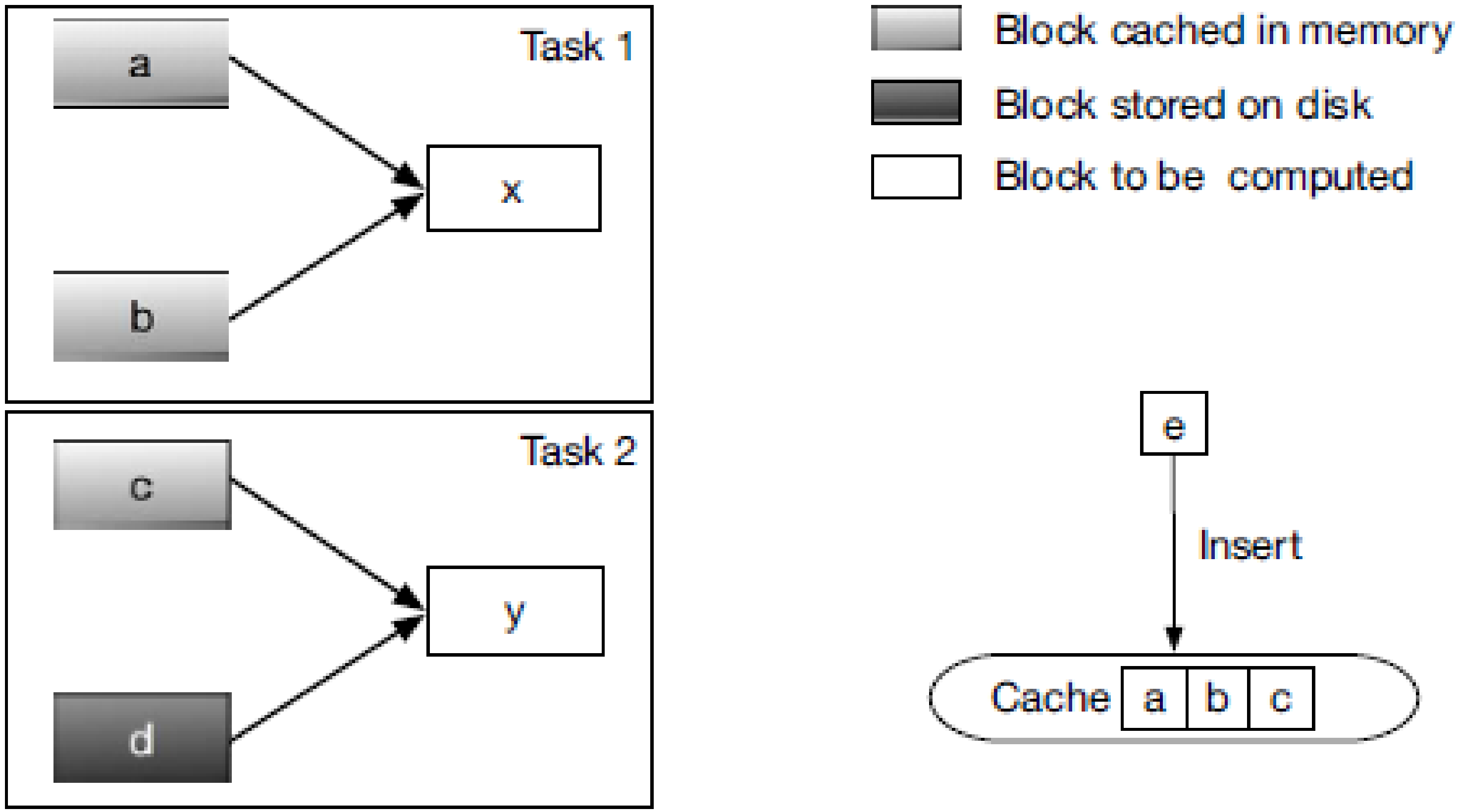
- Unused blocks with zero active references are evicted.
- Reference count is a better indicator for caching.



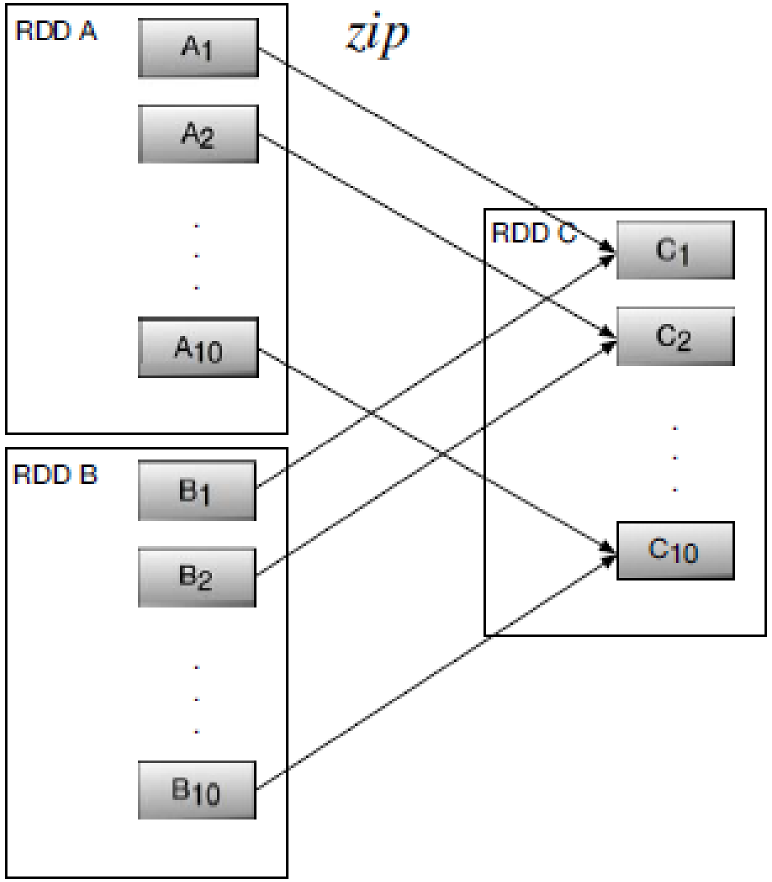
# Solution - Architecture



# Problem – Is this enough?



# Problem – Peer Dependencies

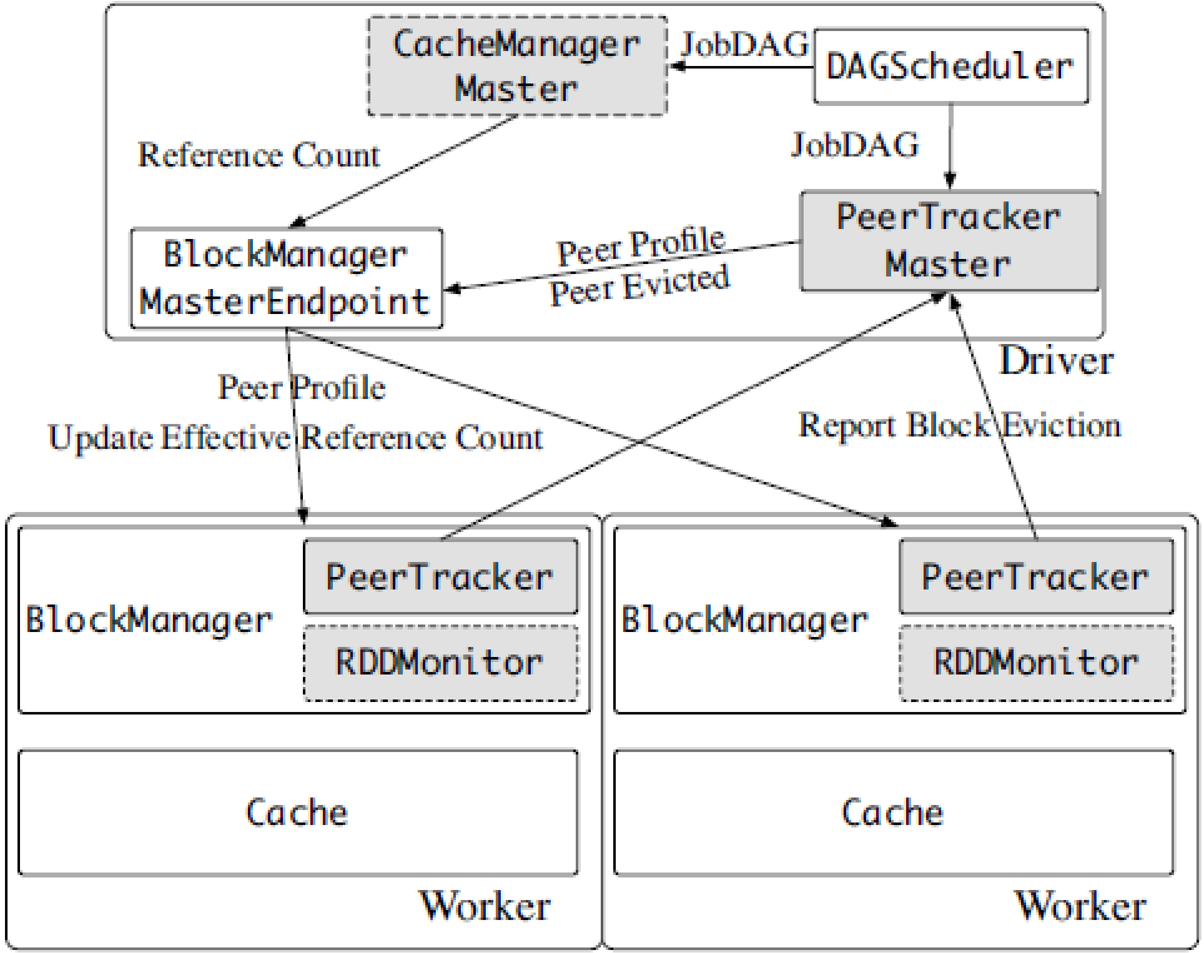



- If results of $A_i$ are not cached, then $B_i$ results should not be cached and vice-versa.
- Latency will remain the same if $A_i$ and $B_i$ results have similar size even if we cache one of them (the other is going to be the bottleneck.

**Definition** (*Effective Reference Count*)**:**

Let block $b$ be referenced by task $t$. We say this reference is effective if task $t$'s dependent blocks, if computed, are all cached in memory.

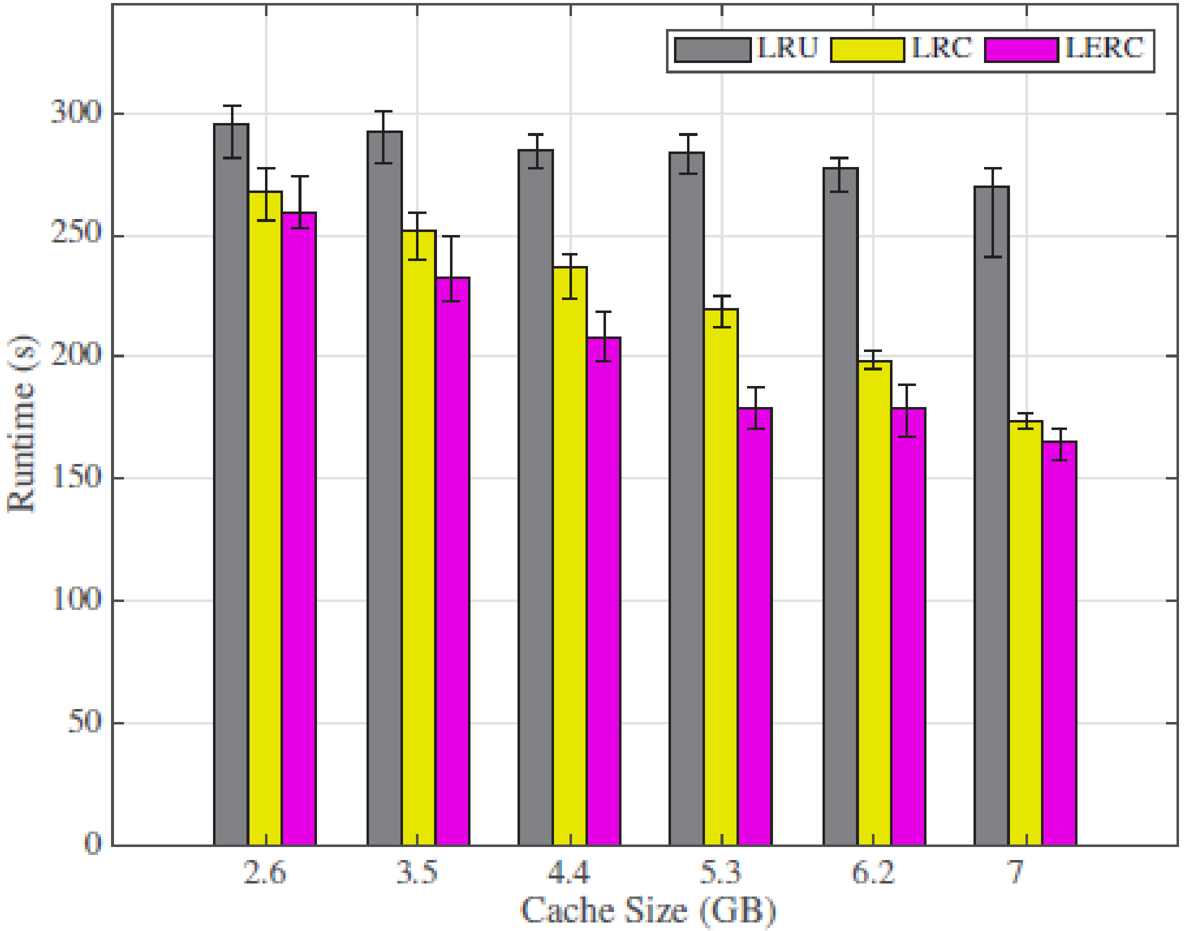
# Solution - LERC



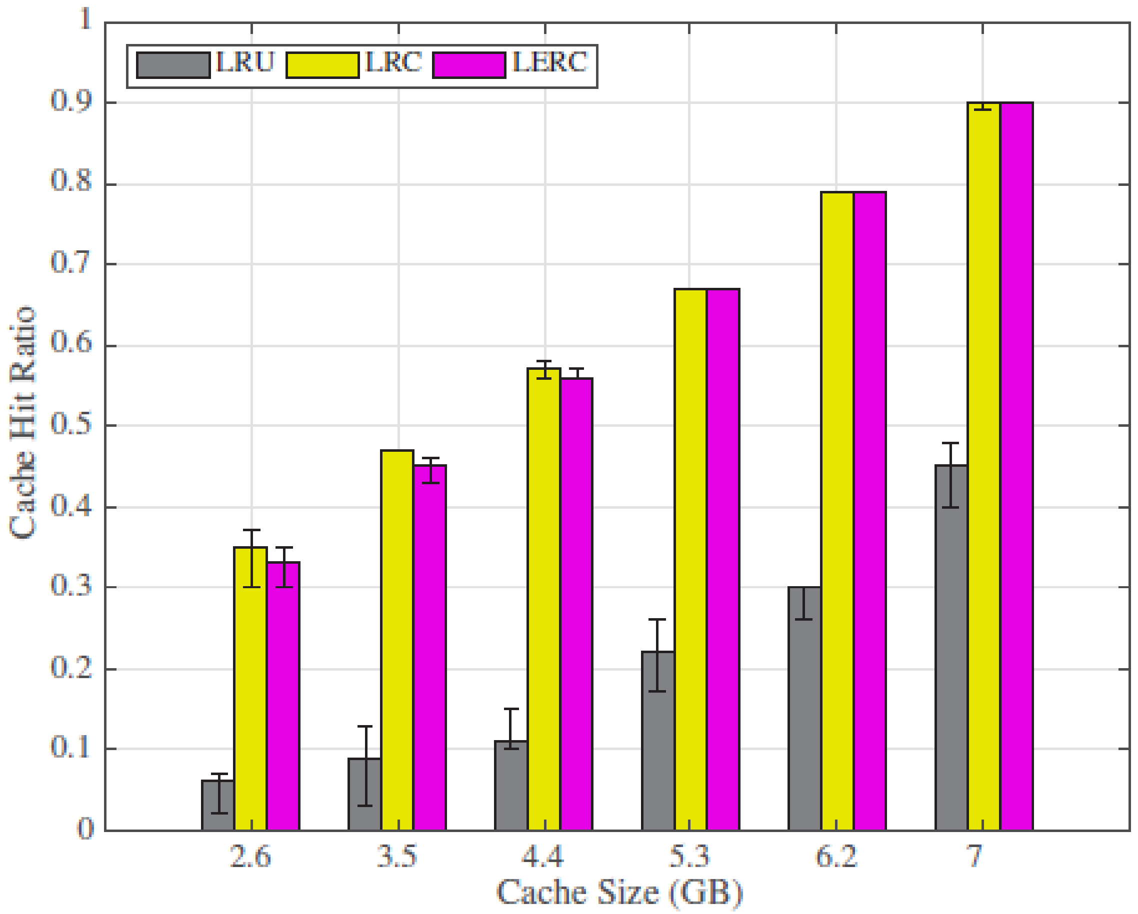
# Experiments – Platform and Setting

- Amazon EC2
  - Cluster with 20 nodes of type m4.large
    - 2.4 GHz Intel Xeon E5-2676 v3 (Haswell) processor
    - 8 GB memory
- Zip application
  - 10 different independent jobs
  - 100 A blocks and 100 B blocks that are zipped together
  - 8 GB total size

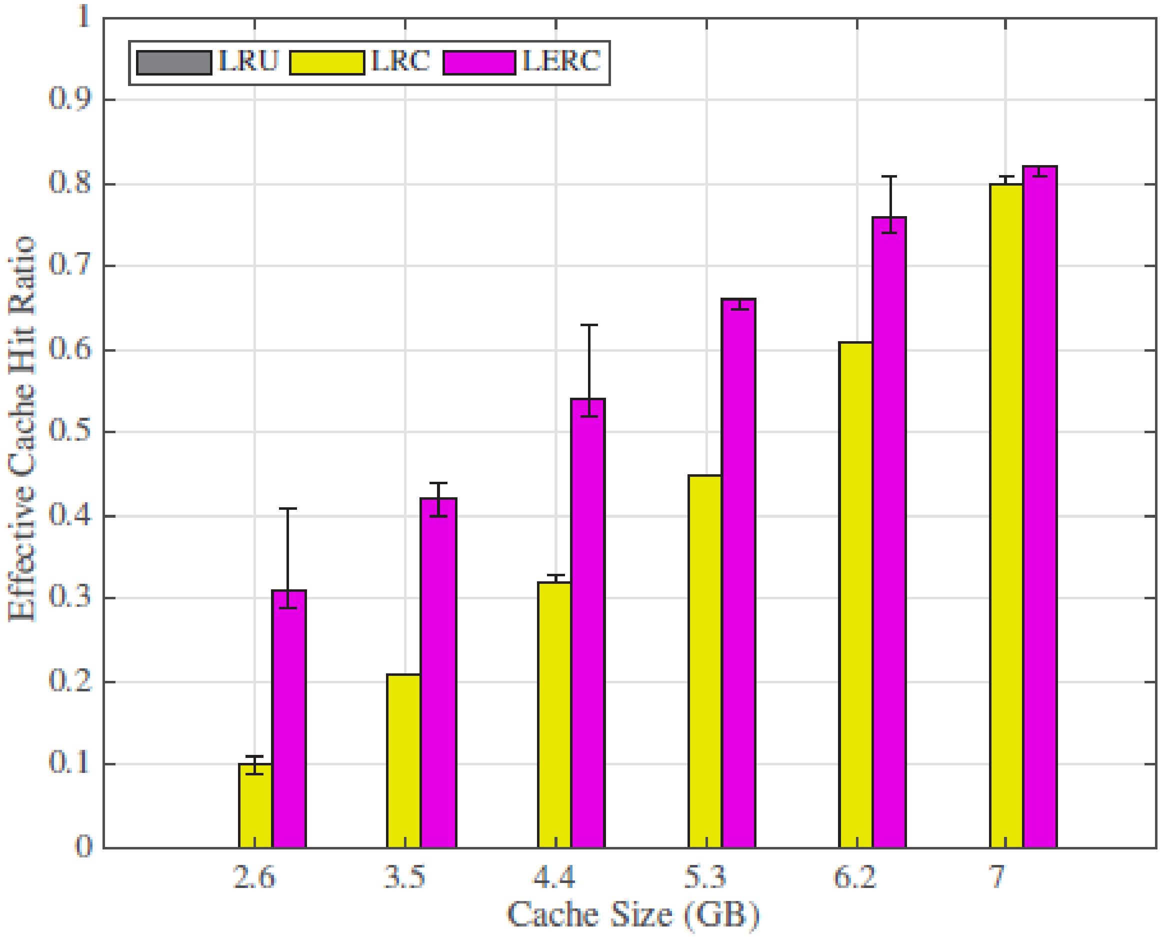
# Experiments - Performance



# Experiments – Overall Cache Hit



# Experiments – Effective Cache Hit



# Temporal Caching
# [Work in Progress]

Theodoros Gkountouvas, Weijia Song, Haoze Wu, Ken Birman

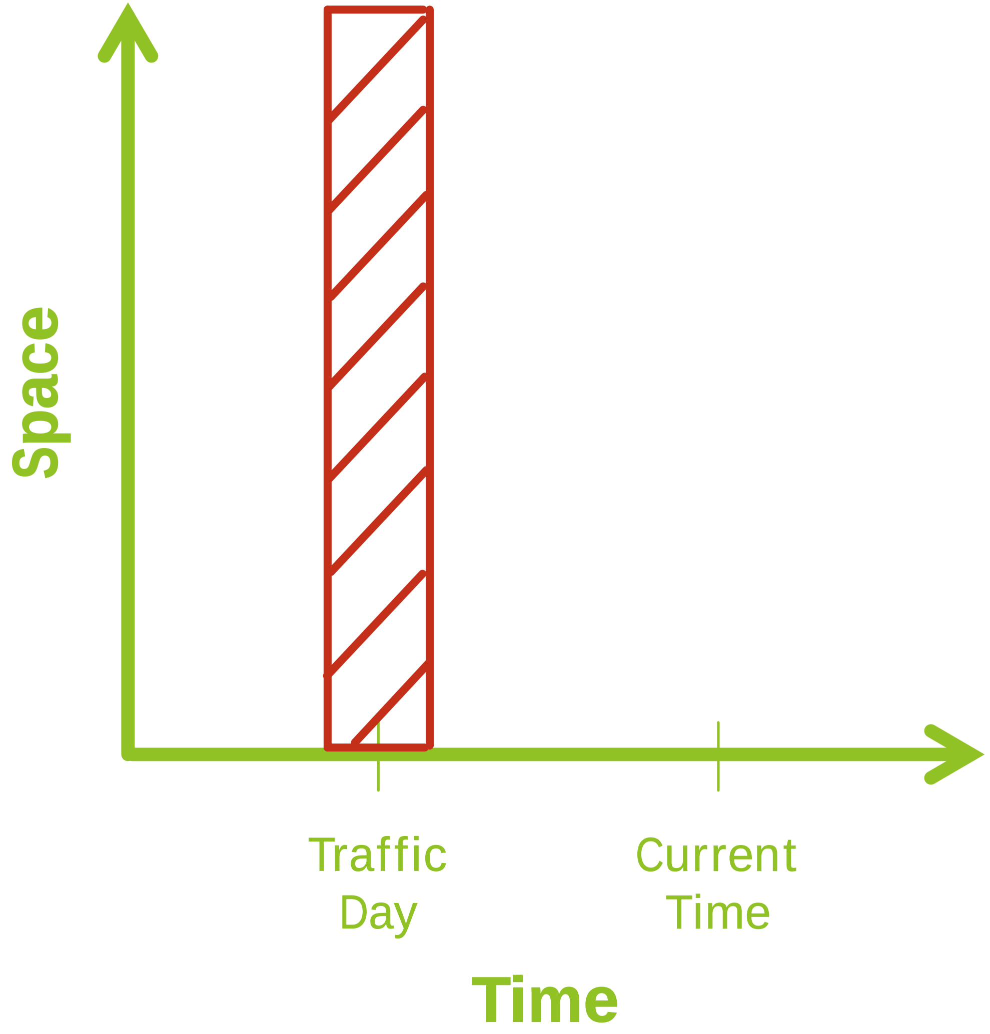
# Time-Series Data

- Timestamped Data
  - Large amount
  - High frequency
- Temporal Queries
  - Sophisticated queries (ML, Optimization)
  - Can be divided to:
    - Fixed Temporal Queries
    - Sliding Temporal Queries

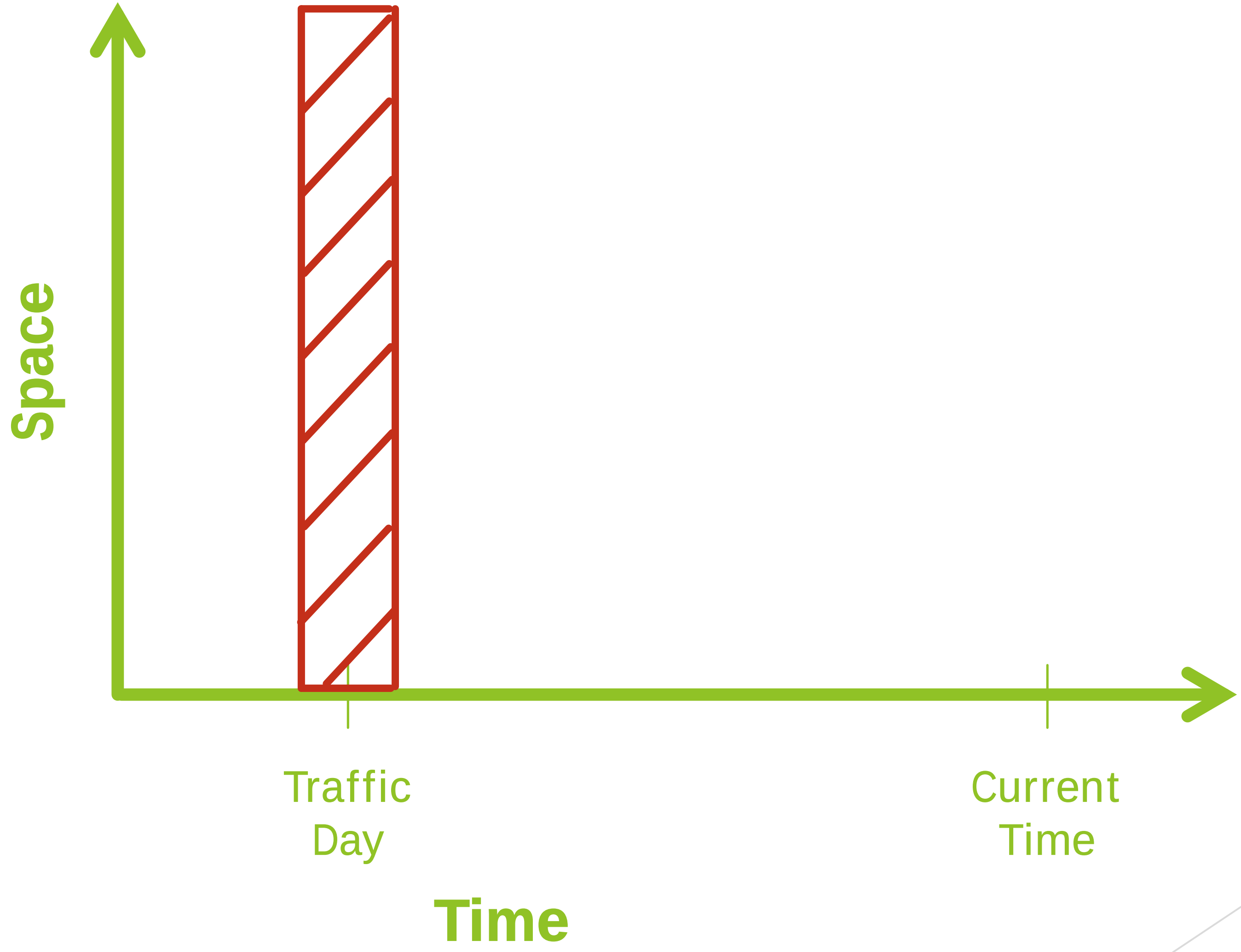
# Example – NYC taxi data

# Fixed Temporal Query - Example

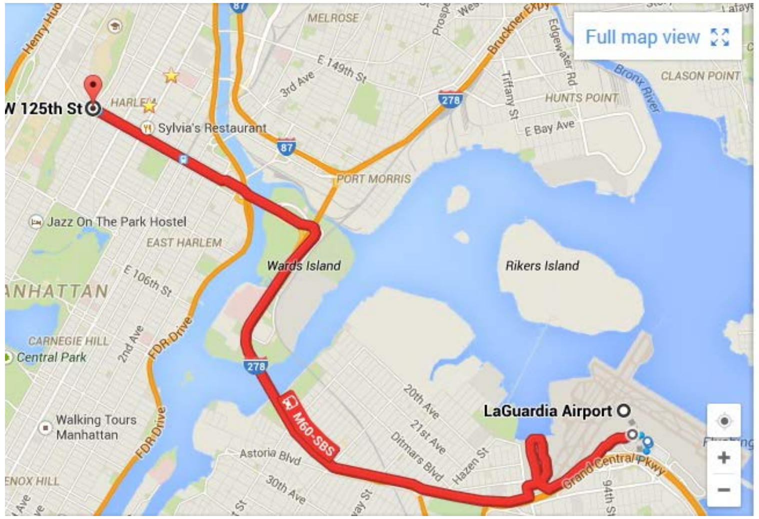
# Fixed Temporal Query - Explanation



# Fixed Temporal Query - Explanation

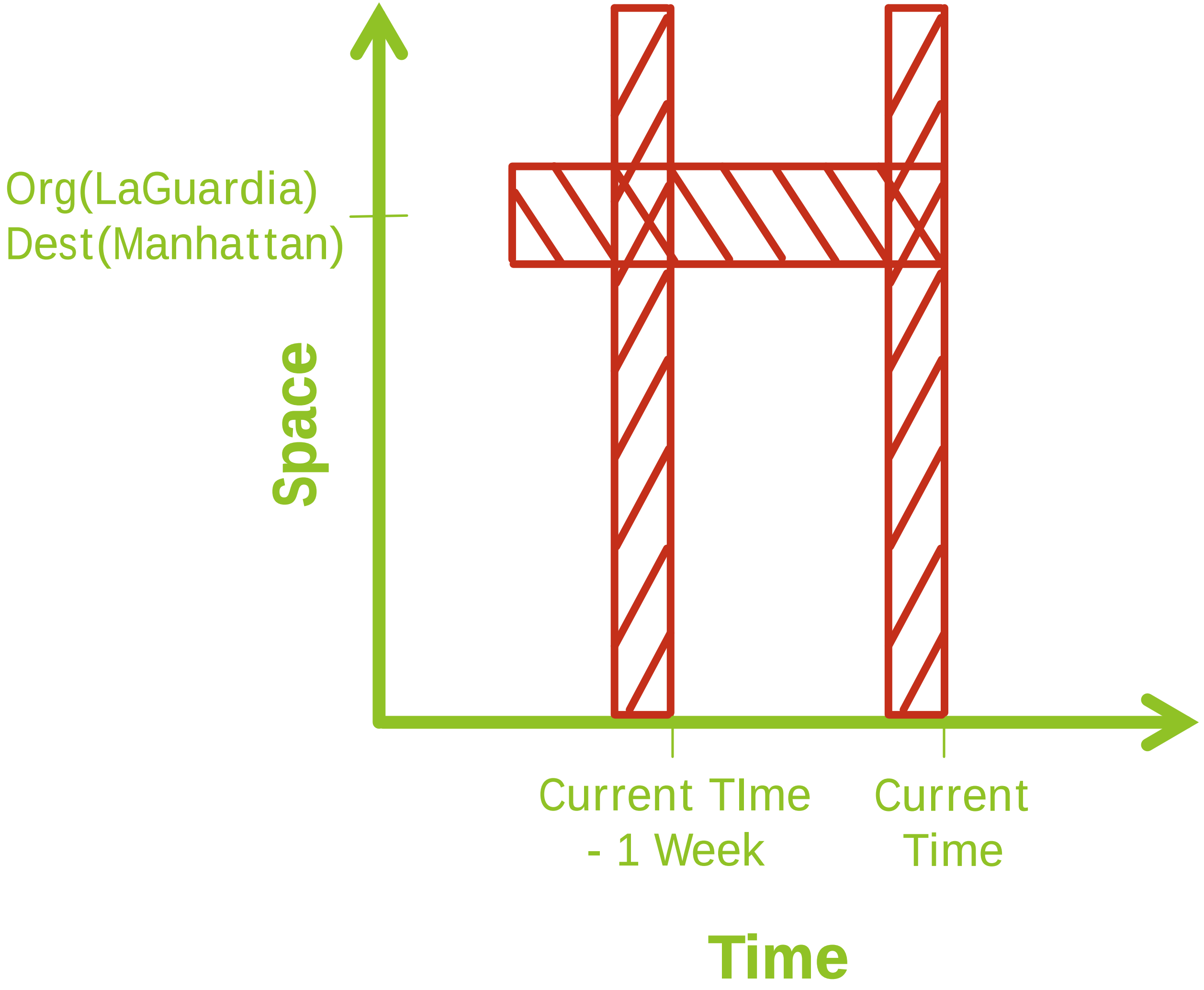


# Sliding Temporal Query - Example

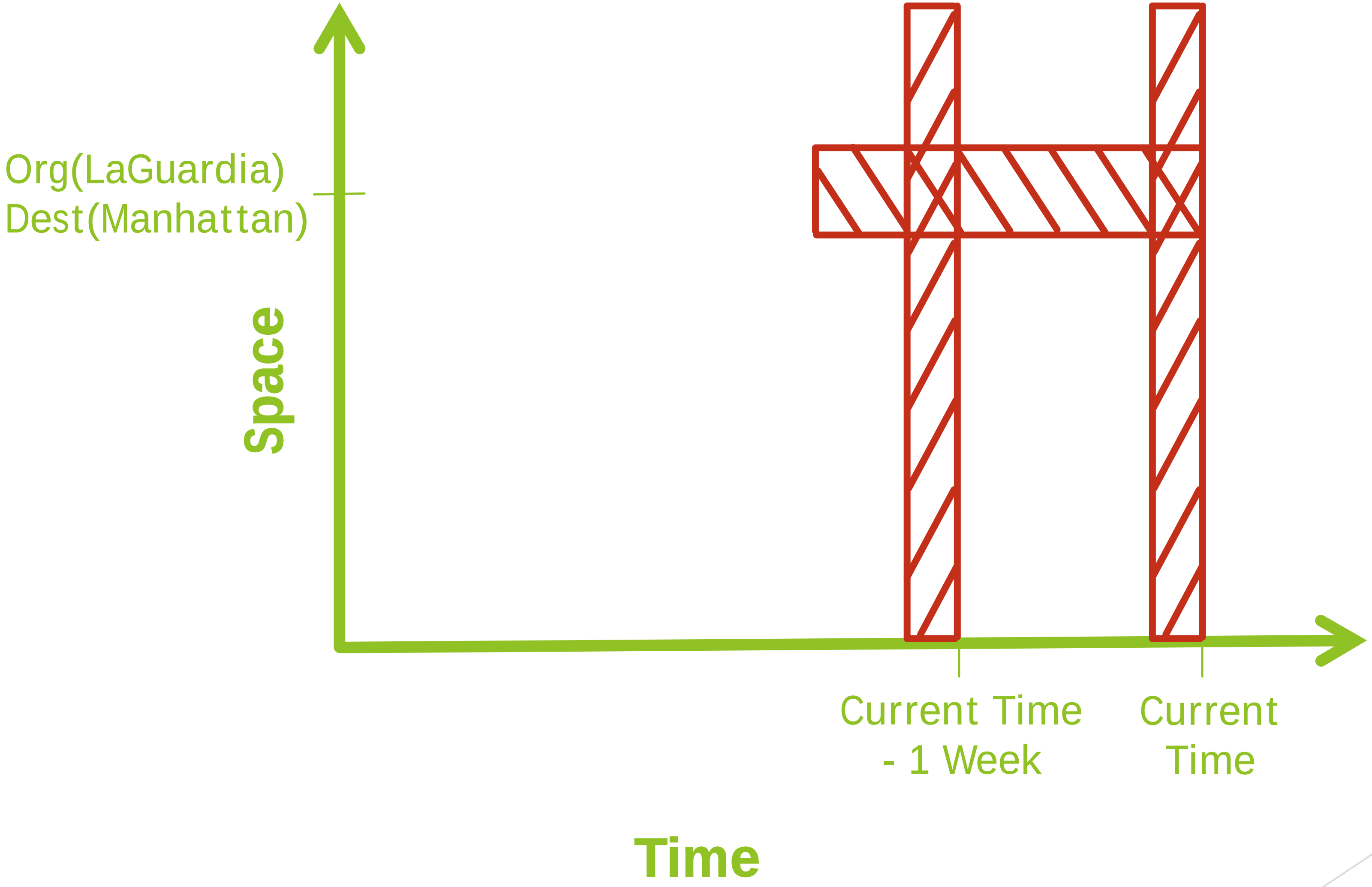



[IEEE TIST,2015,Wang]

# Sliding Temporal Query - Explanation



# Sliding Temporal Query - Explanation



# ARIMA for Time-Series Data

$$\hat{y}_t \quad = \quad \mu + \varphi_1\, y_{t-1} + \ldots + \varphi_p\, y_{t-p} - \theta_1 e_{t-1} - \ldots - \theta_q e_{t-q}$$

- Generic model for making predictions for time-series data.
- Trip Estimation application we saw before uses ARIMA to make the prediction. To date, it is one of the most accurate approaches for this type of prediction.
- ARIMA predictions make by construction sliding temporal queries to the underlying data.

# Temporal Caching

- Claim : Traditional cache eviction techniques (LRU,LFU) are unable to capture the nature of Sliding Temporal Queries.
- Question : Can we devise better cache eviction policies for Sliding Temporal Queries?

# LFU – Counting References

28 Jan 2017
6-7AM

RC:0

21 Jan 2017
6-7AM

RC:0

21 Jan 2017
7-8AM

RC:0

21 Jan 2017
8-9AM

# LFU – Counting References

28 Jan 2017
6-7AM

| | | |
|---|---|---|
| RC:0 | RC:0 | RC:0 |
| 21 Jan 2017<br>6-7AM | 21 Jan 2017<br>7-8AM | 21 Jan 2017<br>8-9AM |

# LFU – Counting References

28 Jan 2017
6-7AM

| RC:1 | RC:0 | RC:0 |
|---|---|---|
| 21 Jan 2017<br>6-7AM | 21 Jan 2017<br>7-8AM | 21 Jan 2017<br>8-9AM |

# LFU – Counting References

28 Jan 2017
6-7AM

RC:1

21 Jan 2017
6-7AM

RC:0

21 Jan 2017
7-8AM

RC:0

21 Jan 2017
8-9AM

# LFU – Counting References

28 Jan 2017
7-8AM

| RC:1 | RC:0 | RC:0 |
|---|---|---|
| 21 Jan 2017<br>6-7AM | 21 Jan 2017<br>7-8AM | 21 Jan 2017<br>8-9AM |

# LFU – Counting References

28 Jan 2017
7-8AM

| RC:1 | RC:0 | RC:0 |
| --- | --- | --- |
| 21 Jan 2017<br>6-7AM | 21 Jan 2017<br>7-8AM | 21 Jan 2017<br>8-9AM |

# LFU – Counting References

28 Jan 2017
7-8AM

RC:1

21 Jan 2017
6-7AM

RC:1

21 Jan 2017
7-8AM

RC:0

21 Jan 2017
8-9AM

# LFU – Counting References

28 Jan 2017
8-9AM

RC:1

21 Jan 2017
6-7AM

RC:1

21 Jan 2017
7-8AM

RC:0

21 Jan 2017
8-9AM

# LFU – Counting References

28 Jan 2017
8-9AM

RC:1 | RC:1 | RC:0

21 Jan 2017 6-7AM | 21 Jan 2017 7-8AM | 21 Jan 2017 8-9AM

# LFU – Counting References

28 Jan 2017
8-9AM

RC:1

21 Jan 2017
6-7AM

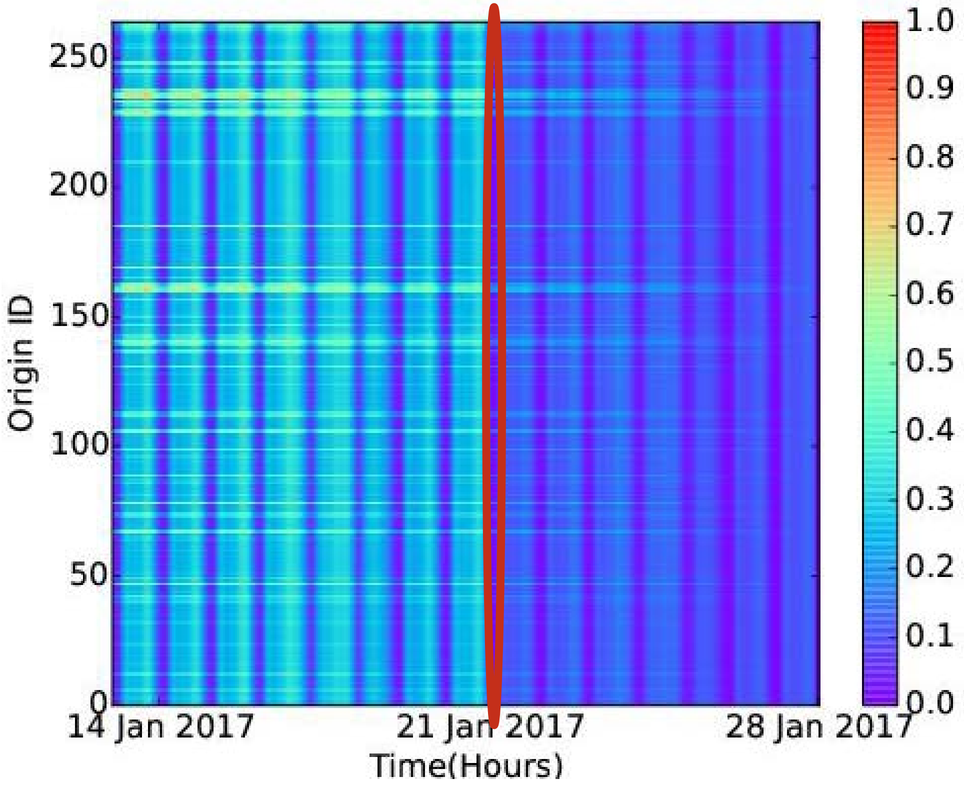
RC:1

21 Jan 2017
7-8AM

RC:1

21 Jan 2017
8-9AM

# LFU – Sliding Temporal Queries




We calculate:

$$rr(rid, wt) = \frac{rc(rid, wt)}{nrQueries}$$

We normalize:

$$nrr(rid, wt) = \frac{rr(rid, wt)}{max_{rid', wt'}\{rr(rid', wt')\}}$$

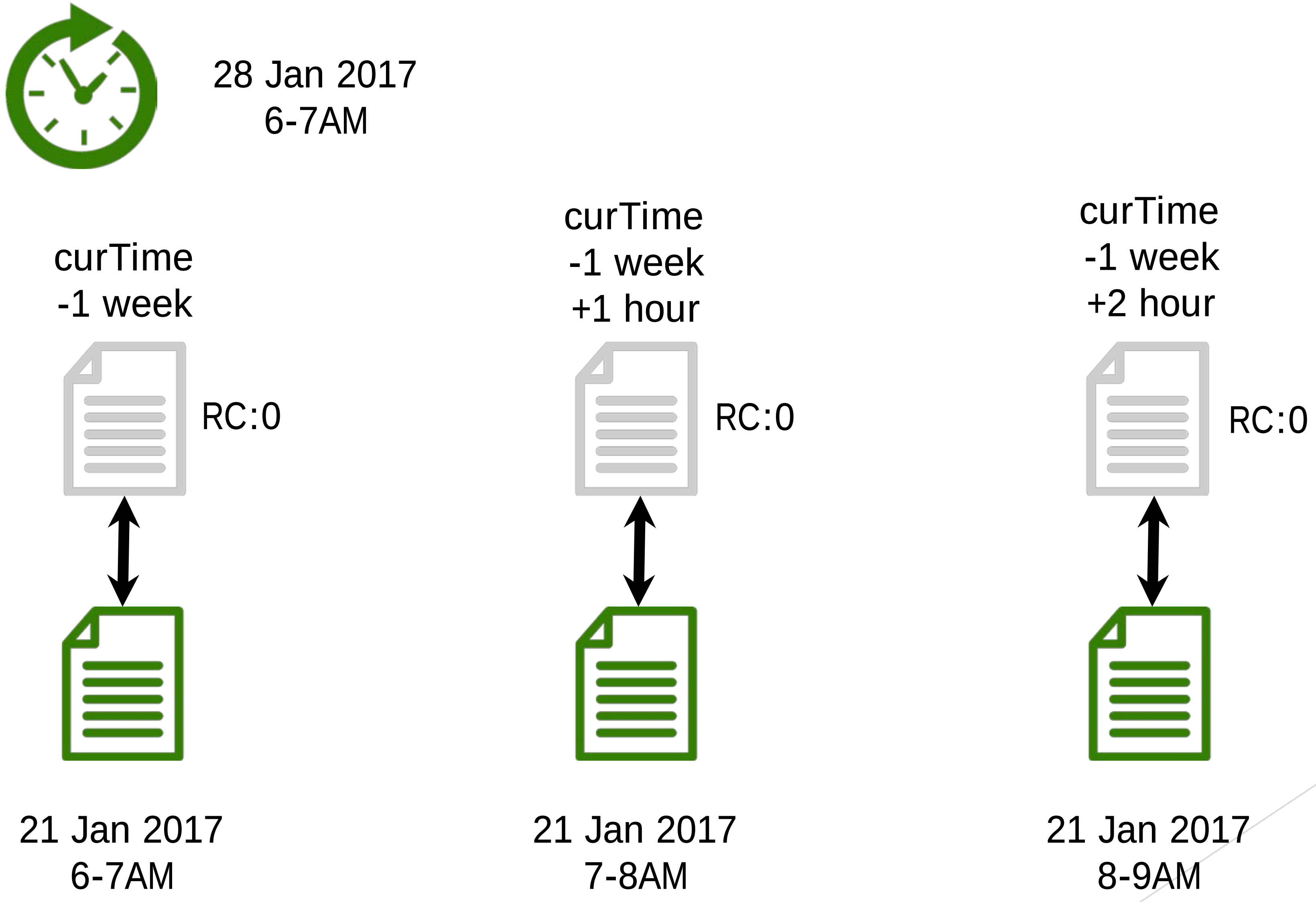
# Count References in Relative Timeline

- Pin current time as a constant time point (no shift).
- Sliding temporal queries will access data that is identified by constant time now. For our previous example we would access data at time:

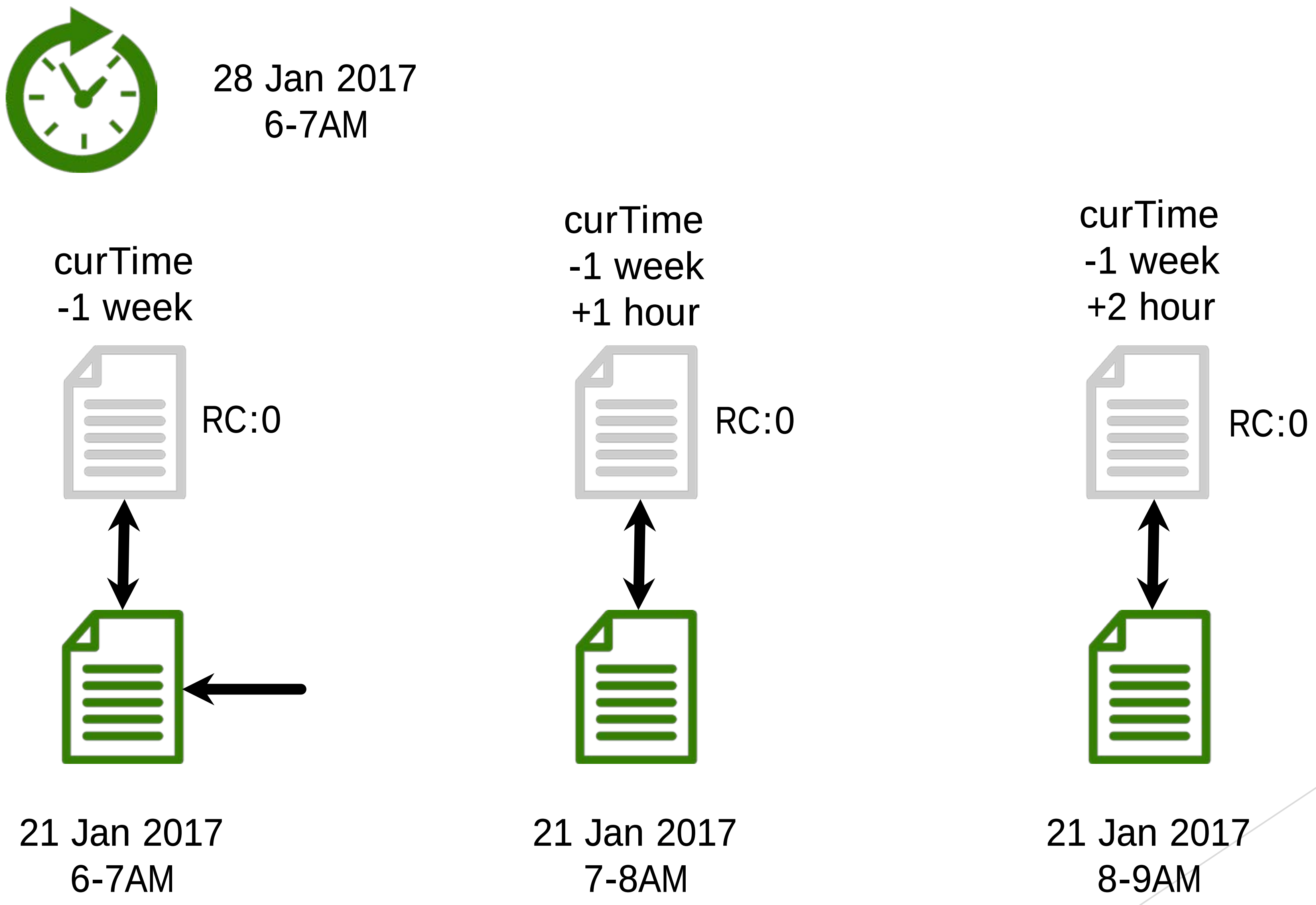
  Current Time – 1 Week

  no matter when we make the query.
- Effectively, sliding temporal queries look like fixed queries for the relative timeline now.

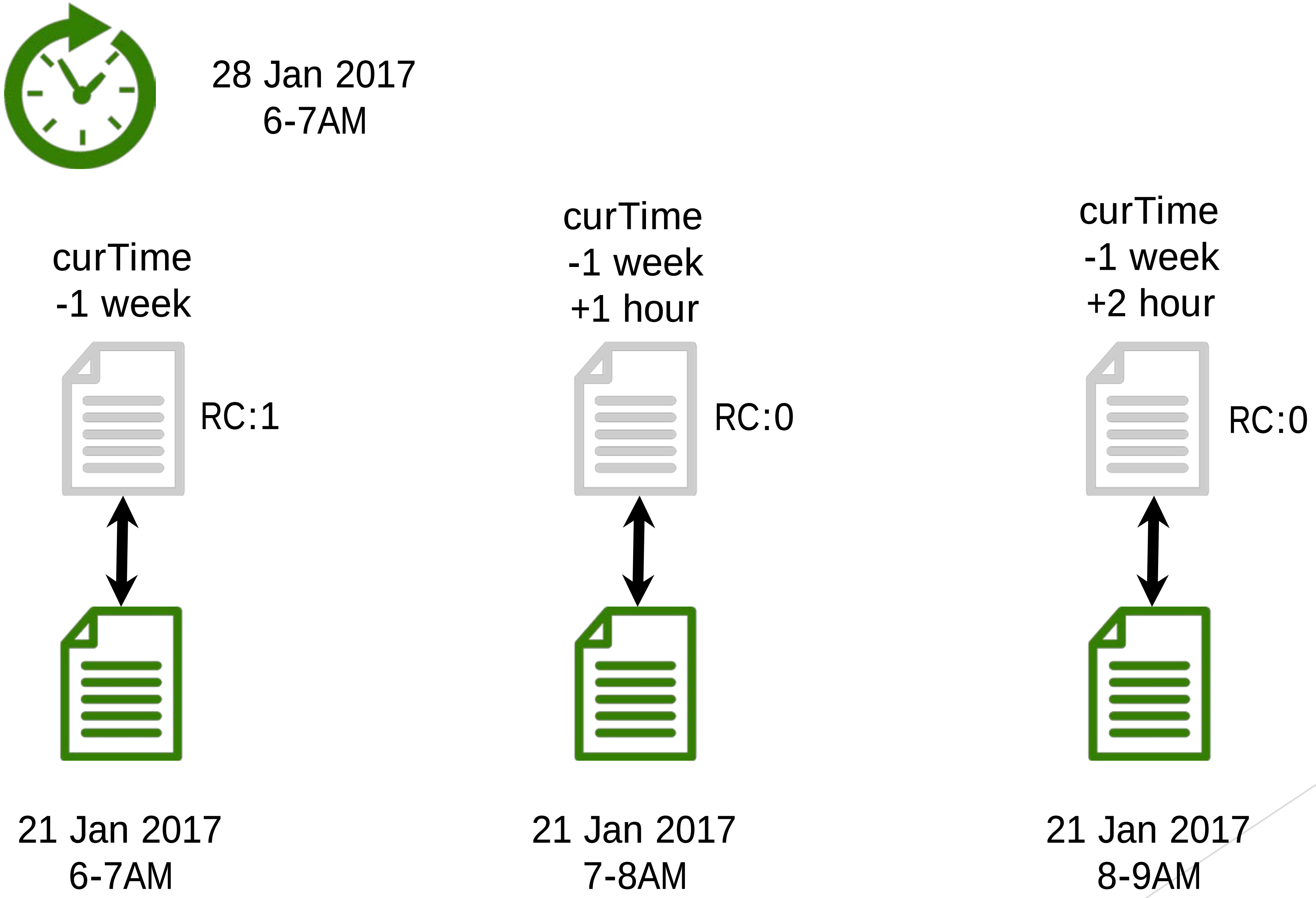
# Counting References on Relative Timeline



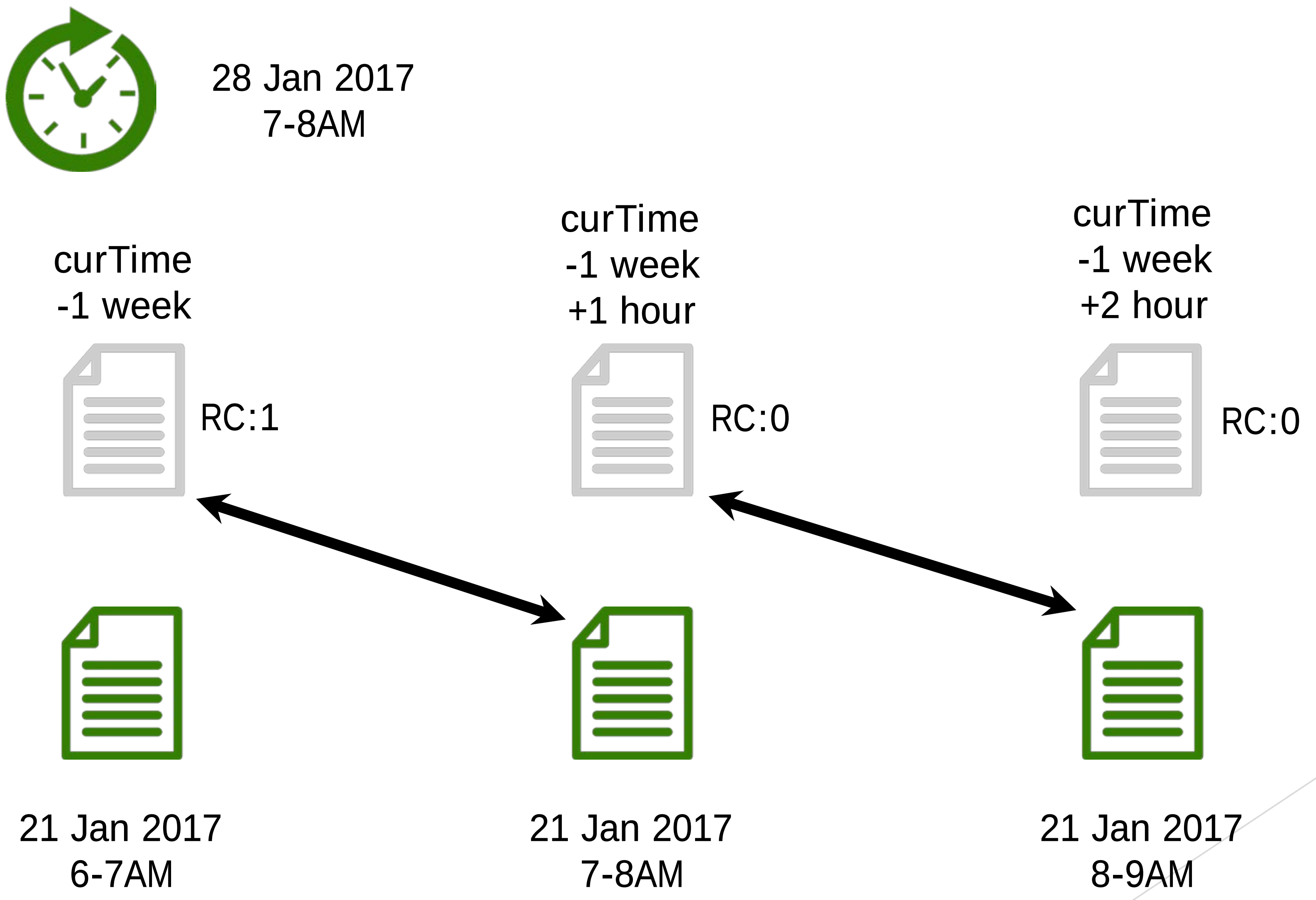
# Counting References on Relative Timeline



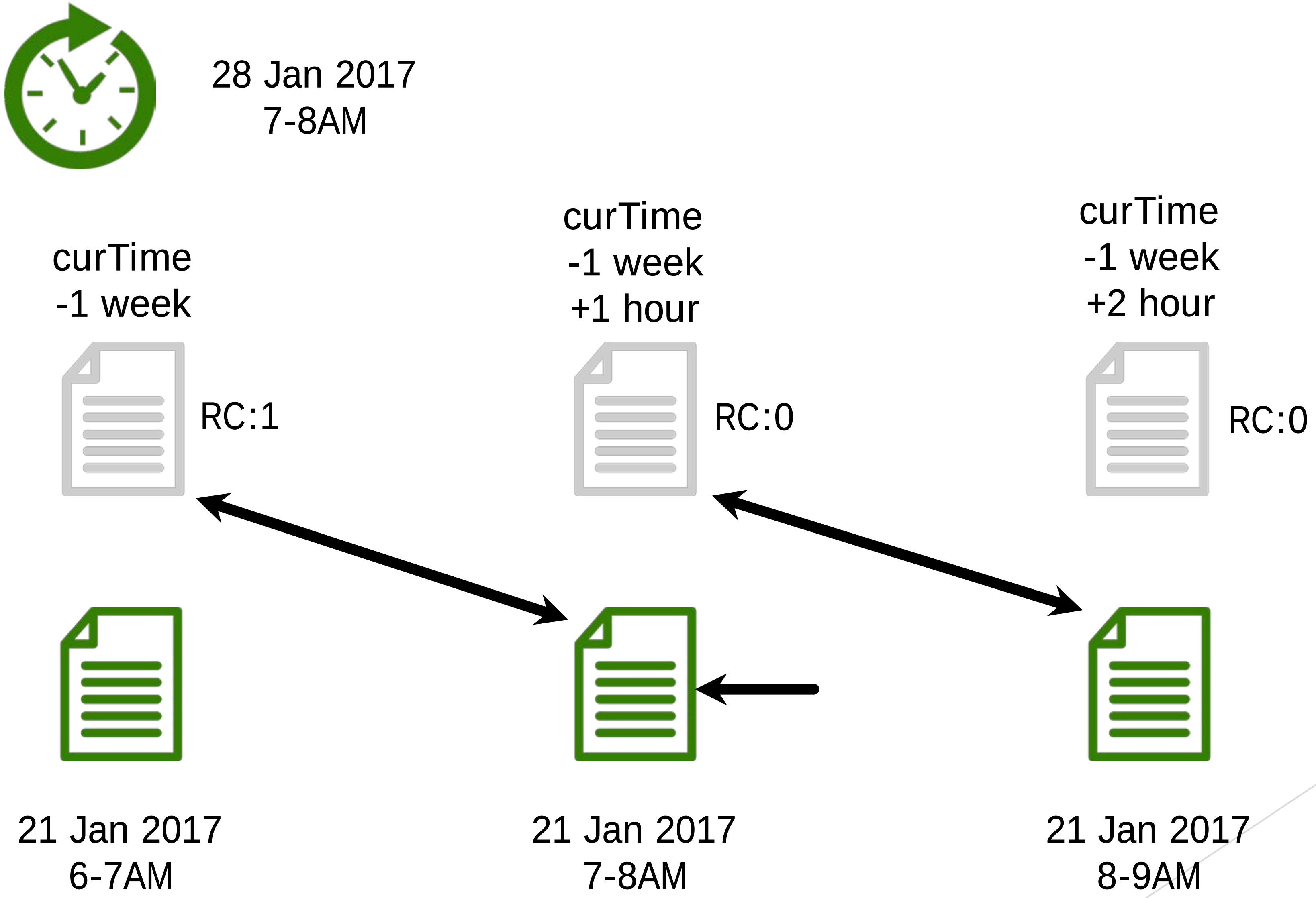
# Counting References on Relative Timeline



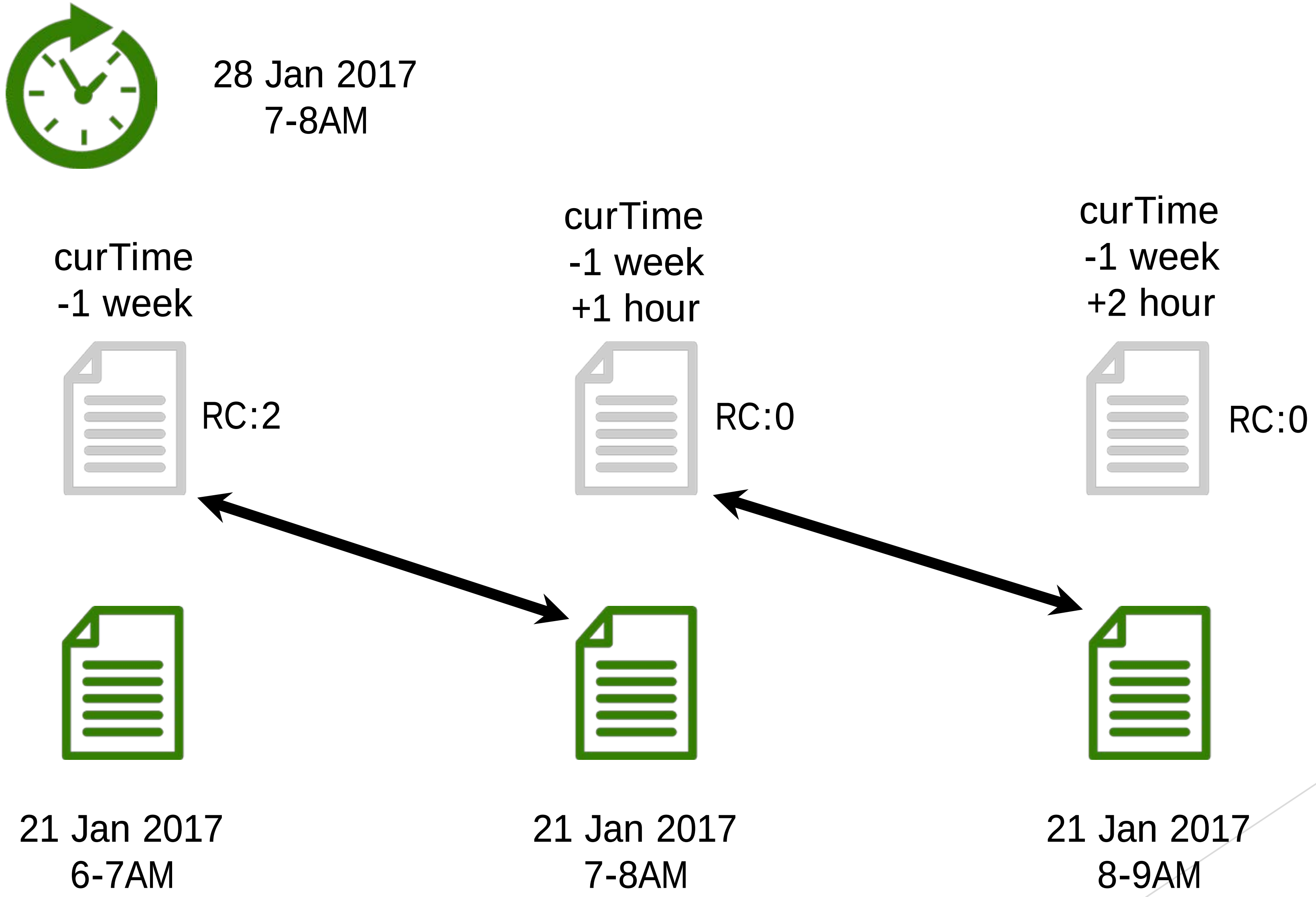
# Counting References on Relative Timeline



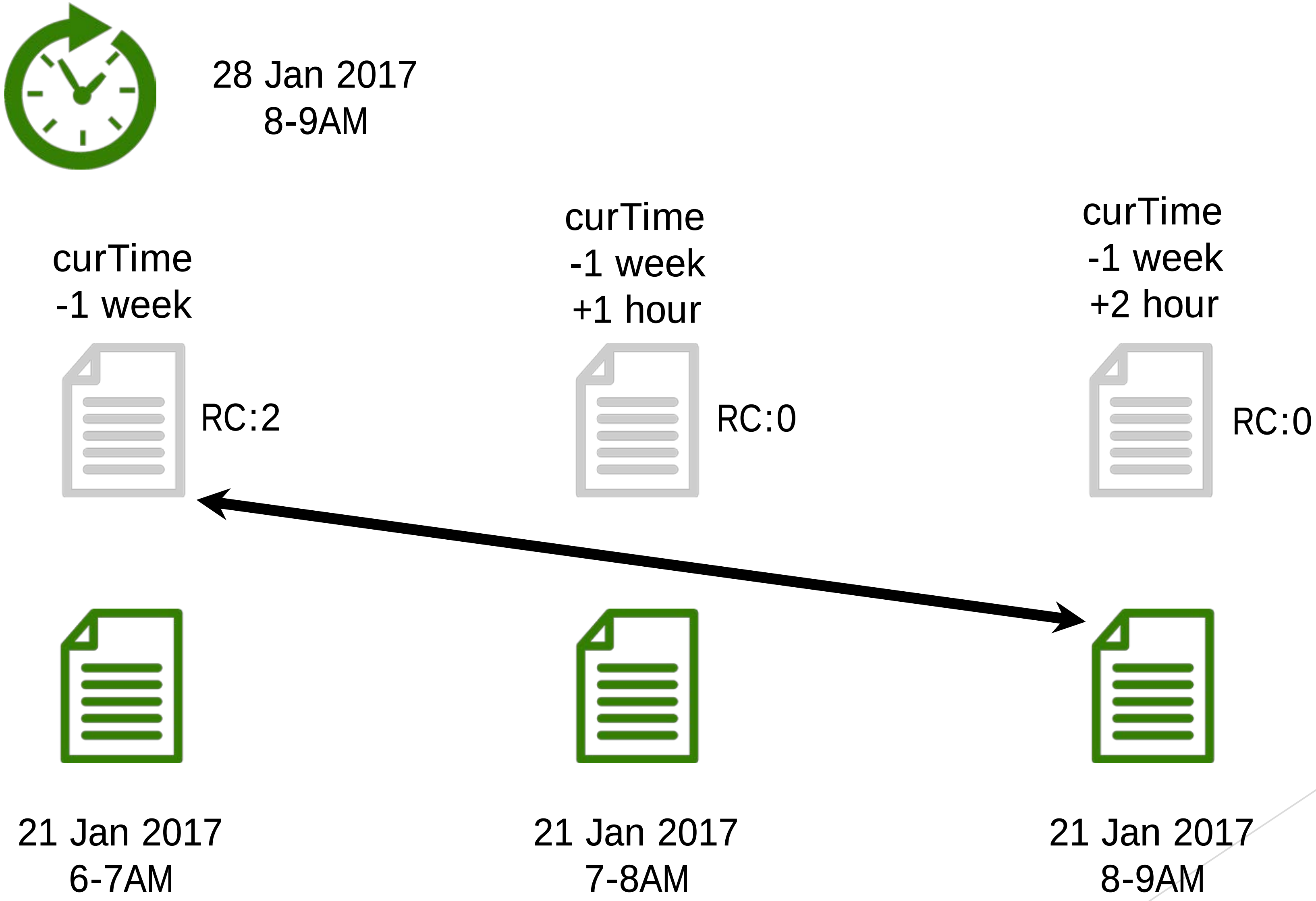
# Counting References on Relative Timeline



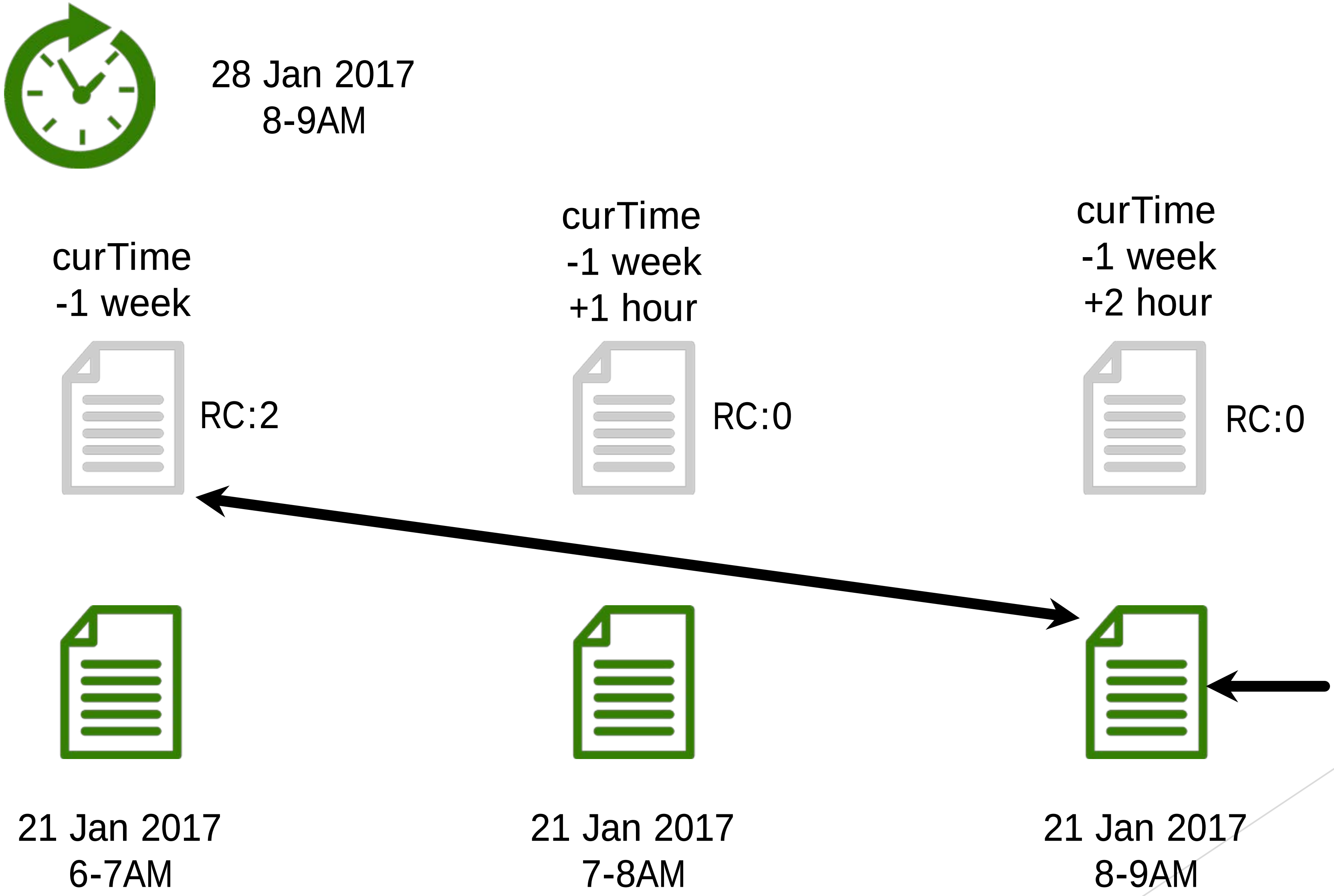
# Counting References on Relative Timeline



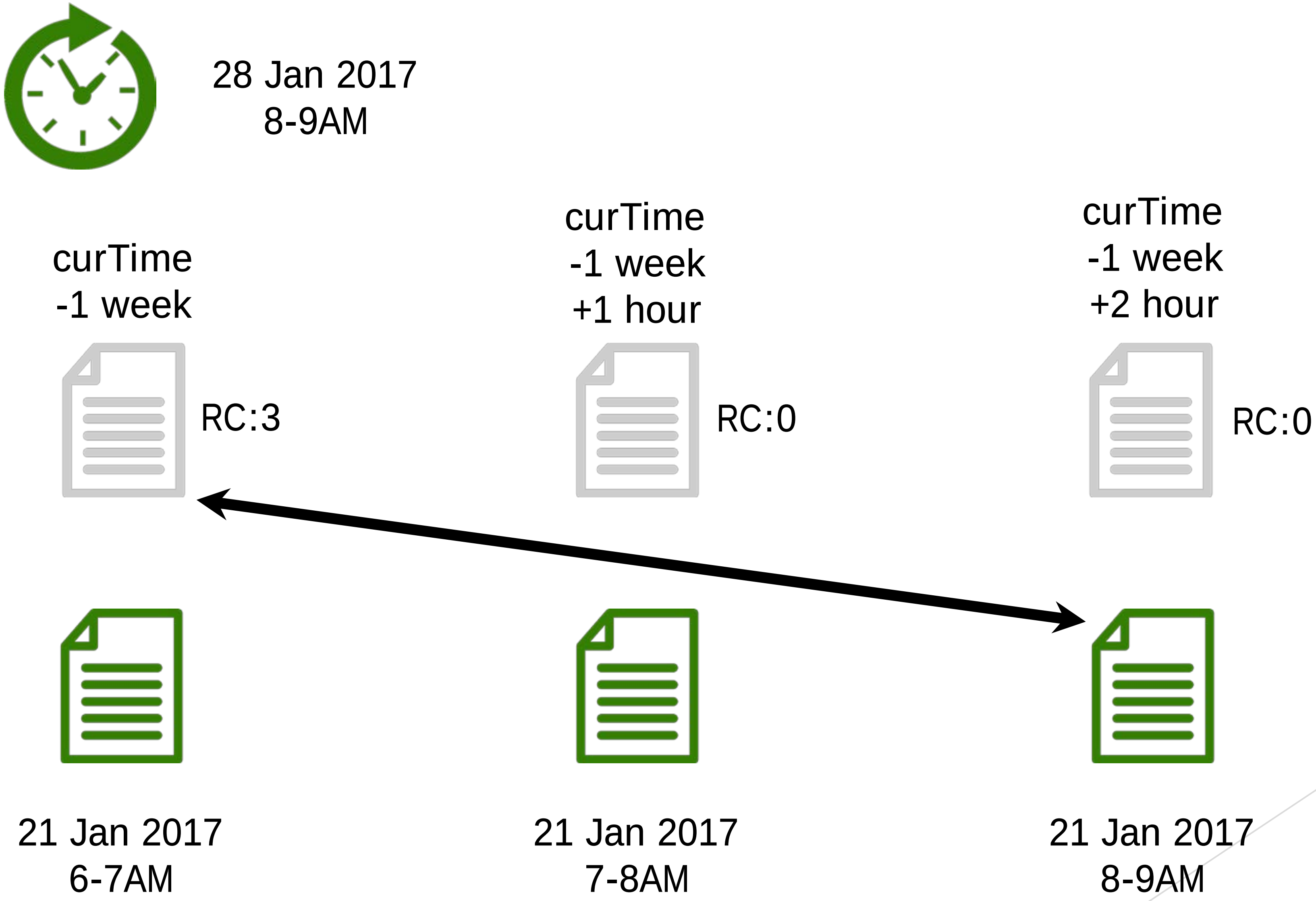
# Counting References on Relative Timeline



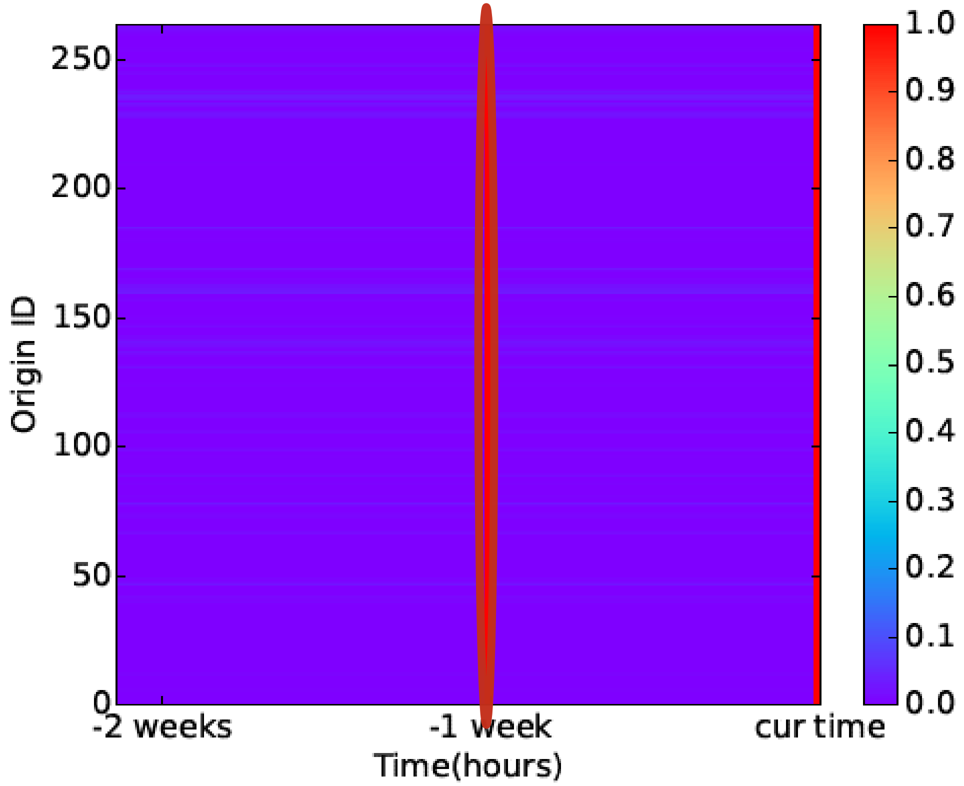
# Counting References on Relative Timeline



# Counting References on Relative Timeline



# LFU on Relative Timeline – Sliding Temporal Queries



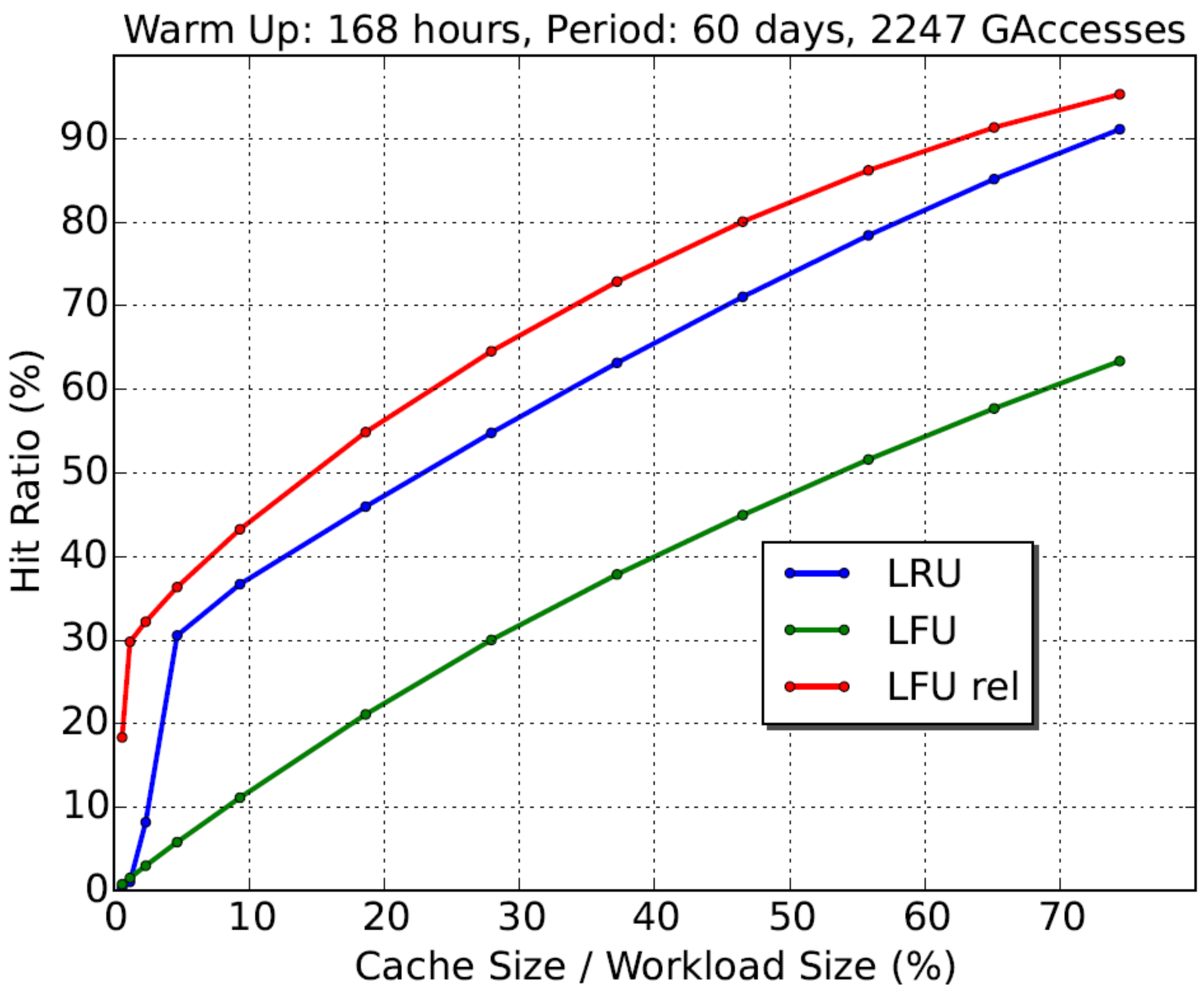

We calculate:

$$rr(rid, wt) = \frac{rc(rid, wt)}{nrQueries}$$

We normalize:

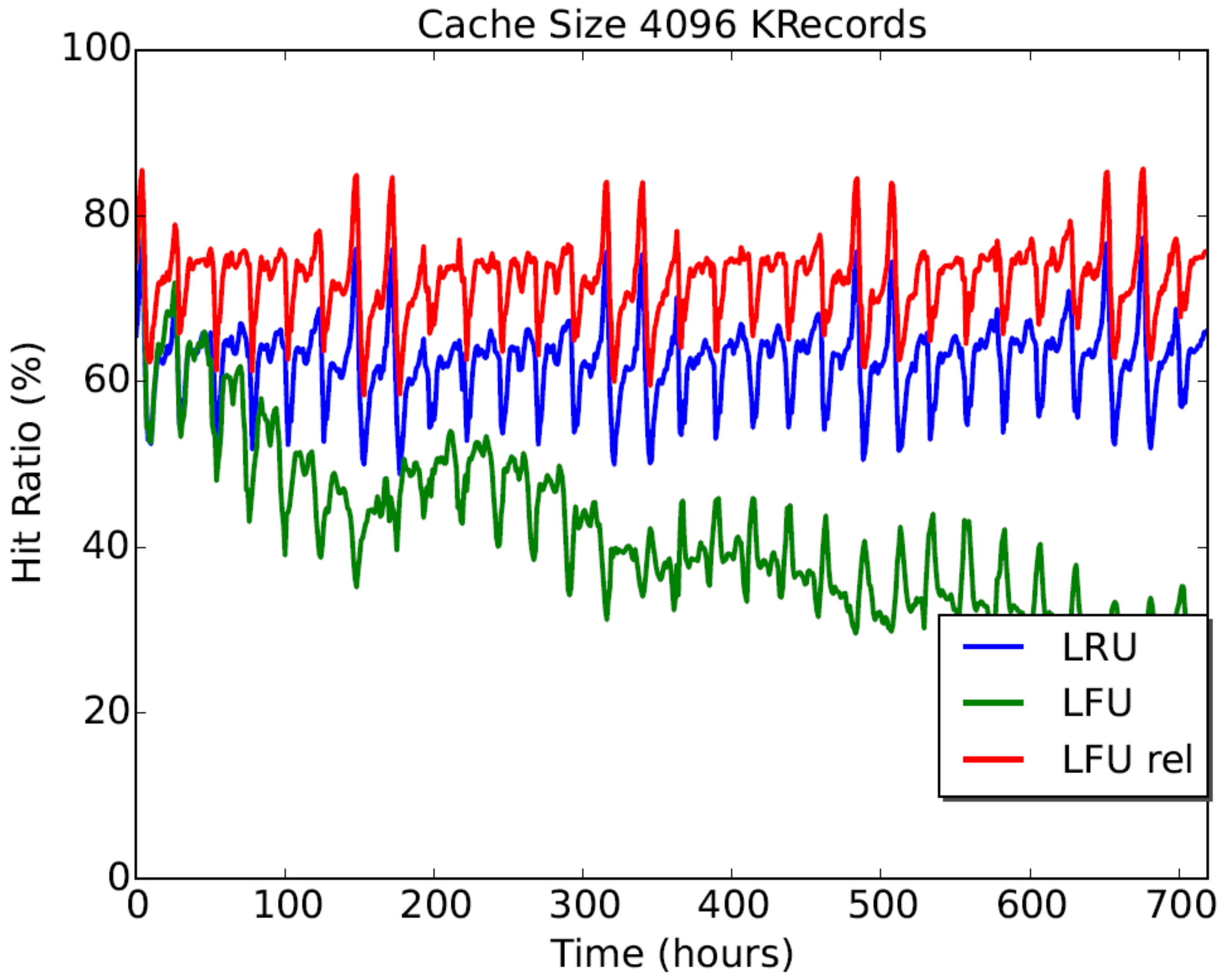
$$nrr(rid, wt) = \frac{rr(rid, wt)}{max_{rid', wt'}\{rr(rid', wt')\}}$$

# Evaluation



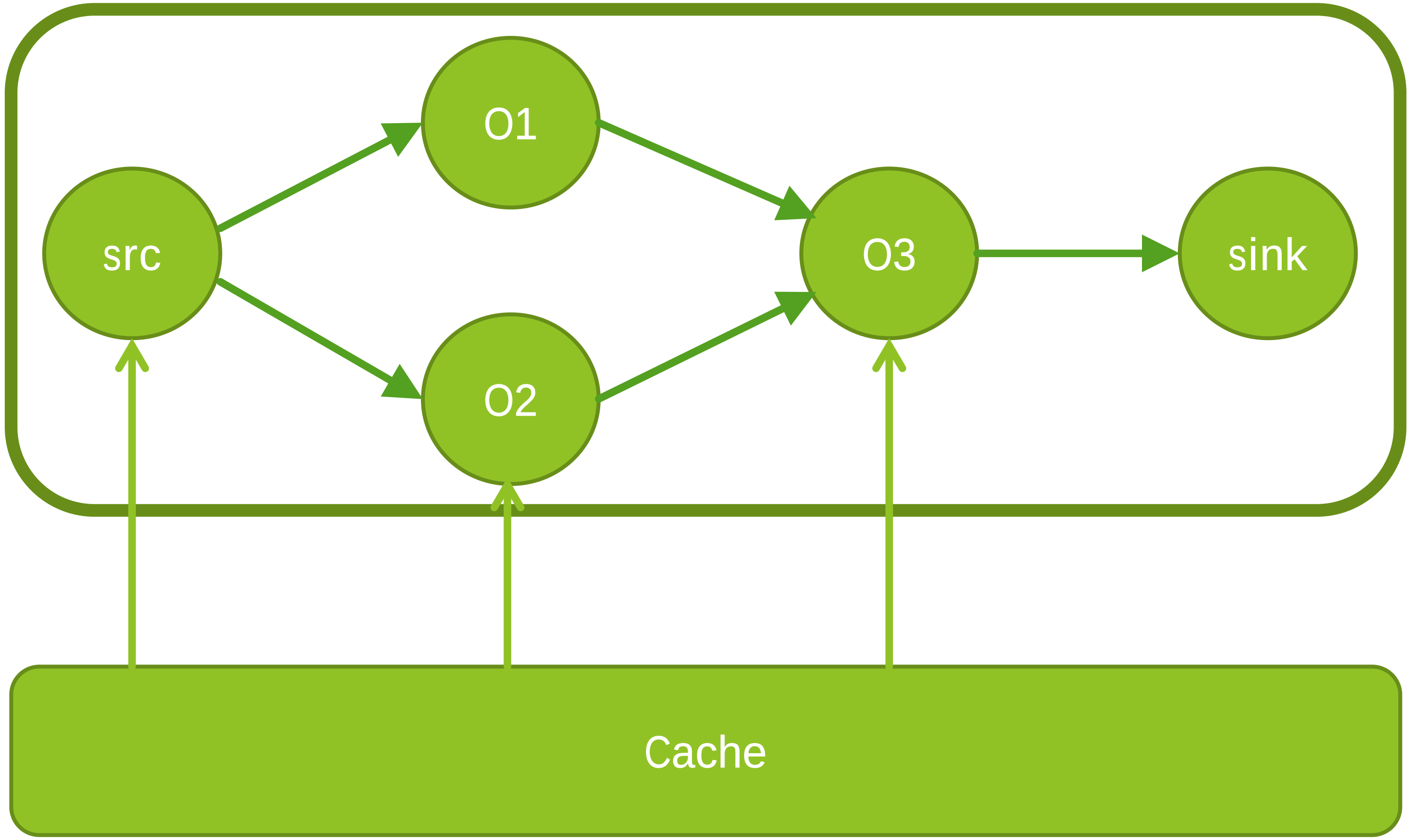
Warm Up: 168 hours, Period: 60 days, 2247 GAccesses
Hit Ratio (%)
0
10
20
30
40
50
60
70
80
90
0
10
20
30
40
50
60
70
Cache Size / Workload Size (%)
LRU
LFU
LFU rel

# Evaluation



Cache Size 4096 KRecords
100
80
60
40
20
0
Hit Ratio (%)
0
100
200
300
400
500
600
700
Time (hours)
LRU
LFU
LFU rel

# Future Work-Dataflow Cache for Time-Series Data.



# Questions