Concurrent programming: From theory to practice

Concurrent Algorithms 2020

Vasileios Trigonakis
Principal Member of Technical Staff
Oracle Labs, Zurich

From theory to practice

Theoretical (design)	Practical (design)	Practical (implementation)

From theory to practice

Theoretical (design)	Practical (design)	Practical (implementation)

- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs

Design (pseudo-code)

From theory to practice

Theoretical (design)	Practical (design)	Practical (implementation)

- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs

Design (pseudo-code)

- System models
  - shared memory
  - message passing
- **Finite memory**
- Practicality issues
  - re-usable objects
- **Performance**

Design (pseudo-code, prototype)

From theory to practice

Theoretical (design)

- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs

Design (pseudo-code)

Practical (design)

- System models
  - shared memory
  - message passing
- **Finite memory**
- Practicality issues
  - re-usable objects
- **Performance**

Design (pseudo-code, prototype)

Practical (implementation)

- **Hardware**
- Which atomic ops
- Memory consistency
- Cache coherence
- Locality
- **Performance**
- **Scalability**

Implementation (code)

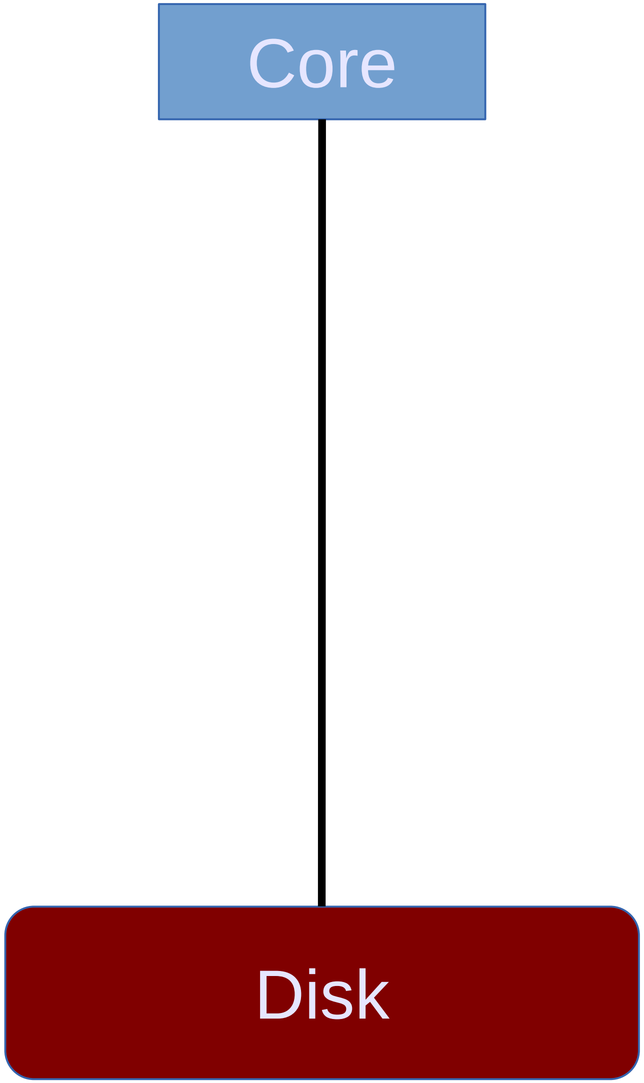
Outline

- CPU caches
- Cache coherence
- Placement of data
- Graph processing: Concurrent data structures

Outline

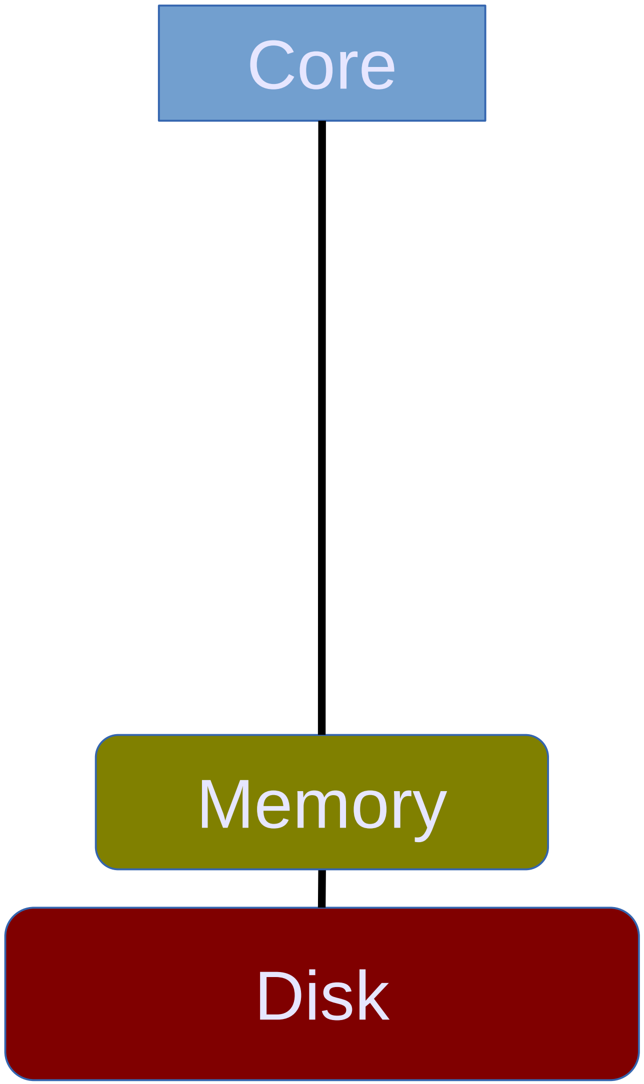
- **CPU caches**
- Cache coherence
- Placement of data
- Graph processing: Concurrent data structures

Why do we use caching?



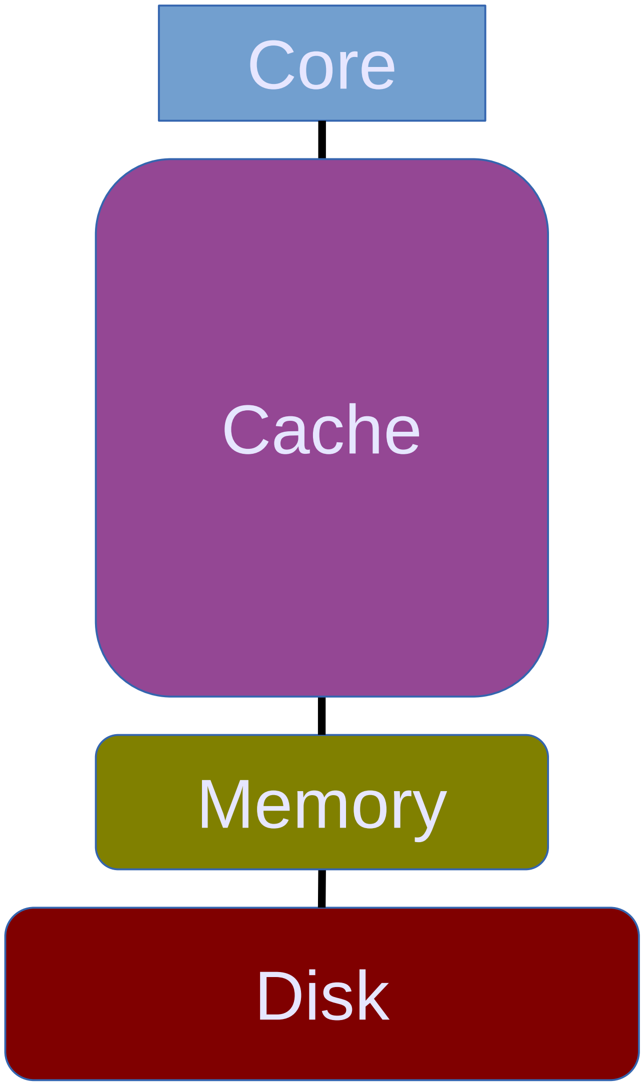
- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms

Why do we use caching?



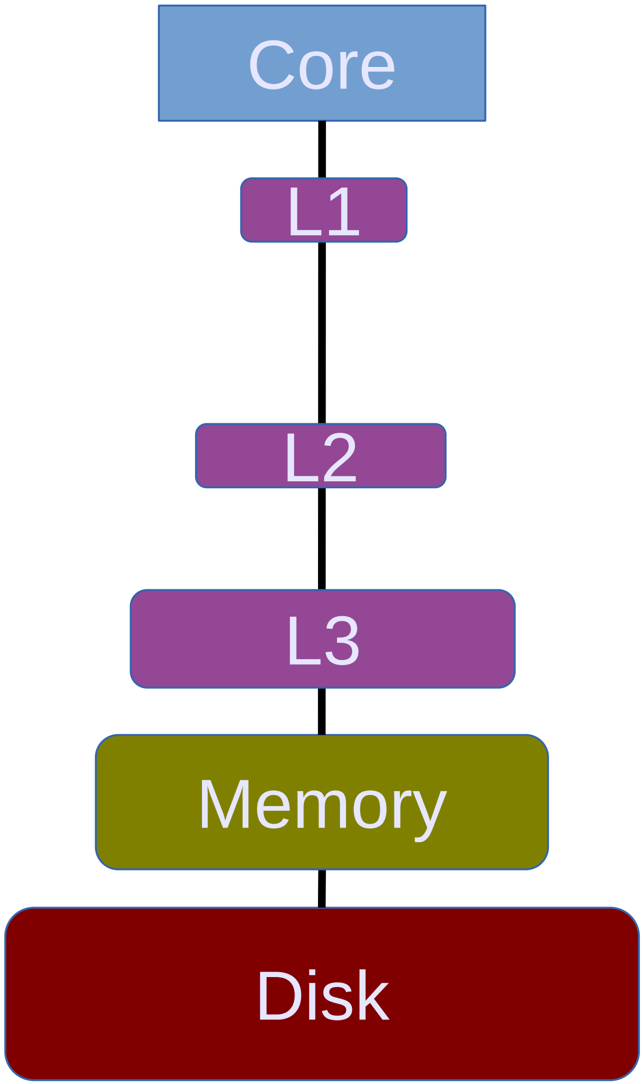
- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns

Why do we use caching?



- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns
- Cache
  - Large = slow
  - Medium = medium
  - Small = fast

Why do we use caching?



- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns
- Cache
  - Core → L3 = ~20ns
  - Core → L2 = ~7ns
  - Core → L1 = ~1ns

Typical server configurations

- **Intel Xeon**
  - 14 cores @ 2.4GHz
  - L1: 32KB
  - L2: 256KB
  - L3: 40MB
  - Memory: 256GB
- **AMD Opteron**
  - 18 cores @ 2.4GHz
  - L1: 64KB
  - L2: 512KB
  - L3: 20MB
  - Memory: 256GB

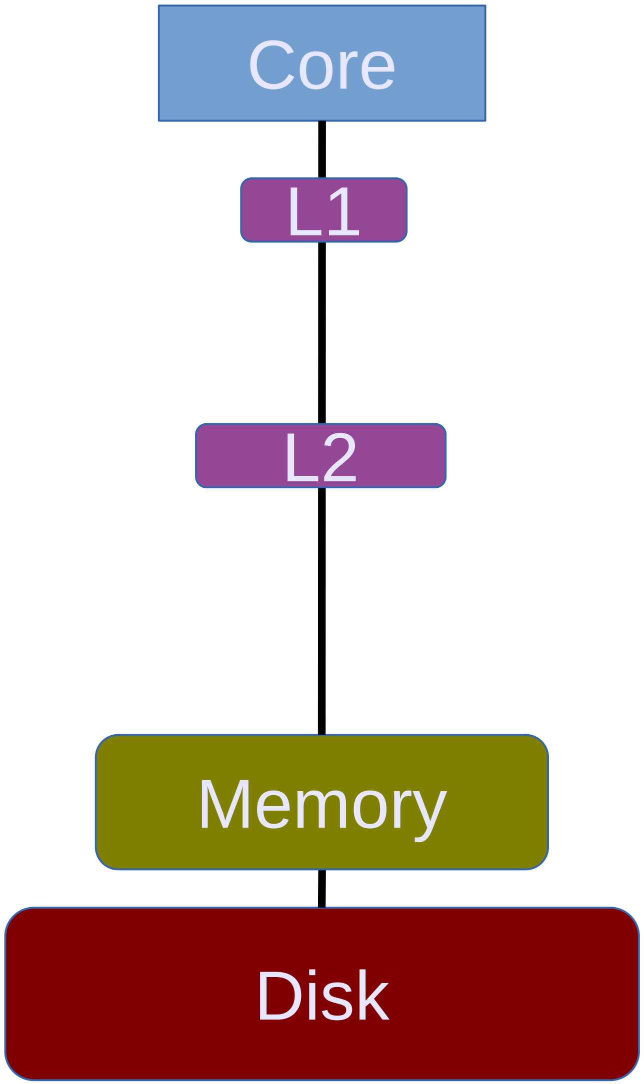
Experiment

Throughput of accessing some memory, depending on the memory size

Outline

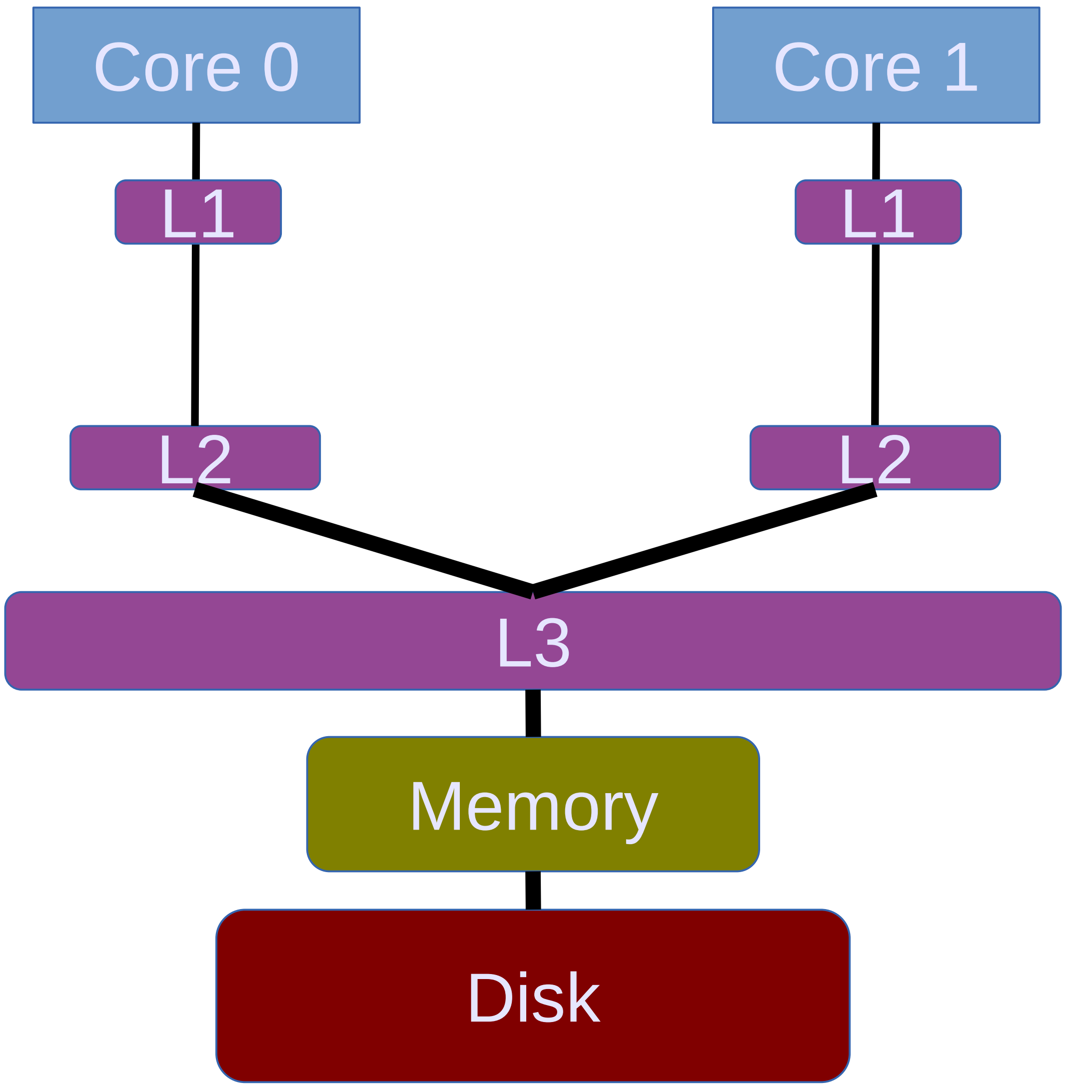
- CPU caches
- **Cache coherence**
- Placement of data
- Graph processing: Concurrent data structures

Until ~2004: single-cores



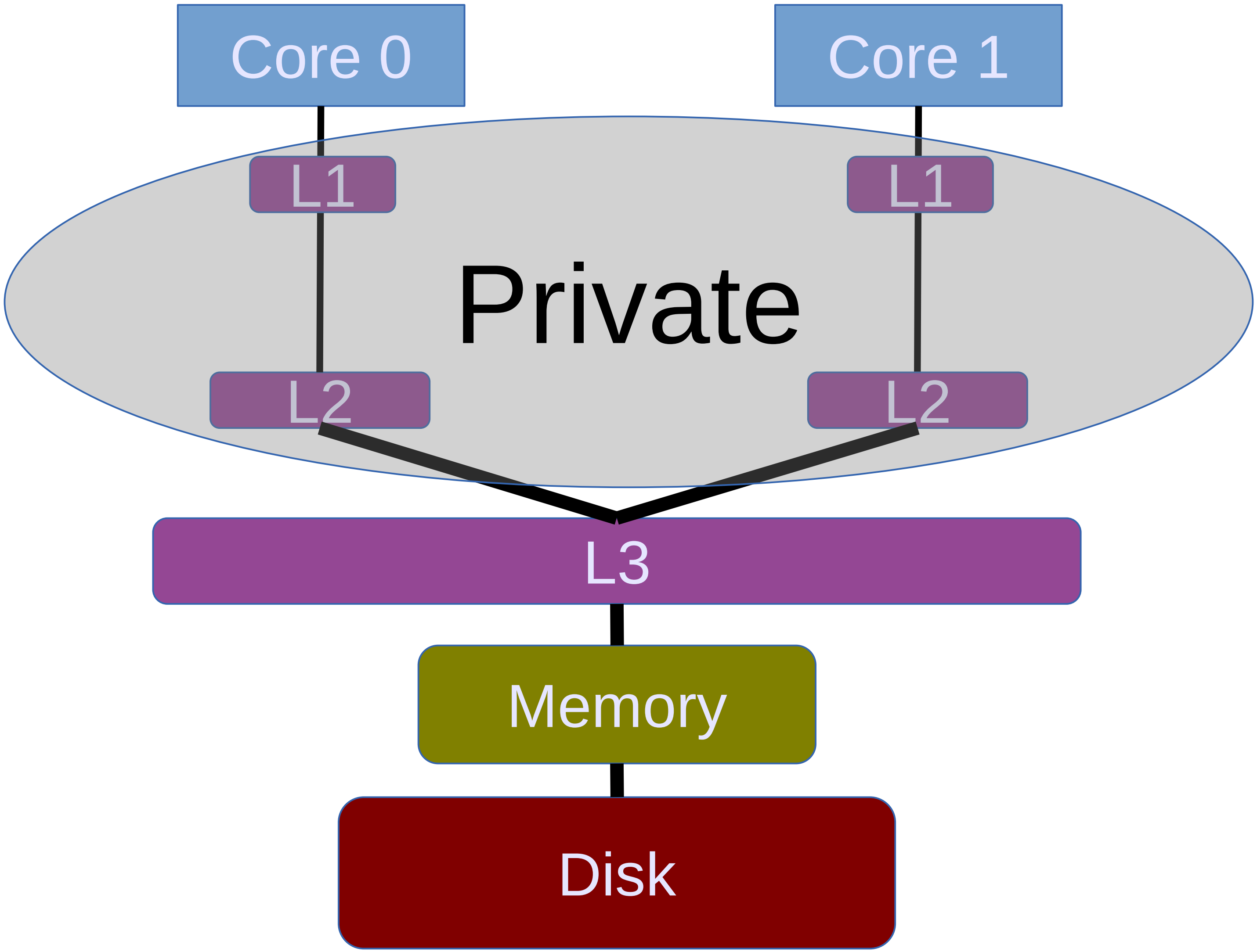
- Core freq: 3+GHz
- Core → Disk
- Core → Memory
- Cache
  - Core → L3
  - Core → L2
  - Core → L1

After ~2004: multi-cores



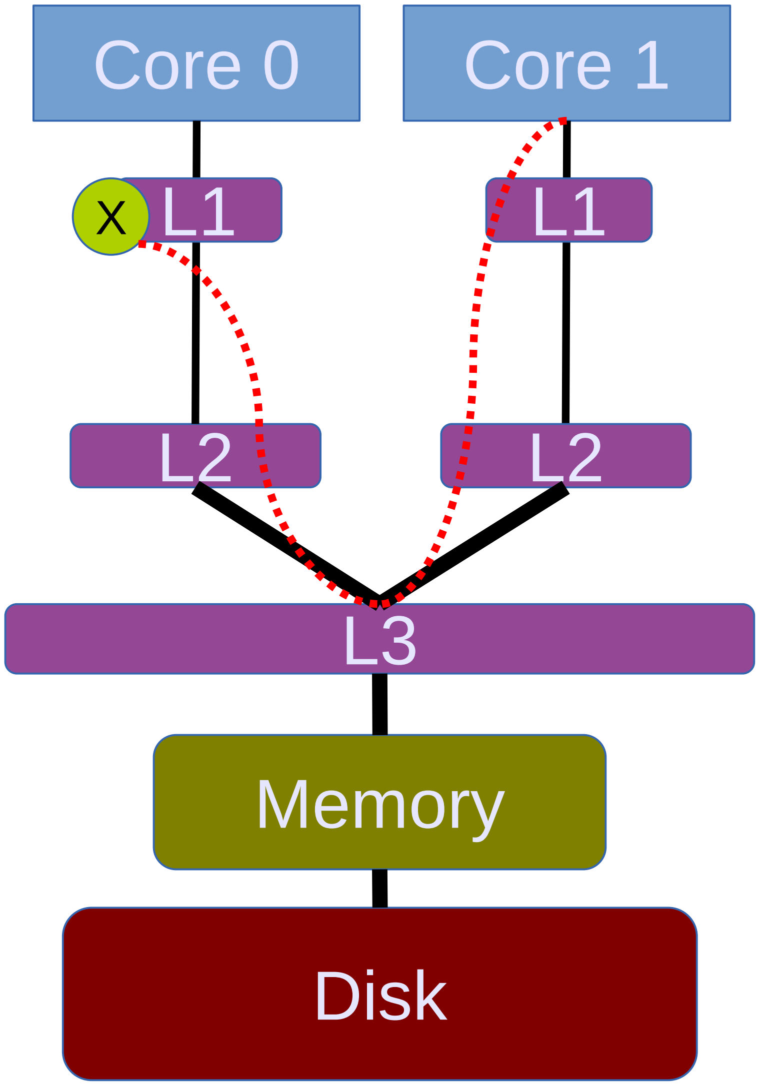
- Core freq: ~2GHz
- Core → Disk
- Core → Memory
- Cache
  - Core → **shared** L3
  - Core → L2
  - Core → L1

Multi-cores with private caches



= multiple copies

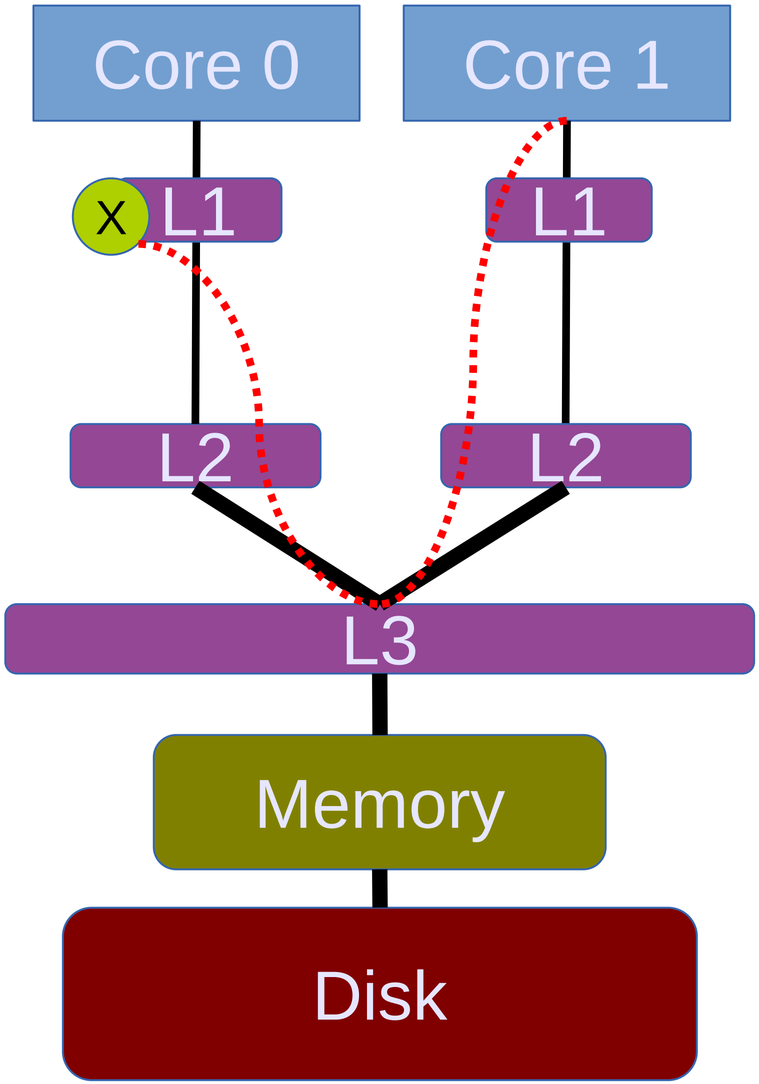
Cache coherence for consistency



Core 0 has **X** and Core 1

- wants to write on **X**
- wants to read **X**
- did Core 0 write or read **X**?

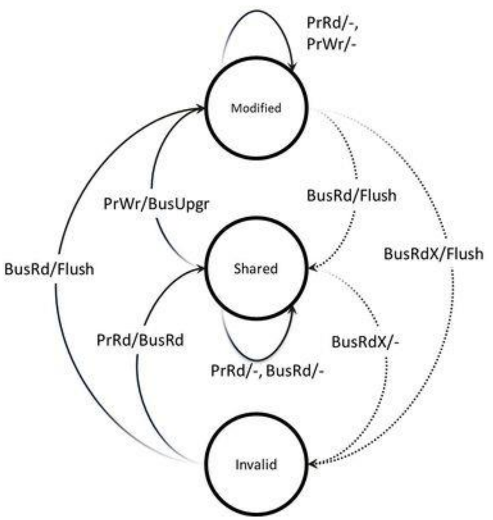
Cache coherence principles



- To perform a write
  - invalidate all readers, or
  - previous writer
- To perform a read
  - find the latest copy

Cache coherence with MESI

- A state diagram
- State (per cache line)
  - **M**odified: the only dirty copy
  - **E**xclusive: the only clean copy
  - **S**hared: a clean copy
  - **I**nvalid: useless data



The ultimate goal for scalability

- Possible states
  - **M**odified: the only dirty copy
  - **E**xclusive: the only clean copy
  - **S**hared: a clean copy
  - **I**nvalid: useless data

- **Which state is our “favorite”?**

The ultimate goal for scalability

- Possible states
  - **M**odified: the only dirty copy
  - **E**xclusive: the only clean copy
  - **S**hared: a clean copy
  - **I**nvalid: useless data

= threads can keep the data close (L1 cache)

= faster

<u>**Experiment**</u>

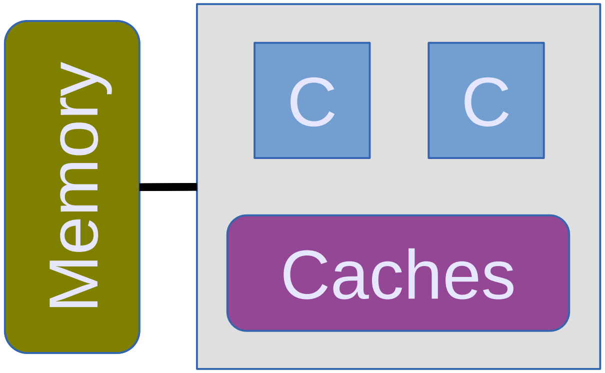
The effects of false sharing

Outline

- CPU caches
- Cache coherence
- **Placement of data**
- Graph processing: Concurrent data structures

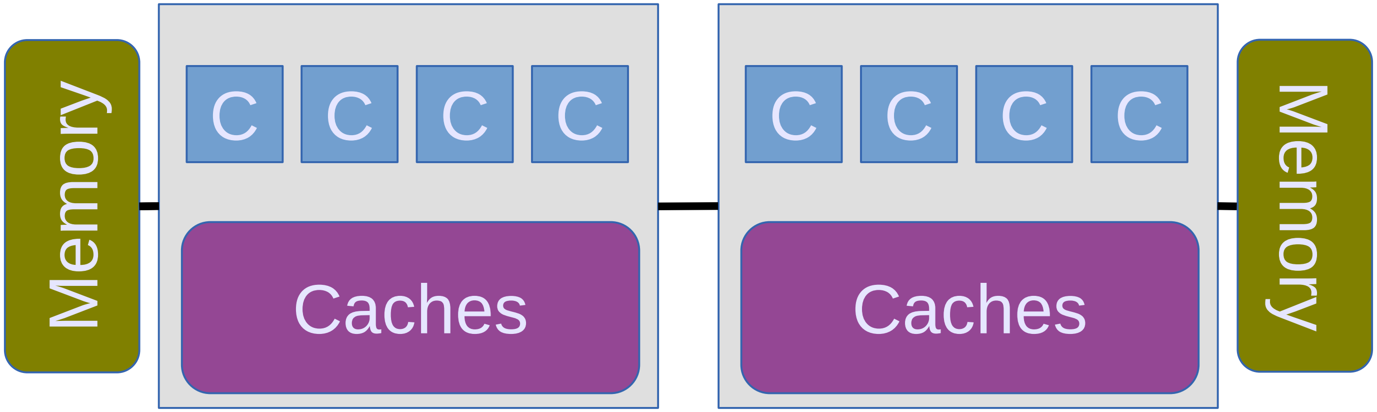
Uniformity vs. non-uniformity

- Typical **desktop** machine



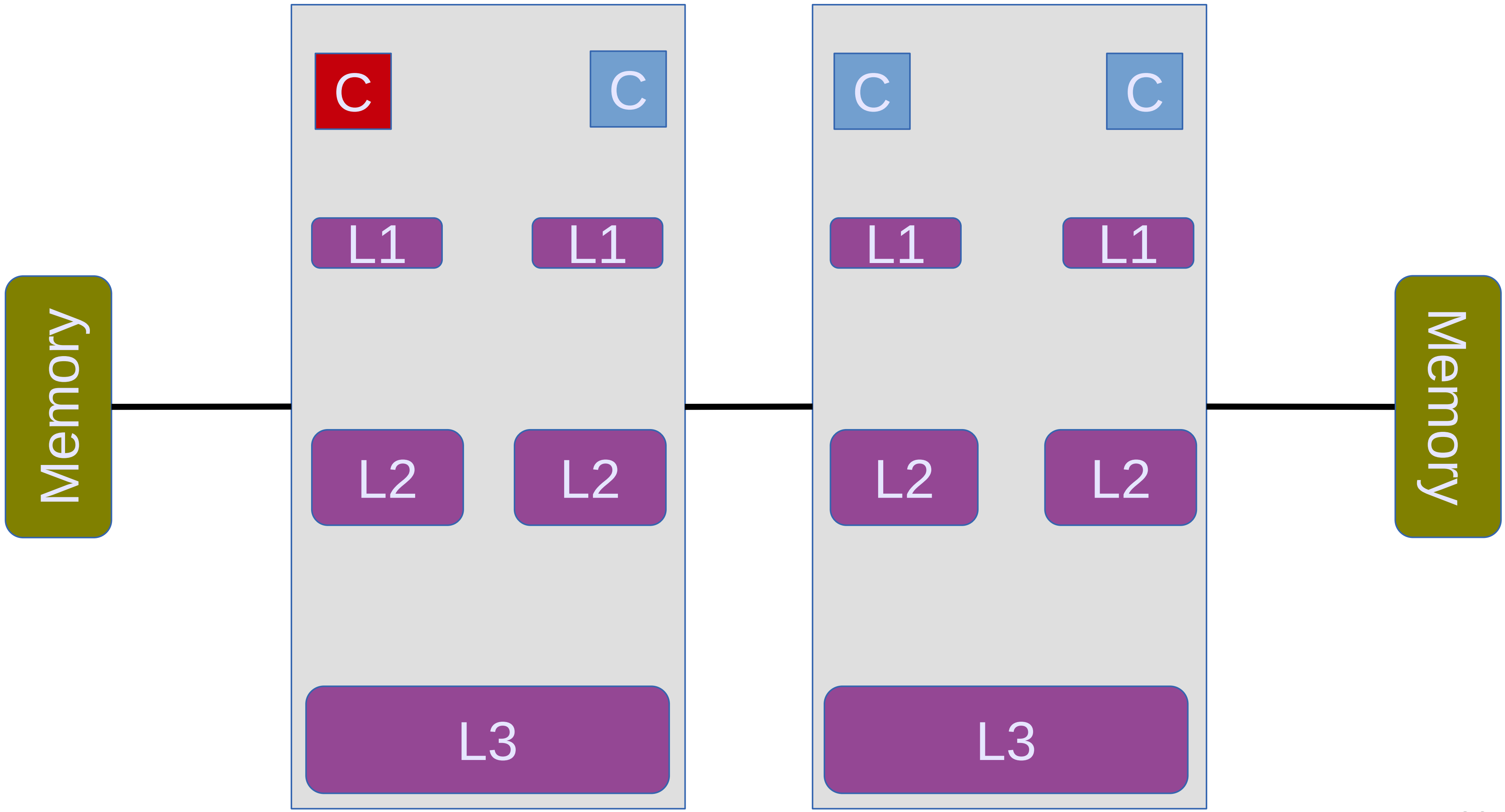
= Uniform

- Typical **server** machine

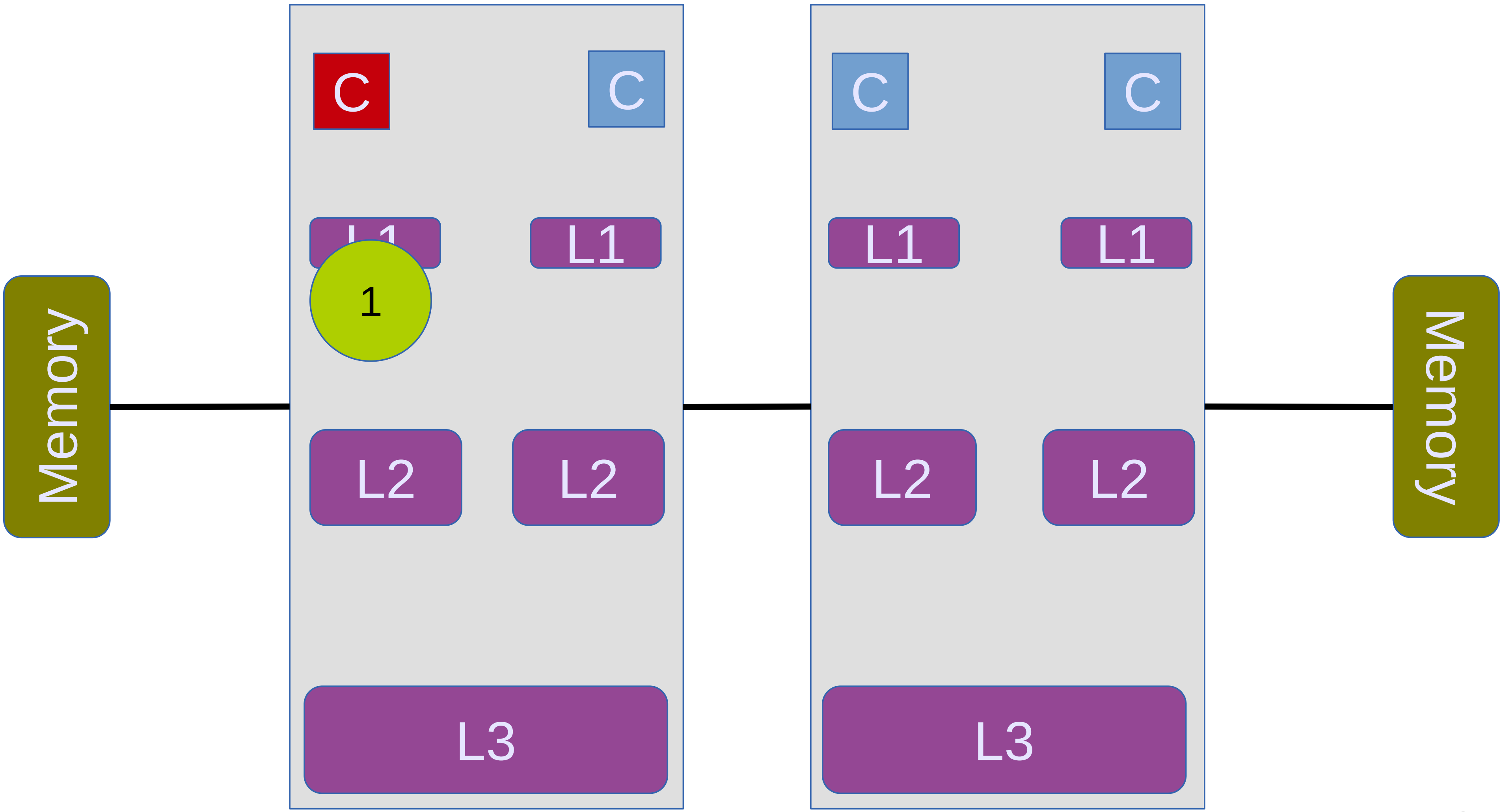


= non-Uniform

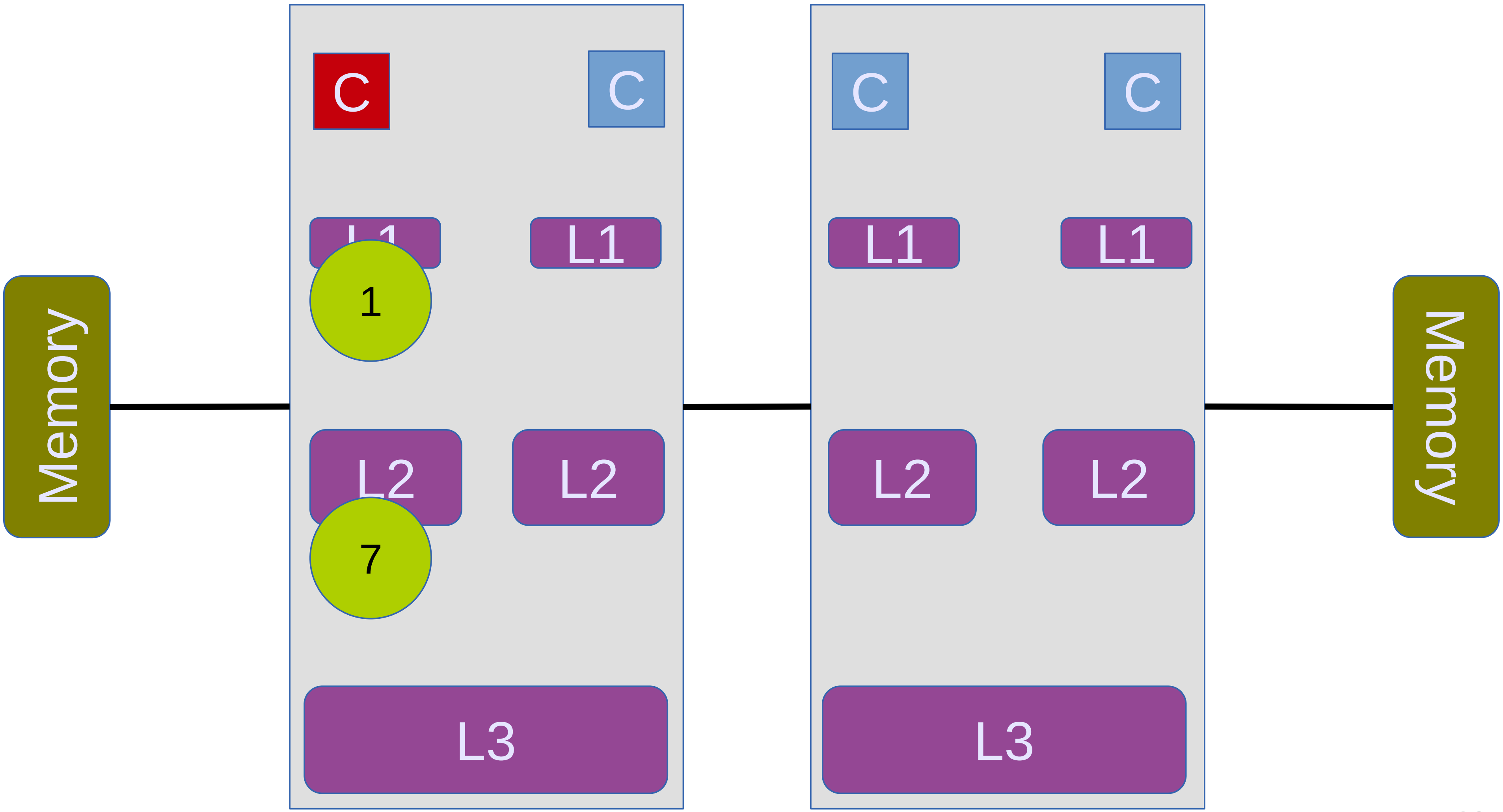
Latency (ns) to access data



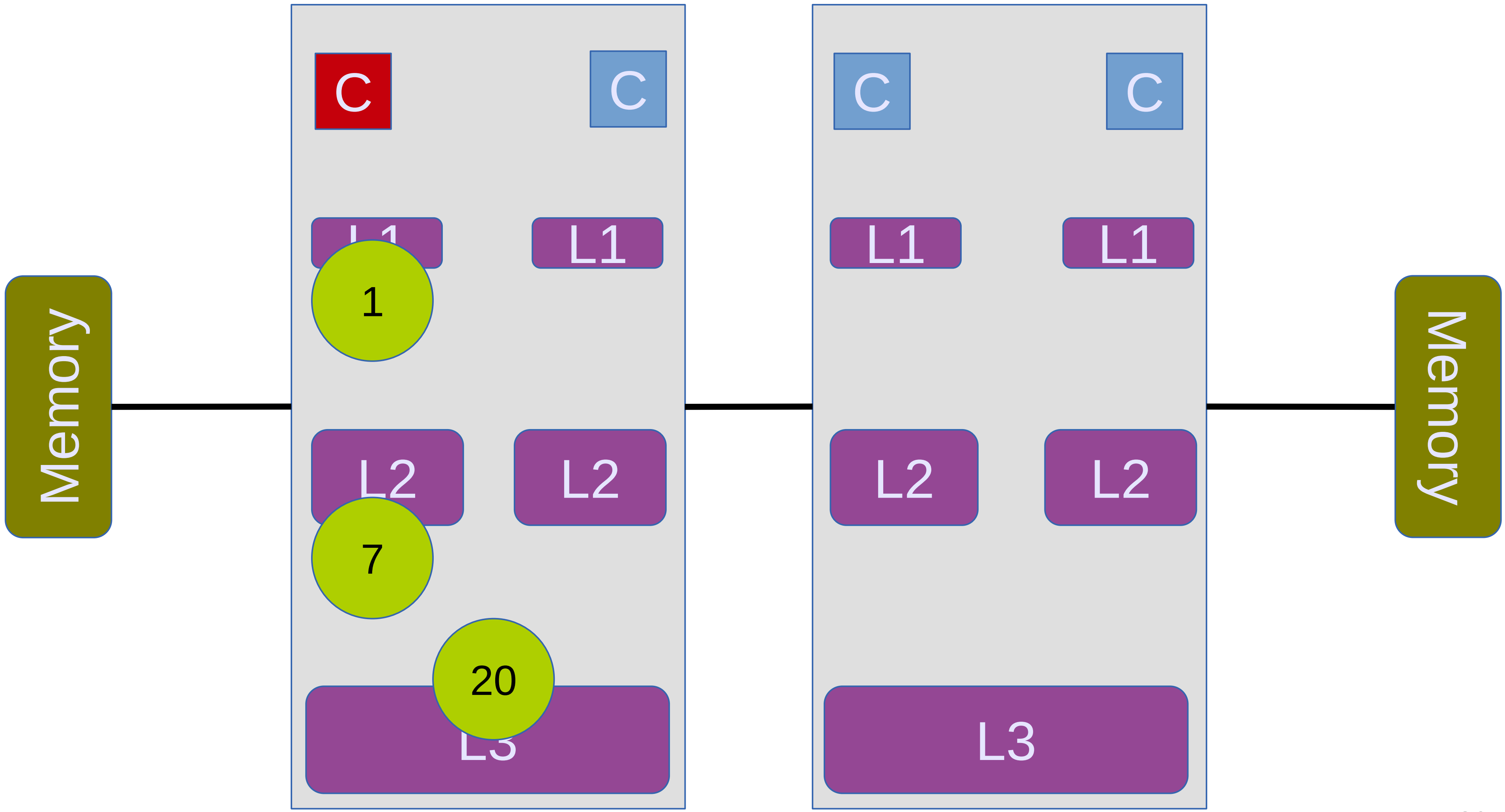
Latency (ns) to access data



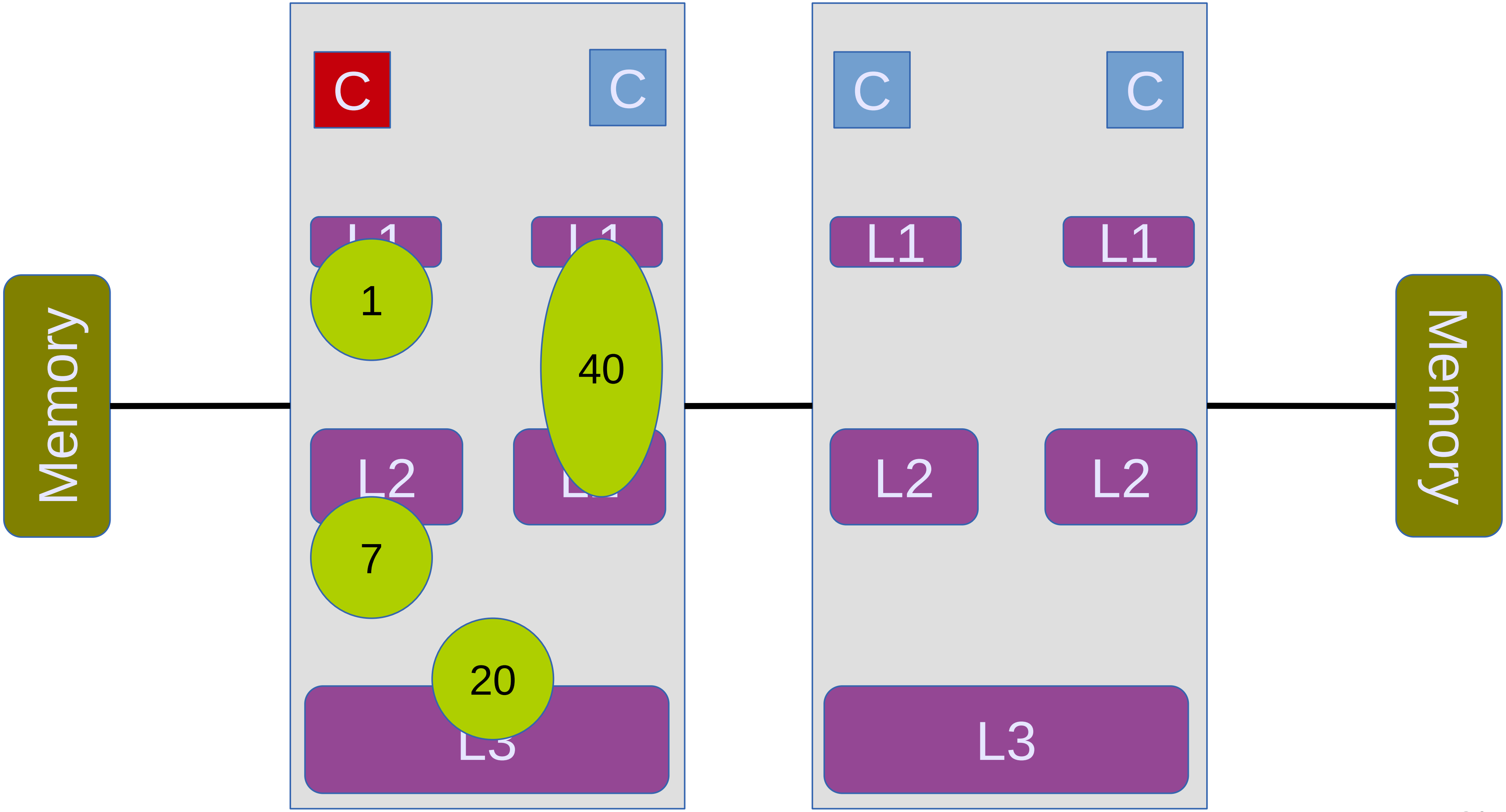
Latency (ns) to access data



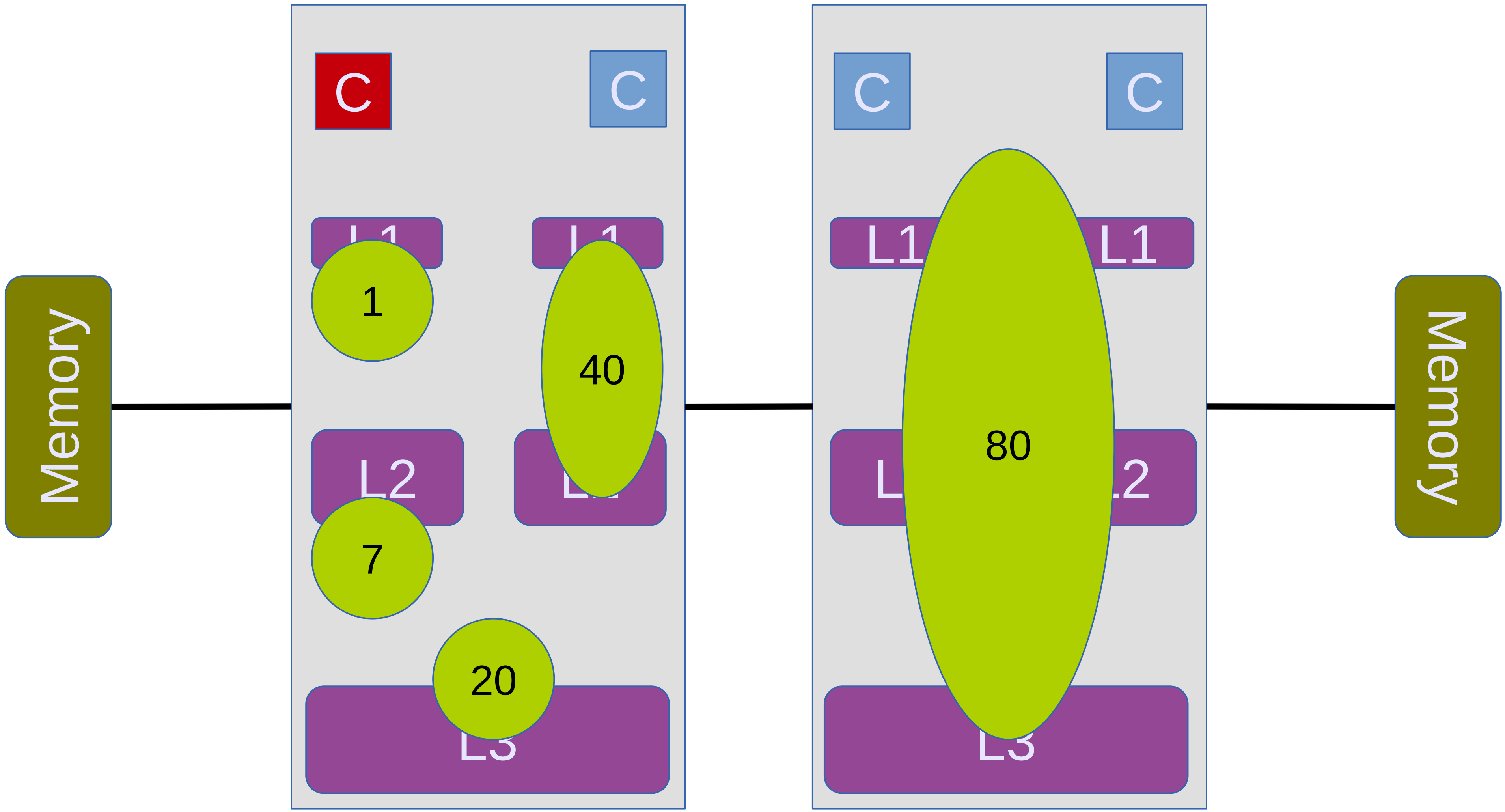
Latency (ns) to access data



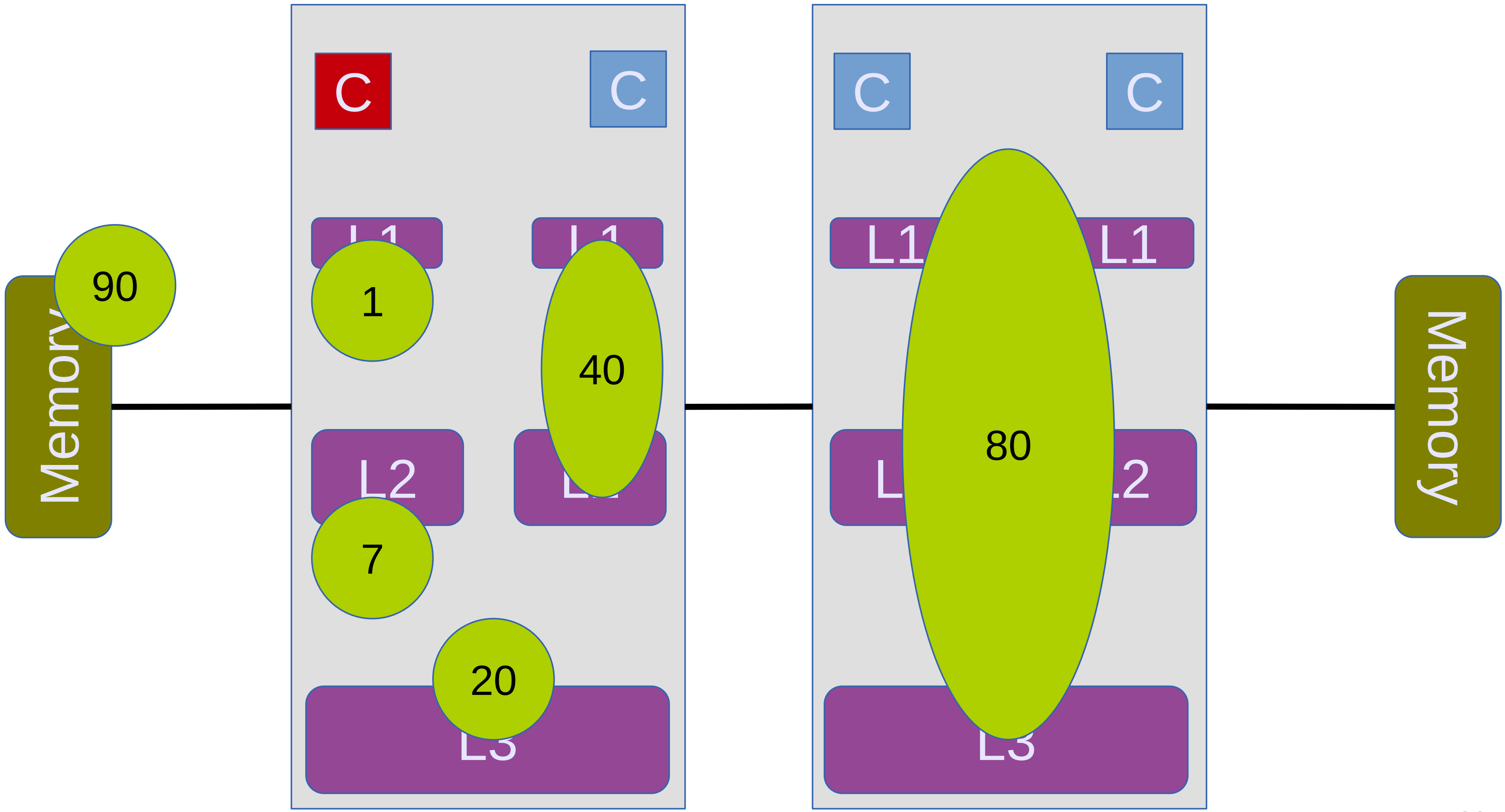
Latency (ns) to access data



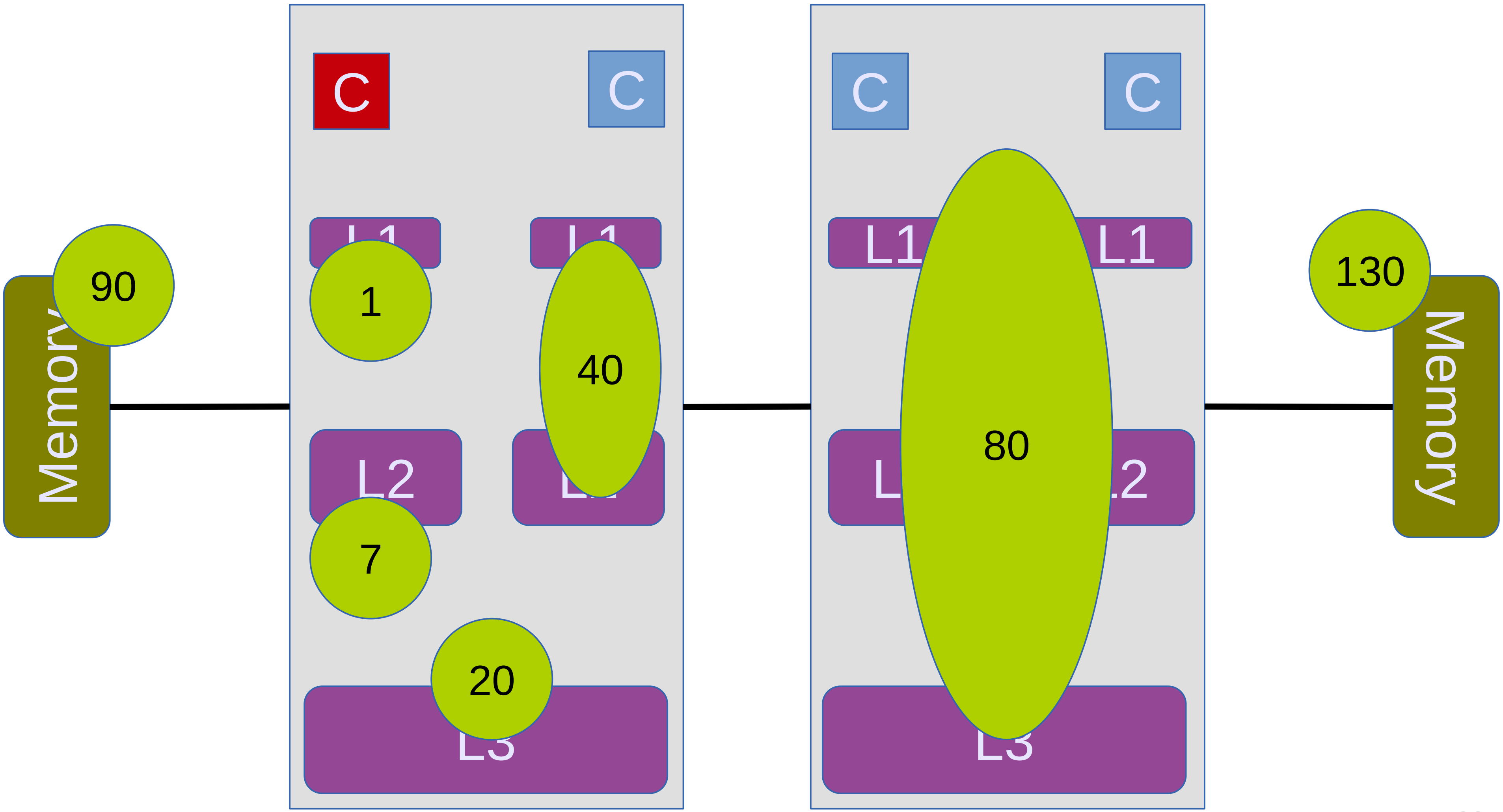
Latency (ns) to access data



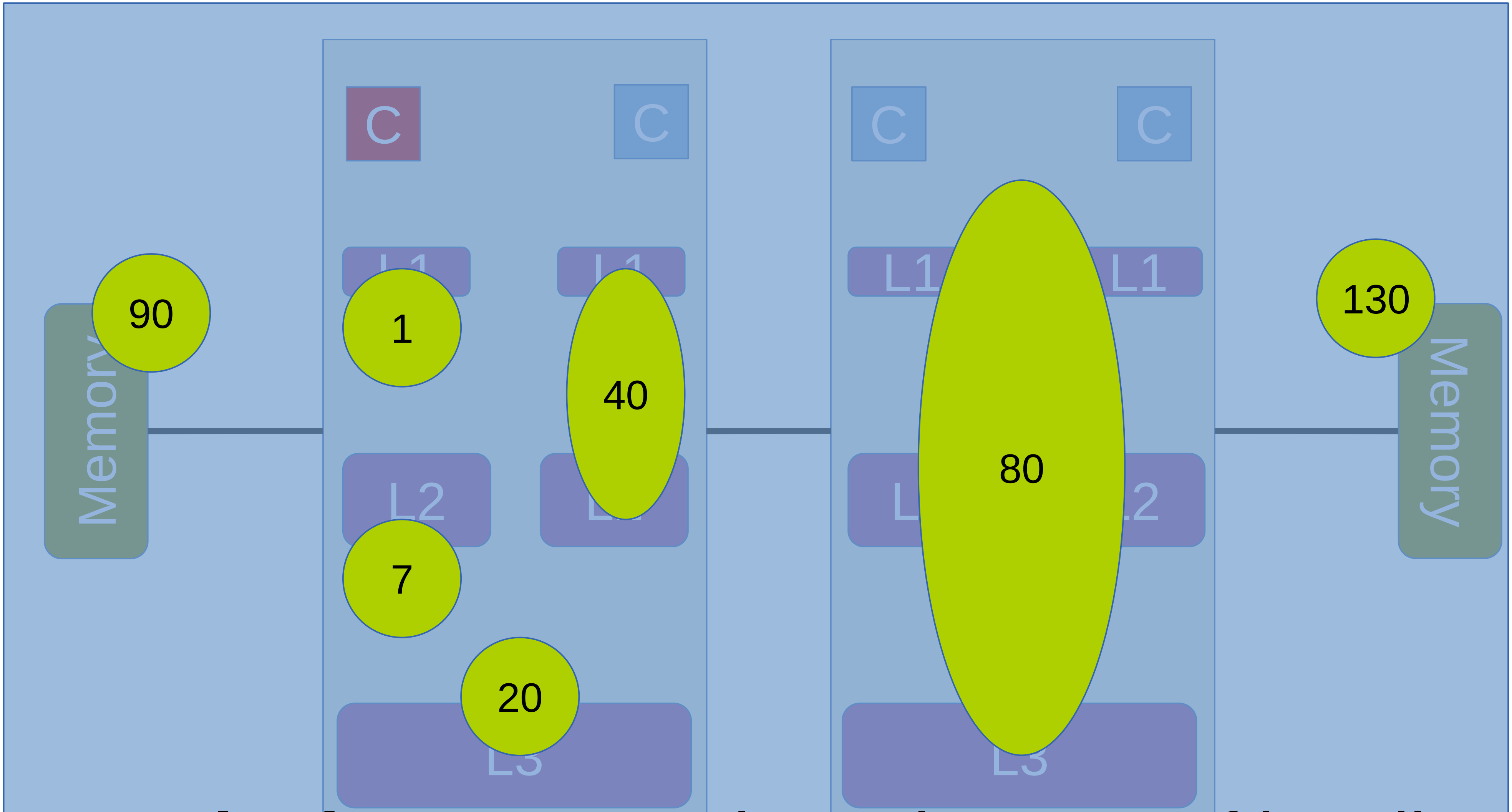
Latency (ns) to access data



Latency (ns) to access data



Latency (ns) to access data



Conclusion: we need to take care of locality

Experiment

The effects of locality

Experiment

The effects of locality

```
vtrigona $ ./test_locality -x0 -y1
Size:            8 counters = 1 cache lines
Thread 0 on core : 0
Thread 1 on core : 2
Number of threads: 2
Throughput       : 104.27 Mop/s
```

Same memory node

```
vtrigona $ ./test_locality -x0 -y10
Size:            8 counters = 1 cache lines
Thread 0 on core : 0
Thread 1 on core : 10
Number of threads: 2
Throughput       : 43.16 Mop/s
```

Different memory nodes

Outline

- CPU caches
- Cache coherence
- Placement of data
- **Graph processing: Concurrent data structures**

Graph processing

Relational view

People Table

Name	Likes
Vasilis	Breaking bad
Rachid	Dexter
Vasilis	Dexter

Series Table

Name	Similar
Breaking bad	Dexter
Dexter	Breaking bad

Graph processing

Relational view

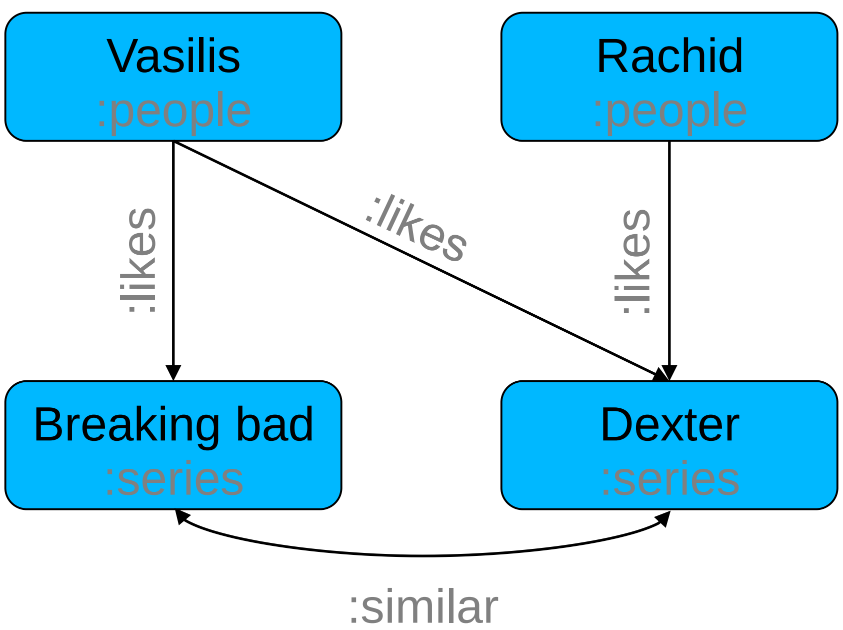
People Table

Name	Likes
Vasilis	Breaking bad
Rachid	Dexter
Vasilis	Dexter

Series Table

Name	Similar
Breaking bad	Dexter
Dexter	Breaking bad

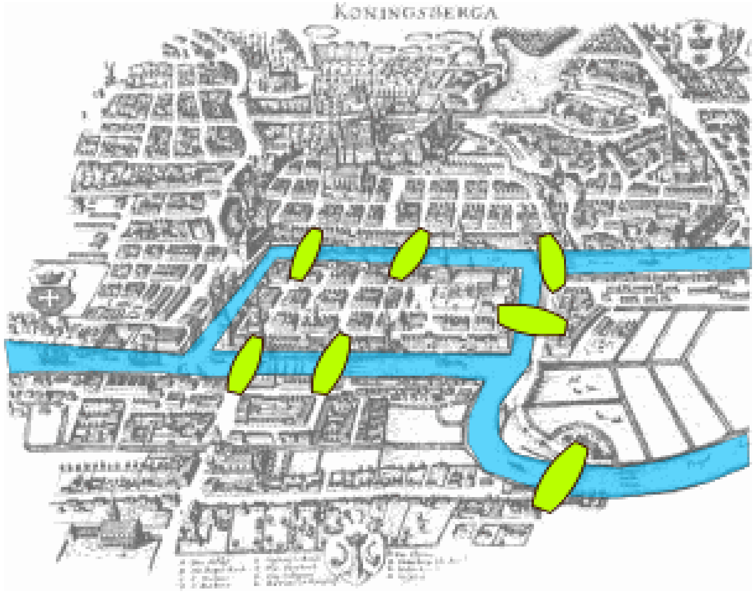
Graph view



Graphs keep the connections among entities materialized

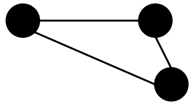
Graph analytics

- Graphs have been studied in Math for centuries
  - Since Euler's "Seven Bridges of Königsberg", 1736
- **Repeatedly traverse your graph and calculate math properties**
- Classic graph problems
  - Graph isomorphism
  - Travelling salesman's problem
  - Max flow, min cut
  - …
- More recent developments
  - Pagerank
  - Infomap



Graph queries

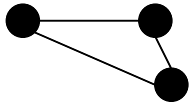
- Graph pattern matching
    - Query graphs to find sub-graphs that match a pattern e.g., triangle counting



- Essentially: SQL for graphs

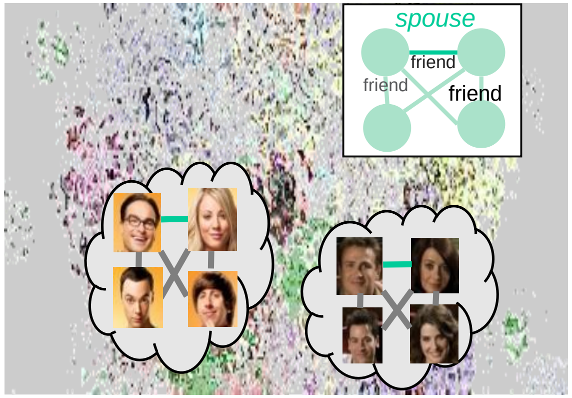
Graph queries

- Graph pattern matching
  - Query graphs to find sub-graphs that match a pattern e.g., triangle counting



- Essentially: SQL for graphs
- Example: Friends of my friends

```
SELECT p1, p3, COUNT(p2)
MATCH (p1)-[:friend]->(p2)->[:friend]->(p3),
          ! (p1)-[:friend]->(p3)
WHERE p1.country = p2.country
GROUP BY p1, p3
ORDER BY COUNT(p2) DESC
```



Graph processing frequently involves both analytics and queries

Dissecting a graph processing system

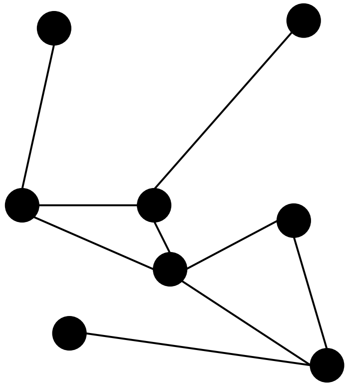
with a focus on (concurrent) data structures

~~Dissecting a graph processing system~~ Preparing for a job interview

with a focus on (concurrent) data structures

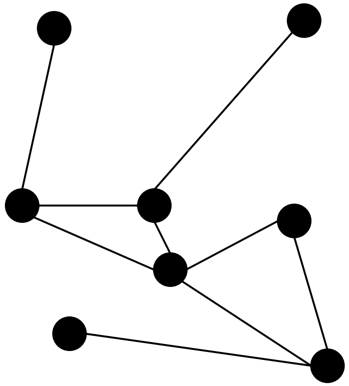
Architecture of a graph processing system

Graph



Architecture of a graph processing system

Graph



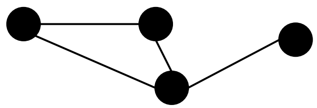
Tons of other data and metadata to store

Graph

tmp graph structure

"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
"Vasilis", "Dexter", :likes
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar

graph structure



user-ids - internal ids

Vasilis → 0	0 → Vasilis
Rachid → 1	1 → Rachid
Breaking bad → 2	2 → Breaking bad
Dexter → 3	3 → Dexter

labels

:likes, :people, :similar, …

properties

"Vasilis", {people, male}, 33, Zurich
"Rachid", {people, male}, ??, Lausanne

lifetime management

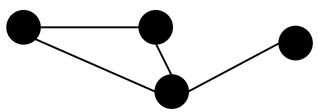
number_of_references: X

Graph

tmp graph structure

"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
"Vasilis", "Dexter", :likes
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar

graph structure



user-ids - internal ids

Vasilis → 0
Rachid → 1
Breaking bad → 2
Dexter → 3

0 → Vasilis
1 → Rachid
2 → Breaking bad
3 → Dexter

labels

:likes, :people, :similar, …

properties

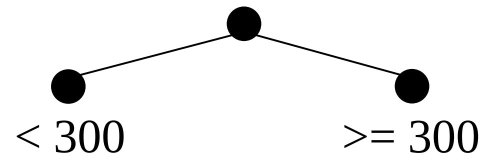
"Vasilis", {people, male}, 33, Zurich
"Rachid", {people, male}, ??, Lausanne

lifetime management

number_of_references: X

Runtime

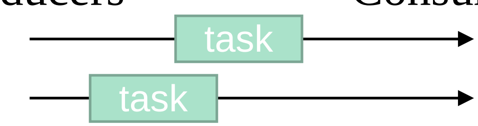
indices / metadata



buffer management

1MB	1MB	1MB	1MB

task / job scheduling



labels

:likes, :people, :similar, :male …

:likes	:people	:similar	:male
1	2	3	4

{people, male} → {2,4}

renaming (ids)

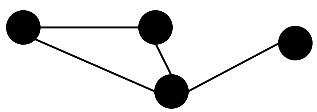
used		used	used

Graph

tmp graph structure

"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
"Vasilis", "Dexter", :likes
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar

graph structure



user-ids - internal ids

Vasilis → 0	0 → Vasilis
Rachid → 1	1 → Rachid
Breaking bad → 2	2 → Breaking bad
Dexter → 3	3 → Dexter

labels

:likes, :people, :similar, …

properties

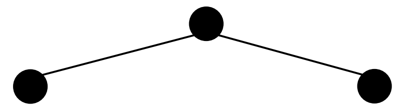
"Vasilis", {people, male}, 33, Zurich
"Rachid", {people, male}, ??, Lausanne

lifetime management

number_of_references: X

Runtime

indices / metadata



buffer management

1MB | 1MB | 1MB | 1MB

task / job scheduling

Producers → task → Consumers
task

labels

:likes, :people, :similar, :male …
↓ 1, ↓ 2, ↓ 3, ↓ 4

{people, male} → {2,4}

renaming (ids)

used		used	used

Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

→

Vasilis, 2
Rachid, 1

distinct

Vasilis
Rachid
Vasilis

→

Vasilis
Rachid

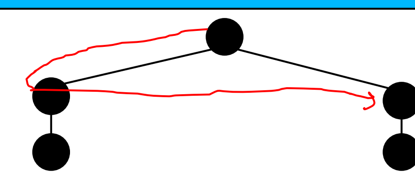
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

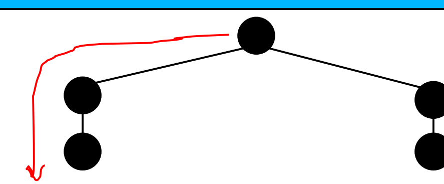
→

32
23
13

BFS



DFS

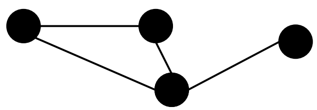


Graph

tmp graph structure

"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
"Vasilis", "Dexter", :likes
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar

graph structure



user-ids - internal ids

Vasilis → 0	0 → Vasilis
Rachid → 1	1 → Rachid
Breaking bad → 2	2 → Breaking bad
Dexter → 3	3 → Dexter

labels

:likes, :people, :similar, …

properties

"Vasilis", {people, male}, 33, Zurich
"Rachid", {people, male}, ??, Lausanne

lifetime management

number_of_references: X

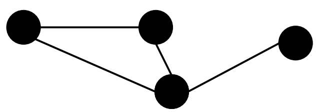
- tmp graph structure
  - append only
  - dynamic schema

Graph

tmp graph structure

"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
"Vasilis", "Dexter", :likes
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar

graph structure



user-ids - internal ids

Vasilis → 0	0 → Vasilis
Rachid → 1	1 → Rachid
Breaking bad → 2	2 → Breaking bad
Dexter → 3	3 → Dexter

labels

:likes, :people, :similar, …

properties

"Vasilis", {people, male}, 33, Zurich
"Rachid", {people, male}, ??, Lausanne

lifetime management

number_of_references: X

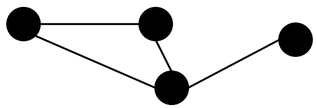
- tmp graph structure
  - append only
  - dynamic schema
  - → **segmented table**

Graph

tmp graph structure

“Vasilis”, “Breaking bad”, :likes
“Rachid”, “Dexter”, :likes
“Vasilis”, “Dexter”, :likes
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

graph structure



user-ids - internal ids

Vasilis → 0
Rachid → 1
Breaking bad → 2
Dexter → 3

0 → Vasilis
1 → Rachid
2 → Breaking bad
3 → Dexter

labels

:likes, :people, :similar, …

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X

- tmp graph structure
  - append only
  - dynamic schema

  → **segmented table**

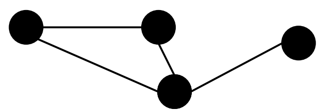
- Classic graph structures

Graph

tmp graph structure

“Vasilis”, “Breaking bad”, :likes
“Rachid”, “Dexter”, :likes
“Vasilis”, “Dexter”, :likes
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

graph structure



user-ids - internal ids

Vasilis → 0
Rachid → 1
Breaking bad → 2
Dexter → 3

0 → Vasilis
1 → Rachid
2 → Breaking bad
3 → Dexter

labels

:likes, :people, :similar, …

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X

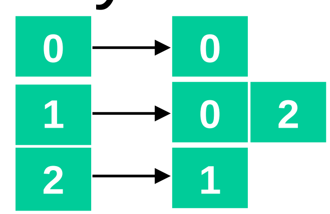
- tmp graph structure
  - append only
  - dynamic schema

  → **segmented table**

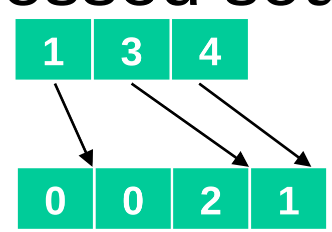
- Classic graph structures
  1. connectivity matrix

	0	1	2
0	x		
1	x		x
2		x	

  2. adjacency list



  3. compressed source row (CSR)



Graph

tmp graph structure

“Vasilis”, “Breaking bad”, :likes
“Rachid”, “Dexter”, :likes
“Vasilis”, “Dexter”, :likes
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

segmented buffer

graph structure

CSR

user-ids - internal ids

Vasilis → 0	0 → Vasilis
Rachid → 1	1 → Rachid
Breaking bad → 2	2 → Breaking bad
Dexter → 3	3 → Dexter

labels

:likes, :people, :similar, …

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X

- Mapping user ids to internal ids
  - create once
  - read-only after

Graph

tmp graph structure

segmented buffer

“Vasilis”, “Breaking bad”, :likes
“Rachid”, “Dexter”, :likes
“Vasilis”, “Dexter”, :likes
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

graph structure

user-ids - internal ids

Vasilis → 0	0 → Vasilis
Rachid → 1	1 → Rachid
Breaking bad → 2	2 → Breaking bad
Dexter → 3	3 → Dexter

labels

:likes, :people, :similar, …

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X

- Mapping user ids to internal ids
  - create once
  - read-only after
  - → **hash map, lock-free reads**

Graph

tmp graph structure

segmented buffer

“Vasilis”, “Breaking bad”, :likes
“Rachid”, “Dexter”, :likes
“Vasilis”, “Dexter”, :likes
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

graph structure

user-ids - internal ids

Vasilis → 0	0 → Vasilis
Rachid → 1	1 → Rachid
Breaking bad → 2	2 → Breaking bad
Dexter → 3	3 → Dexter

labels

:likes, :people, :similar, …

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X

- Mapping user ids to internal ids
  - create once
  - read-only after
  - → **hash map, lock-free reads**
- Mapping internal ids to user ids
  - create once
  - read-only after
  - fixed key range: [0, N}

Graph

tmp graph structure

segmented buffer

"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar

graph structure

user-ids - internal ids

Vasilis → 0	0 → Vasilis
Rachid → 1	1 → Rachid
Breaking bad → 2	2 → Breaking bad
Dexter → 3	3 → Dexter

labels

:likes, :people, :similar, …

properties

"Vasilis", {people, male}, 33, Zurich
"Rachid", {people, male}, ??, Lausanne

lifetime management

number_of_references: X

- Mapping user ids to internal ids
  - create once
  - read-only after
  - → **hash map, lock-free reads**

- Mapping internal ids to user ids
  - create once
  - read-only after
  - fixed key range: [0, N}
  - → **(sequential) array**

Graph

tmp graph structure

segmented buffer

“Vasilis”, “Breaking bad”, :likes
“Rachid”, “Dexter”, :likes
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

graph structure

user-ids - internal ids

hash map / array

Vasilis → 0 | 0 → Vasilis
Rachid → 1 | 1 → Rachid
Breaking bad → 2 | 2 → Breaking bad
Dexter → 3 | 3 → Dexter

labels

:likes, :people, :similar, …

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X

- Storing labels
  - usually a small enumeration e.g., person, female, male
  - storing strings is expensive “person” → ~ 7 bytes
  - comparing strings is expensive

Graph

tmp graph structure

segmented buffer

“Vasilis”, “Breaking bad”, :likes
“Rachid”, “Dexter”, :likes
“Vasilis”, “Dexter”, :likes
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

graph structure

CSR

user-ids - internal ids

hash map / array

Vasilis → 0	0 → Vasilis
Rachid → 1	1 → Rachid
Breaking bad → 2	2 → Breaking bad
Dexter → 3	3 → Dexter

labels

:likes, :people, :similar, …

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

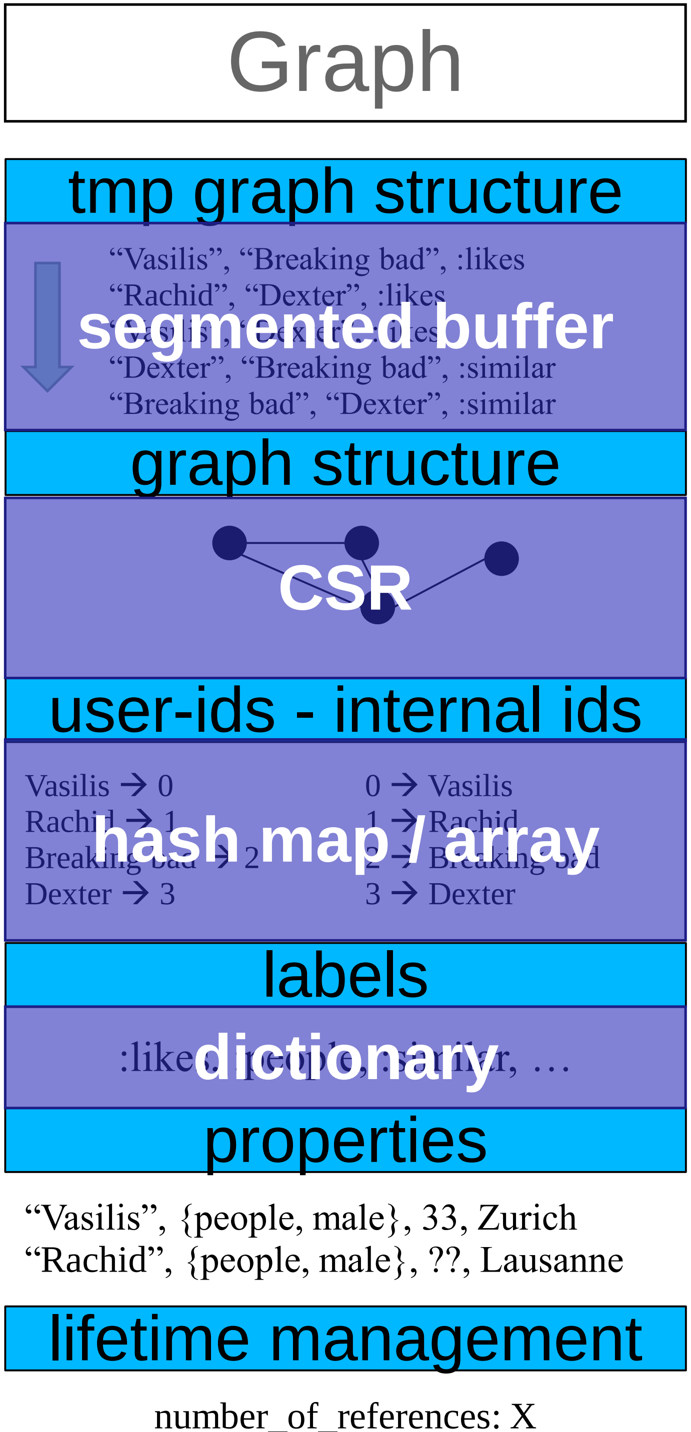
lifetime management

number_of_references: X

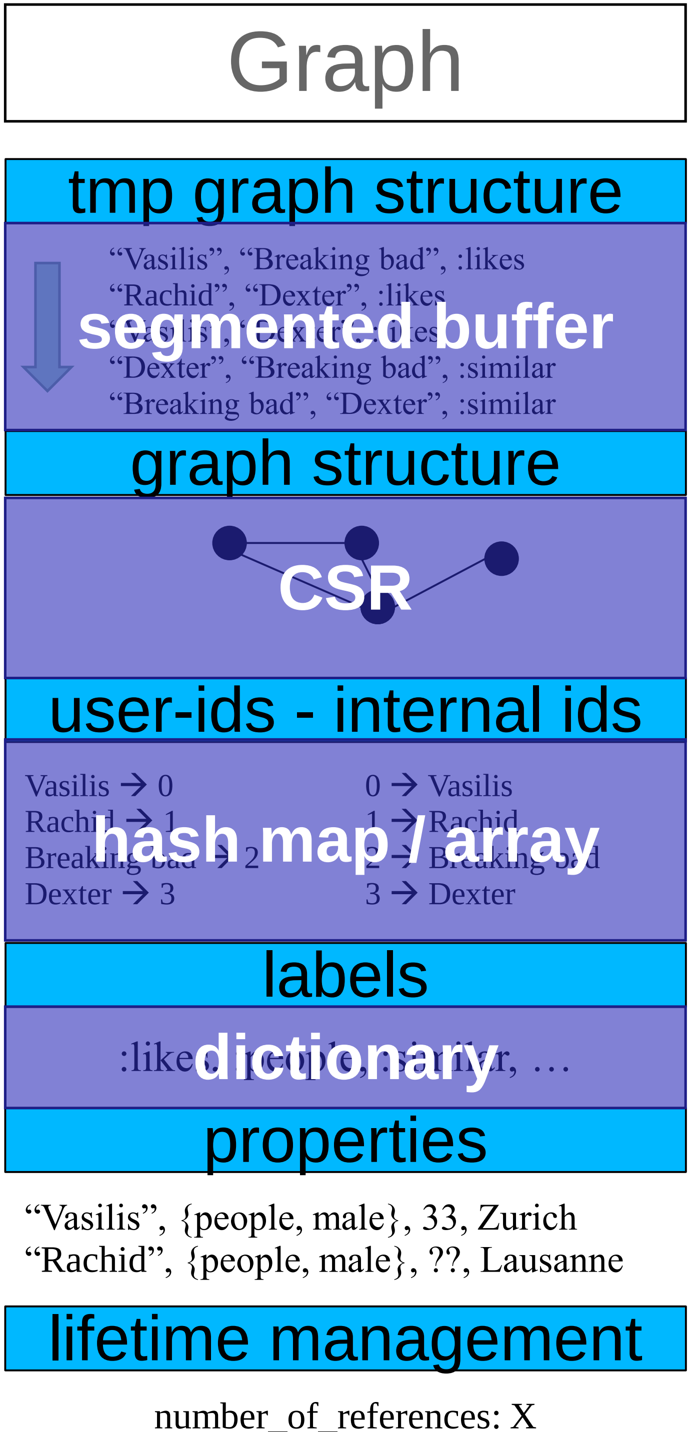
- Storing labels
  - usually a small enumeration e.g., person, female, male
  - storing strings is expensive “person” → ~ 7 bytes
  - comparing strings is expensive

  → **dictionary encoding**, e.g.,
    - person → 0
    - female → 1
    - male → 2

- Ofc, **hash map** to
  - store those
  - translate during runtime

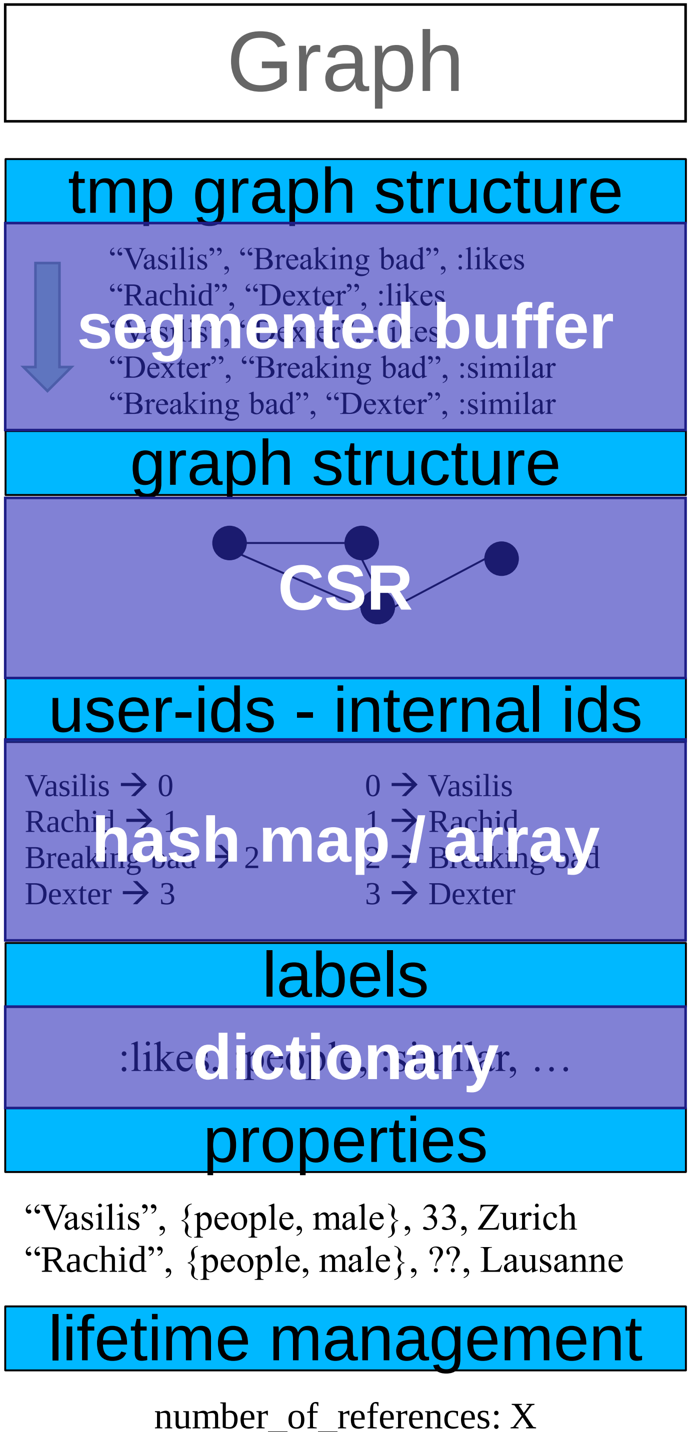


- Property
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges



- Property
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges

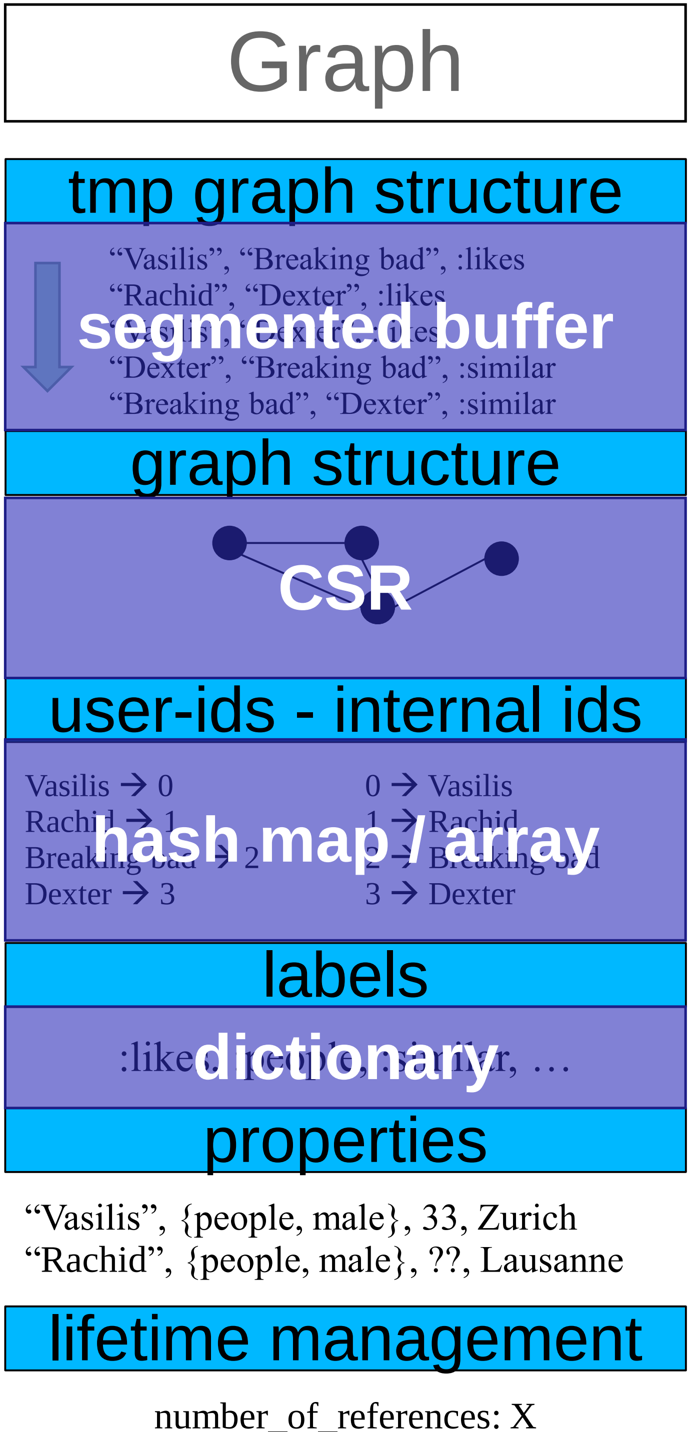
  → **(sequential) arrays**



- Property
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges

  → **(sequential) arrays**

- Lifetime management (and other counters)
  - cache coherence: atomic counters can be expensive



- Property
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges

  → **(sequential) arrays**

- Lifetime management (and other counters)
  - cache coherence: atomic counters can be expensive
  - Two potential solutions
    1. **approximate counters**
    2. **stripped counters**

Thread local:	counter[0]	counter[1]	counter[2]

```
increment(int by) { counter[my_thread_id] += by; }
int value() {
    int sum = 0;
    for (int i = 0; i < num_threads; i++) { sum += counter[i]; }
    return sum;
}
```

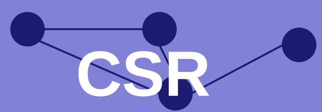
Graph

tmp graph structure

"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
"Vasilis", "Dexter", :likes
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar

segmented buffer

graph structure



CSR

user-ids - internal ids

Vasilis → 0
Rachid → 1
Breaking bad → 2
Dexter → 3

0 → Vasilis
1 → Rachid
2 → Breaking bad
3 → Dexter

hash map / array

labels

:likes, :people, :similar

dictionary (= map)

properties

"Vasilis", {people, male}, 33, Zurich
"Rachid", {people, male}, ??, Lausanne

array

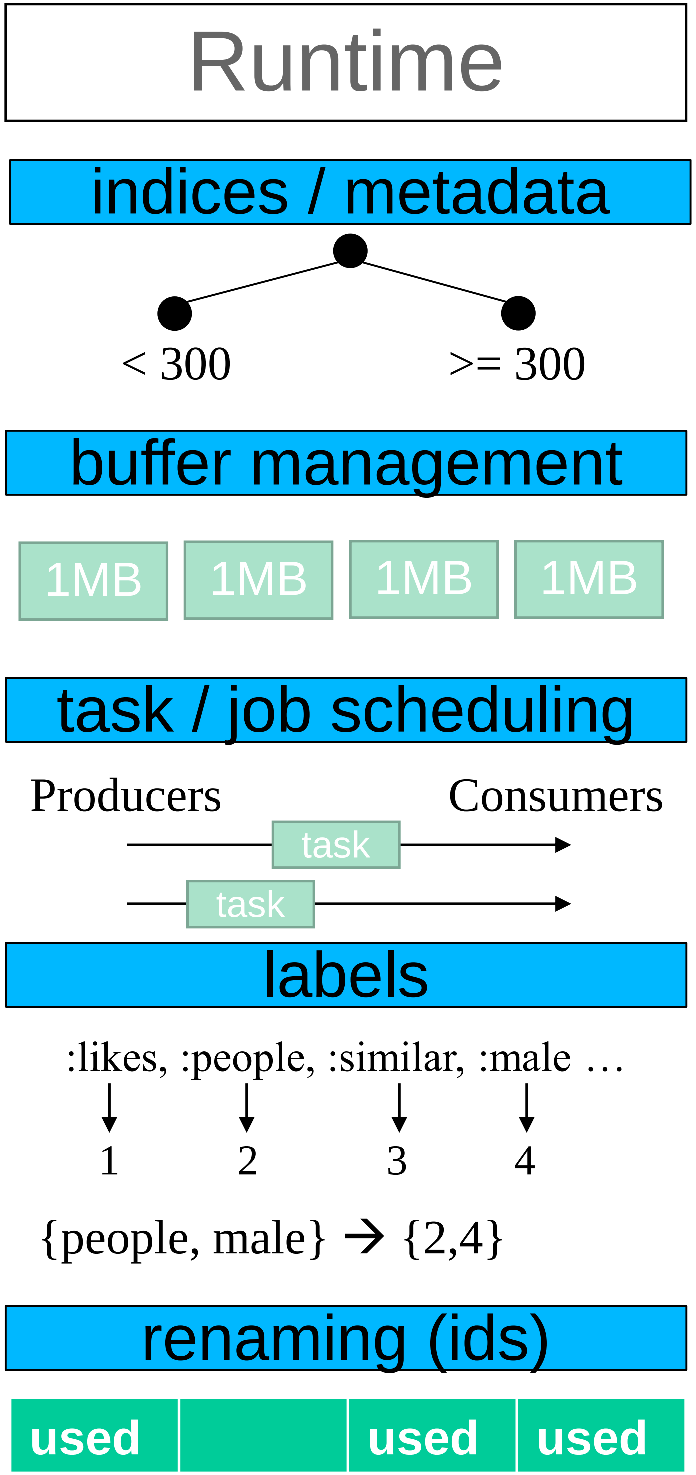
lifetime management

number_of_references: X

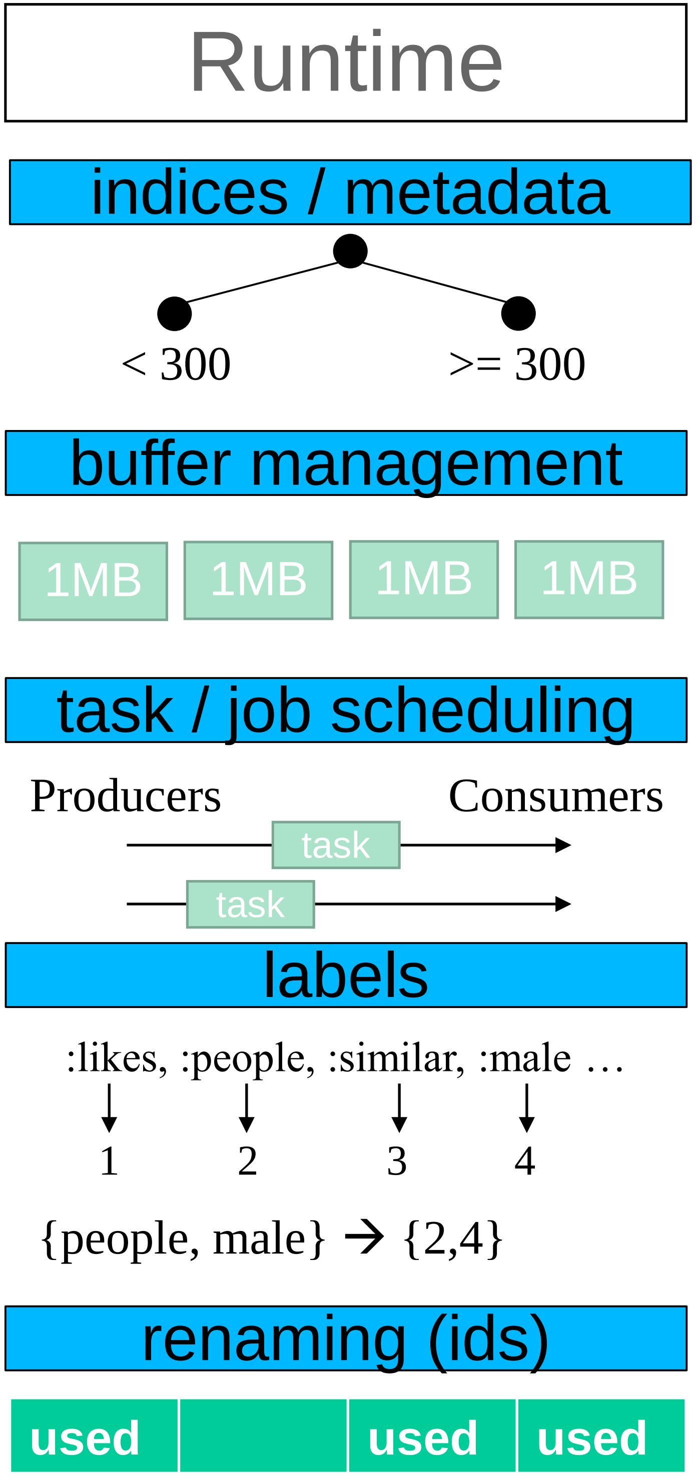
stripped counter

Score

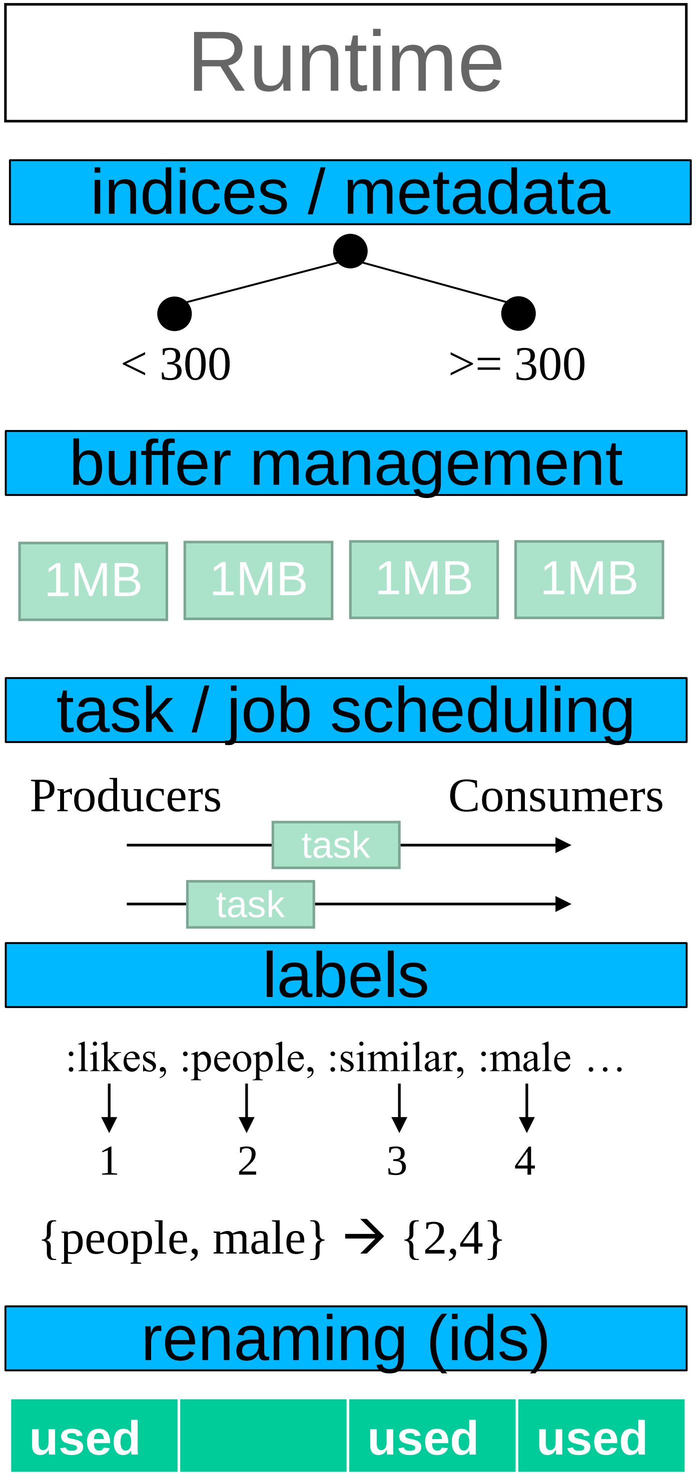
Structure	# Usages
array / buffer	5
map	2



- Indices
  - Used for speeding up “queries”
    - Which vertices have label :person?
    - Which edges have value > 1000?

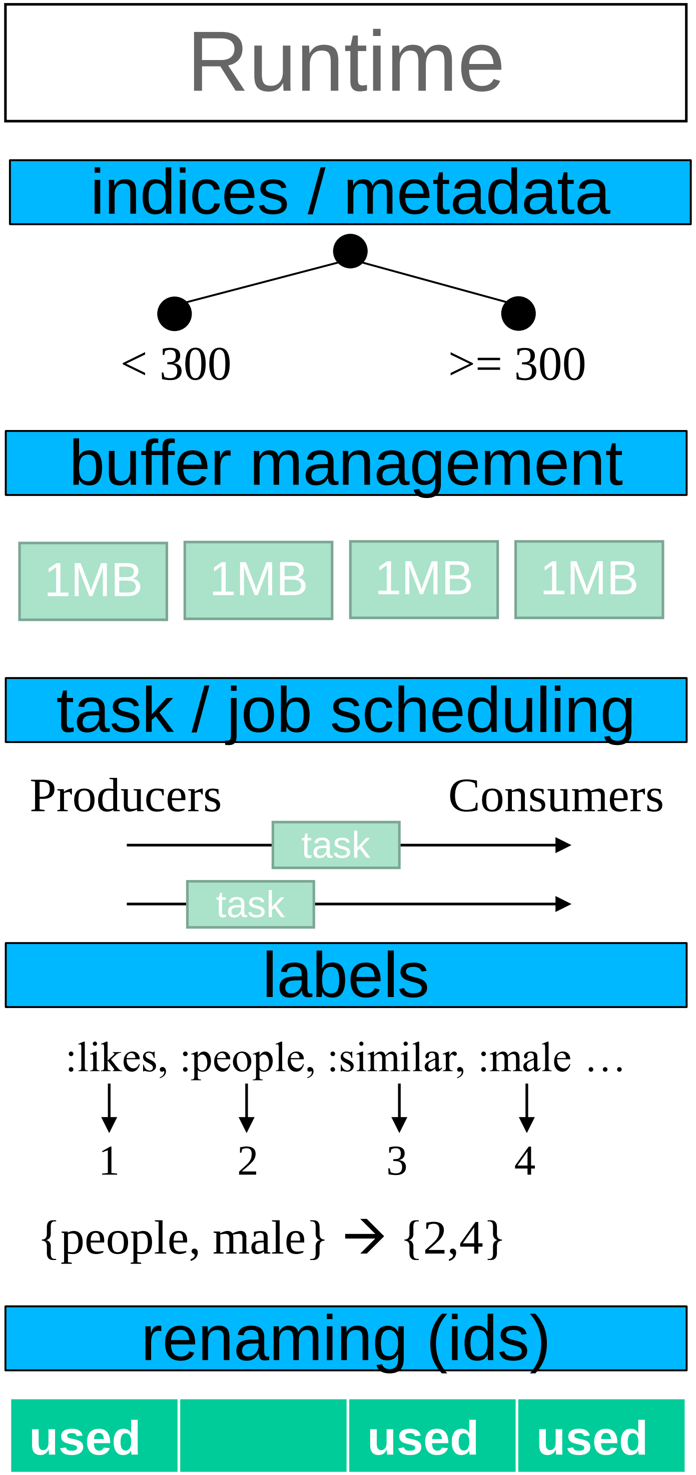


- Indices
  - Used for speeding up “queries”
    - Which vertices have label :person?
    - Which edges have value > 1000?
  - →**maps, trees**



- Indices
  - Used for speeding up “queries”
    - Which vertices have label :person?
    - Which edges have value > 1000?
  - →**maps, trees**

- Buffer management
  - In “real” systems, resource management is very important
  - buffer pools
    - no order
    - insertions and deletions
    - no keys



- Indices
  - Used for speeding up “queries”
    - Which vertices have label :person?
    - Which edges have value > 1000?

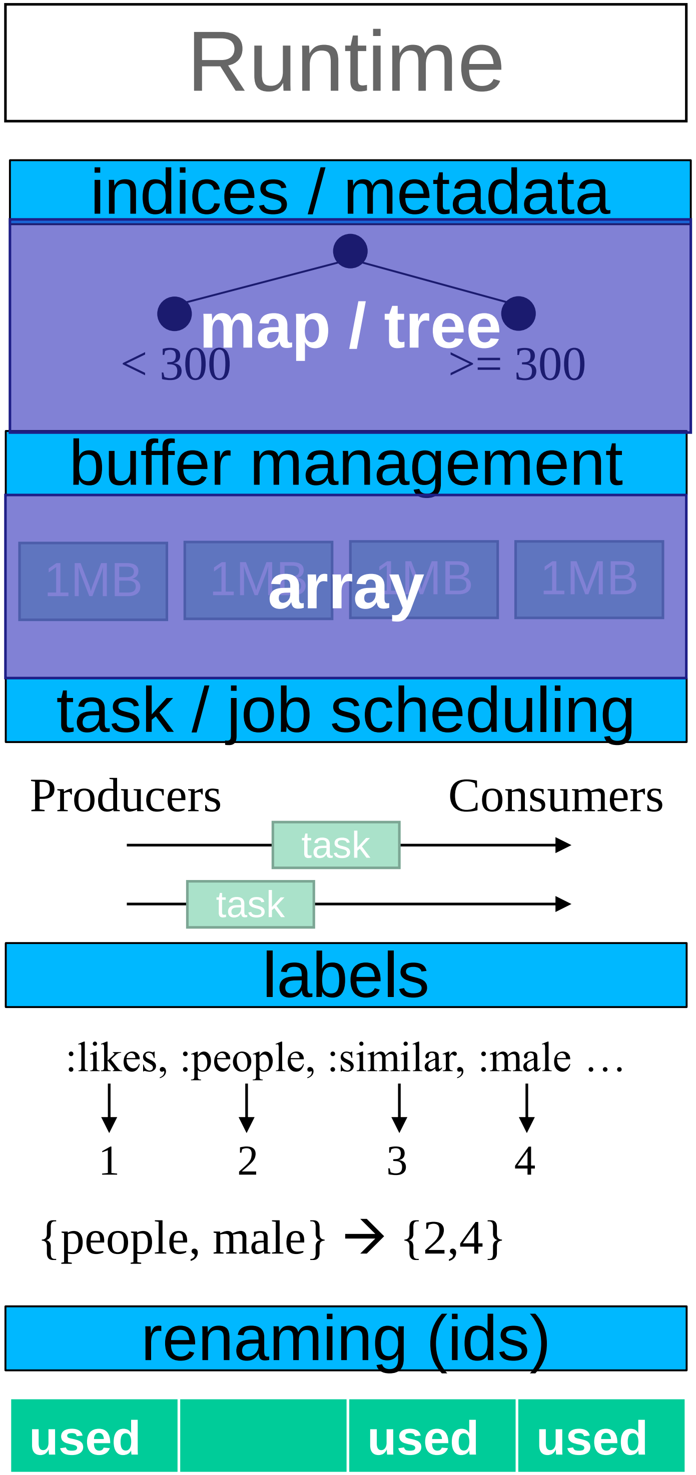
  → **maps, trees**

- Buffer management
  - In “real” systems, resource management is very important
  - buffer pools
    - no order
    - insertions and deletions
    - no keys

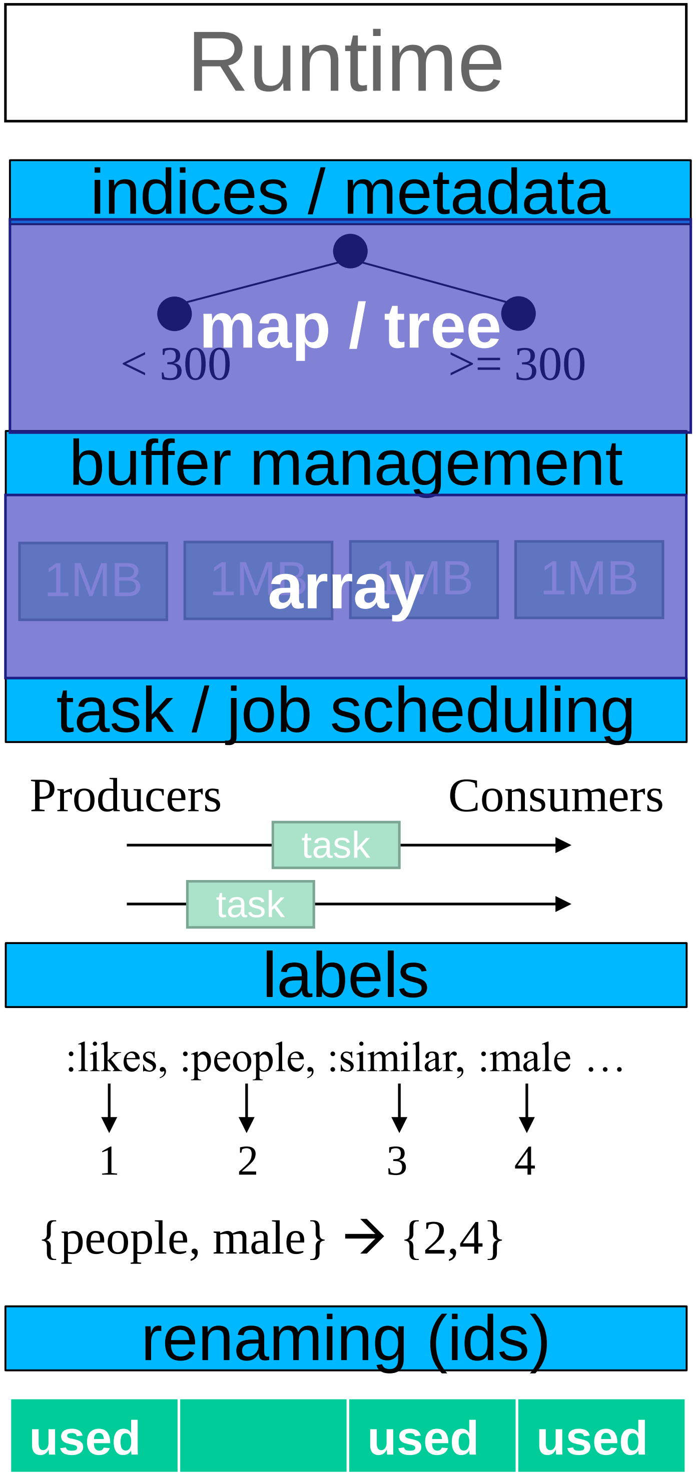
  → Fixed num object pool: **array**

  → Otherwise: **list**

  → Variable-sized elements: **heap**



- Task and job scheduling
  - producers create and share tasks
  - consumers get and handle tasks
  - insertions and deletions
  - usually FIFO requirements

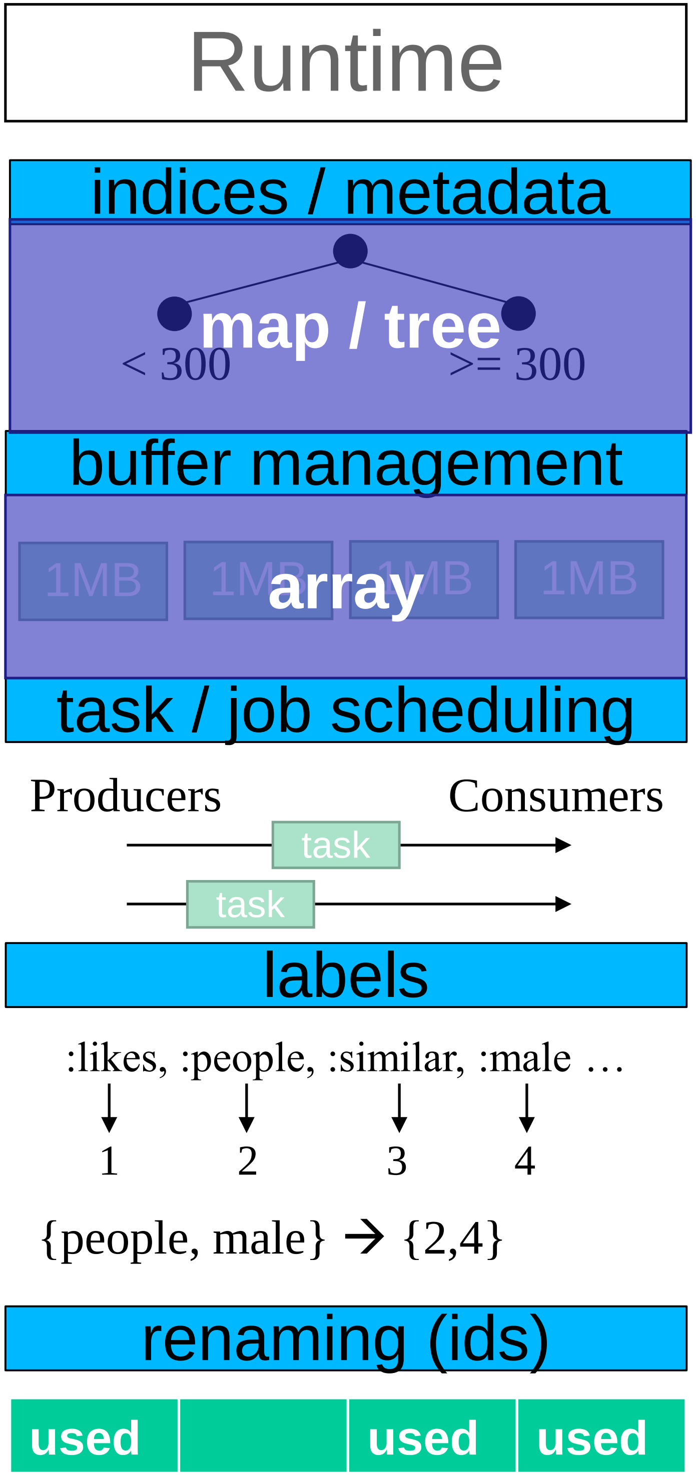


- Task and job scheduling
  - producers create and share tasks
  - consumers get and handle tasks
  - insertions and deletions
  - usually FIFO requirements

  → **queues**

- Storing / querying sets of labels
  - set equality expensive
  - usually common groups

    e.g., {person, female}, {person, male}



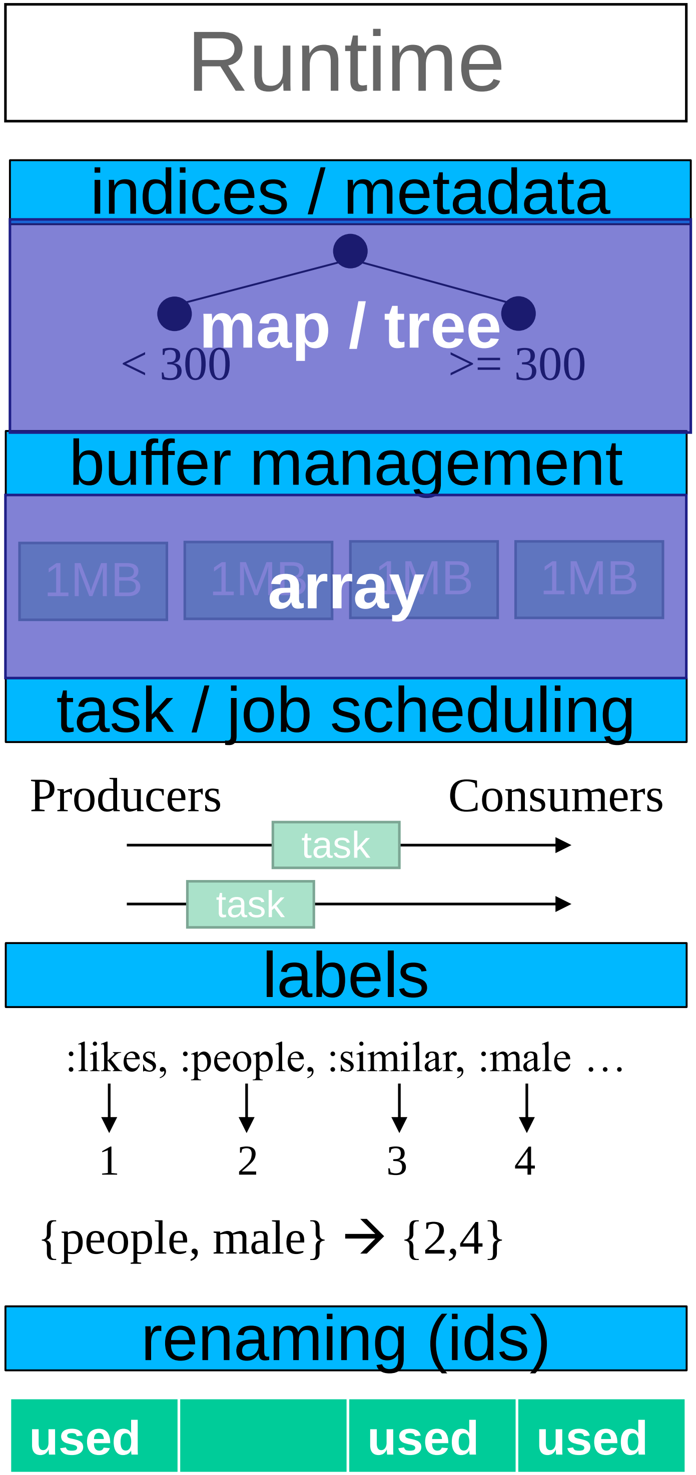
- Task and job scheduling
  - producers create and share tasks
  - consumers get and handle tasks
  - insertions and deletions
  - usually FIFO requirements

  → **queues**

- Storing / querying sets of labels
  - set equality expensive
  - usually common groups

    e.g., {person, female}, {person, male}

  → 2-level **dictionary** encoding
    - {person, female} → 0
    - {person, male} → 1



- Task and job scheduling
  - producers create and share tasks
  - consumers get and handle tasks
  - insertions and deletions
  - usually FIFO requirements

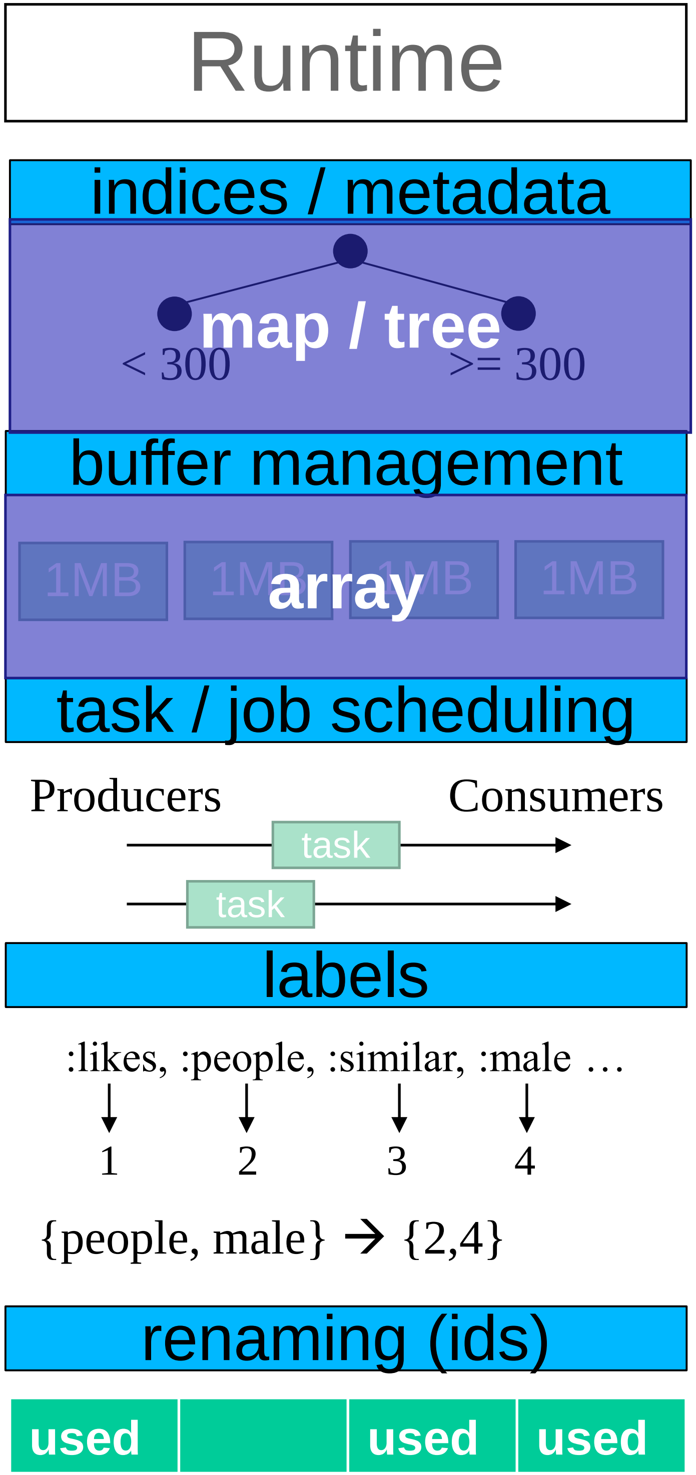
  → **queues**

- Storing / querying sets of labels
  - set equality expensive
  - usually common groups

    e.g., {person, female}, {person, male}

  → 2-level **dictionary** encoding
    - {person, female} → 0
    - {person, male} → 1

- Giving unique ids (renaming)



- Task and job scheduling
  - produces create and share tasks
  - consumers get and handle tasks
  - insertions and deletions
  - usually FIFO requirements

  → **queues**

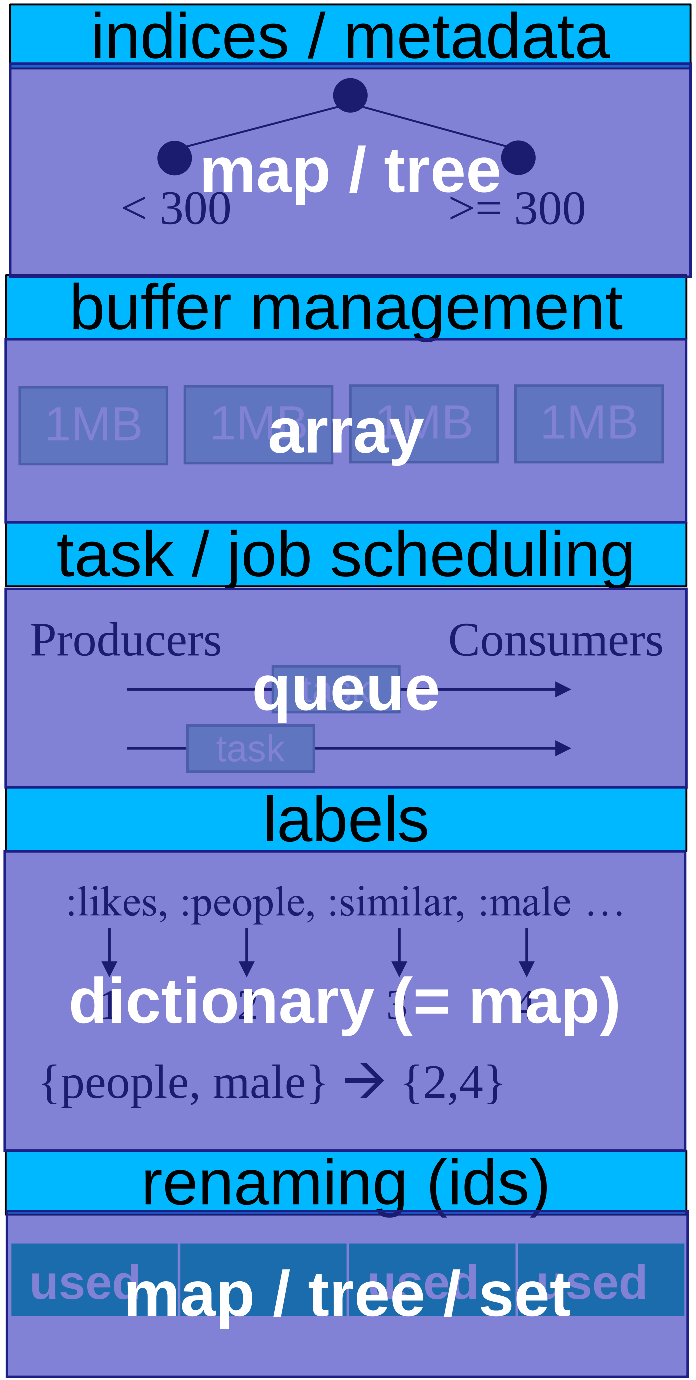
- Storing / querying sets of labels
  - set equality expensive
  - usually common groups
    e.g., {person, female}, {person, male}

  → 2-level **dictionary** encoding
    - {person, female} → 0
    - {person, male} → 1

- Giving unique ids (renaming)

  → **tree, map, set**, **counter**, other?

Runtime



Score

Structure	# Usages
array / buffer	6
map	5
tree / heap	2
set	1
queue	1

Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

→

Vasilis, 2
Rachid, 1

distinct

Vasilis
Rachid
Vasilis

→

Vasilis
Rachid

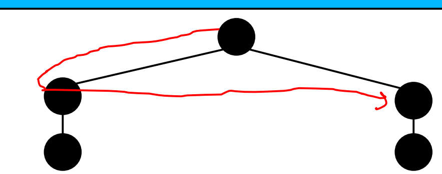
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

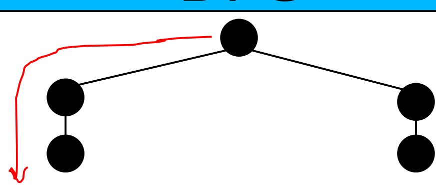
→

32
23
13

BFS



DFS



- Group by
  1. Mapping from keys to values
  2. Atomic value aggregations e.g., COUNT, SUM, MAX
  - insertion only

Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter
→
Vasilis, 2
Rachid, 1

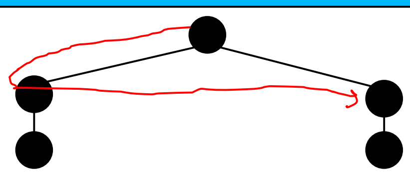
distinct

Vasilis
Rachid
Vasilis
→
Vasilis
Rachid

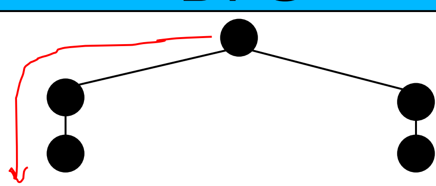
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0
→
32
23
13

BFS



DFS



- Group by
  1. Mapping from keys to values
  2. Atomic value aggregations e.g., COUNT, SUM, MAX
  - insertion only

  → **hash map**

  → **atomic inc / sum / max**, etc.

Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

Vasilis, 2
Rachid, 1

distinct

Vasilis
Rachid
Vasilis

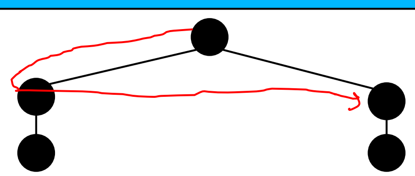
Vasilis
Rachid

limit (top k)

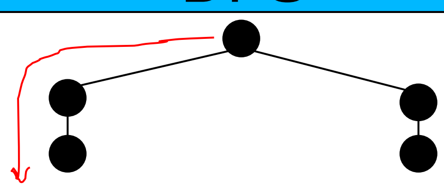
11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

32
23
13

BFS



DFS



- Group by
  1. Mapping from keys to values
  2. Atomic value aggregations
     e.g., COUNT, SUM, MAX
  - insertion only

  → **hash map**

  → **atomic inc / sum / max**, etc.

- Join
  - create a map of the small table
  - insertion phase, followed by
  - probing phase

Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

→

Vasilis, 2
Rachid, 1

distinct

Vasilis
Rachid
Vasilis

→

Vasilis
Rachid

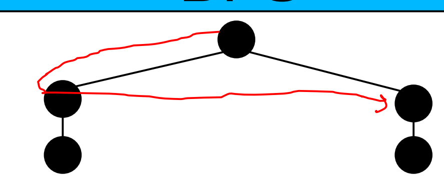
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

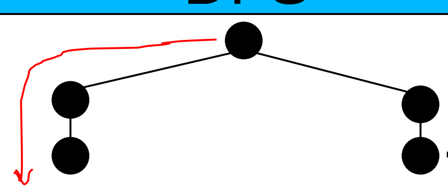
→

32
23
13

BFS



DFS



- Group by
  1. Mapping from keys to values
  2. Atomic value aggregations e.g., COUNT, SUM, MAX
  - insertion only

  → **hash map**

  → **atomic inc / sum / max**, etc.

- Join
  - create a map of the small table
  - insertion phase, followed by
  - probing phase

  → **hash map, lock-free probing**

Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

Vasilis, 2
Rachid, 1

map / atomics

distinct

Vasilis
Rachid
Vasilis

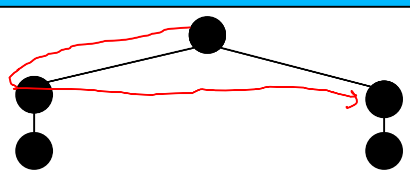
Vasilis
Rachid

limit (top k)

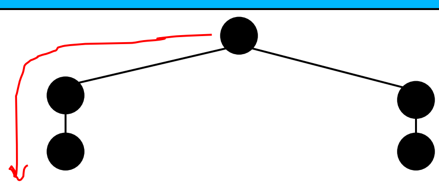
11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

32
23
13

BFS



DFS



- Distinct
  - can be solved with sorting, or

Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

→

Vasilis, 2
Rachid, 1

map / atomics

distinct

Vasilis
Rachid
Vasilis

→

Vasilis
Rachid

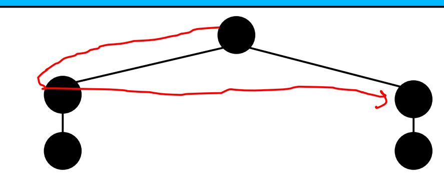
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

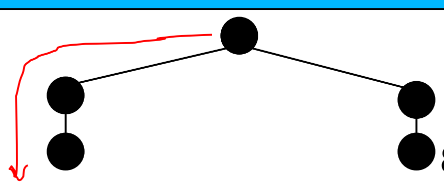
→

32
23
13

BFS



DFS



- Distinct
  - can be solved with sorting, or

→ **hash set**

Operations

group by / join

map / atomics

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

→

Vasilis, 2
Rachid, 1

distinct

Vasilis
Rachid
Vasilis

→

Vasilis
Rachid

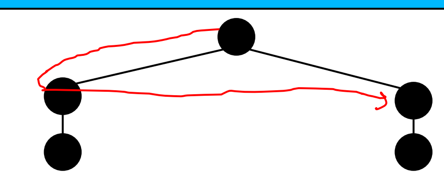
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

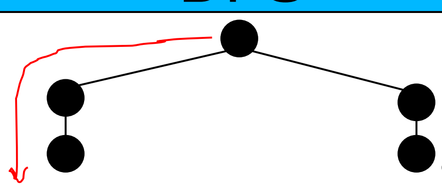
→

32
23
13

BFS



DFS



- Distinct
  - can be solved with sorting, or

  → **hash set**

- Limit (top k)
  - can be solved with sorting, or
  - different specialized structures

Operations

group by / join

map / atomics

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexte

→

Vasilis, 2
Rachid, 1

distinct

Vasilis
Rachid
Vasilis

→

Vasilis
Rachid

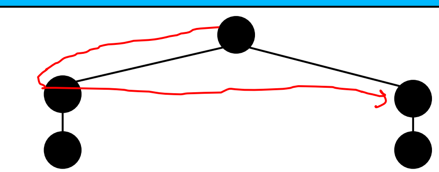
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

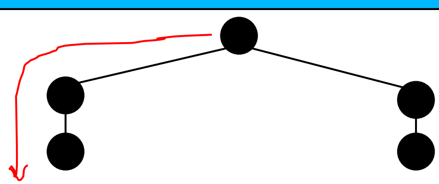
→

32
23
13

BFS



DFS



- Distinct
  - can be solved with sorting, or
  - → **hash set**

- Limit (top k)
  - can be solved with sorting, or
  - different specialized structures
  - → **tree**
  - → **heap**
  - → ~ **list**
  - → **array** (e.g., 2 elements only)
  - → **register** (1 element only)

Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

Vasilis, 2
Rachid, 1

map / atomics

distinct

Vasilis
Rachid
Vasilis

Vasilis
Rachid

hash set

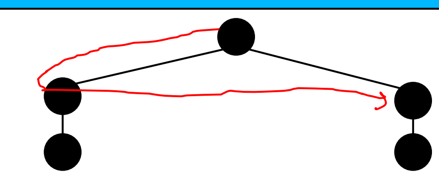
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

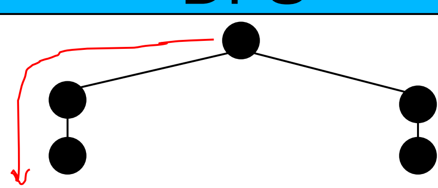
32
23
13

tree / heap / list

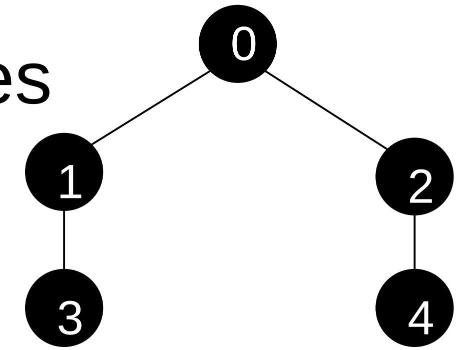
BFS



DFS



- Breadth-first search (BFS)
  - FIFO order
  - track visited vertices



Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

Vasilis, 2
Rachid, 1

map / atomics

distinct

Vasilis
Rachid
Vasilis

Vasilis
Rachid

hash set

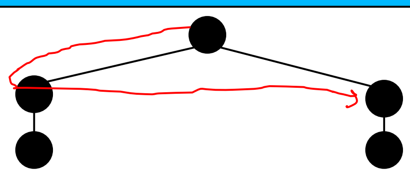
limit (top k)

11 12 0 9 8 13
8 9 11 22 32 9
1 2 3 5 7 3 2 0

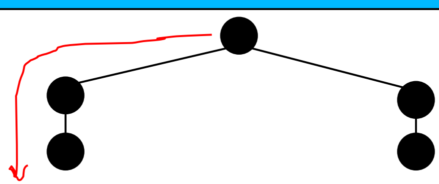
32
23
13

tree / heap / list

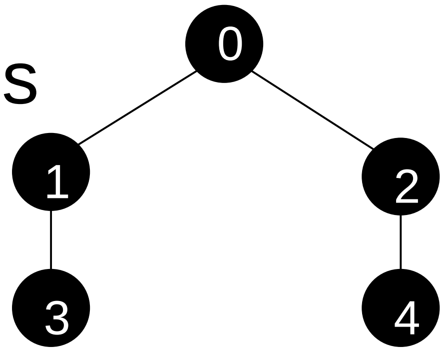
BFS



DFS



- Breadth-first search (BFS)
  - FIFO order
  - track visited vertices
  - → **queue**
  - → **set**



Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

Vasilis, 2
Rachid, 1

map / atomics

distinct

Vasilis
Rachid
Vasilis

Vasilis
Rachid

hash set

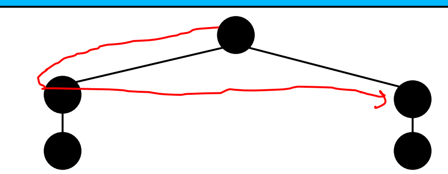
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

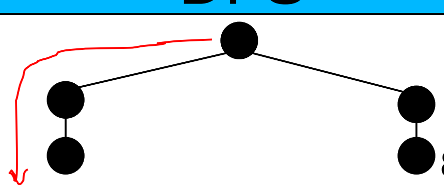
32
23
13

tree / heap / list

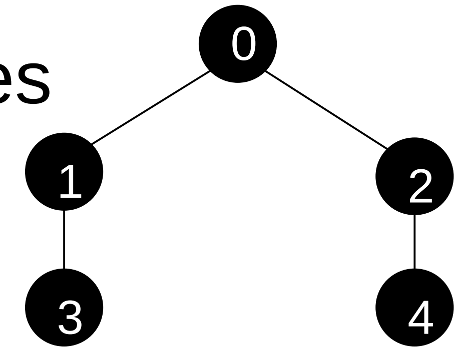
BFS



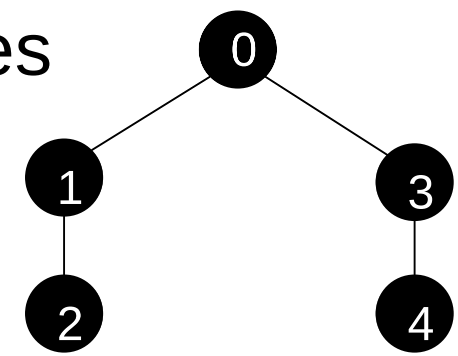
DFS



- Breadth-first search (BFS)
  - FIFO order
  - track visited vertices
  - → **queue**
  - → **set**



- Depth-first search (DFS)
  - LIFO order
  - track visited vertices



Operations

group by / join

map / atomics

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

Vasilis, 2
Rachid, 1

distinct

hash set

Vasilis
Rachid
Vasilis

Vasilis
Rachid

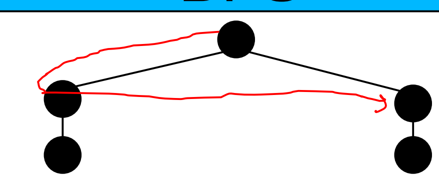
limit (top k)

tree / heap / list

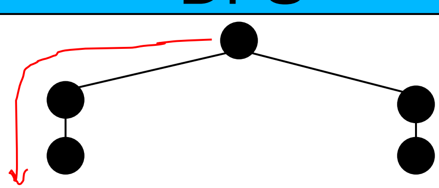
11 12 0 9 8 13
8 9 11 22 32 9
1 2 3 5 7 3 2 0

32
23
13

BFS

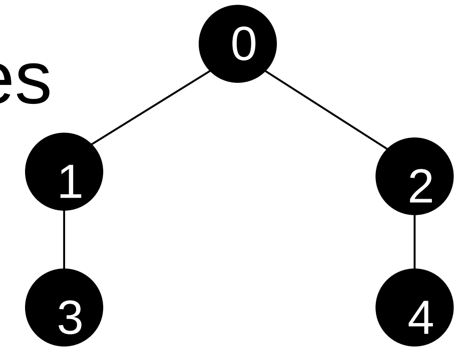


DFS



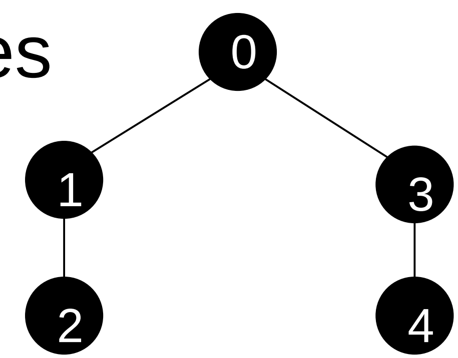
- Breadth-first search (BFS)
  - FIFO order
  - track visited vertices

  → **queue**
  → **set**



- Depth-first search (DFS)
  - LIFO order
  - track visited vertices

  → **stack**
  → **set**



Operations

group by / join

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

Vasilis, 2
Rachid, 1

map / atomics

distinct

Vasilis
Rachid
Vasilis

Vasilis
Rachid

hash set

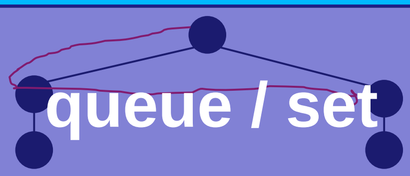
limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

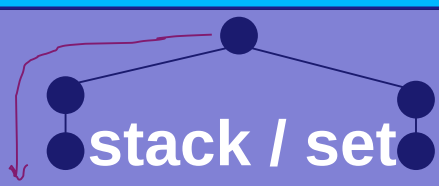
32
23
13

tree / heap / list

BFS

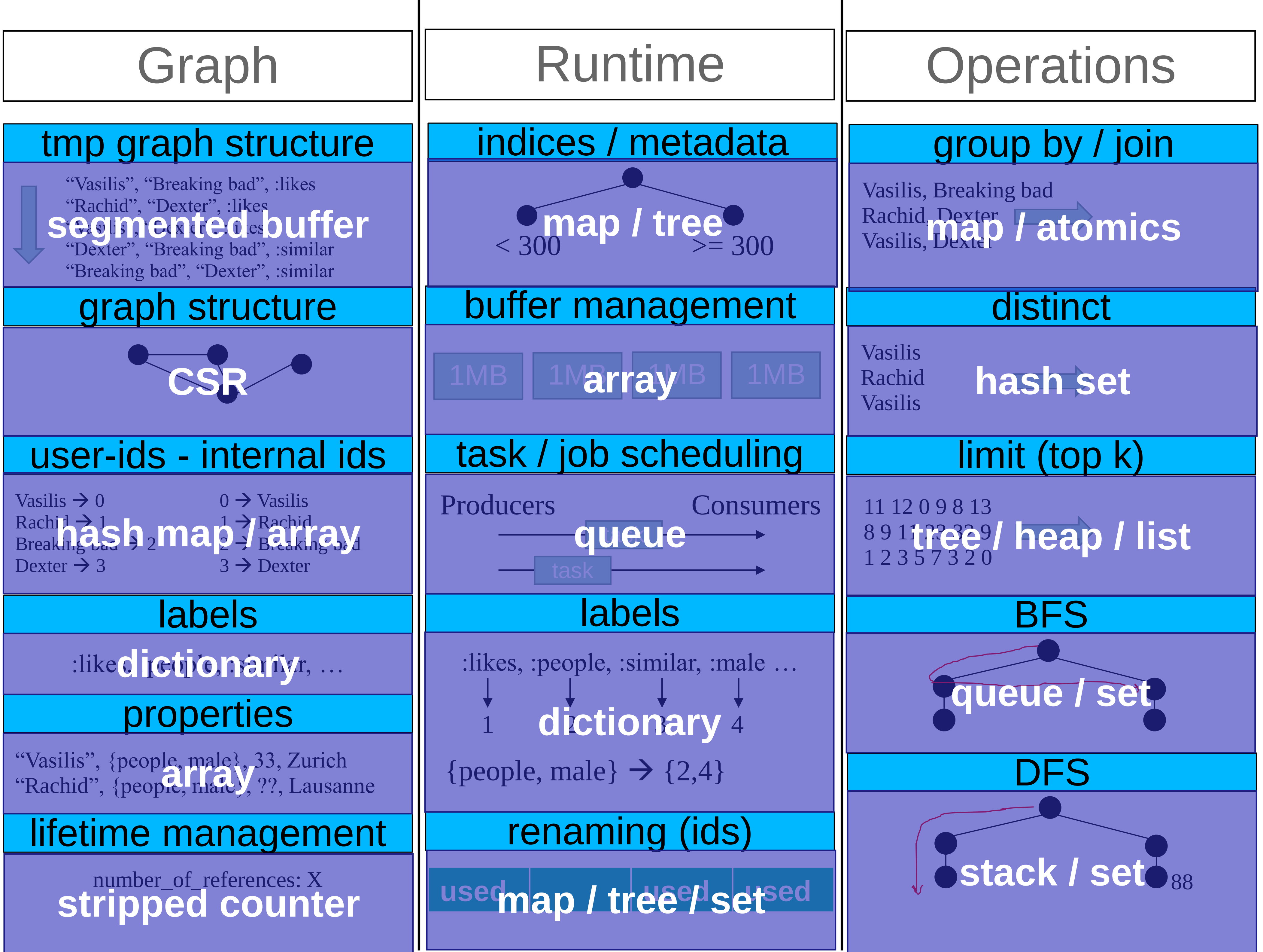


DFS



Score

Structure	# Usages
array / buffer	7
map	6
set	4
tree / heap	3
queue	2
stack	1
list	1


Graph
Runtime
Operations
tmp graph structure
"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
segmented buffer
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar
graph structure
CSR
user-ids - internal ids
Vasilis → 0
0 → Vasilis
hash map / array
Dexter → 3
3 → Dexter
labels
dictionary
properties
"Vasilis", {people, male}, 33, Zurich
array
"Rachid", ??, Lausanne
lifetime management
number_of_references: X
stripped counter
indices / metadata
map / tree
< 300
>= 300
buffer management
1MB
1MB
array
task / job scheduling
Producers
Consumers
queue
task
labels
:likes, :people, :similar, :male …
1
dictionary
4
{people, male} → {2,4}
renaming (ids)
used
map / tree / set
used
group by / join
Vasilis, Breaking bad
map / atomics
distinct
Vasilis
Rachid
Vasilis
hash set
limit (top k)
11 12 0 9 8 13
tree / heap / list
1 2 3 5 7 3 2 0
BFS
queue / set
DFS
stack / set

Conclusions

- Both theory and practice are necessary for
  - Designing, and
  - Implementing fast / scalable data structures
- Hardware plays a huge role on implementations
  - How and which memory access patterns to use
- **(Concurrent) Data structures**
  - **The backbone of every system**
  - **An “open” and challenging area or research**

vasileios.trigonakis@oracle.com – internships++