Concurrent programming: From theory to practice

Concurrent Computing 2024

Vasileios Trigonakis
Consulting Member of Technical Staff
Oracle Labs Zurich

09.Dec.2024

From theory to practice

Theoretical (design)	Practical (design)	Practical (implementation)
- Impossibilities - Upper/Lower bounds - Techniques - System models - Correctness proofs	- System models - shared memory - message passing **Finite memory** - Practicality issues - re-usable objects **Performance**	**Hardware** - Which atomic ops - Memory consistency - Cache coherence - Locality **Performance** **Scalability**
↓	↓	↓
Design (pseudo-code)	**Design (pseudo-code, prototype)**	**Implementation (code)**

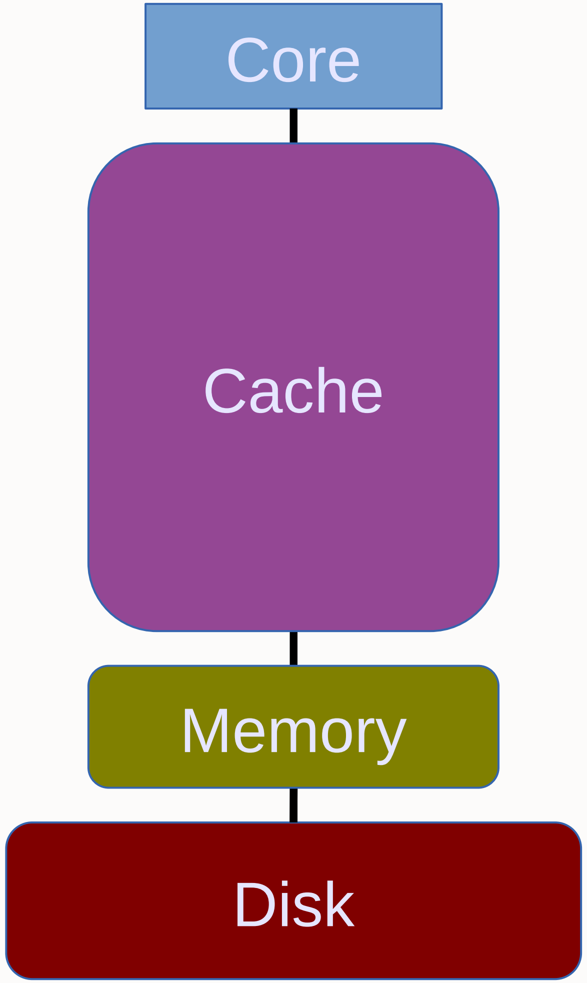
Outline

- 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

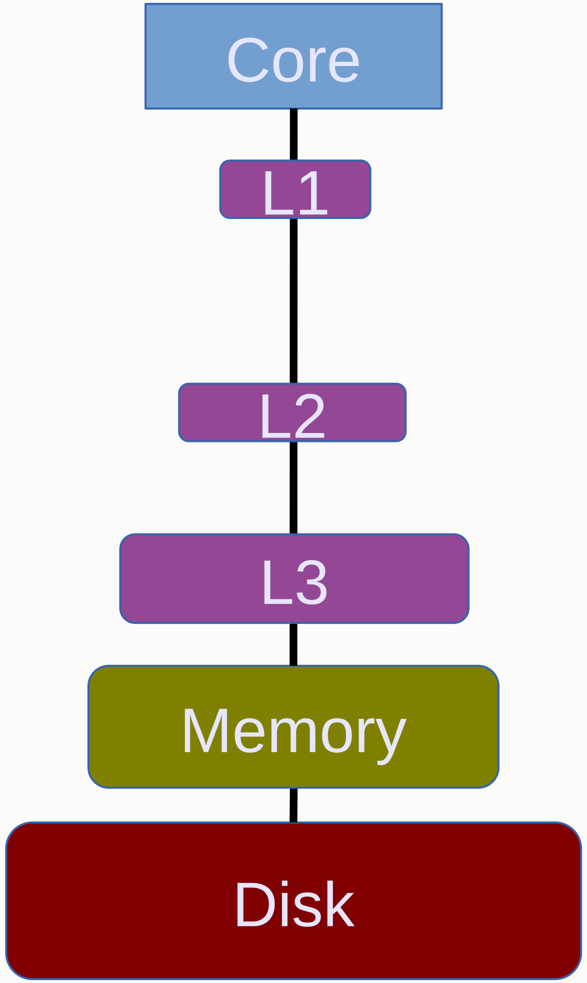
Core → Disk = ~ms

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

From typical server configurations a few years back to the ERA of Gen AI

Intel® Xeon®

- 14 cores @ 2.4GHz
- L1: 32KB
- L2: 256KB
- L3: 40MB
- Memory: 512GB

Intel® Xeon® 6 Processors with P(erformance)-Cores
> 70 cores, > 400MB L3

&

Intel® Xeon® 6 Processors with E(nergy)-Cores
> 60 cores, > 90ML L3

https://www.intel.com/content/www/us/en/products/details/processors/xeon.html

AMD Opteron™

- 18 cores @ 2.4GHz
- L1: 64KB
- L2: 512KB
- L3: 20MB
- Memory: 512GB

AMD EPYC™ 9005 Series

Max config:
192 cores, 384MB L3

&

AMD EPYC™ 9004, 8004, 7003, 4004 Series

https://www.amd.com/en/products/processors/server/epyc.html

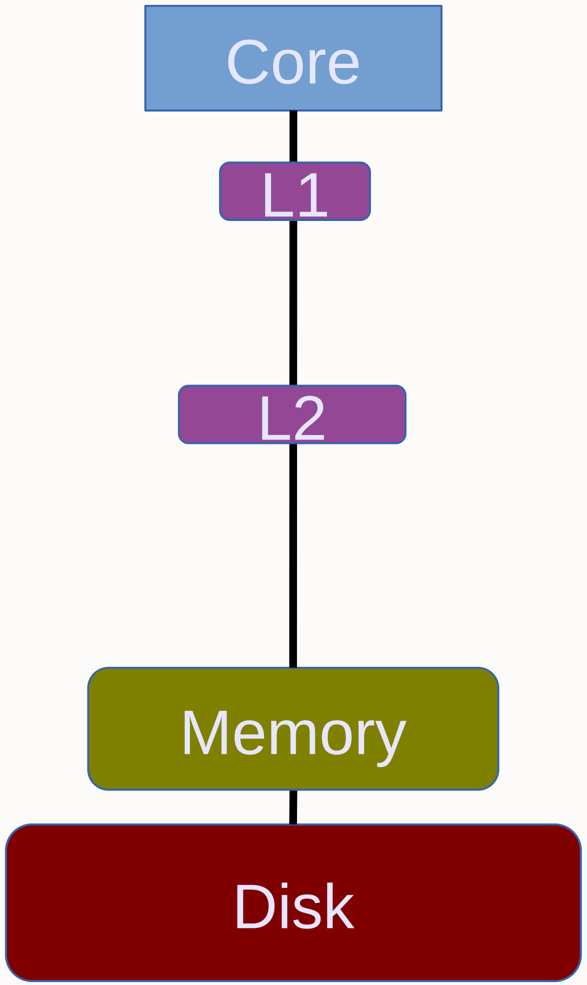
Experiment

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



Single core

Core freq: 3+GHz

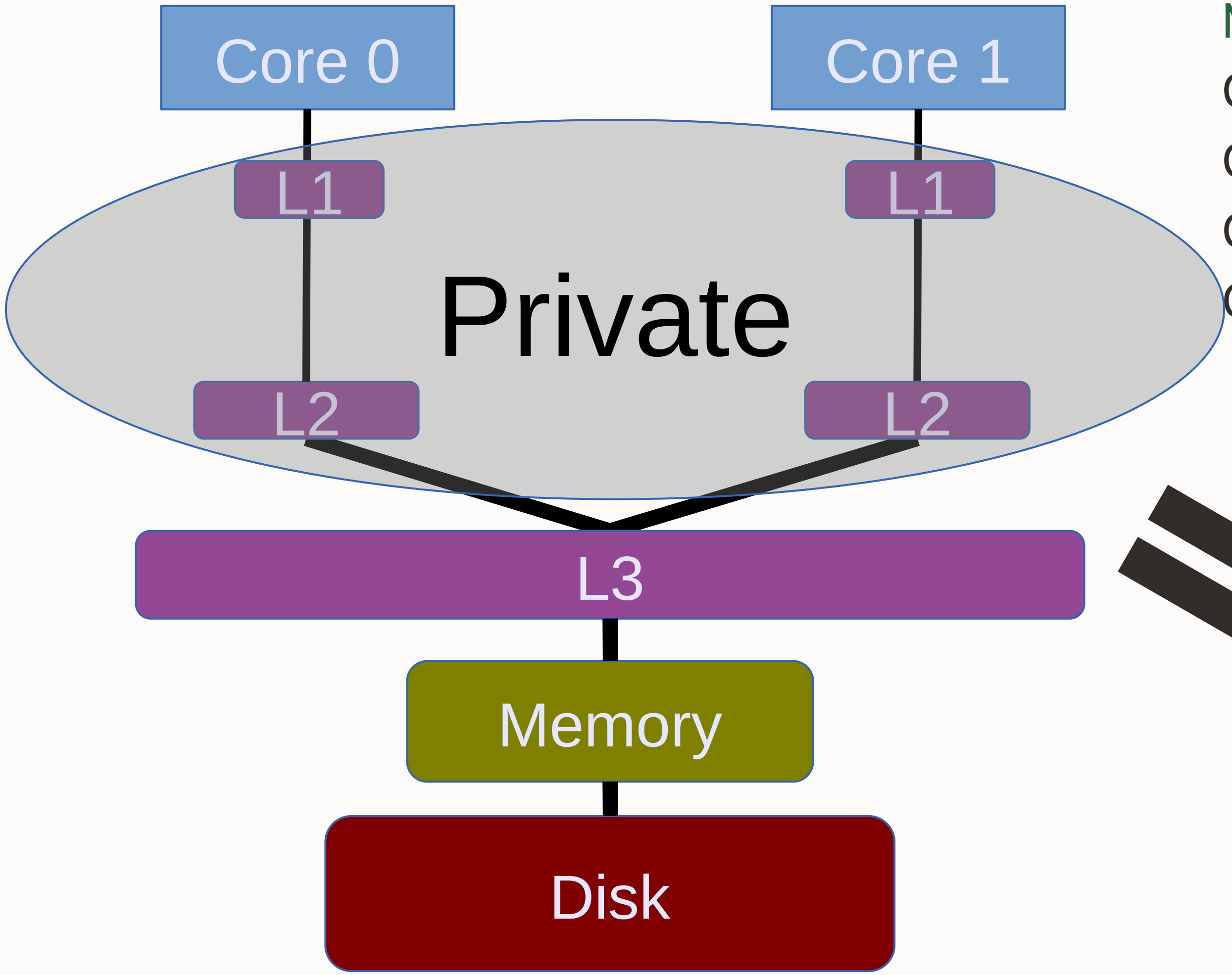
Core → Disk

Core → Memory

Cache

- Core → L2
- Core → L1

After ~2004: multi-cores



Multiple cores

Core freq: ~2GHz

Core → Disk

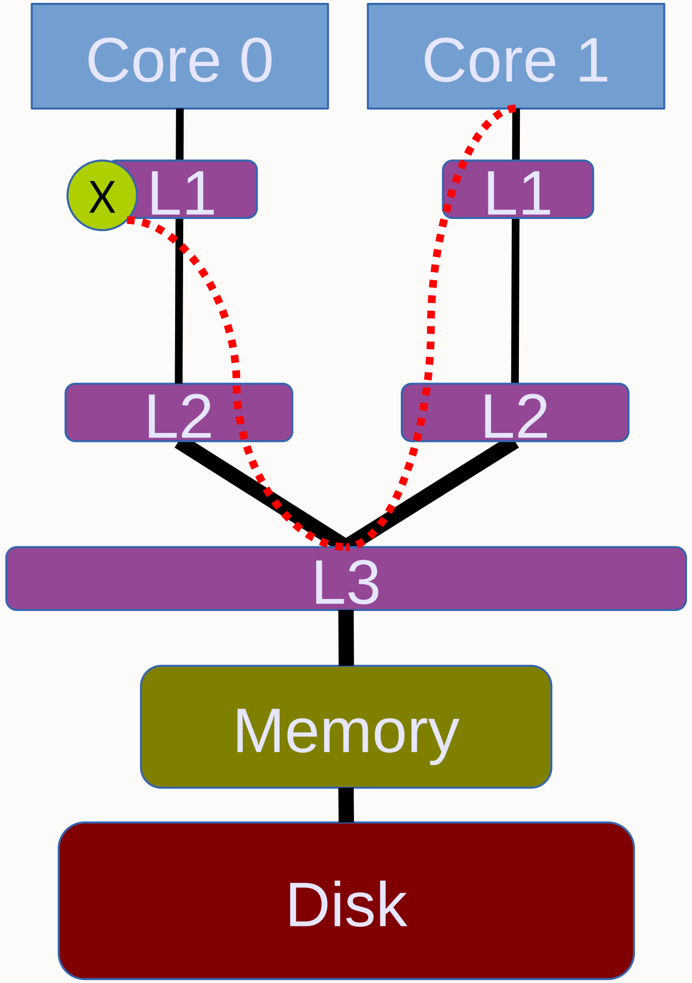
Core → Memory

Cache

- Core → shared L3
- Core → L2
- Core → L1

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?

To perform a **write**

- invalidate all readers, or
- previous writer

To perform a **read**

- find the latest copy

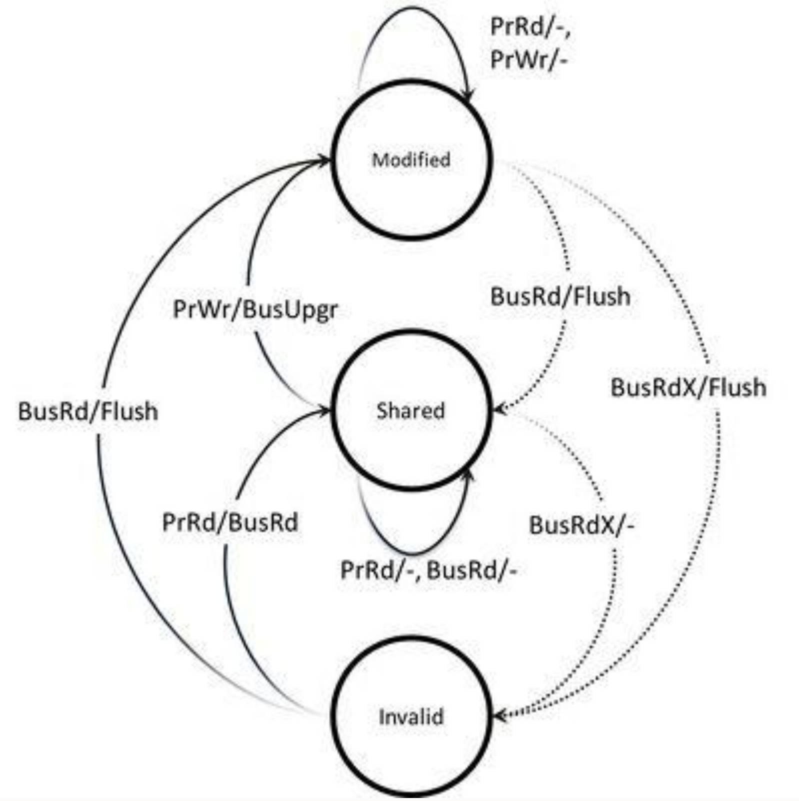
Cache coherence with MESI

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

Which state is our "favorite?"



The ultimate goal for scalability

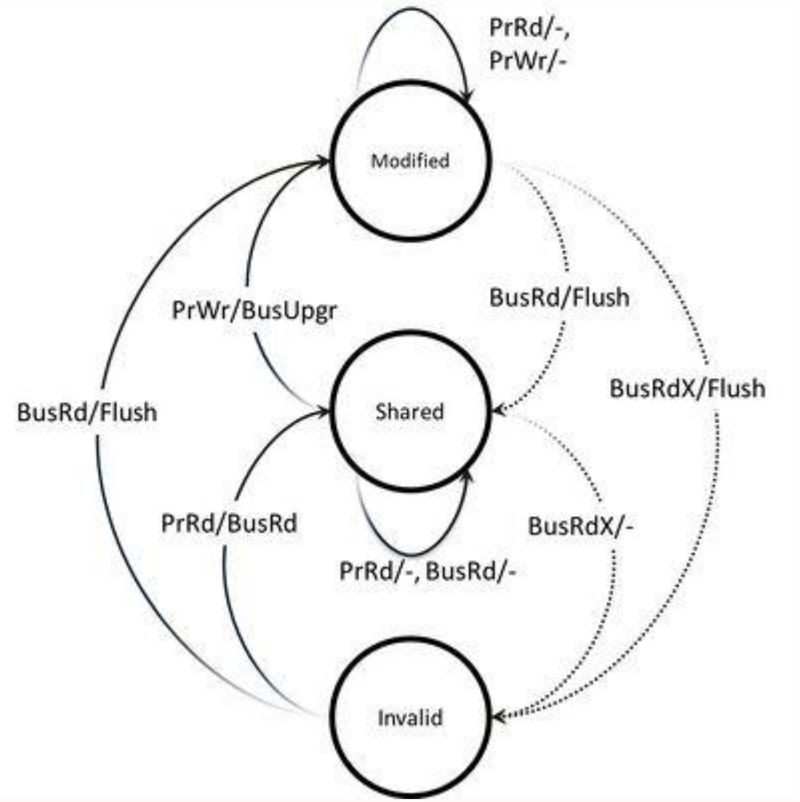
A **state diagram**

State (per cache line)

- **M**odified: the only dirty copy
- **E**xclusive: the only clean copy
- **Shared**: a clean copy
- **I**nvalid: useless data

= threads can keep the data close (L1 cache)

= faster



Experiment

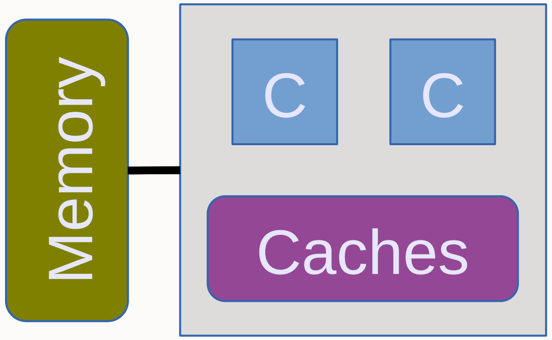
The effects of false sharing

Outline

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

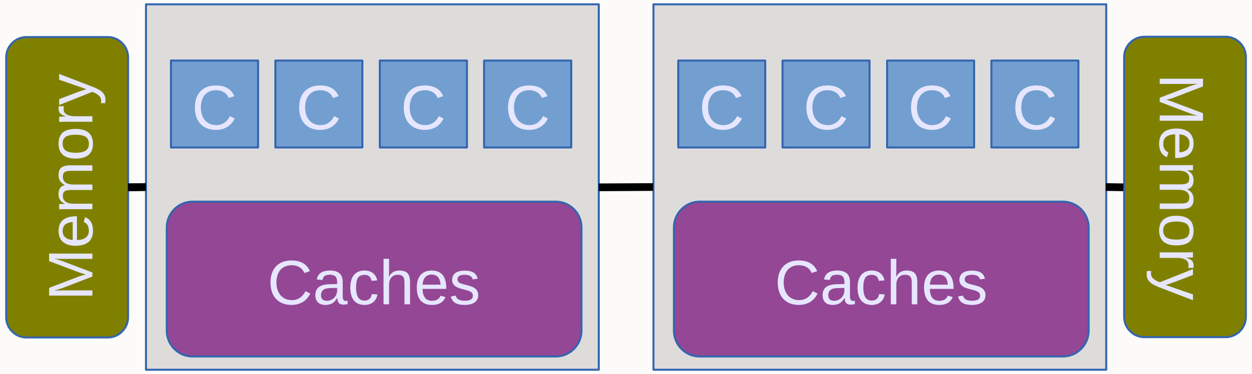
Uniformity vs. non-uniformity

Typical desktop machine



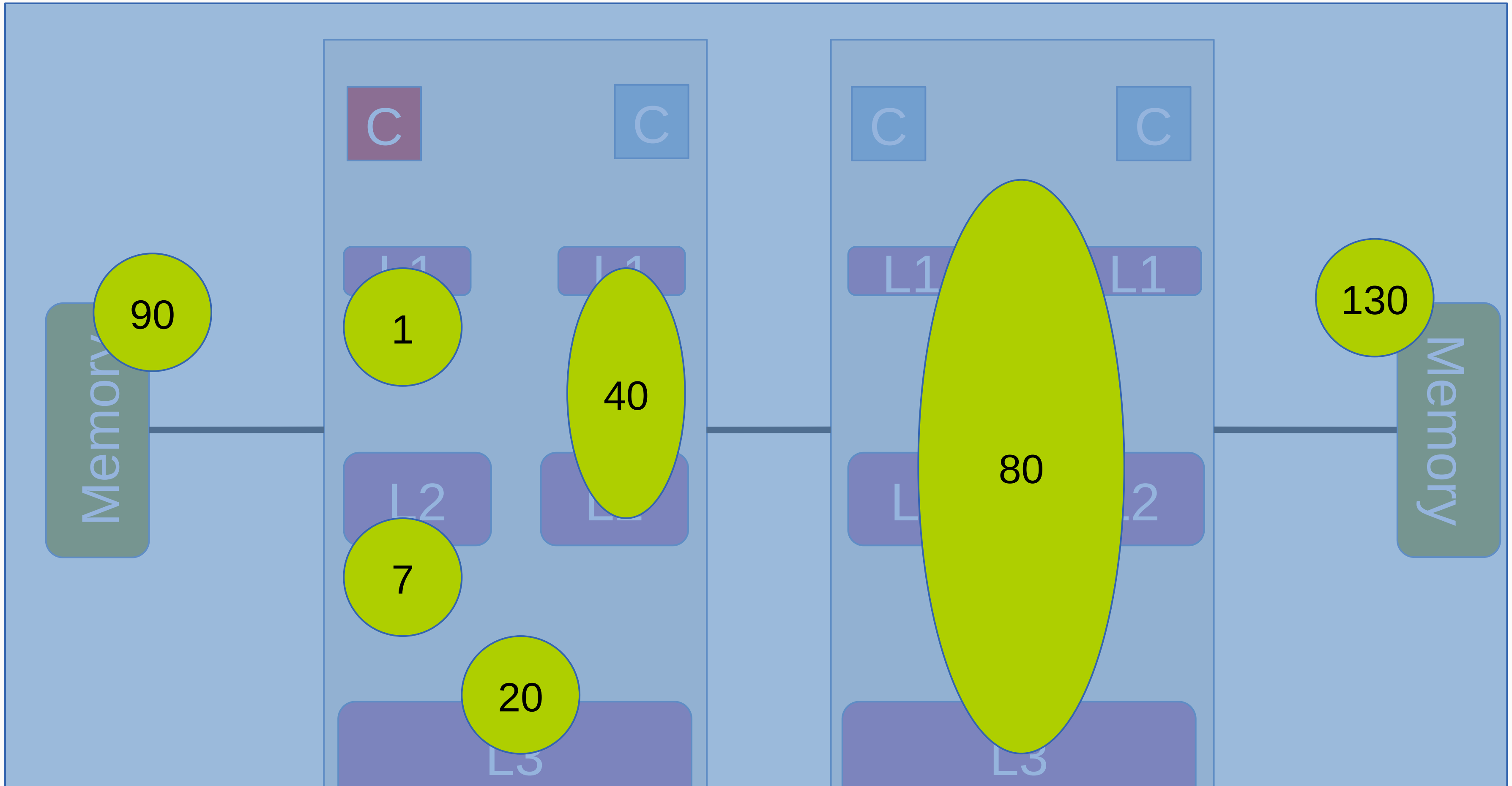
= Uniform

Typical server machine



= non-Uniform
(aka. NUMA)

Latency (ns) to access data in a NUMA multi-core server



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
```

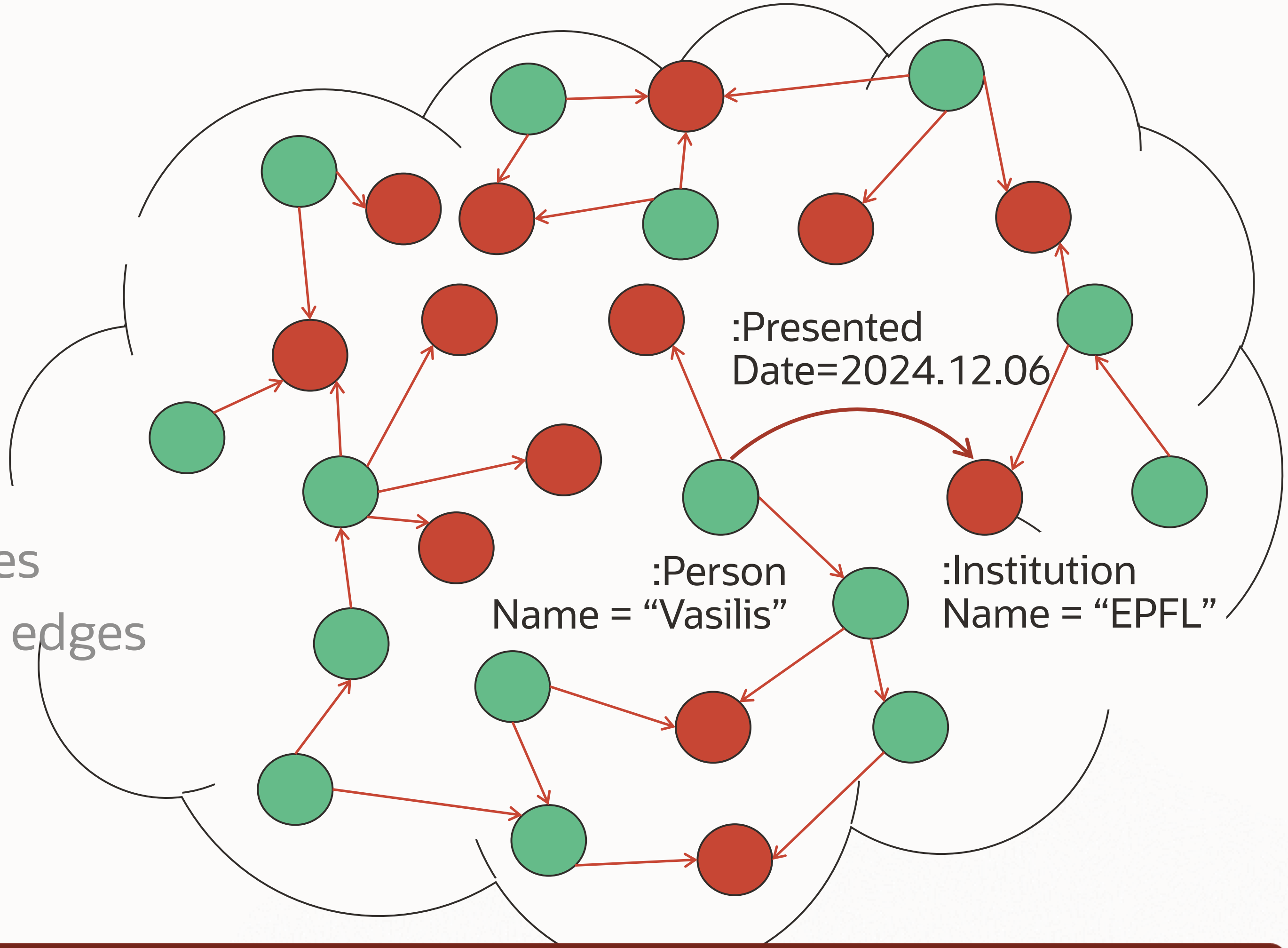
Different memory nodes

Outline

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

Your Data is a Graph!

- Represent it as a property graph
  - Entities are vertices
  - Relationships are edges
- Annotate your graph
  - Labels identify vertices and edges
  - Properties describe vertices and edges
- For the purpose of
  - Data modeling
  - Data analysis



Navigate multi-hop relationships quickly (instead of joins)

Relational (Database) Model → Property Graph Model

users

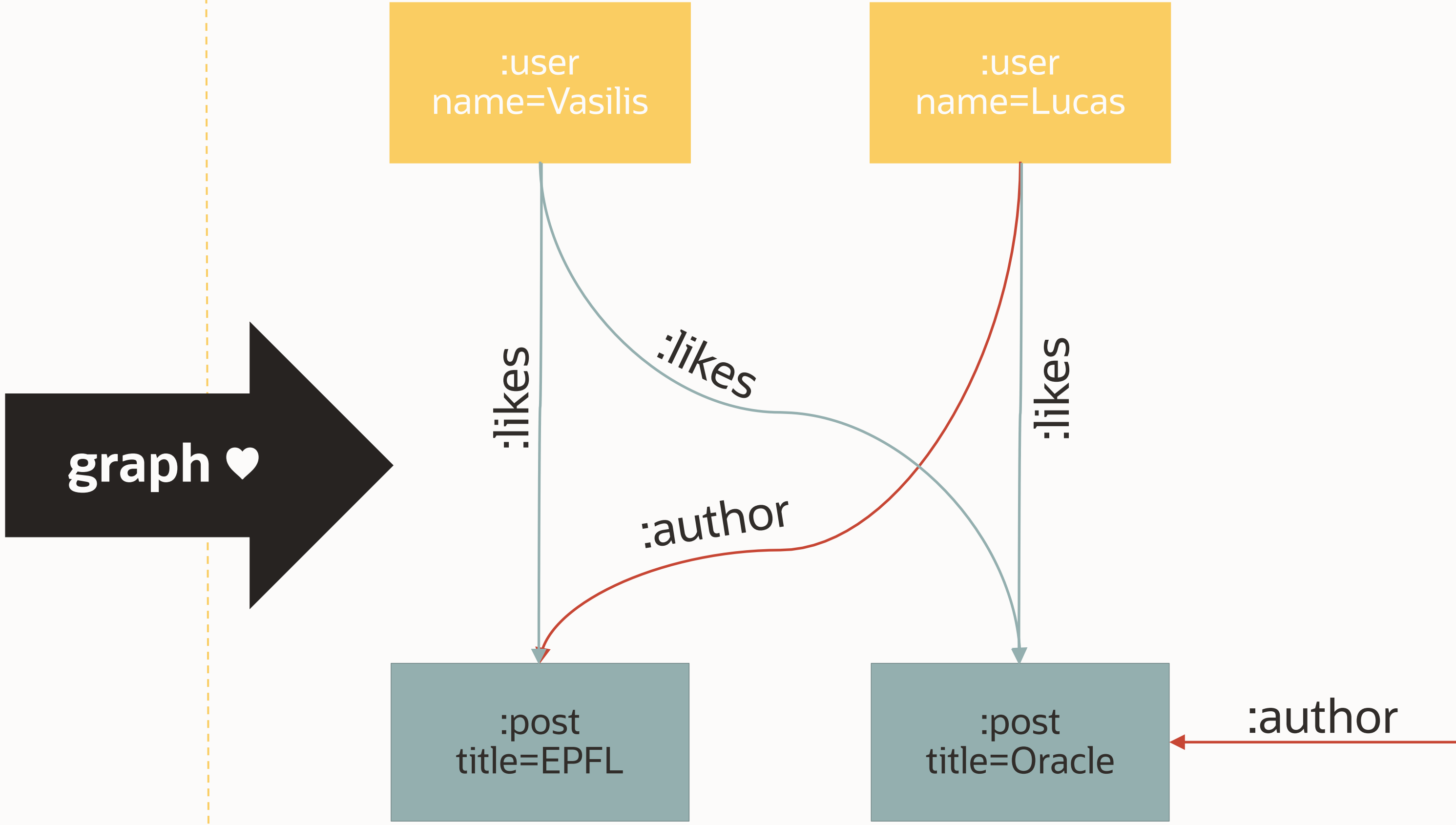
user_id (PK)	name
0	Vasilis
1	Rachid
…	…

user_likes

user_id	post_id
0	0
0	1
1	1

posts

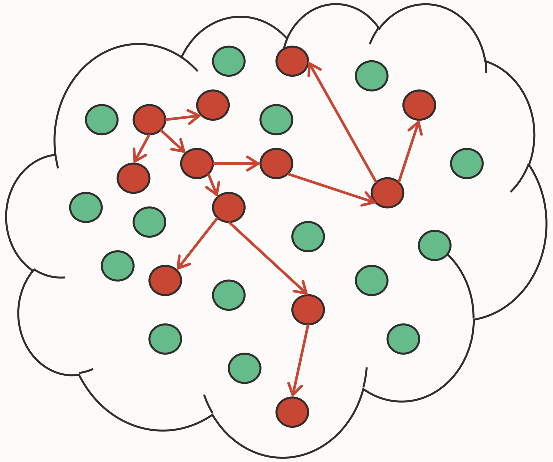
author_id	post_id (PK)	title
1	0	EPFL
123	1	Oracle
…	…	…



Essentially having “materialized joins”

Main Approaches of Graph Processing

1. Computational graph analytics [ASPLOS'12, VLDB'16]
   - Iterate the graph multiple times and compute mathematical properties using Greenmarl / PGX Algorithm (e.g., Pagerank)
   - e.g, `graph.getVertices().forEach(n -> …)`
2. Graph querying and pattern matching [GRADES'16/23, VLDB'16, Middleware Ind. 23]
   - Query the graph using PGQL or SQL/PGQ to find sub-graphs that match to the given relationship pattern
   - e.g., `SELECT … MATCH (a) -[edge]-> (b) …`
3. Graph ML
   - Use the structural information latent in graphs
   - e.g., graph similarity



$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

4. Vector similarity graph indices
   - Hierarchical navigable small world (HNSW)
5. Graph RAG
   - Retrieval-Augmented Generation (RAG)
   - Enhancing RAG with knowledge graphs

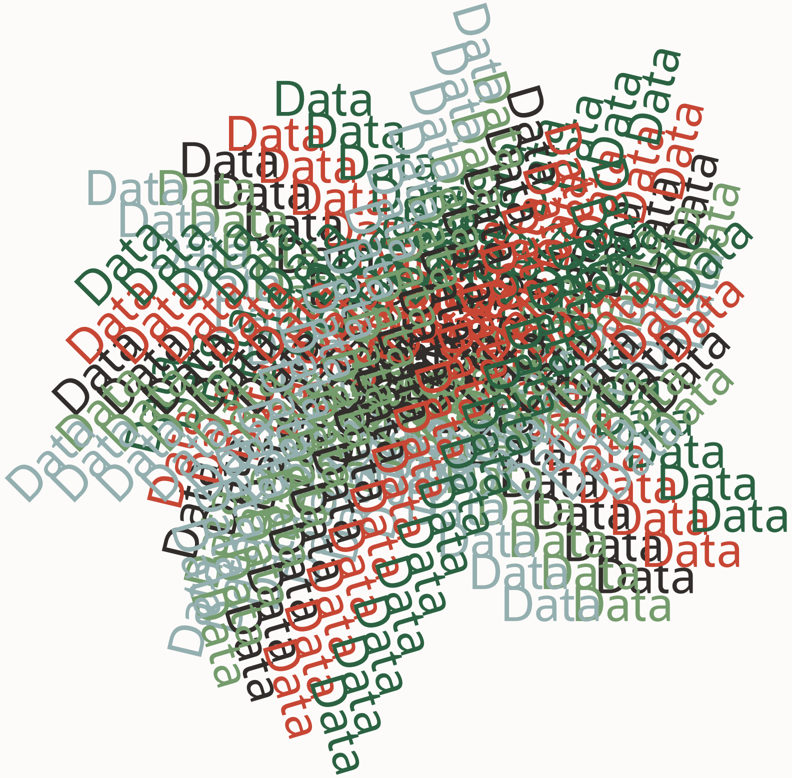
Dissecting a graph processing system

with a focus on (concurrent) data structures

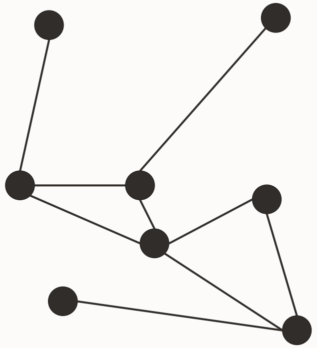
Dissecting a graph processing system and preparing for a job interview

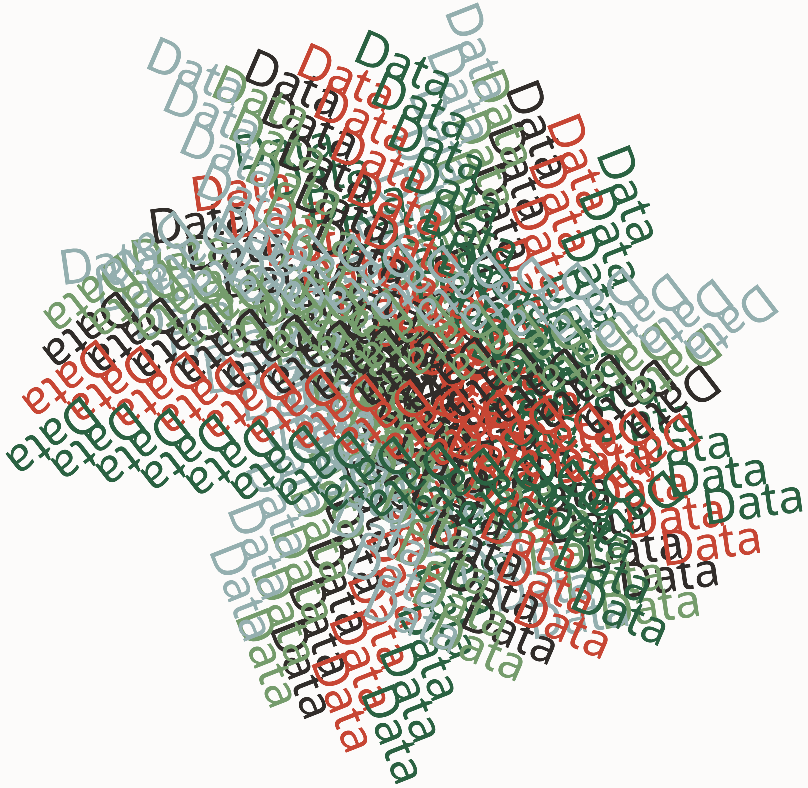
with a focus on (concurrent) data structures

Architecture of a graph processing system



Graph





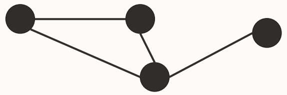
Tons of other data and metadata to store

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
Rachid → 1
Breaking bad → 2
Dexter → 3

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

labels

:likes, :people, :similar, …

properties

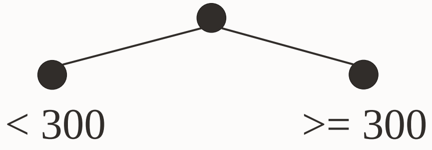
"Vasilis", {people, male}, 20, Zurich
"Rachid", {people, male}, ??, Lausanne

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

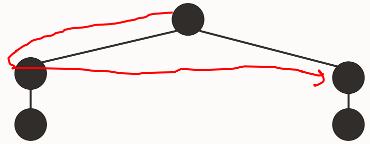
limit (top k)

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

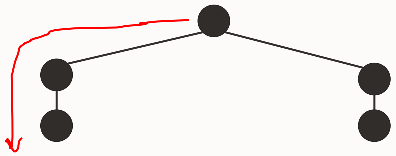
→

32
23
13

BFS



DFS

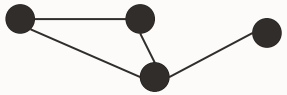


Graph

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}, 20, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X

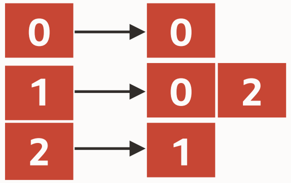
- tmp graph structure
  - append only
  - dynamic schema

  → **dataframe** = segmented buffer

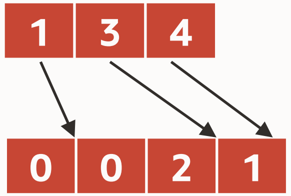
- Classic graph structures
  1. adjacency matrix

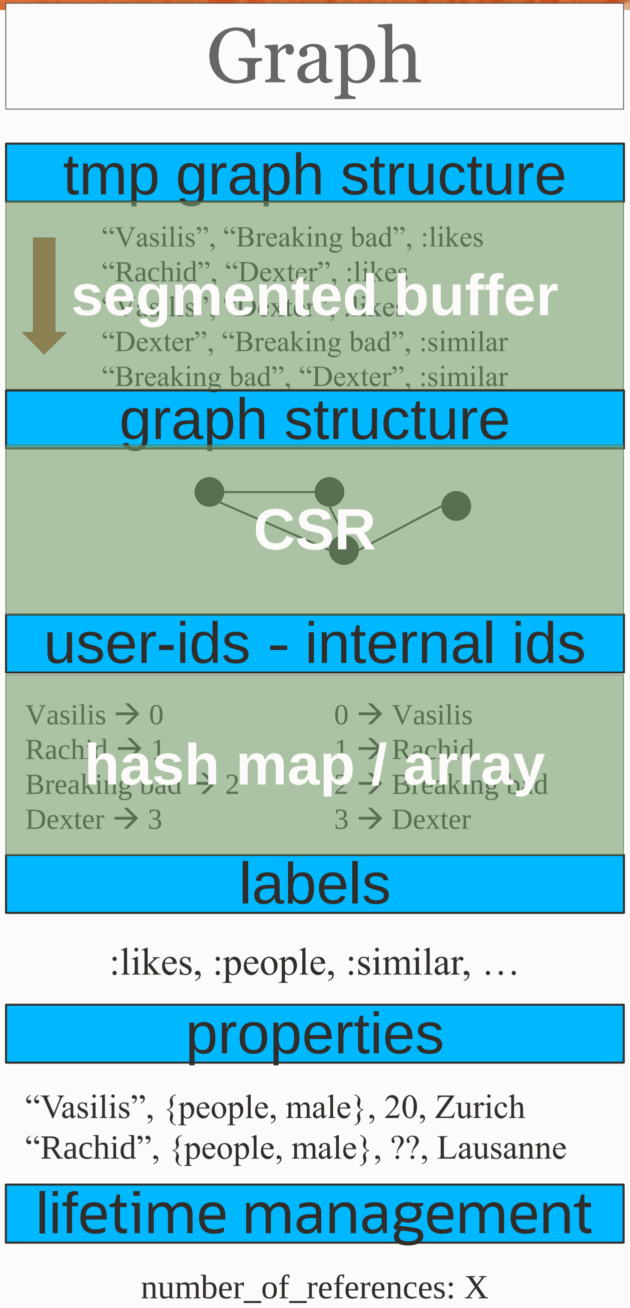
	0	1	2
0	x		
1	x		x
2		x	

  2. adjacency list



  3. compressed source row (CSR)





- 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

Graph

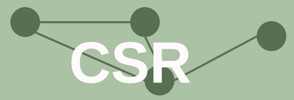
tmp graph structure

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

dictionary

:likes, :people, :similar, …

properties

“Vasilis”, {people, male}, 20, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X

- 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

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
Rachid → 1
Breaking bad → 2
Dexter → 3

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

labels

dictionary (= map)

:likes, :people, :similar

properties

array

“Vasilis”, {people, male}, 20, Zurich
“Rachid”, {people, male}, ??, Lausanne

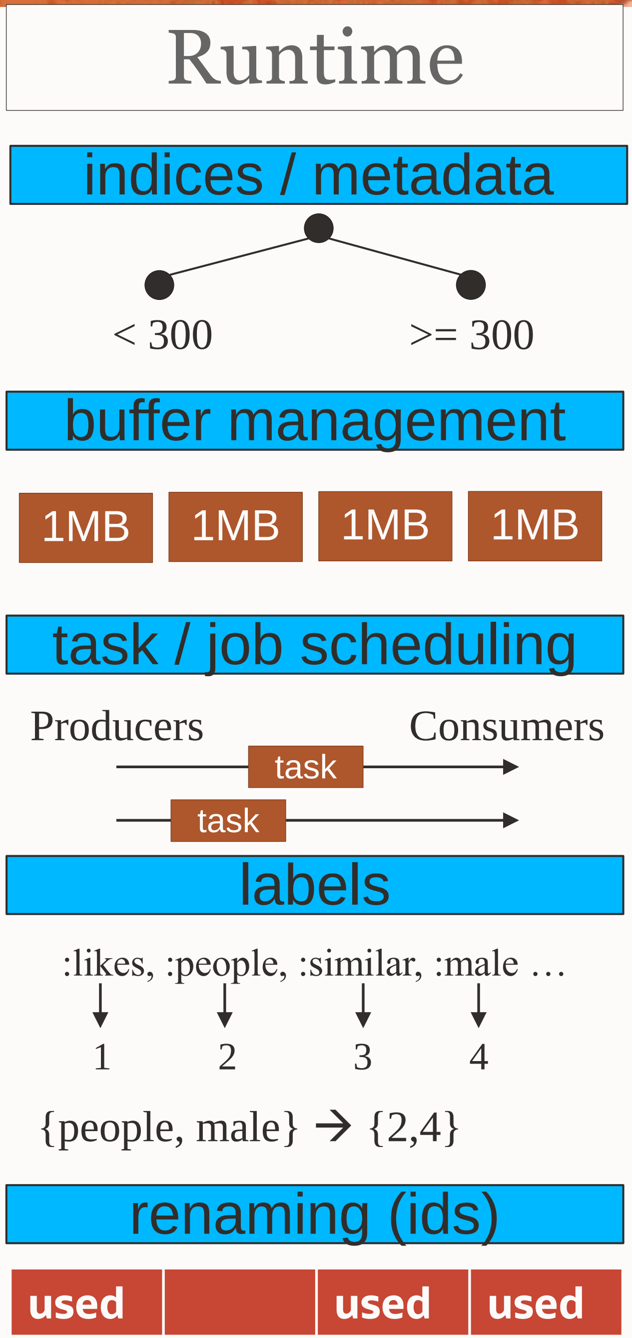
lifetime management

stripped counter

number_of_references: X

Score

Structure	# Usages
array / buffer	5
map	2



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

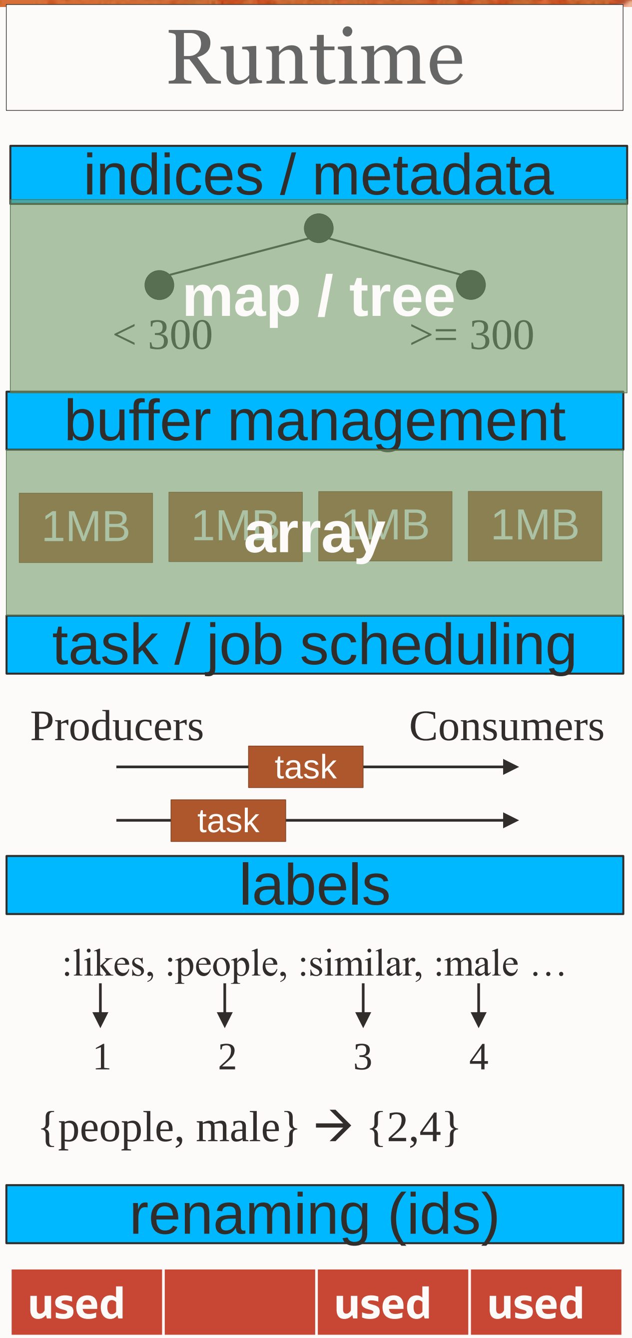
  → **maps, trees**

- 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

  → **queues**

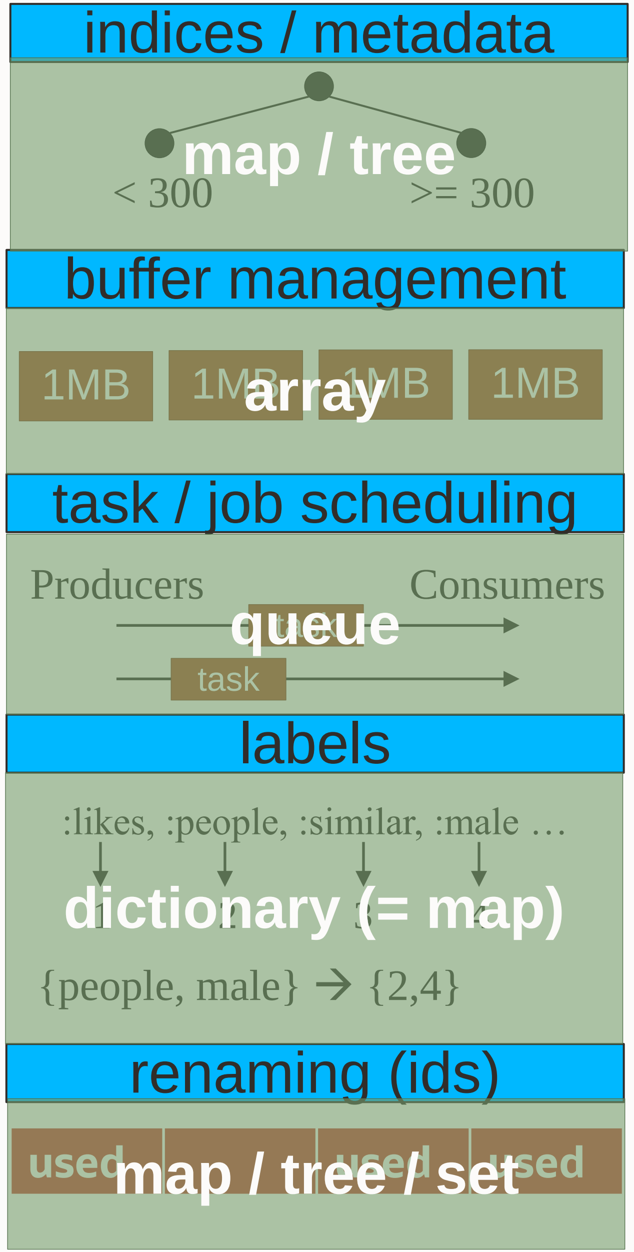
- Storing / querying sets of labels
  - set equality expensive
  - usually common groups
    e.g., {person, female}, {person, male}

  → 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

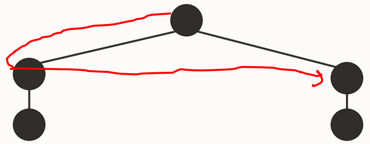
distinct

Vasilis
Rachid
Vasilis
→
Vasilis
Rachid

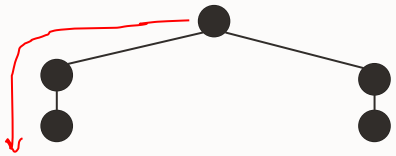
limit (top k)

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

  → **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

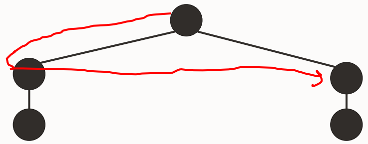
limit (top k)

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

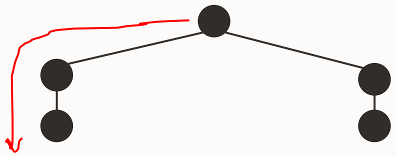
→

32
23
13

BFS

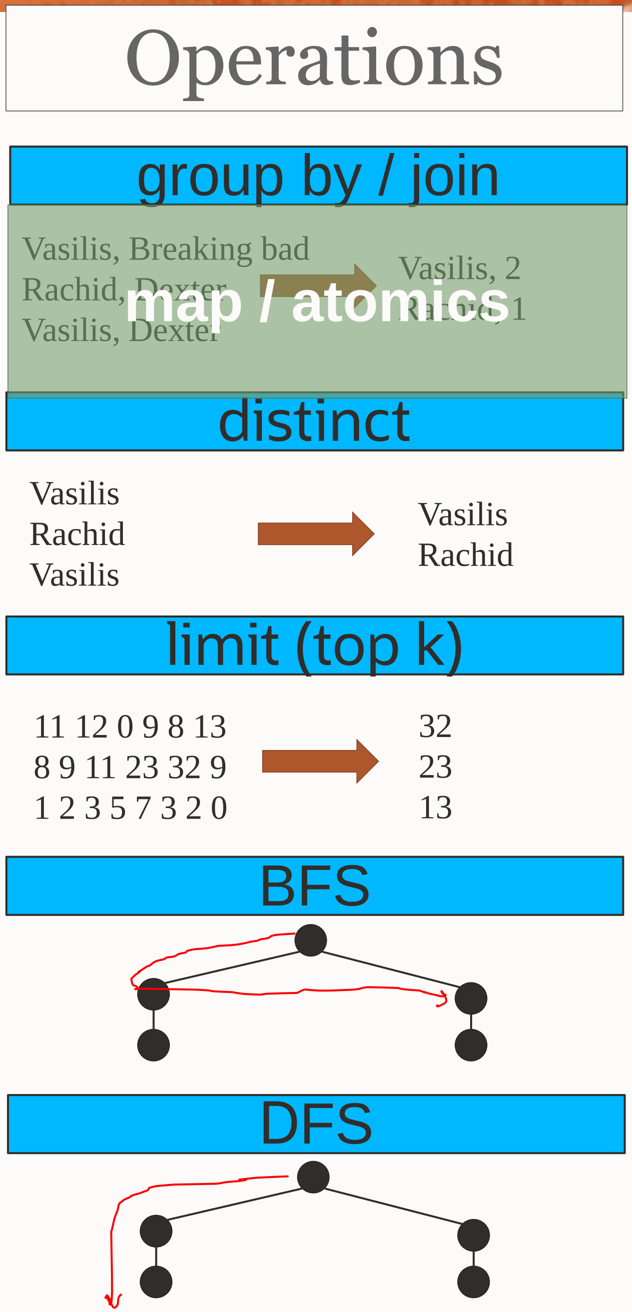


DFS



Distinct

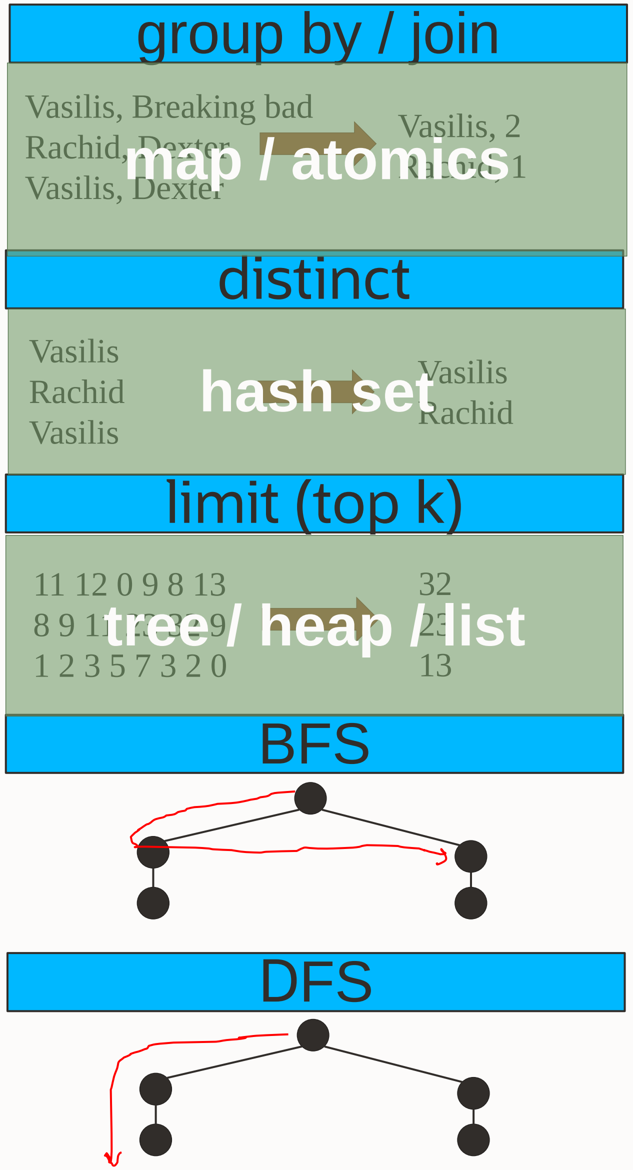
- can be solved with sorting, or



- 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)

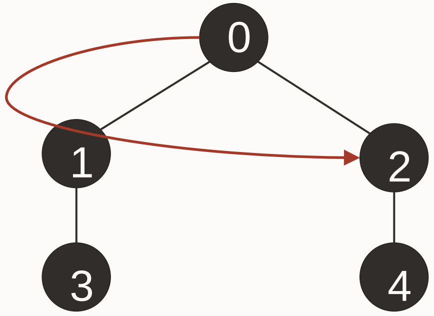
Operations



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

  → **queue**

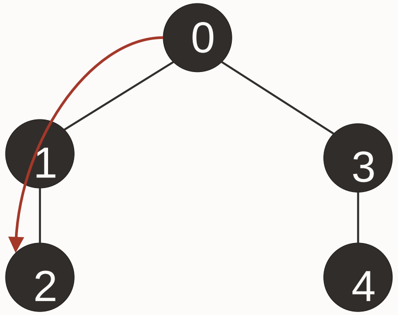
  → **set**



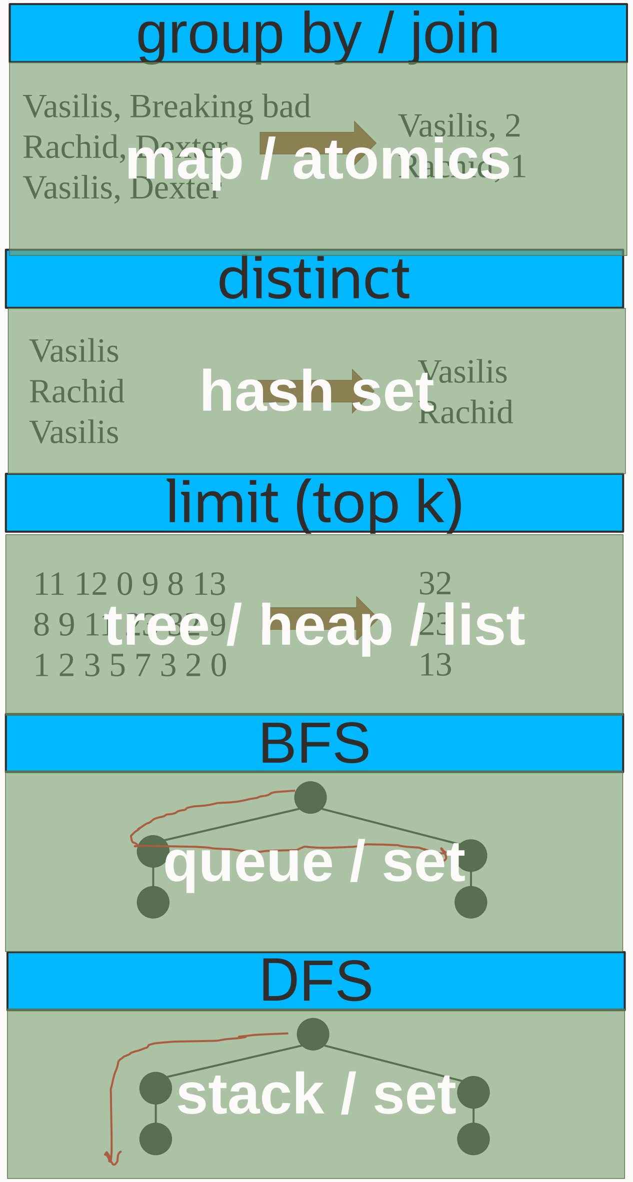
- Depth-first search (DFS)
  - LIFO order
  - track visited vertices

  → **stack**

  → **set**



Operations



Score

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

Your new cheatsheet for interviews!

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
Rachid → 1
Breaking bad → 2
Dexter → 3

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

labels
dictionary

:likes, :people, :similar, …

properties
array

"Vasilis", {people, male}, 20, Zurich
"Rachid", {people, male}, ??, Lausanne

lifetime management
stripped counter

number_of_references: X

Runtime

indices / metadata
map / tree

< 300 >= 300

buffer management
array

1MB 1MB 1MB 1MB

task / job scheduling
queue

Producers Consumers
task
task

labels
dictionary

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

{people, male} → {2,4}

renaming (ids)
map / tree / set

used used used used

Operations

group by / join
map / atomics

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter

distinct
hash set

Vasilis
Rachid
Vasilis

limit (top k)
tree / heap / list

11 12 0 9 8 13
8 9 11 12 22 9
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

Identify, **explore**, and **transfer** new technologies that have the potential to substantially improve Oracle's business.

Oracle Labs Mission Statement

Internship and job opportunities

Visit the Oracle Labs Internship Page: labs.oracle.com/pls/apex/labs/r/labs/internships

or find our topics in EPFL's portal

- Automated Machine Learning with Explainability (AutoMLx)
- Cloud Security at Oracle
- Extending a Distributed Graph Engine (Oracle Labs PGX)
- Extending Application Dependency Management Cloud Service
- Extending a Web-Based Enterprise Data Science Platform
- Extending Oracle (Labs) Security and Compliance Applications
- Graph Machine Learning at Oracle
- Graph Support in the Oracle Database
- LLMs for Assistants and Code Generation at Oracle
- Machine Learning for Data Integration
- Machine Learning in Database Systems
- Oracle Database Multilingual Engine - Modern Programming Languages in the Database

or just send us an email at epfl-labs_ch@oracle.com

Using the Oracle Cloud for free

Everybody
Oracle Cloud Always-Free Tier: oracle.com/cloud/free/

Universities and Schools
Oracle Academy: academy.oracle.com

Research Institutions
Oracle For Research: oracle.com/oracle-for-research/

Our mission is to help people
see data in new ways, discover insights,
unlock endless possibilities.