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

Practical (design)
- System models
  - shared memory
  - message passing
- Finite memory
- Practicality issues
  - re-usable objects
- Performance

Practical (implementation)

Design (pseudo-code)

Design (pseudo-code, prototype)
From theory to practice

Theoretical (design)
- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs

Practical (design)
- System models
  - shared memory
  - message passing
- Finite memory
- Practicality issues
  - re-usable objects
- Performance

Practical (implementation)
- 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
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

Core 0
  
  L1
  
  L2
  
  L3
  
  Memory
  
  Disk

Core 1
  
  L1
  
  L2
  
  L3
  
  Memory
  
  Disk

Private

= 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)
  - **Modified**: the only dirty copy
  - **Exclusive**: the only clean copy
  - **Shared**: a clean copy
  - **Invalid**: useless data
The ultimate goal for scalability

● Possible states
  - Modified: the only dirty copy
  - Exclusive: the only clean copy
  - Shared: a clean copy
  - Invalid: useless data

● Which state is our “favorite”? 
The ultimate goal for scalability

- Possible states
  - **Modified**: the only dirty copy
  - **Exclusive**: the only clean copy
  - **Shared**: a clean copy
  - **Invalid**: 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
Latency (ns) to access data

Memory

C
L1
L2
L3

C
L1
L2
L3

C
L1
L2
L3

C
L1
L2
L3

Memory
Latency (ns) to access data
Latency (ns) to access data

- L1: 1 ns
- L2: 7 ns
- L3:
Latency (ns) to access data

C
L1
L2
L3

C
L1
L2

C
L1
L2

Memory

Memory
Latency (ns) to access data
Latency (ns) to access data

Memory

<table>
<thead>
<tr>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>20</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

Memory
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

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
Outline

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

## Relational view

<table>
<thead>
<tr>
<th>Name</th>
<th>Likes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>Breaking bad</td>
</tr>
<tr>
<td>Rachid</td>
<td>Dexter</td>
</tr>
<tr>
<td>Vasilis</td>
<td>Dexter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Similar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breaking bad</td>
<td>Dexter</td>
</tr>
<tr>
<td>Dexter</td>
<td>Breaking bad</td>
</tr>
</tbody>
</table>
Graph processing

Relational view

<table>
<thead>
<tr>
<th>Name</th>
<th>Likes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>Breaking bad</td>
</tr>
<tr>
<td>Rachid</td>
<td>Dexter</td>
</tr>
<tr>
<td>Vasilis</td>
<td>Dexter</td>
</tr>
</tbody>
</table>

Graph view

<table>
<thead>
<tr>
<th>Name</th>
<th>Similar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breaking bad</td>
<td>Dexter</td>
</tr>
<tr>
<td>Dexter</td>
<td>Breaking bad</td>
</tr>
</tbody>
</table>

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
    
    ```sql
    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
Architecture of a graph processing system

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 \rightarrow 0 \quad 0 \rightarrow Vasilis
Rachid \rightarrow 1 \quad 1 \rightarrow Rachid
Breaking bad \rightarrow 2 \quad 2 \rightarrow Breaking bad
Dexter \rightarrow 3 \quad 3 \rightarrow 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

tmp graph structure

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 Consumers

labels
:likes, :people, :similar, :male ...

renaming (ids)

used used used used

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**
  - < 300
  - >= 300

- **buffer management**
  - 1MB 1MB 1MB 1MB

**Operations**

- **group by / join**
  - Vasilis, Breaking bad
  - Rachid, Dexter
  - Vasilis, Dexter
  - Rachid, 1

- **distinct**
  - Vasilis
  - Rachid
  - 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
  - append only
  - dynamic schema

Graph structure

User-ids - internal ids

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

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

<table>
<thead>
<tr>
<th>User</th>
<th>Id</th>
<th>Internal Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rachid</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Breaking bad</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Dexter</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**labels**
- :likes, :people, :similar, ...

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

**lifetime management**

number_of_references: X
Graph

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

- **Classic graph structures**

### Graph Structure

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

```
0 Vasilis
1 Rachid
2 "Breaking bad"
3 "Dexter"
```

```
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**
- append only
- dynamic schema

→ **segmented table**

**graph structure**

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

**user-ids - internal ids**

<table>
<thead>
<tr>
<th>User</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>0</td>
</tr>
<tr>
<td>Rachid</td>
<td>1</td>
</tr>
<tr>
<td>Breaking bad</td>
<td>2</td>
</tr>
<tr>
<td>Dexter</td>
<td>3</td>
</tr>
</tbody>
</table>

**labels**

:likes, :people, :similar, ...

**properties**

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

**lifetime management**

number_of_references: X

### Classic graph structures

1. **connectivity matrix**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

2. **adjacency list**

```
0 → 0
1 → 0 2
2 → 1
```

3. **compressed source row (CSR)**

```
1 3 4
0 0 2 1
```
• Mapping user ids to internal ids
  • create once
  • read-only after

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
Graph

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

**tmp graph structure**

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

**user-ids - internal ids**

<table>
<thead>
<tr>
<th>User</th>
<th>Internal ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>0</td>
</tr>
<tr>
<td>Rachid</td>
<td>1</td>
</tr>
<tr>
<td>Breaking bad</td>
<td>2</td>
</tr>
<tr>
<td>Dexter</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Internal ID</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Vasilis</td>
</tr>
<tr>
<td>1</td>
<td>Rachid</td>
</tr>
<tr>
<td>2</td>
<td>Breaking bad</td>
</tr>
<tr>
<td>3</td>
<td>Dexter</td>
</tr>
</tbody>
</table>

**labels**

:likes, :people, :similar, …

**properties**

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

**lifetime management**

number_of_references: X
**Graph**

- 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]

---

**tmp graph structure**

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

---

**graph structure**

```
CSR
```

---

**user-ids - internal ids**

<table>
<thead>
<tr>
<th>User</th>
<th>Internal Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>0</td>
</tr>
<tr>
<td>Rachid</td>
<td>1</td>
</tr>
<tr>
<td>Breaking bad</td>
<td>2</td>
</tr>
<tr>
<td>Dexter</td>
<td>3</td>
</tr>
</tbody>
</table>

```
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
Graph

- Mapping user ids to internal ids
  - create once
  - read-only after
  \(\rightarrow\) hash map, lock-free reads

- Mapping internal ids to user ids
  - create once
  - read-only after
  - fixed key range: \([0, N]\)
  \(\rightarrow\) (sequential) array

### tmp graph structure

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

### segmented buffer

### graph structure

 CSR

### user-ids - internal ids

<table>
<thead>
<tr>
<th>Name</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>0</td>
</tr>
<tr>
<td>Rachid</td>
<td>1</td>
</tr>
<tr>
<td>Breaking bad</td>
<td>2</td>
</tr>
<tr>
<td>Dexter</td>
<td>3</td>
</tr>
</tbody>
</table>

### 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” $\rightarrow$ ~ 7 bytes
  • comparing strings is expensive

user-ids - internal ids

Vasilis $\rightarrow$ 0
Rachid $\rightarrow$ 1
Breaking bad $\rightarrow$ 2
Dexter $\rightarrow$ 3

hash map / array

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
  - “Dexter”, “Breaking bad”, :similar
- **graph structure**: segmented buffer
- **user-ids - internal ids**
  - Vasilis \(\to 0\)
  - Rachid \(\to 1\)
  - Breaking bad \(\to 2\)
  - Dexter \(\to 3\)
- **hash map / array**
  - Vasilis \(\to 0\)
  - Rachid \(\to 1\)
  - Breaking bad \(\to 2\)
  - Dexter \(\to 3\)
- **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” \(\to\) \(\sim 7\) bytes
- comparing strings is expensive
  - \(\rightarrow\) **dictionary encoding**, e.g.,
    - person \(\to 0\)
    - female \(\to 1\)
    - male \(\to 2\)

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

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

user-ids - internal ids

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

labels
dictionary
properties

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

lifetime management

number_of_references: X
Graph

- **Property**
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges → (sequential) arrays

```
user-ids - internal ids
Vasilis → 0
Rachid → 1
Breaking bad → 2
Dexter → 3

labels
dictionary
properties
“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne
```

number_of_references: X
Graph

- **Property**
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges
  \( \rightarrow \) (sequential) arrays

- **Lifetime management**
  - cache coherence: atomic counters can be expensive

### Graph structure

- **Property**
  - :likes
  - :similar

- **Segmented buffer**
  - CSR

- **user-ids - internal ids**

- **Hash map / array**

- **Dictionary**

- **Lifetime management**
  - number_of_references: X

### Properties

- \("Vasilis", \{people, male\}, 33, Zurich\)
- \("Rachid", \{people, male\}, ??, Lausanne\)
Graph

- **Property**
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges
    $\rightarrow$ (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:**

```c
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;
}
```

**CSR**

**Segmented buffer**

**User-ids - Internal ids**

```
Vasilis $\rightarrow$ 0
Rachid $\rightarrow$ 1
Breaking bad $\rightarrow$ 2
Dexter $\rightarrow$ 3
0 $\rightarrow$ Vasilis
1 $\rightarrow$ Rachid
2 $\rightarrow$ Breaking bad
3 $\rightarrow$ Dexter
```

**Labels**

```
:likes, :people, :similar, ...
```

**Dictionary**

```
"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
"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

Dictionary (= map)

Labels

Properties

Dictionary (= map)

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

Array

Lifetime management

Number of references: X

Structure

# Usages

array / buffer
5

map
2

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

---

Indices
- :likes, :people, :similar, :male ...
- {people, male} → {2,4}

Buffer Management
- < 300
- ≥ 300

Task / Job Scheduling
- Producers
- Consumers

Labels
- :likes, :people, :similar, :male ...
- 1 → 2 → 3 → 4

Runtime
- Indices / Metadata
- 1MB
- Used

Renaming (ids)
- used
- used
- used
• 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

labels
:likes, :people, :similar, :male …

renaming (ids)
used used used

Producers

Consumers

map / tree
< 300
>= 300

Runtime

indices / metadata

buffer management

array

1MB 1MB 1MB 1MB
- 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., \{\text{person, female}\}, \{\text{person, male}\}
  → 2-level \textbf{dictionary} encoding
  • \{\text{person, female}\} → 0
  • \{\text{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\} \rightarrow 0
  - \{person, male\} \rightarrow 1

- Giving unique ids (renaming)
  -> tree, map, set, counter, other...
buffer management

indices / metadata

map / tree

< 300

>= 300

array

task / job scheduling

Producers

Consumers

queue

dictionary (= map)

{people, male} → {2,4}

renaming (ids)

map / tree / set

labels

:likes, :people, :similar, :male ...

map / tree

Runtime

Structure

# Usages

array / buffer

6

map

5

tree / heap

2

set

1

queue

1
- **Group by**
  1. Mapping from keys to values
  2. Atomic value aggregations e.g., COUNT, SUM, MAX
- insertion only
- **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.
**Group by**
1. Mapping from keys to values
2. Atomic value aggregations e.g., COUNT, SUM, MAX
   - insertion only
   → hash map
   → atomic $\text{inc} / \text{sum} / \text{max}$, etc.

**Join**
- create a map of the small table
- insertion phase, followed by
- probing phase
- **Group by**
  1. Mapping from keys to values
  2. Atomic value aggregations e.g., COUNT, SUM, MAX
- **distinct**
  - 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
- Distinct
  - can be solved with sorting, or

- group by / join

- map / atomics

- distinct

- limit (top k)

- BFS

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

- **Limit (top k)**
  - can be solved with sorting, or
  - different specialized structures
- **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)
- Breadth-first search (BFS)
  - FIFO order
  - track visited vertices

Operations

- group by / join
- map / atomics
- distinct
- hash set
- limit (top k)
- tree / heap / list

BFS

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

- **Depth-first search (DFS)**
  - LIFO order
  - track visited vertices
• 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
- distinct
- hash set
- limit (top k)
- tree / heap / list
- map / atomics

**Score**

<table>
<thead>
<tr>
<th>Structure</th>
<th># Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>array / buffer</td>
<td>7</td>
</tr>
<tr>
<td>map</td>
<td>6</td>
</tr>
<tr>
<td>set</td>
<td>4</td>
</tr>
<tr>
<td>tree / heap</td>
<td>3</td>
</tr>
<tr>
<td>queue</td>
<td>2</td>
</tr>
<tr>
<td>stack</td>
<td>1</td>
</tr>
<tr>
<td>list</td>
<td>1</td>
</tr>
</tbody>
</table>
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++