Concurrent programming: From theory to practice

Concurrent Algorithms 2018
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)
- 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**
  - 12 cores @ 2.4GHz
  - L1: 32KB
  - L2: 256KB
  - L3: 40MB
  - Memory: 128GB

- **AMD Opteron**
  - 12 cores @ 2.4GHz
  - L1: 64KB
  - L2: 512KB
  - L3: 20MB
  - Memory: 128GB
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

Private

= multiple copies
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

  ![Desktop machine diagram]

  = Uniform

- Typical **server** machine

  ![Server machine diagram]

  = non-Uniform
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data

- L1: 1 ns
- L2: 7 ns
- L3: 20 ns
Latency (ns) to access data
Latency (ns) to access data

Memory

1 7 20

L1 L2 L3

C

L1 L2 L3

C

L1 L2 L3

80

C

Memory
Latency (ns) to access data

C

L2

L1

L3

Memory

C

L2

L1

L3

Memory

90

1

7

20

40

80
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
### 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>

### Series 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>Vasilis :people</th>
</tr>
</thead>
<tbody>
<tr>
<td>:likes</td>
</tr>
<tr>
<td>:likes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rachid :people</th>
</tr>
</thead>
<tbody>
<tr>
<td>:likes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Breaking bad :series</th>
</tr>
</thead>
<tbody>
<tr>
<td>:similar</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dexter :series</th>
</tr>
</thead>
</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
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 \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, ...

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 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
task / job scheduling

Runtime

indices / metadata

buffer management

tasks used

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

number_of_references: X

renaming (ids)

1MB 1MB 1MB 1MB
Graph structure

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

User-ids - internal ids

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

Properties

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

Lifetime management

- number_of_references: X

Operations

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

- distinct
  - 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

- BFS

- DFS
• 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 \rightarrow 0  \quad \text{0} \rightarrow \text{Vasilis}
Rachid \rightarrow 1  \quad \text{1} \rightarrow \text{Rachid}
Breaking bad \rightarrow 2  \quad \text{2} \rightarrow \text{Breaking bad}
Dexter \rightarrow 3  \quad \text{3} \rightarrow \text{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>User-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

\(\rightarrow\) segmented table
• tmp graph structure
  • append only
  • dynamic schema
  → segmented table

• Classic graph structures

Graph

user-ids - internal ids

- 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

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**

- 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

**Classic graph structures**

1. connectivity matrix

```
0 1 2
0 x
1 x x
2 x
```

2. adjacency list

```
0 → 0
1 → 0 2
2 → 1
```

3. compressed source row (CSR)

```
1 3 4
```

```
Graph

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

<table>
<thead>
<tr>
<th>user-ids - internal ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis → 0</td>
</tr>
<tr>
<td>Rachid → 1</td>
</tr>
<tr>
<td>Breaking bad → 2</td>
</tr>
<tr>
<td>Dexter → 3</td>
</tr>
<tr>
<td>0 → Vasilis</td>
</tr>
<tr>
<td>1 → Rachid</td>
</tr>
<tr>
<td>2 → Breaking bad</td>
</tr>
<tr>
<td>3 → 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

user-ids - internal ids

- 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

- 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: \([0, N]\)

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]
  → (sequential) array

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

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

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

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

**Graph Structure**
- One type per property, e.g., int
- 1:1 mapping with vertices/edges

**User-Ids - Internal Ids**
- Vasilis \(\rightarrow 0\)
- Rachid \(\rightarrow 1\)
- Breaking Bad \(\rightarrow 2\)
- Dexter \(\rightarrow 3\)

**Hash Map / Array**
- \(0 \rightarrow\) Vasilis
- \(1 \rightarrow\) Rachid
- \(2 \rightarrow\) Breaking Bad
- \(3 \rightarrow\) Dexter

**Dictionary**
- :likes
- :similar
- :people

**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
  \(\rightarrow\) (sequential) arrays

---

**Sentence:**

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

- 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

**Graph structure**

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

- segmented buffer

- CSR

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

- hash map / array

- liftime management

- labels

- dictionary

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

- number_of_references: X
**Graph**

**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**

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

- **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:**

|------------|------------|------------|

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
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

segmented buffer

csr

graph structure

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

dictionary (= map)

properties

array

number of references: X

lifetime management

Structure | # Usages
---|---
array / buffer | 5
map | 2

Score

63
• **Indices**
  - Used for speeding up “queries”
  - Which vertices have label :person?
  - Which edges have value > 1000?

---

**Runtime**

- **indices / metadata**
  - < 300
  - >= 300

**buffer management**

- 1MB 1MB 1MB 1MB

**task / job scheduling**

- Producers → Consumers
  - task
  - task

**labels**

- :likes, :people, :similar, :male …

- 1 2 3 4

- \{people, male\} → \{2, 4\}

**renaming (ids)**

- used 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

• 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?
  \rightarrow \textit{maps, trees}

- Buffer management
  - In “real” systems, resource management is very important
  - buffer pools
    - no order
    - insertions and deletions
    - no keys
  \rightarrow \text{Fixed num object pool: array}
  \rightarrow \text{Otherwise: list}
  \rightarrow \text{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\}

labels

:likes, :people, :similar, :male …

1 2 3 4

\{people, male\} → \{2,4\}

buffer management

map / tree

< 300  →  ≥ 300

indices / metadata

Runtime

task / job scheduling

Producers → Consumers

labels

renaming (ids)

used used used
• **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)
### Runtime

**indices / metadata**
- map / tree
  - < 300
  - >= 300

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

**task / job scheduling**
- Producers → Consumers
  - task

**labels**
- :likes, :people, :similar, :male …
  - 1
  - 2
  - 3
  - 4
- {people, male} → {2,4}

**renaming (ids)**
- used
- used
- used

### 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

indices / metadata

buffer management

array

map / tree

< 300

>= 300

>= 300

>= 300

runtime

indices / metadata

buffer management

array

map / tree

< 300

>= 300

task / job scheduling

Producers

Consumers

queue

task

dictionary (= map)

{people, male} \rightarrow \{2,4\}

labels

:likes, :people, :similar, :male …

renaming (ids)

map / tree / set

Score

<table>
<thead>
<tr>
<th>Structure</th>
<th># Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>array / buffer</td>
<td>6</td>
</tr>
<tr>
<td>map</td>
<td>5</td>
</tr>
<tr>
<td>tree / heap</td>
<td>2</td>
</tr>
<tr>
<td>set</td>
<td>1</td>
</tr>
<tr>
<td>queue</td>
<td>1</td>
</tr>
</tbody>
</table>
• 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.

---

### Operations

- **group by / join**
  - Vasilis, Breaking bad → Vasilis, 2
  - Rachid, Dexter → Rachid, 1
  - Vasilis, Dexter

- **distinct**
  - Vasilis
  - Rachid
  - Vasilis
  - Rachid

- **limit (top k)**
  - 11 12 0 9 8 13 → 32
  - 8 9 11 23 32 9 → 23
  - 1 2 3 5 7 3 2 0 → 13

- **BFS**

- **DFS**
**Operations**

**group by / join**

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

**distinct**

- Vasilis → Vasilis
- Rachid → Rachid
- Vasilis

**limit (top k)**

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

**BFS**

- Graph traversal from left to right

**DFS**

- Graph traversal from left to right

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

Operations

- 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
  - \(\rightarrow\) hash set

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

**Operations**

- **group by / join**
- **map / atomics**
- **distinct**

**Limit (top k)**

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

**BFS**

**DFS**

80
Operations

- **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

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**
  - Vasilis, Breaking bad
  - Rachid, Dexter
  - Vasilis, Dexter

- **map / atomics**
  - Vasilis
  - Rachid

- **distinct**
  - Vasilis
  - Rachid

- **limit (top k)**
  - Vasilis
  - Rachid

- **tree / heap / list**
  - 11 12 0 9 8 13
  - 8 9 1 12 3 3 2 0

- **BFS**
  - Queue / set

- **DFS**
  - Stack / set

---

**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