High-Performance Graph Queries *(and friends)*

Alberto Parravicini
alberto.parravicini@polimi.it

2020-11-07

POLITECNICO MILANO 1863
POLITECNICO MILANO 1863 NECST laboratory

Graph Queries – The Big Picture

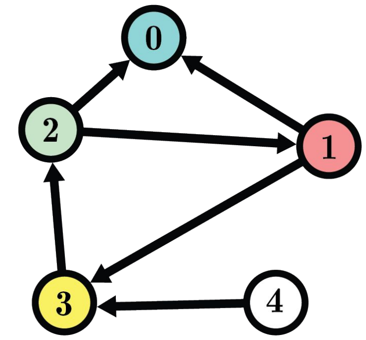
"Find all friends of Alberto who are older than 30"

"What's the lowest number of airport layovers if going from Malpensa to Timbuktu"
(Answer: 1, Casablanca)

"Find all money exchanges between people in Milan in the last 24 hours"

```
SELECT v3.ID
MATCH (v1) -> (v2) -> (v3)
WHERE v1.ID == 1 AND v3.ID > 1
```

Things get out of hands quickly! Graph queries can be extremely complex, operate on extremely large data, and require extremely quick results



Graph Queries – The Big Picture

“Find all friends of Alberto who are older than 30”

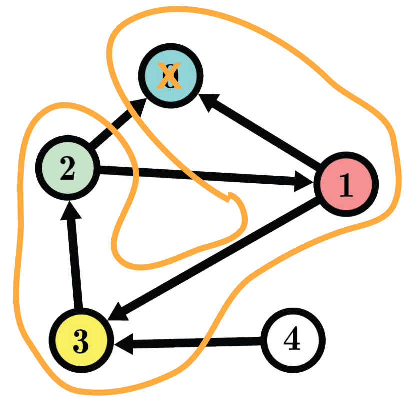
“What’s the lowest number of airport layovers if going from Malpensa to Timbuktu”
(Answer: 1, Casablanca)

“Find all money exchanges between people in Milan in the last 24 hours”

Things get out of hands quickly! Graph queries can be extremely complex, operate on extremely large data, and require extremely quick results

```
SELECT v3.ID
MATCH (v1) -> (v2) -> (v3)
WHERE v1.ID == 1 AND v3.ID > 1
```

ANSWER: **2**



Graph Queries – What do we need

Mostly, we see again - more in depth - topics seen before

Data structures for graphs

- Space-efficient
- Fast to query (and parallel!), and (sometimes) easy to update
- Can we leverage DB data-structures? (Answer: sometimes)

A language to define queries

- SQL doesn't really work well
- We need a query language built with graphs in mind

A set of operators

- We need to map queries in our "language" to actions on the graphs

A way to apply operators

- Broadly speaking "query planning", here we focus on a specific case

Graph Queries – Why do we care

Why do we talk about graph queries in this course?

It's challenging from a Computer Science perspective

- Lots of possible optimizations
- Performance depends on data
- Opportunities for parallelism
- Hardware knowledge required

Plenty of research opportunities

- Ever-increasing data size and new hardware presents new possibilities and challenges
    - Data-driven optimization, heterogeneous architectures, 3D Xpoint SSDs, etc.
- **Even our contest is an open research problem**

Our contest:
Graph-Traversal VS Hash-Join

Don't worry, we'll see later all the technical details!

What are you gonna do?

Neighbour match is a common graph operator

Given a vertex, retrieve neighbours 1, 2, ... , N hops away

There are 2 ways to implement it

- Graph traversal
- Table JOIN

But... which is faster? And when? Can you combine them to get a super fast adaptive implementation?

Graph data structures for everyone

Adjacency matrix

Adjacency list

You should know them already

COO

CSR

CSC

Also used for sparse linear algebra

Other stuff

Adjacency Matrix – 1

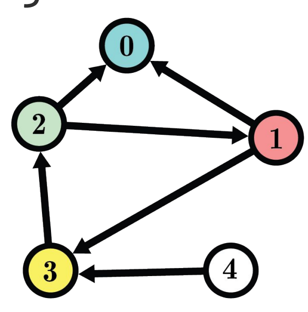
Just a dense matrix with non-zero entries representing edges

Practical uses: almost none, used for *very fast* random access to in-neighbours AND out-neighbours, and algorithms that use dense matrices (e.g. GCN)

Biggest drawback: $|\mathrm{V}|^2$ storage space

Real-world graphs are sparse.
E.g. Wikipedia, 10M vertices, 160M edges

Sparsity:
$160 \cdot 10^6 \,/\, (10 \cdot 10^6)^2 = 0.0000016,$



$$\mathbf{A} = \begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & & & & & \\ 1 & 1 & & & 1 & \\ 2 & 1 & 1 & & & \\ 3 & & & 1 & & \\ 4 & & & & 1 & \end{array}$$

1 out of 625000 is non-zero

if using 1 bit for each value, 12TB

Adjacency Matrix - 2

Cost of operations

V: number of vertices,
E: number of edges

Note: I'm using $\mathrm{O}(\ldots)$ if necessary, precise values where possible

- *Random access:* 1
- *Neighbour iteration:* V, in & out
  - It's bad, neighbourhoods are sparse!
- *Adding edge:* 1
- *Modifying/removing edge:* 1
- *Adding vertex:* $\mathrm{O}($☠️$)$
  - Actually $\mathrm{O}(\mathrm{V})$ amortized if using vector of vectors or $\mathrm{O}(\mathrm{E})$ if using a single vector (see next slide). Either way, if you need to add a lot of vertices, you are using the wrong data structure

Adjacency Matrix - 3

Quick recap on how to implement matrices!

- Vector of vectors

```cpp
std::vector<std::vector<int>> G(V);
// Init sub-vectors...
for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
        G[i][j] = ... // Access;
```

- A single array (or vector)

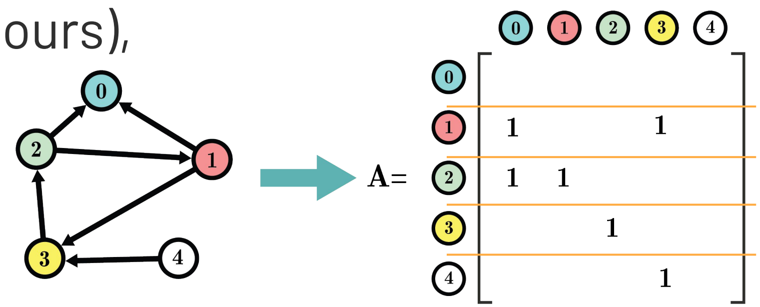
```cpp
std::vector<int> G(V * V);
for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
        G[i * V + j] = ... // Access;
```

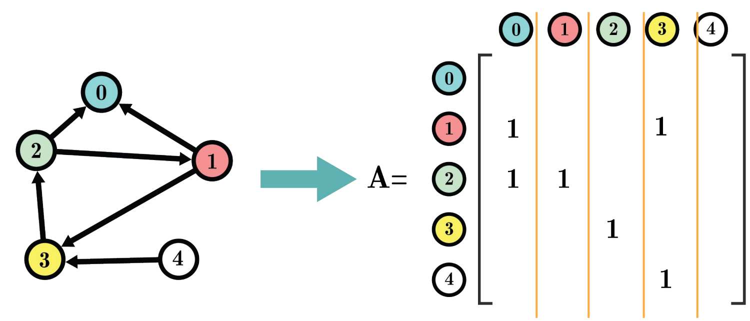
- A single array is usually faster (access is 1 memory access instead of 2, and it's more cache-friendly). But vector of vectors is easier to manipulate

Adjacency Matrix - 4

Keep in mind the difference between column-wise and row-wise allocation

- **Row-wise:** linear scan of rows (out-neighbours), "jumps" between columns
- **Column-wise:** linear scan of columns (in-neighbours), "jumps" between rows
- **Don't mix row-wise allocation with column-wise iteration (or vice-versa)!** You'll get terrible performance due to "jumps" causing cache misses






```
std::vector<int> G(V * V);
for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
        G[j * V + i] = ... // ☠️
```

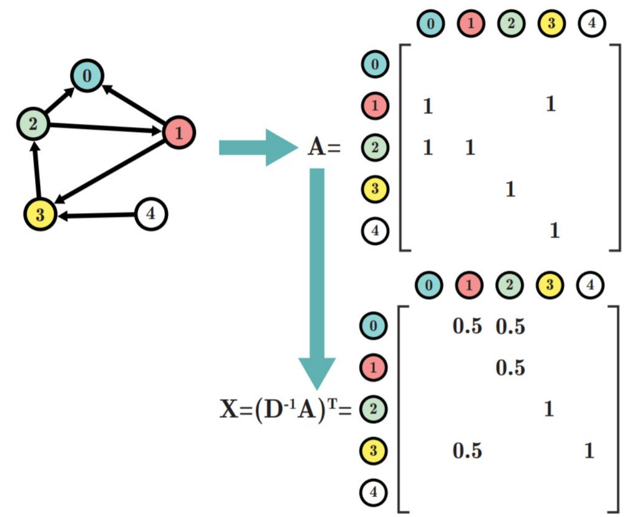
Adjacency Matrix – 5

- **Storing vertex properties:** additional vectors, use linear algebra (e.g. matrix-vector multiplication) to propagate properties across the topology

  Example, PageRank equation (**X** is the graph): $\mathbf{p_{t+1}} = \alpha \mathbf{X}\mathbf{p_t} + \frac{\alpha}{|V|}(\bar{\mathbf{d}}\mathbf{p_t})\mathbf{1} + (1-\alpha)\bar{\mathbf{v}}$

- **Storing edge properties:** use values instead of 0/1 in the matrix. Use more matrices (i.e. a tensor) for additional properties (space inefficient!)

- **Hardware friendliness:** very good, easy to parallelize and exploit cache, plenty of techniques from numerical computing (blocks, rows, columns)



Adjacency List - 1

A vector of vectors, in which we store only non-zero neighbour entries

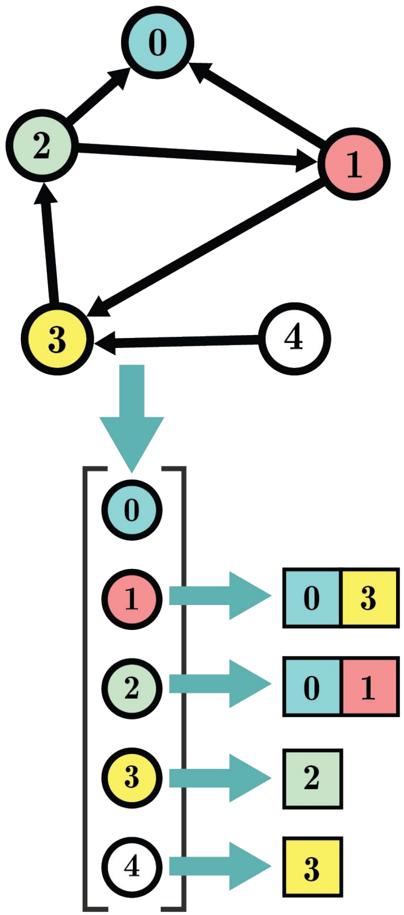
It looks good **on paper**! Constant vertex access, easy access to neighbours, easy to modify, lower memory footprint

- But is has plenty of drawbacks

Practical uses:

easy insertion of vertices and edges, use objects to represent vertices

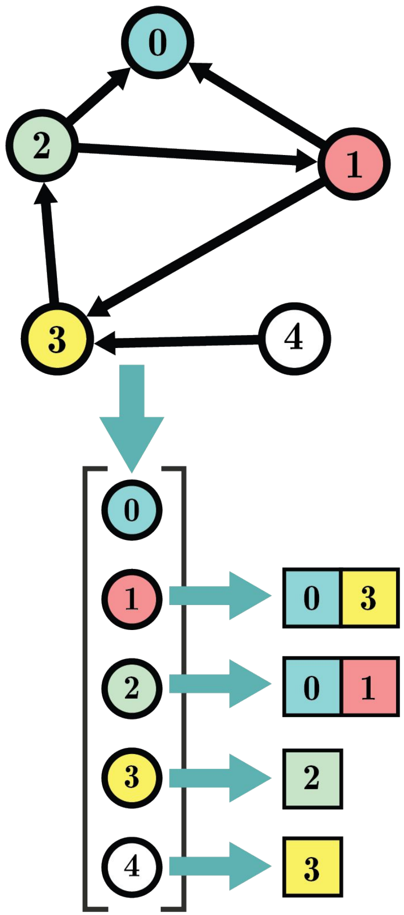
Can also be implemented as HashMap, with some pros and cons



Adjacency List - 2

Cost of operations

- *Random vertex access:* 1
- *Random edge lookup:* $O(V)$ (need to iterate all neighbors). $O(\log V)$ if binary search
- *Neighbour iteration:* $O(V)$ out-neigh., $O($☠️$)$ in-neigh (you need another adj. list)
- *Adding edge:* $O(1)$ amortized, $O(V)$ if inserted sorted
- *Removing/updating edge:* $O(V)$
- *Adding vertex:* $O(1)$ amortized, it depends on the implementation

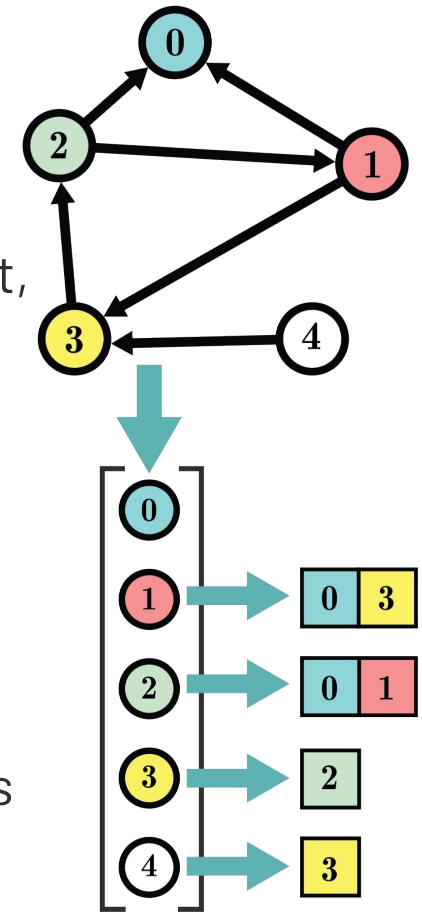


Adjacency List - 3

- **Storing vertex properties:** implementation dependent, e.g. using vertex objects, or additional vectors of size $|V|$

- **Storing edge properties:** implementation dependent, e.g. using Edge objects in a map
`HashMap<Tuple<VertexID, VertexID>, Edge>`

- **Hardware friendliness:** not good, traversal requires many lookups/memory accesses, neighbors arrays are not contiguous.
Even worse if implemented through hashmap, as you have further overheads (hashing) and conflicts



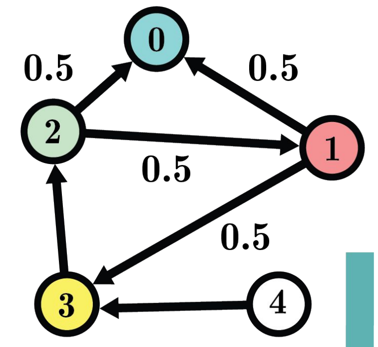
COO - 1

COOrdinate format, just a list of all the edges (not necessarily sorted). Use 1 or more vectors for edge-weights

Also used for the **MTX** file format

Practical uses: the simplest way to store a graph in a file. Streaming computations that require sequential access to all edges (e.g. PageRank)

Storing vertex/edge properties:
extra vectors $|\mathbb{V}|$ and $|\mathbb{E}|$



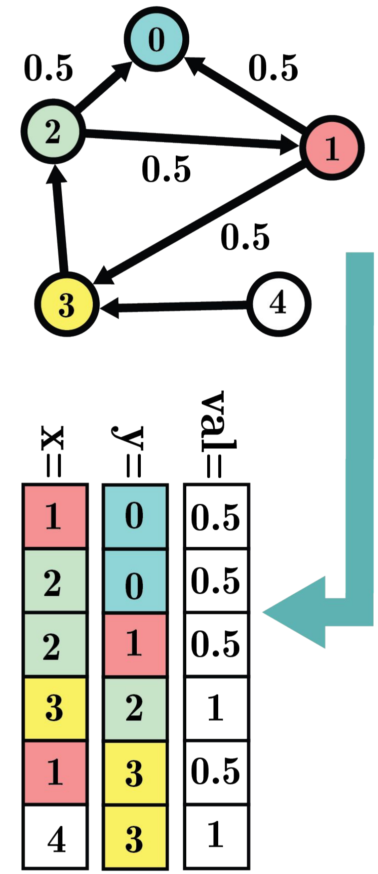
x=	y=	val=
1	0	0.5
2	0	0.5
2	1	0.5
3	2	1
1	3	0.5
4	3	1

COO - 2

Cost of operations

- *Random edge/vertex access:* O(☠️). $O(E)$, don't.
- *Neighbour iteration:* O(☠️).
  Possibly $O(E)$, we don't know where each vertex starts (even worse if vertices are not contiguous!)
- *Adding vertex/edge:* $O(1)$ amortized, if we allow non-contiguous edges. Otherwise $O(E)$
- Removals: $O(E)$

Note: if sorted w.r.t x and y, we can use binary search, with cost $O(\log(E))$, and improve some operations. Complexity is still extremely bad. E.g. find if a random edge exists: binary search on x, then linear scan on y, complexity $O(\log(E) + V)$



COO – 3

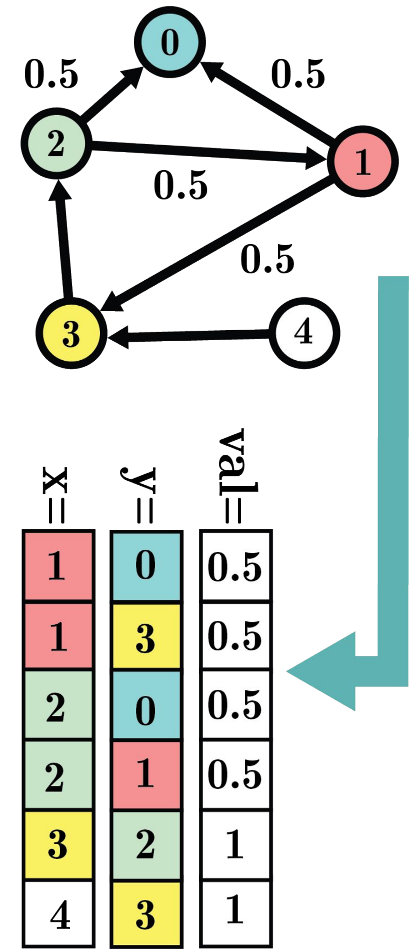
All operations have super bad complexity!
Is this data-structure useless for practical computations?

Not really, it's very very good for streaming edge processing, e.g. count all links with a certain value

Extremely easy to pipeline and parallelize, and cache friendly

Notes on parallelization

- If we just need to scan the edges, simply split the COO in equal partitions
- If we need to aggregate properties vertex-wise (e.g. PageRank), ensure that edges starting from a single vertex are not split, or have additional "aggregation logic"



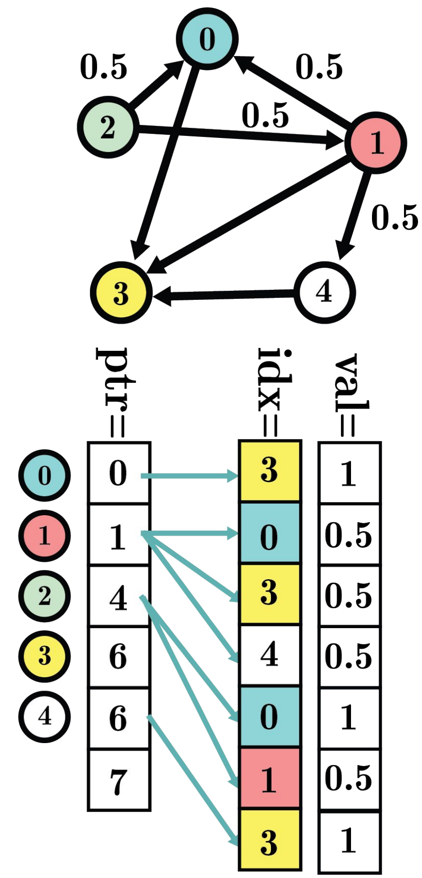
CSR - 1

Compressed Sparse Row (CSR) format

Keep a vector with cumulative degree of all vertices (called **ptr**), then vectors **idx** and **val** identical to the **y** and **val** vectors in COO

- Why cumulative degree? It allows fast access to out-neighbors
- **ptr** has size $V+1$, there is a starting 0

Practical uses: almost every graph algorithm (or sparse matrix computation) on static graphs, e.g. BFS



CSR - 2

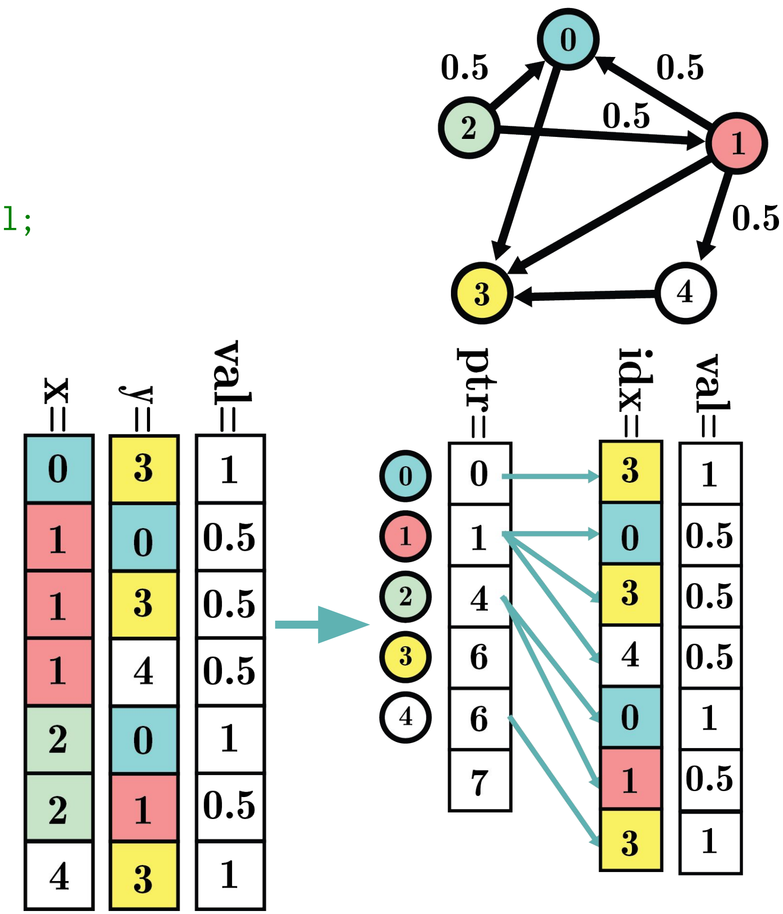
Assume x and y are sorted!

$\mathrm{O}(\mathrm{E})$ complexity if sorted, else $\mathrm{O}(\mathrm{E}\log(\mathrm{E}))$

idx/val must be sorted w.r.t. x in the COO!

COO-to-CSR, if COO is sorted

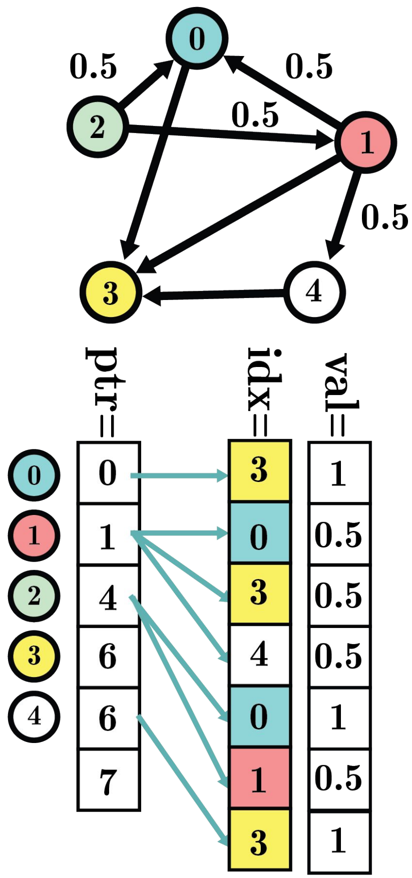
```
std::vector<int> ptr(V + 1, 0); // All zeros;
std::vector<int> idx(y) // Copy y into idx;
std::vector<float> val(val_coo) // Copy val_coo into val;
int curr_row = 0; int curr_sum = 0
for (int i = 0; i < E; i++) {
    int diff = (i > 0) ? (x[i] - x[i - 1]) : x[i];
    if (diff > 0)
        for (int j = 0; j < diff; j++)
            ptr[++curr_row] = curr_sum;
    curr_sum++;
}
// Handle edge-less vertices at the end;
for (int i = curr_row + 1; i < V + 1; i++)
    ptr[i] = curr_sum;
```



CSR – 3

Cost of operations

- *Vertex lookup:* 1
- *Edge lookup:* $\mathrm{O(V)}$, require traversing all neighbors; $\mathrm{O(\log(V))}$ with binary search
- *Out-neighbors iteration:* $\mathrm{O(V)}$, very efficient
- *In-neighbors iteration:* O(☠️)
- *Adding vertices:* $\mathrm{O(1)}$ amortized
- *Adding edges:* $\mathrm{O(1)}$ at the end, else $\mathrm{O(E)}$
- *Removals:* $\mathrm{O(1)}$ at the end, else $\mathrm{O(E)}$



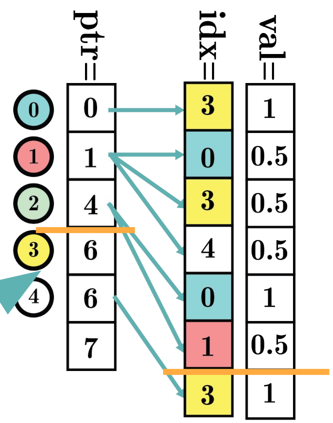
CSR – 4

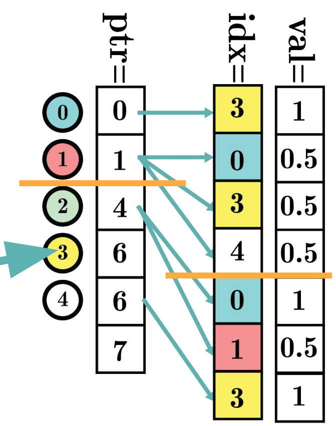
CSR is somewhat similar to adjacency list, but harder to update

But CSR is also much more **hardware-friendly**: based on array lookups, and arrays are contiguous.

Parallelization

- Very easy row parallelization (split **`ptr`**)
- This parallelization is not always ideal (imbalance), but it's easy and ok in most cases
- Other option: create P partitions with average size E / P, and split **`ptr`** accordingly

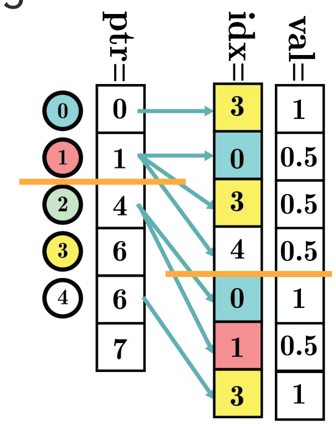




CSR – 5, Smart Partitioning

Building a smart CSR partitioning for parallelization

- Idx has size E, we want P partitions (here, E=7, P=2)
- First partition P_0 should end around floor(E/P) = 3
- Binary search on ptr for 3
  - We might not find 3, instead look for ptr[p_i] <= 3 && ptr[p_i+1] >= 3
  - Here p_i = 1
- Partition P_0 includes vertices 0 and 1, and idx up to ptr[p_i+1]
- Repeat for second partition (it should end around floor(2E/P))
- Repeat for all the other partitions
- Cost: $O(\log(V)P)$

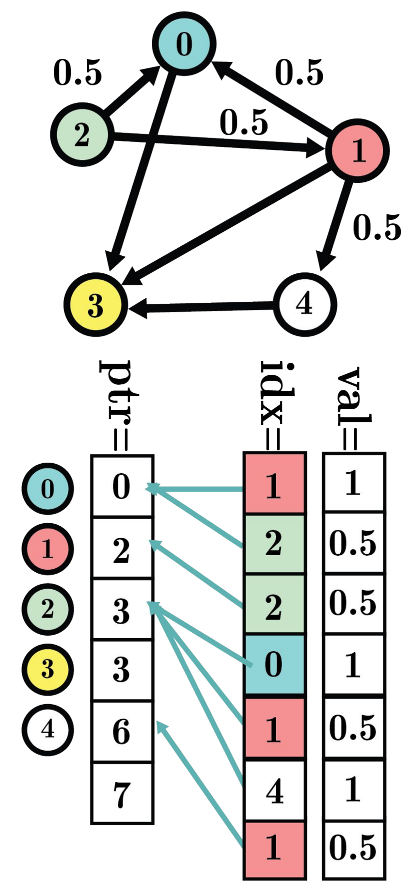


CSC – 1

Compressed Sparse Column (CSC) format

Same as CSR, but store incoming edges instead of outgoing edges

Practical uses: like CSR, useful in applications requiring incoming edges, e.g. PageRank

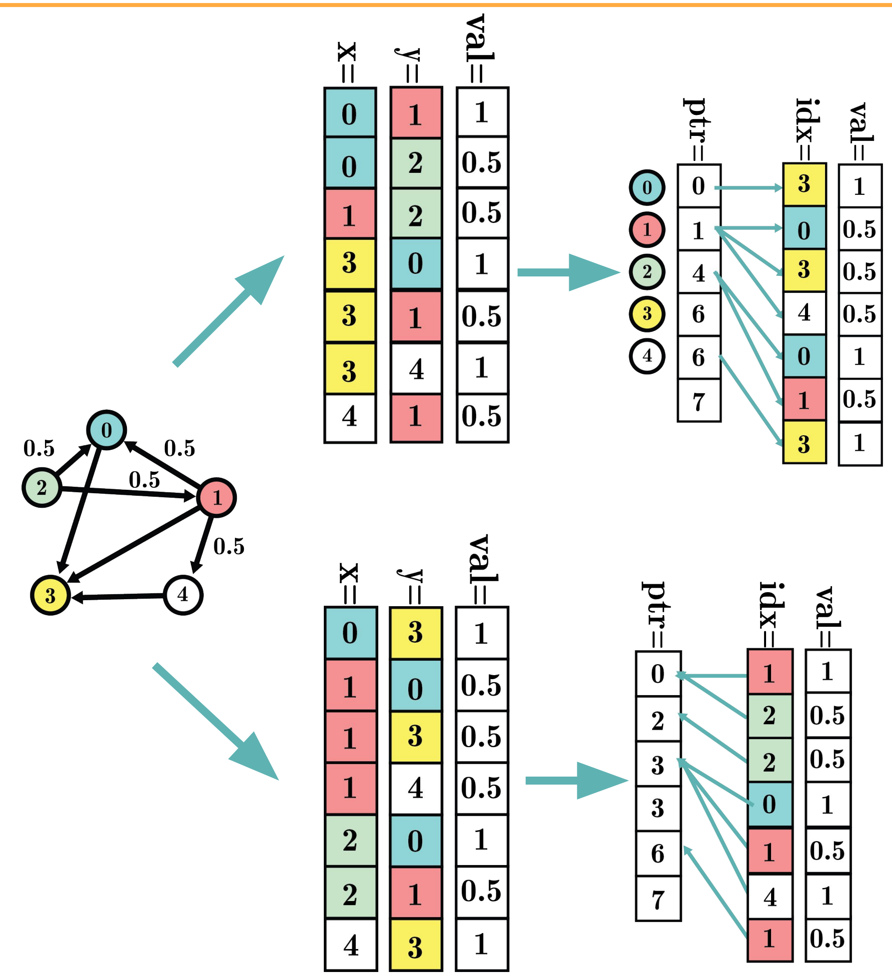


CSC - 2

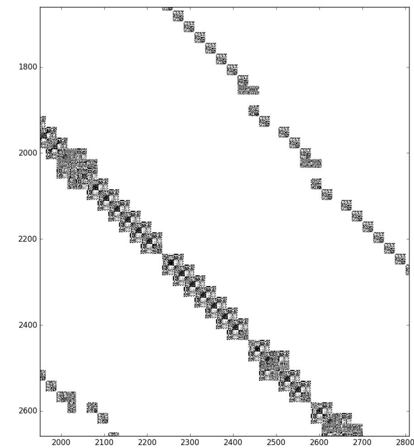
CSC can be created from COO just like CSR, swapping **x** and **y** (transposed matrix)

CSR-to-CSC or vice-versa is terrible, don't do it. Use a COO as temporary data structure

It's common to store both CSR and CSC to represent graphs, to have fast out and in neighbors iteration



Other Data Structures



BSR (Block Compressed Row): a CSR with dense matrices instead of scalar values, used for block-sparse matrices. A list of dense matrices with additional information about start/size of each matrix

GraphTinker and STINGER: very complex data-structures for dynamic graphs.

Extension of CSR with edge-blocks connected through linked lists or hash-maps, and meta-data to check if a value is valid or not. They allow a given number of updates, followed by *compaction/cleanup*

https://stackoverflow.com/questions/37209998/solving-large-linear-systems-with-block-sparse-matrices

https://ieeexplore.ieee.org/document/8821003

https://ieeexplore.ieee.org/document/6408680

Graph query languages and operators

Intro to PGQL

Graph query operators

- Root Match
- Neighbour Match
- Edge Match
- Common Neighbour Match

Intro to PGQL – 1

SQL doesn't really work well with graph data

- Paths on the graph are very complex JOINs
- What about arbitrary length paths (e.g. "*is there a path between ... and ...?*")

We want a language with graphs in mind!

Different options exists, but no common standard

- PGQL, pgql-lang.org/
- SPARQL *(built for RDF, not graphs)*, www.w3.org/TR/rdf-sparql-query
- Gremlin, tinkerpop.apache.org/gremlin.html
- Cypher, neo4j.com/developer/cypher

Intro to PGQL - 2

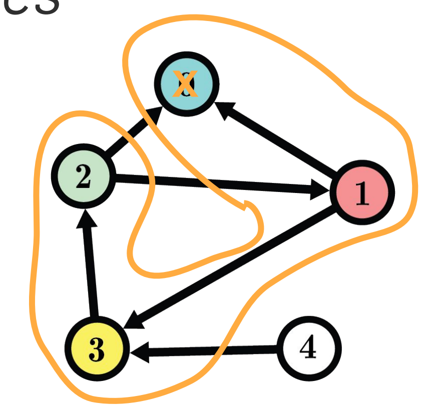
Here we see PGQL (Property Graph Query Language)

PGQL is an SQL-based query language for the property graph data model.
It allows you to specify high-level graph patterns which are matched against vertices and edges in a graph

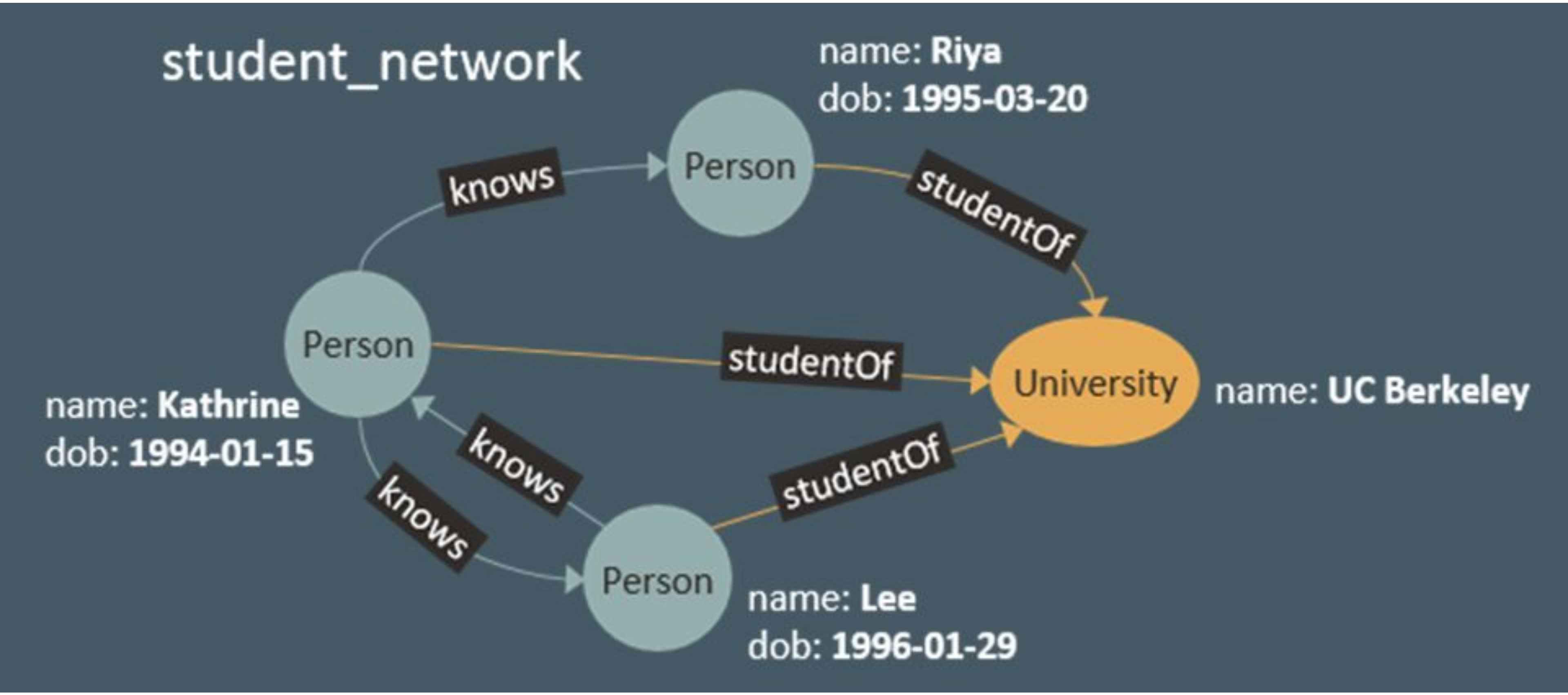
We'll learn how to use it with some examples

```
SELECT v3.ID
MATCH (v1) -> (v2) -> (v3)
WHERE v1.ID == 1 AND v3.ID > 1

ANSWER: 2
```



PGQL by Examples



PGQL by example

Selection of properties to be displayed

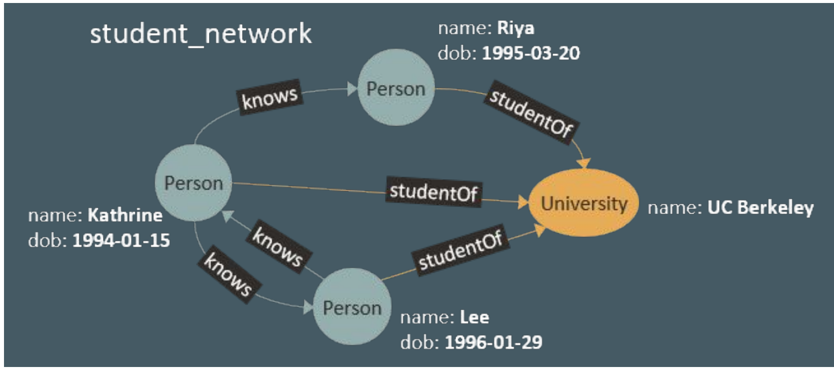
Data source from which we extract data

Graph pattern to search

```
SELECT n.name, n.dob
  FROM student_network
 MATCH (n:Person)
```

student_network: graph label
n: variable name
Person: vertex label
n.name: name is a property
(n:Person): vertex pattern

```
+-----------------------+
| n.name    | n.dob     |
+-----------------------+
| Riya      | 1995-03-20 |
| Kathrine  | 1994-01-15 |
| Lee       | 1996-01-29 |
+-----------------------+
```



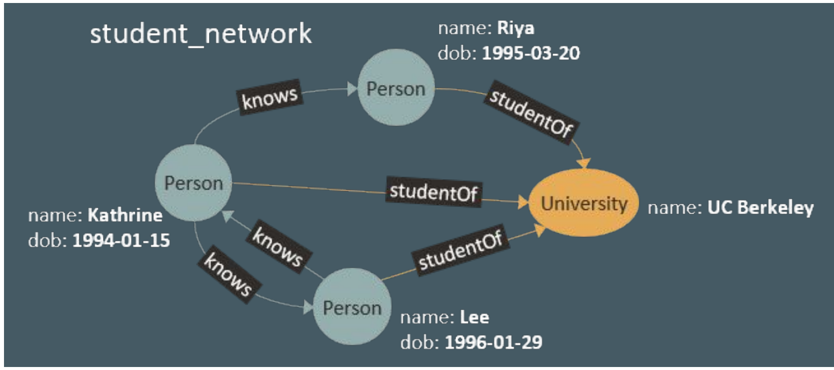
Edge Patterns


```
SELECT a.name, b.name
  FROM student_network
 MATCH (a:Person) -[e:knows]-> (b:Person)
```

-[e:knows]-> is an edge pattern in which **e** is a variable name and **:knows** a label expression

-> indicates edges outgoing from **a**

```
+---------------------+
| a.name   | b.name   |
+---------------------+
| Kathrine | Riya     |
| Kathrine | Lee      |
| Lee      | Kathrine |
+---------------------+
```

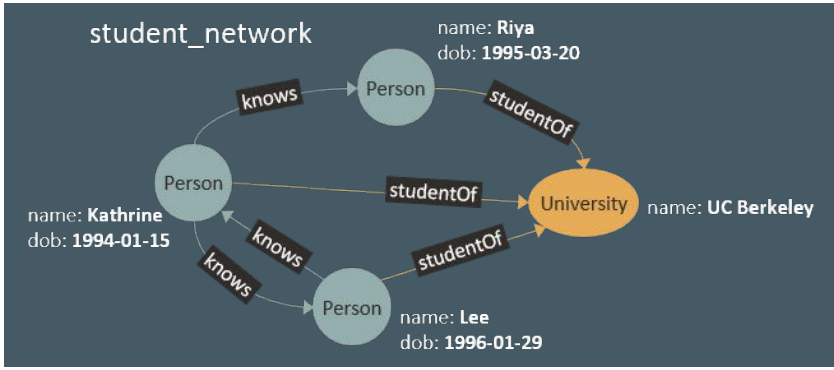


Label Disjunction

The bar operator **(|)** is a logical OR for specifying that a vertex or edge should match as long as it has either of the specified labels.

```
SELECT n.name, n.dob
  FROM student_network
 MATCH (n:Person|University)
```

```
+--------------------------+
| n.name      | n.dob      |
+--------------------------+
| Riya        | 1995-03-20 |
| Kathrine    | 1994-01-15 |
| Lee         | 1996-01-29 |
| UC Berkeley | <null>     |
+--------------------------+
```

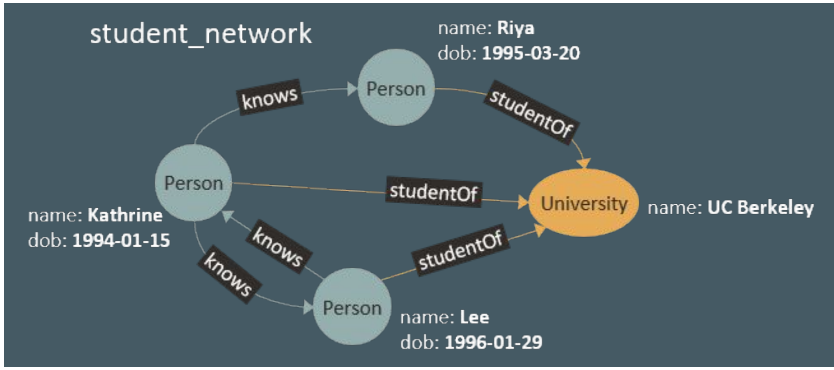


Label Omission

Label expressions may be omitted so that the vertex or edge pattern will then match any vertex or edge.

```
SELECT n.name, n.dob
  FROM student_network
 MATCH (n)
```

```
+-------------------------+
| n.name      | n.dob     |
+-------------------------+
| Riya        | 1995-03-20 |
| Kathrine    | 1994-01-15 |
| Lee         | 1996-01-29 |
| UC Berkeley | <null>    |
+-------------------------+
```

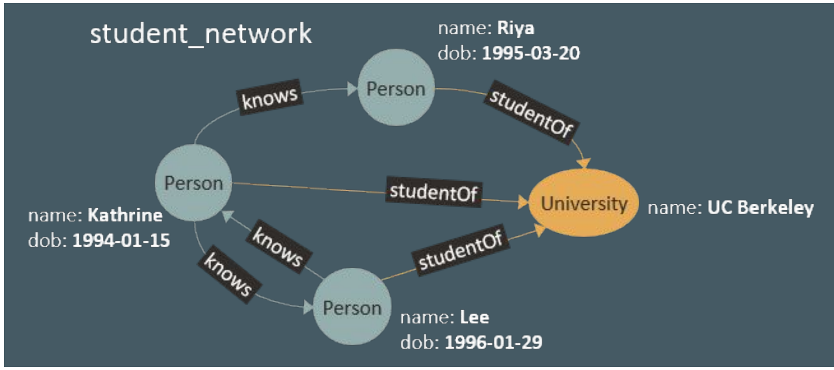


Filter Predicates

Filter predicates provide a way to further restrict which vertices or edges may bind to patterns. A filter predicate is a boolean value expression and is placed in a **WHERE** clause.

```
SELECT m.name AS name, m.dob AS dob
  FROM student_network
 MATCH (n) -[e]-> (m)
 WHERE n.name = 'Kathrine' AND n.dob <= m.dob
```

```
+-------------------+
| name | dob        |
+-------------------+
| Riya | 1995-03-20 |
| Lee  | 1996-01-29 |
+-------------------+
```



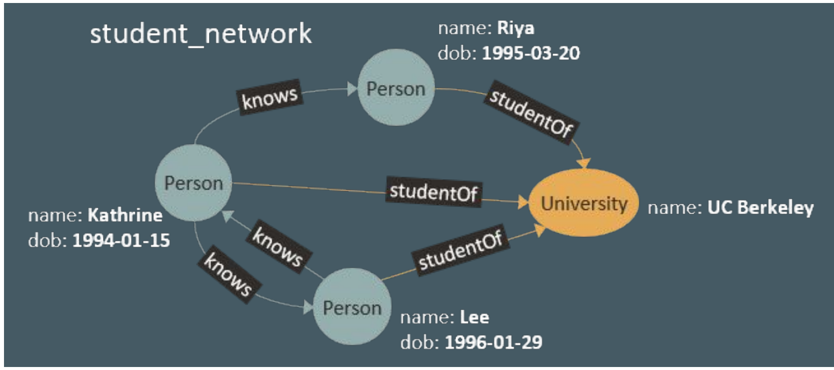
Complext Patterns

"find people that Lee knows and that are a student at the same university as Lee"

```
SELECT p2.name AS friend, u.name AS university
  FROM student_network
 MATCH (u:University) <-[:studentOf]- (p1:Person) -[:knows]-> (p2:Person) -[:studentOf]-> (u)
 WHERE p1.name = 'Lee'
```

Above, in the **MATCH** clause there is only one path pattern that consists of four vertex patterns and three edge patterns. Note that the first and last vertex pattern both have the variable **u**.

```
+------------------------+
| friend   | university  |
+------------------------+
| Kathrine | UC Berkeley |
+------------------------+
```

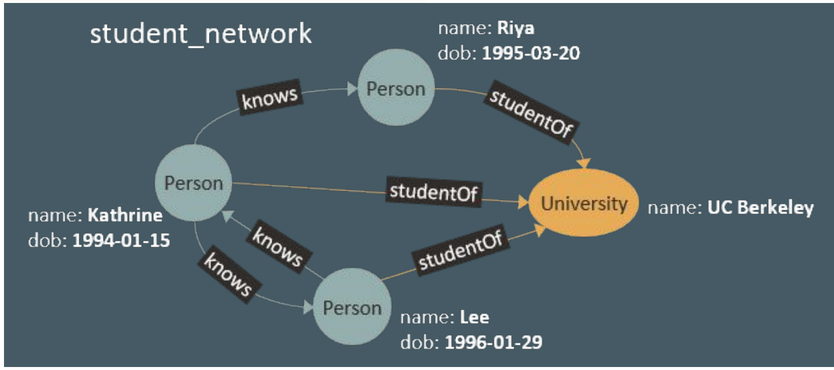


Separating match patterns

The previous query may be expressed through
multiple comma-separated path patterns, like this:

```
SELECT p2.name AS friend, u.name AS university
  FROM student_network
 MATCH (p1:Person) -[:knows]-> (p2:Person)
     , (p1) -[:studentOf]-> (u:University)
     , (p2) -[:studentOf]-> (u)
 WHERE p1.name = 'Lee'
```

```
+------------------------+
| friend   | university  |
+------------------------+
| Kathrine | UC Berkeley |
+------------------------+
```

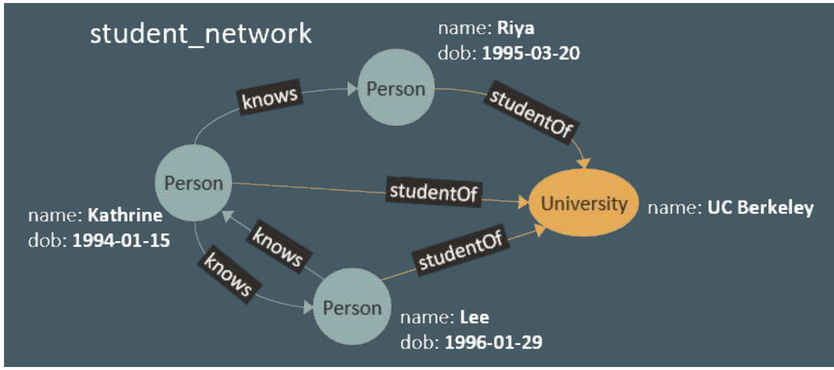


Binding a vertex many times

In a single solution it is allowed for **a vertex or an edge to be bound to multiple variables at the same time**, i.e. **(p1)** and **(p3)** can be the same vertex
For example, *"find friends of friends of Lee"* (friendship being defined by the presence of a 'knows' edge):

```
SELECT p1.name AS p1, p2.name AS p2, p3.name AS p3
  FROM student_network
 MATCH (p1:Person) -[:knows]-> (p2:Person) -[:knows]-> (p3:Person)
 WHERE p1.name = 'Lee'
```

```
+-----------------------+
| p1  | p2       | p3   |
+-----------------------+
| Lee | Kathrine | Riya |
| Lee | Kathrine | Lee  |
+-----------------------+
```



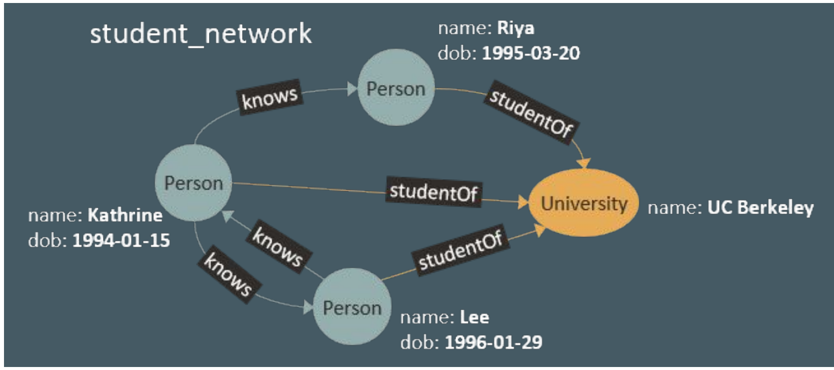
Non-equalities

If such binding of vertices to multiple variables is not desired, one can use either non-equality constraints or the ALL_DIFFERENT predicate.

```
SELECT p1.name AS p1, p2.name AS p2, p3.name AS p3
  FROM student_network
 MATCH (p1:Person) -[:knows]-> (p2:Person) -[:knows]-> (p3:Person)
 WHERE p1.name = 'Lee' AND p1 <> p3
```

predicate **p1 <> p3** in the query below adds the restriction that Lee, which has to bind to variable **p1**, cannot also bind to variable **p3**

```
+-----------------------+
| p1  | p2       | p3   |
+-----------------------+
| Lee | Kathrine | Riya |
+-----------------------+
```

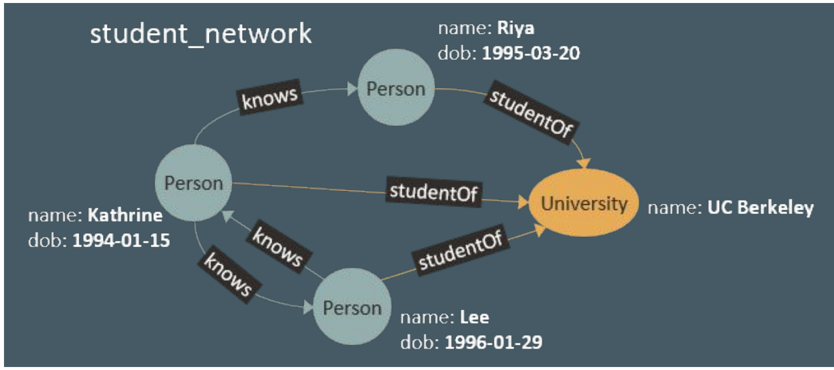


Binding an edge many times

It is also possible for edges to bind to multiple variables (i.e. different names but they refer to the same edge).
For example, *"find two people that both know Riya"*

```
SELECT p1.name AS p1, p2.name AS p2, e1 = e2
  FROM student_network
 MATCH (p1:Person) -[e1:knows]-> (riya:Person)
     , (p2:Person) -[e2:knows]-> (riya)
 WHERE riya.name = 'Riya'
```

```
+-----------------------------+
| p1       | p2       | e1 = e2 |
+-----------------------------+
| Kathrine | Kathrine | true    |
+-----------------------------+
```



Match edges in any direction

Any-directed edge patterns match edges in the graph no matter if they are incoming or outgoing

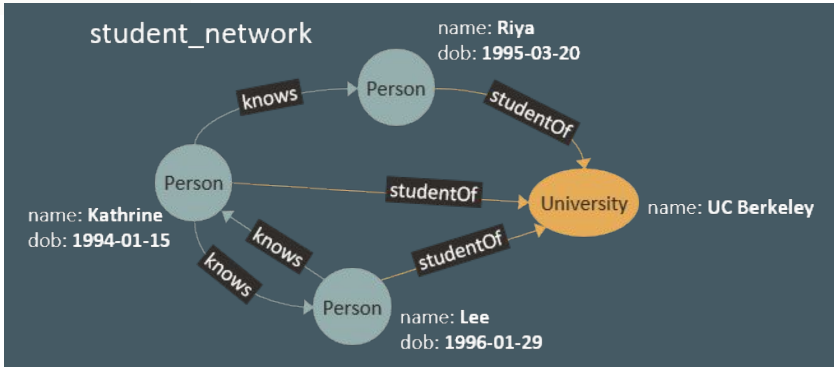
```
SELECT *
  FROM g MATCH (n) -[e1]- (m) -[e2]- (o)
```

In case there are both incoming and outgoing data edges between two data vertices, there will be separate result bindings for each of the edges.

Common path expressions:

```
  PATH two_hops AS () -[e1]- () -[e2]- ()
SELECT *
  FROM g MATCH (n) -/:two_hops*/-> (m)
```

The above query will return all pairs of vertices **n** and **m** that are **reachable via a multiple of two edges**, each edge being either an incoming or an outgoing edge.



Graph Query Operators – 1

We need to translate our *high-level* queries to basic operations on our data-structures

It's a complex problem! What operators do we need, how do we apply them?

Here we see a few basic operators. In the contest, you will implement and optimize one of them (**neighbour match**)

Graph Query Operators – 2

Root Match: matches all root vertices

Constant vertex match: root match optimized for unique vertices

Neighbor Match: given a vertex, matches all its neighbors

Edge Match: given two vertices, checks if they are connected via an edge

Common Neighbor Match: given two vertices, matches all common neighbors

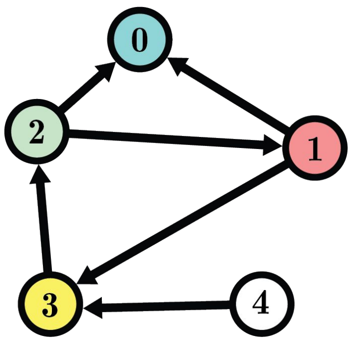
Cartesian product: combine results of different operators

Root Match

```
SELECT a
MATCH (a)
```


Similar to a table scan in a DB, it fetches all vertices. Optionally, apply filters or projections

Done by scanning the data-structure representing vertices




```
SELECT a
MATCH (a)
```

a
0
1
2
3
4

```
SELECT a
MATCH (a)
WHERE a > 2
```

a
3
4

```
SELECT SUM(a)
MATCH (a)
WHERE a > 2
```

SUM(a)
7

Constant Vertex Match

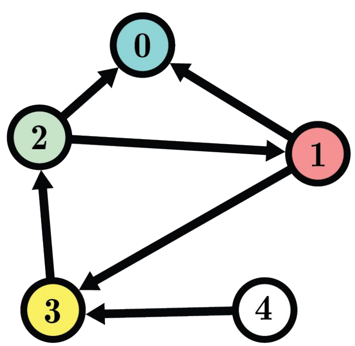
```
SELECT a
MATCH (a)
WHERE a.ID = 1
```


Root match operator specialized for unique vertices

If we are matching a vertex that we know is unique (e.g. filter condition on index/key), we can (and should) be faster than standard root match

Implementation: key match on a set/hash-map, $O(1)$, but requires additional data-structure

Queries rooted on a unique vertex are common, still worth optimizing for! Think about queries like *"find all passengers who took a flight from MPX last week"*




```
SELECT a
MATCH (a)
WHERE a.ID = 1
```

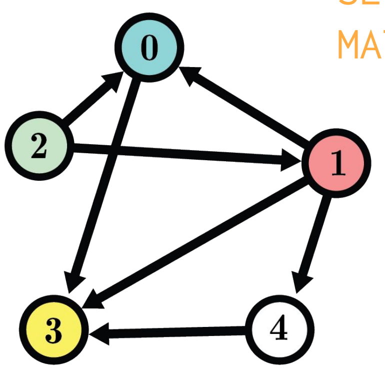
a
1

Neighbour Match – 1

```
SELECT a, b
MATCH (a)->(b)
```


Similar to a table JOIN in a DB, it retrieves the neighbours of one or more input vertices

Leverage the CSR for fast traversal, or perform a table JOIN




```
SELECT a, b
MATCH (a)->(b)
```

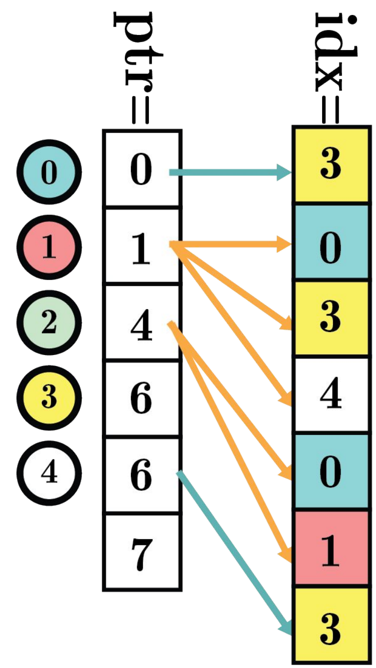
Consider only some vertices in this example

a
1
2

IN

a	b
1	0
1	3
1	4
2	0
2	1

OUT



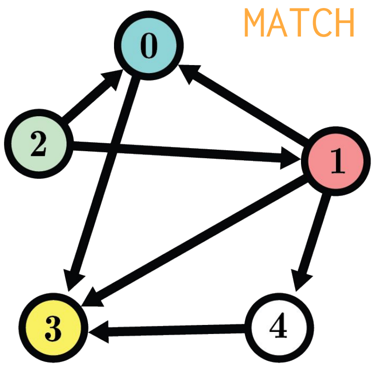
Neighbour Match - 2

```
SELECT a, b, c
MATCH (a)->(b)->(c)
```


Matching with depth > 1 requires care

- Avoid repeating matches for the same vertex (e.g. 0)
- Some vertices don’t have outgoing edges (e.g. 3)

```
SELECT a, b, c
MATCH (a)->(b)->(c)
```



a
1
2

a	b
1	0
1	3
1	4
2	0
2	1

a	b	c
1	0	3
1	4	3
2	0	3
2	1	0
2	1	3
2	1	4

BF Traversal

Neighbour Match - 3

```
SELECT a, b, c
MATCH (a)->(b)->(c)
```


With depth > 1, we do a Breadth-First or Depth-First Traversal

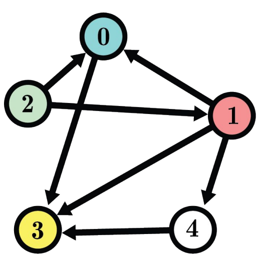
BF: match all **(a)**, then all **(b)**, then all **(c)**

Easy to parallelize, but requires storing a lot of intermediate results

DF: match one **(a)**, then one **(b)**, then all **(c)** w.r.t. that **(b)**, then another **(b)**, then all **(c)** w.r.t. that **(b)**, etc.

Low memory consumption, $O(\text{depth})$ instead of $O(\text{width})$, but difficult to parallelize, and might require multiple accesses to repeated neighbours

We can combine both approaches for best performance!



a
1
2

a	b
1	0

a	b	c
1	0	3

a	b
1	3

☠️

...

DF Traversal

Skeleton of BF and DF traversal

BF: use a queue (FIFO). DF: use a stack (LIFO)

In both cases keep track of visited vertices (e.g. with a set)

Here I visit the entire graph and store distances from the source

```cpp
void bf(std::vector<int> &ptr, std::vector<int> &idx,
std::vector<int> &res, int start_index = 0) {
    std::queue<int> frontier;
    frontier.push(start_index);
    std::unordered_set<int> seen;
    res[start_index] = 0;
    while (frontier.size() > 0) {
        int curr_elem = frontier.front();
        frontier.pop();
        seen.insert(curr_elem);
        for (int i = ptr[curr_elem]; i < ptr[curr_elem + 1]; i++) {
            int child = idx[i];
            res[child] = std::min(res[child], res[curr_elem] + 1);
            if (seen.find(child) == seen.end()) {
                frontier.push(child);
            }        }     }}
```

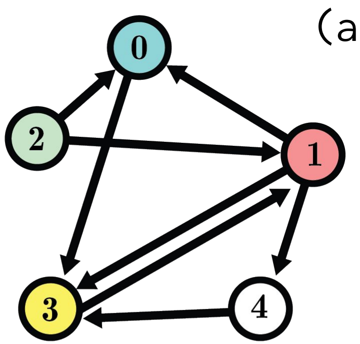
```cpp
void df(std::vector<int> &ptr, std::vector<int> &idx, std::vector<int>
&res, int start_index = 0) {
    std::stack<int> stack;
    stack.push(start_index);
    std::unordered_set<int> seen;
    res[start_index] = 0;
    while (stack.size() > 0) {
        int curr_elem = stack.top();
        stack.pop();
        seen.insert(curr_elem);
        for (int i = ptr[curr_elem]; i < ptr[curr_elem + 1]; i++) {
            int child = idx[i];
            res[child] = std::min(res[child], res[curr_elem] + 1);
            if (seen.find(child) == seen.end()) {
                stack.push(child);
            }        }     }}
```

Edge Match

```
SELECT a, b
MATCH (a)->(b)->(a)
```


After matching **(b)**, don't apply neighbour match to **(b)**

Instead, apply binary search in the outgoing neighbourhood of **(b)** to find **(a)**



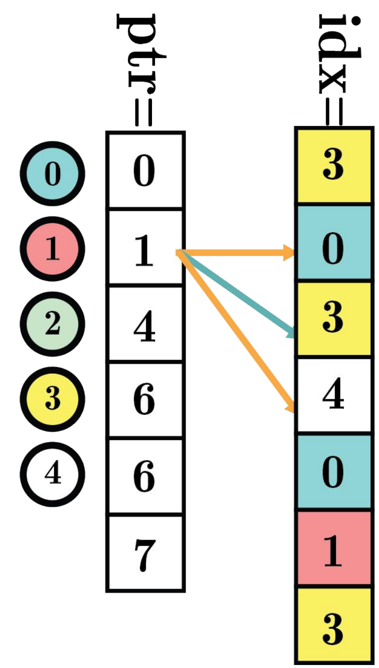

```
SELECT a, b
MATCH
(a)->(b)->(a)
```

a
2
3

a	b
2	0
2	1
3	1

```
Binary search
instead of
linear scan from
idx[1] to idx[3]
```

a	b
3	1

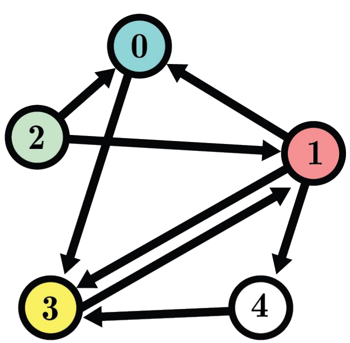


Common Neighbor Match

```
SELECT a, b, c
MATCH (a)->(b)<-(c)
```


Option 1: neighbourhood match from **(a)**, neighbourhood match from **(b)** to **(c)** using a CSC. $\text{Cost} = O(V) + O(V)^2$

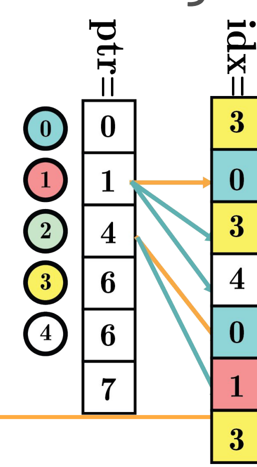
Option 2: neighbourhood match from **(a)**, then binary search in the neighbourhood of **(c)** to find common neighbours. Alternatively, neighbour match from **(c)** followed by set intersection. $\text{Cost} = O(V \cdot \log(V))$ or $3 \cdot O(V)$
Cost is misleading as very dependent on number of neighbours




```
SELECT a, b, c
MATCH (a)->(b)<-(c)
```


a	c
1	2

a	b	c
1	0	2



Cartesian Product

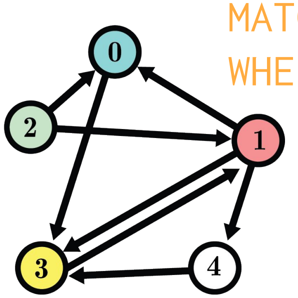
```
SELECT a, b, c, d
MATCH (a)->(b), (c)->(d)
WHERE a.ID = 1, c.ID = 2
```


Combine results from different operators, by computing all possible combinations

Used when no other operator can be applied, e.g. when combining separate MATCH patterns

It's your last resort: it's expensive, and causes a quadratic increase in result size

```
SELECT a, b, c, d
MATCH (a)->(b), (c)->(d)
WHERE a.ID = 1, b.ID = 2
```



a	b
1	0
1	2
1	4

c	d
2	0
2	1

a	b	c	d
1	0	2	0
1	0	2	1
1	2	2	0
1	2	2	1
1	4	2	0
1	4	2	1

Cartesian Product – 2

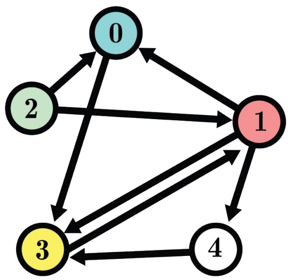
```
SELECT a, b, c, d
MATCH (a)->(b), (c)->(d)
WHERE a.ID = 1, c.ID = 2
```


Small pills of query planning

Think what happens if you apply Cartesian Product before WHERE

- You compute all edges in the graph *twice*, $2\mathrm{O}(\mathrm{E})$
- You compute all combinations of edges, $\mathrm{O}(\mathrm{E})^2$
- Then you filter the edges, $\mathrm{O}(\mathrm{E})^2$

Instead, computing the filter before neighbour match and Cartesian is way way better!




```
Worst case plan on our graph:
1.  Neighbourhood match, twice: 2*8
2.  All combinations: 8^2
3.  Filter edges: 8^2
Total: 208 operations
```

Best case plan on our graph:
1. Root match on **a** and **c**: 2*1
2. Neigh. match on **a=1** and **c=2**: 3 + 2
3. Cartesian product: 3 * 2

Total: 13 operations, **16x better!**

Quick overview of Hash-Join

Hash-Join

More Hash-Join

Even more Hash-Join

Storing graphs as tables

Starting point: a graph stored as a table

Here, 2 columns **x, y** but we could have other columns (edge properties)

index	x	y
0	A	B
1	A	D
2	A	E
3	B	C
4	B	F
5	B	H
6	B	L
7	C	A
8	C	B
9	D	A
10	D	C
11	D	F

We are not limited to integer values

To quickly retrieve neighbours, we can build a hash-table on column **x**

- Not much different from an adjacency list on **x** as key, built with an underlying hash-table
- But we must be aware of the underlying hash-table implementation!
- Here, simplified situation as we have only in-memory data

Hash-table, idealized view

Ideally, each vertex will map to a different row of the hash-table

Hash function: a function s.t. (ideally) $h(x_1) = h(x_2) \leftrightarrow x_1 = x_2$

index	x	y
0	A	B
1	A	D
2	A	E
3	B	C
4	B	F
5	B	H
6	B	L
7	C	A
8	C	B
9	D	A
10	D	C
11	D	F

→

key	val
h(x=A)	0,1,2
h(x=B)	3,4,5,6
h(x=C)	7,8
h(x=D)	9,10,11

We store rows indices. If you have just 2 columns, might as well store **y** directly

What if $h(x_1) = h(x_2)$ for $x_1 \mathrel{!=} x_2$?
We have a conflict

Conflicts will happen unless the codomain of $h(.)$ is $|V|$

Using hash-tables gives the flexibility of non-int keys and dynamic graphs

Hash-table, real implementation

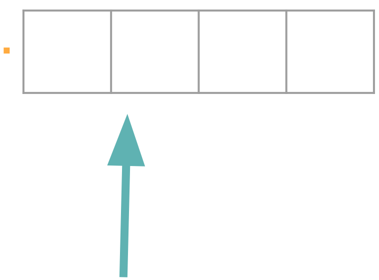
In practice, we have a fixed number of buckets/blocks, equal to the codomain of $\mathrm{h}(.)$

- Each block is a list (usually a fixed-size array, the block size)
- After computing $\mathrm{h}(.)$, linear scan of the block to find the desired key (if lookup) or to find an empty spot (if storing a value)
- If the block is full, we add a new block after it (overflow chain, a linked list of blocks). An extensible vector is also ok in our case
- If blocks are too full, we can increase the number of blocks (and change $\mathrm{h}(.)$ accordingly). This is expensive, as we might have to recompute all the existing blocks (if exists a stored value x for which $\mathrm{h}_1(\mathrm{x})$ $!=$ $\mathrm{h}_2(\mathrm{x})$)
- Rule of thumb: if blocks are filled above 80%, the probability of conflicts is so high that the current hash-table is no longer worth using

Hash-table, real implementation

index	x	y
0	A	B
1	A	D
2	A	E
3	B	C
4	B	F
5	B	H
6	B	L
7	C	A
8	C	B
9	D	A
10	D	C
11	D	F

key	BLOCK	B[0]	B[1]	B[2]	B[3]
h(x=A), h(x=C)	1	A:0,1,2	C:7,8		
h(x=B), h(x=D)	2	B: 3,4,5,6	D: 9,10,11		



Overflow chain

2 blocks, each block has size 4

In some implementations, store rows directly in block cells, e.g. [A,B], [A,D], [A,E] (in 3 blocks cells) instead of A:[0,1,2] in 1 cell

Hash-join - 1


```
SELECT a, b, c
MATCH (a)->(b)->(c)
```

```
SELECT a.x, b.x, c.x
FROM graph_table a, graph_table b,
graph_table c
WHERE a.y = b.x AND b.y = c.x
```

OPTIMIZED:

```
SELECT a.x, b.x, b.y
FROM graph_table a,
graph_table b
WHERE a.y = b.x
```

A single join is done as `SELECT a.x, b.x FROM a, b WHERE a.y = b.x`

1. Find the smaller table (let's say **a**)
2. Create a hash-table for **b** if it doesn't exist already
3. Iterate on rows of **a**
4. For each row, lookup the value of **a.y** on the hash-table of **b**
   a. First find the bucket with h(a.y), then scan to find results
5. Add results of **b.x** (from the hash-table) to the result

Hash-join - 2


```
SELECT a.x, b.y FROM a, b WHERE a.y = b.x
```

We can optimize this query with an additional hash-table on **`a.y`**, using the same hash function used for **`b.x`** (that's ok, the range of values is the same)

Now, values values of **`a.y`** will have the same block index of values in **`b.x`**, $\mathbf{block_a(y) = block_b(x)}$

We can perform a block-wise join by processing pairs of blocks (one block from a, one from b) in parallel. Each graph vertex will fall in the same block in both hash-tables! This also enables efficient processing disk-resident data

More info: www.csd.uoc.gr/~hy460/pdf/p63-mishra.pdf

Hash-join – 2

index	x	y
0	A	B
1	A	D
2	A	E
3	B	C
4	B	D
5	B	E
6	B	F
7	C	A
8	C	B
9	D	A
10	D	C
11	D	F

H1: Hash-table on x

key	BLOCK	B[0]	B[1]	B[2]	B[3]
h(x=A), h(x=C)	1	A:0,1,2	C:7,8		
h(x=B), h(x=D)	2	B: 3,4,5,6	D: 9,10,11		

H2: Hash-table on y

key	BLOCK	B[0]	B[1]	B[2]	B[3]
h(x=A), h(x=C), h(x=E)	1	A:7,9	C:3,10	E: 2,5	
h(x=B), h(x=D), h(x=F)	2	B: 0,8	D: 1,4	F:6,11	

Overflow chain

Join B1 in H1 with B1 in H2, and B2 in H1 with B2 in H2

Start from rows in H2: B[0] tells us that rows 7,9 ends with A. Now find key A in H1, and create results joining rows 7,9 with 0,1,2

And finally…

Graph-Traversal VS Hash-Join Contest Overview

Graph-Traversal VS Hash-Join
Contest Overview

The goal of this challenge consists in implementing what you learned about CSR and Hash-Join, and implement a simple query execution engine able to perform a set of predefined simple queries.

Important references

- *Repository with README and code:* github.com/AlbertoParravicini/high-performance-graph-analytics-2020
- *For any question:* alberto.parravicini@polimi.it
- *Contest start:* **NOW**
- *Contest end:* December 9th 2020, 11.59 PM (Milan Time!)

Graph-Traversal VS Hash-Join
Contest Overview

Dataset

POCEK, the most popular online social network in Slovakia

1.6M vertices, 30M edges, we only care about the graph topology (i.e. friendship relations)

Protip: start loading and working with a smaller subgraph!

https://snap.stanford.edu/data/soc-Pokec.html

Graph-Traversal VS Hash-Join
Contest Overview

4 Tasks

1. Load the dataset, and store the graph in a CSR and a Tabular format. You should be able to load a subgraph too
2. CSR Traversal and Hash-Join, you should implement the Neighbour Match operator in these 2 ways
3. Benchmark some queries! `(a)->(b)`, `(a)->(b)->(c)`, `(a)->(b)->(c)->(d)`, etc. Which implementation is faster? Which uses your hardware more efficiently/effectively?
4. Build a data-driven heuristic, to pick the best implementation based on the data and query

Graph-Traversal VS Hash-Join
Contest Overview

And finally... **Write a report**

- Submission before December 9th 11.59 PM 2020, Milan time
- Email to alberto.parravicini@polimi.it, CC to guidowalter.didonato@polimi.it and marco.santambrogio@polimi.it
- In the email:
  names of participants, link to GitHub repo, PDF copy of report

Repository:

- The source code
- A README that explains how to execute your solution
- A 4-pages report written in Latex describing your findings in tasks 3 and 4, a description of your heuristic, and any other implementation decision you took that you'd like to share with us

Graph-Traversal VS Hash-Join
Contest Overview

Additional notes (please refer to github.com/AlbertoParravicini/high-performance-graph-analytics-2020/blob/main/track-graph-query/README.md)

- Your code must be buildable with standard tools like Maven
- Use Java. Other JVM based languages (e.g. Scala) are ok if you can **properly** justify their usage
- Tests must be runnable using a Bash or Python script
- **The easier for us to replicate your results, the better it is for you!**
- External libraries are allowed, as long as you justify their usage and the **core** of the implementation is written by you. You can use existing CSR/Hash-Join implementations, but only as a performance comparison against your custom implementation
- The report should be 4 pages long at most, and written in double-column Latex, with font-size 10pt

Graph-Traversal VS Hash-Join

Contest Overview

In the repository you'll find the skeleton of 4 classes. Feel free to extend them as you like. You **can** change the existing interfaces, but justify any change!

`CompressedSparseRow` Basic CSR class, it offers 2 methods

```
void buildFromFile(String filepath)
ArrayList<Integer> getNeighbors(Integer vertex_id)
```

`Table` Basic tabular graph implementation

```
void BuildFromFile(String filepath)
```

`CSREngine` Given a CSR and a Integer, return neighbours

```
ArrayList<Integer> traverse(CompressedSparseRow csr, Integer vertex_id)
```

`HashJoinEngine` Given a Table and a Integer, return neighbours

```
ArrayList<Integer> join(Table tab1, Integer element_id);
```

Graph-Traversal VS Hash-Join
Contest Overview

These functions are just a sketch. You'll need something more!

Query parser: to turn queries into a list of operations. It's very simple, as all queries have form `(a)->(b)->(c)->...`

Extend the query operators: instead of providing just an integer to the traverse/join functions, you can pass a list of vertices or even a full graph/table, to optimize the overall computation

Use a Graph or Vertex class: using objects to represent vertices might help in some cases (e.g. track seen vertices in traversal). Be careful with overheads though! Also, instead of building CSR/Table directly from a file you can use an intermediate Graph data structure and build CSR/Table from it

Evaluating index creation overheads: building CSR and Hash-tables has a cost that must be properly accounted for in benchmarks. For example, you can amortize the creation cost over the cost of 100 queries vs just 1 query.

Graph-Traversal VS Hash-Join
Additional References

Graph Analytics at MIT, 2018 https://people.csail.mit.edu/jshun/6886-s18/

Roussopoulos, Nick, and Hyunchul Kang. "A pipeline n-way join algorithm based on the 2-way semijoin program." *IEEE Transactions on Knowledge and Data Engineering* 3.4 (1991): 486-495
https://ieeexplore.ieee.org/iel3/69/3315/00109109.pdf?casa_token=etWCzgbsSZQAAAAA:LZNQ3sWFfgfoIL-43D8xS5Ak9zPb6-29g_3SUNUS2Y3mdJl37Av92EXtiuluj486H9PMMHI

Chavan, Shasank, et al. "Accelerating joins and aggregations on the oracle in-memory database." 2018 ICDE. IEEE, 2018
https://ieeexplore.ieee.org/iel7/8476188/8509221/08509384.pdf?casa_token=ZNA-5bDQenYAAAAA:VK7OdgHiJHE-zqpk7WGZUyO8rLcbAK4FS-a6le1NKZHnnuL-bDgcwDY04pZ2jV3Shtx5Ezg

Dees, Jonathan, and Peter Sanders. "Efficient many-core query execution in main memory column-stores." 2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE, 2013
https://ieeexplore.ieee.org/iel7/6530811/6544790/06544838.pdf?casa_token=gGdMOuyC0vQAAAAA:mafYsFjMgSyR-ahMzqRa0chO5uFhb_4TThYc3THGjBh4xqUxMWIbyZlkIcLB0kmZOVKKyCo

Also look for recent conference proceedings of VLDB and SIGMOD