
ORACLE®

Domain Specific Languages for Parallel Graph AnalytiX (PGX)

Guido Wachsmuth

Principal Member of Technical Staff
Oracle Labs Zurich, Switzerland

January 8, 2019

Meet Oracle Labs

Guido Wachsmuth

Principal Member of Technical Staff

Oracle Labs Zurich, Switzerland

Meet Oracle Labs

Martijn Dwars

Member of Technical Staff

Oracle Labs Zurich, Switzerland

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Oracle Corporation

Scale

- US$40 billion total GAAP revenue in FY 2018
- 43k customers in 175 countries
- 25k partners
- More than 137k employees
- 14k support and services specialists
- 19k implementation consultants

Innovation

- More than US$48 billion in R&D over the last 10 years
- 38k developers and engineers
- 6.3M students supported annually in 128 countries
- More than 18K patents
- 5M registered members of the Oracle Developer Community
- 484 independent user communities in 92 countries

Oracle Leadership

#1

Technologies

- Application Server
- Database
- Database on Linux and Unix
- Data warehouse
- Deployment-centric applications platform
- Embedded database
- Engineered systems
- Middleware

Applications

- Business analytics
- Database management
- Enterprise performance management
- Lead management
- Marketing automation
- Structural data management
- Supply chain execution
- Talent management

Oracle Customers

AEROSPACE AND DEFENSE
10 out of 10 top companies

AUTOMOTIVE
20 out of 20 top companies

CLOUD
10 out of 10 top SaaS providers

CONSUMER GOODS
9 out of 10 top companies

EDUCATION AND RESEARCH
20 out of 20 top universities

#1

ENG. AND CONSTRUCTION
9 out of 10 top companies

#1

FINANCIAL SERVICES
20 out of 20 top banks

HIGH TECHNOLOGY
20 out of 20 top companies

#1

INSURANCE
20 out of 20 top insurers

MANUFACTURING
20 out of 20 top companies

#1

MEDICAL DEVICES
20 out of 20 top companies

OIL AND GAS
20 out of 20 top companies

#1

PHARMACEUTICALS
20 out of 20 top companies

#1

PUBLIC SECTOR
20 out of 20 top governments

#1

RETAIL
20 out of 20 top companies

#1

SUPPLY CHAINS
20 out of 20 top companies

#1

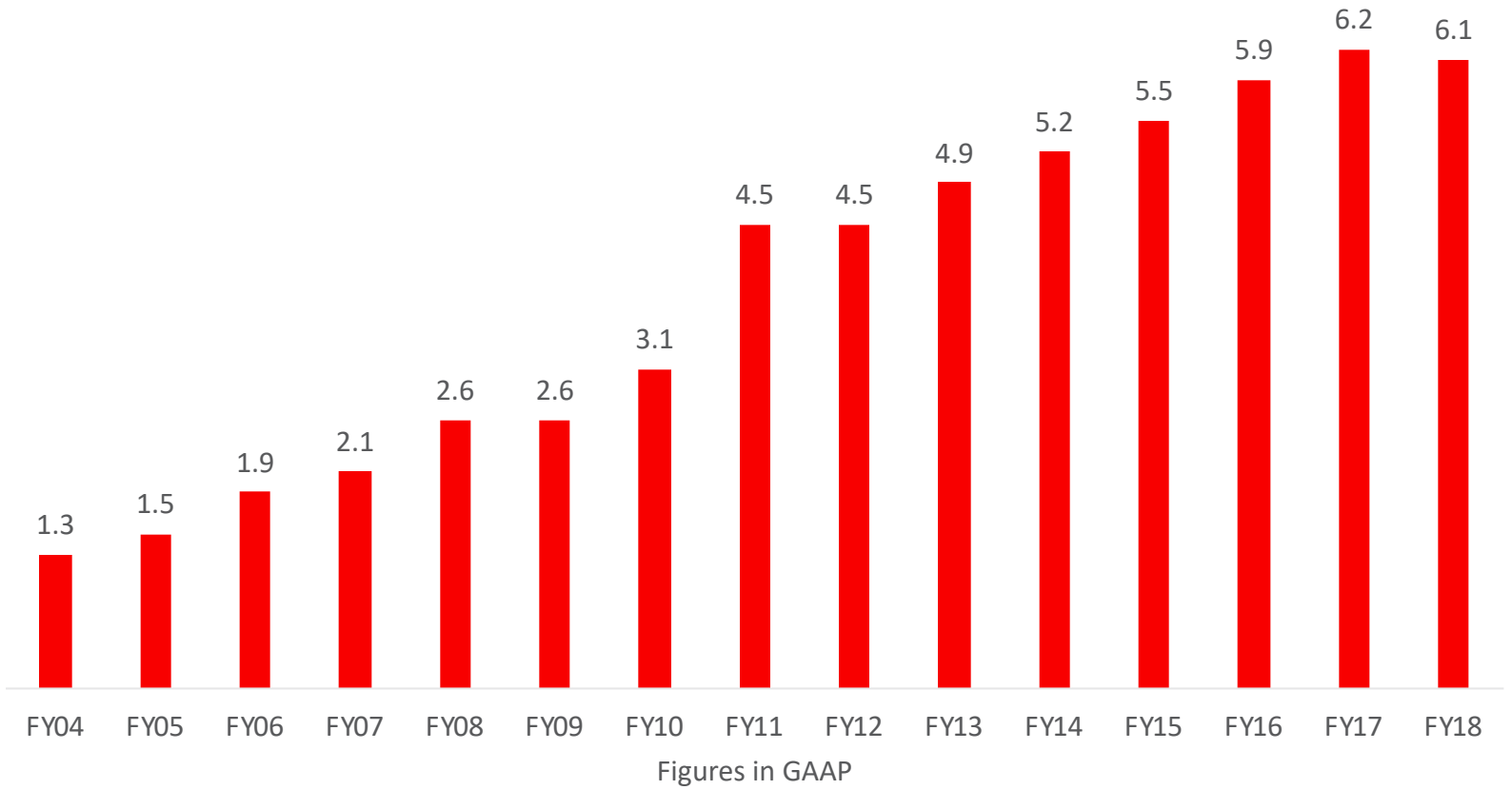
TELECOMMUNICATION
20 out of 20 top companies

TRAVEL AND TRANSPORT
20 out of 20 top airlines

#1

UTILITIES
10 out of 10 top companies

Investment in Research and Development



Figures in GAAP

MORE THAN $57 BILLIONS SINCE 2004

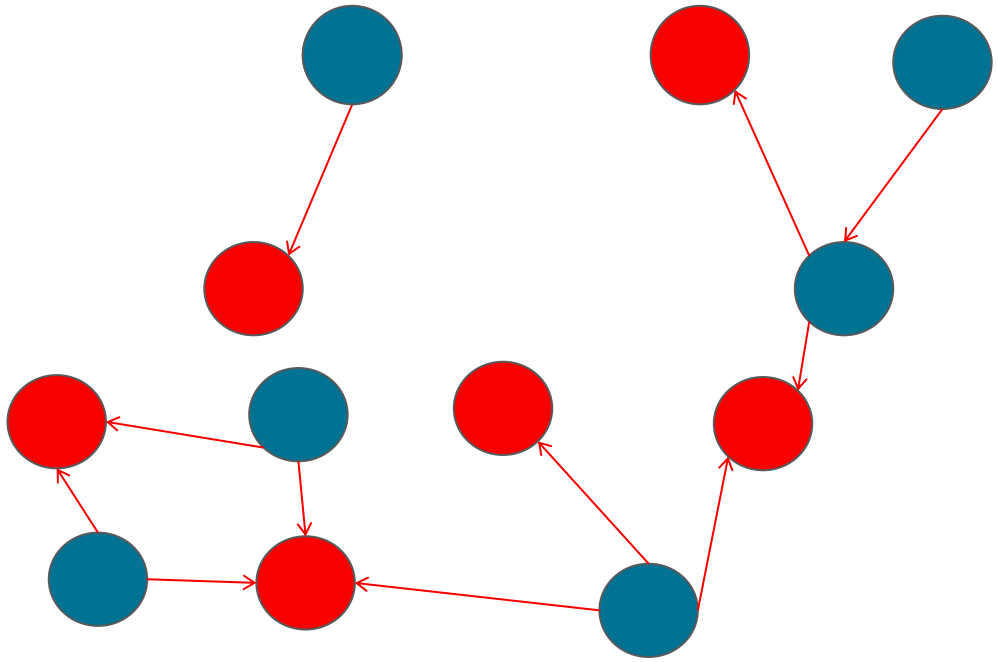
ORACLE LABS

"The Mission of Oracle Labs is straightforward:

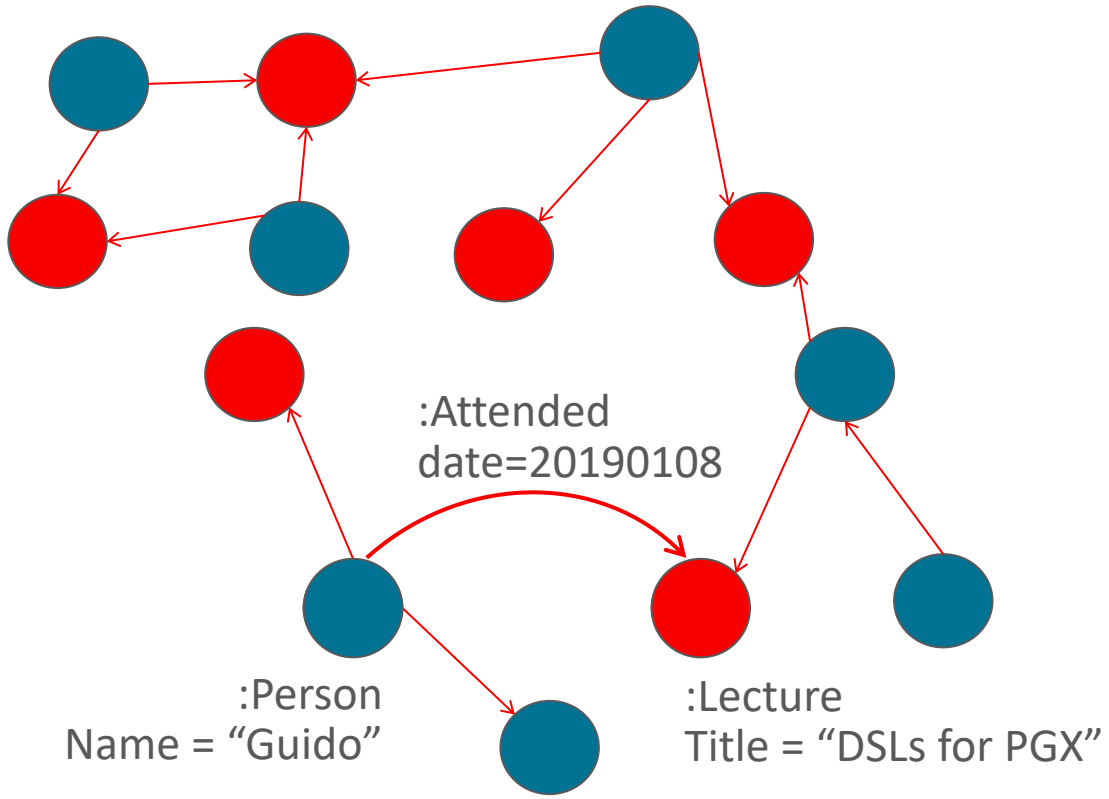
Identify, **explore**, and **transfer** *new technologies that have the potential to substantially improve Oracle's business*."

– Oracle Labs mission statement

Parallel Graph AnalytiX



Data as a Graph



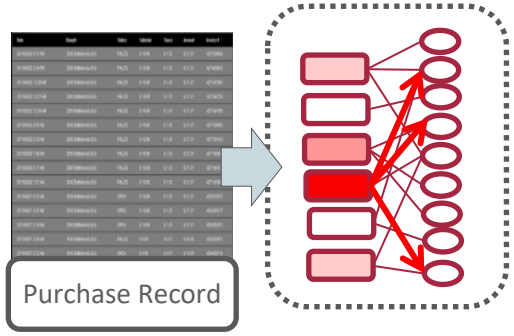
- Vertex: entity
- Edge: relationship

- Labels: identify vertices and edges
- Properties: describe vertices and edges

Example Applications

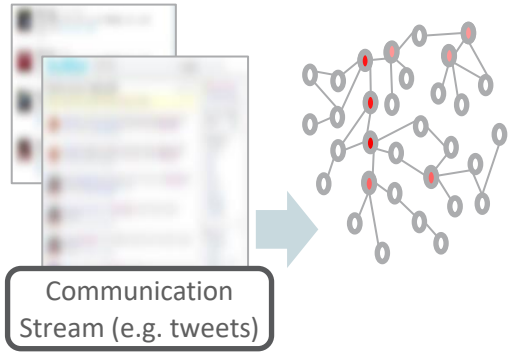
Product recommendation

Recommend the most **similar** item purchased by **similar** people



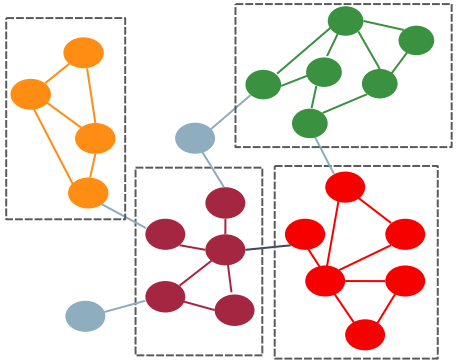
Influencer identification

Find out people that are **central** in the given network – e.g. influencer marketing



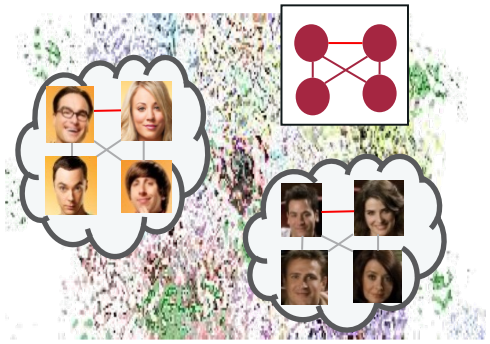
Community detection

Identify group of people that are **close to each other** – e.g. target group marketing



Graph pattern matching

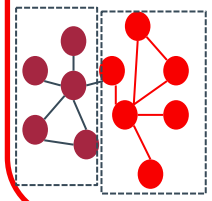
Find out all sets of entities that **match a given pattern** – e.g. fraud detection



Built-in Algorithms

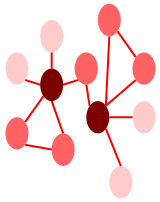
- PGX provides a rich set of built-in (parallel) graph algorithms

Detecting Components and Communities



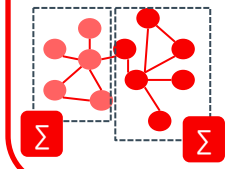
Tarjan's, Kosaraju's, Weakly Connected Components, Label Propagation (w/ variants), Soman and Narang's Spacification

Ranking and Walking



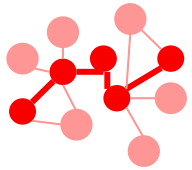
Pagerank, Personalized Pagerank, Betwenness Centrality (w/ variants), Closeness Centrality, Degree Centrality, Eigenvector Centrality, HITS, Random walking and sampling (w/ variants)

Evaluating Community Structures



Conductance, Modularity Clustering Coefficient (Triangle Counting) Adamic-Adar

Path-Finding

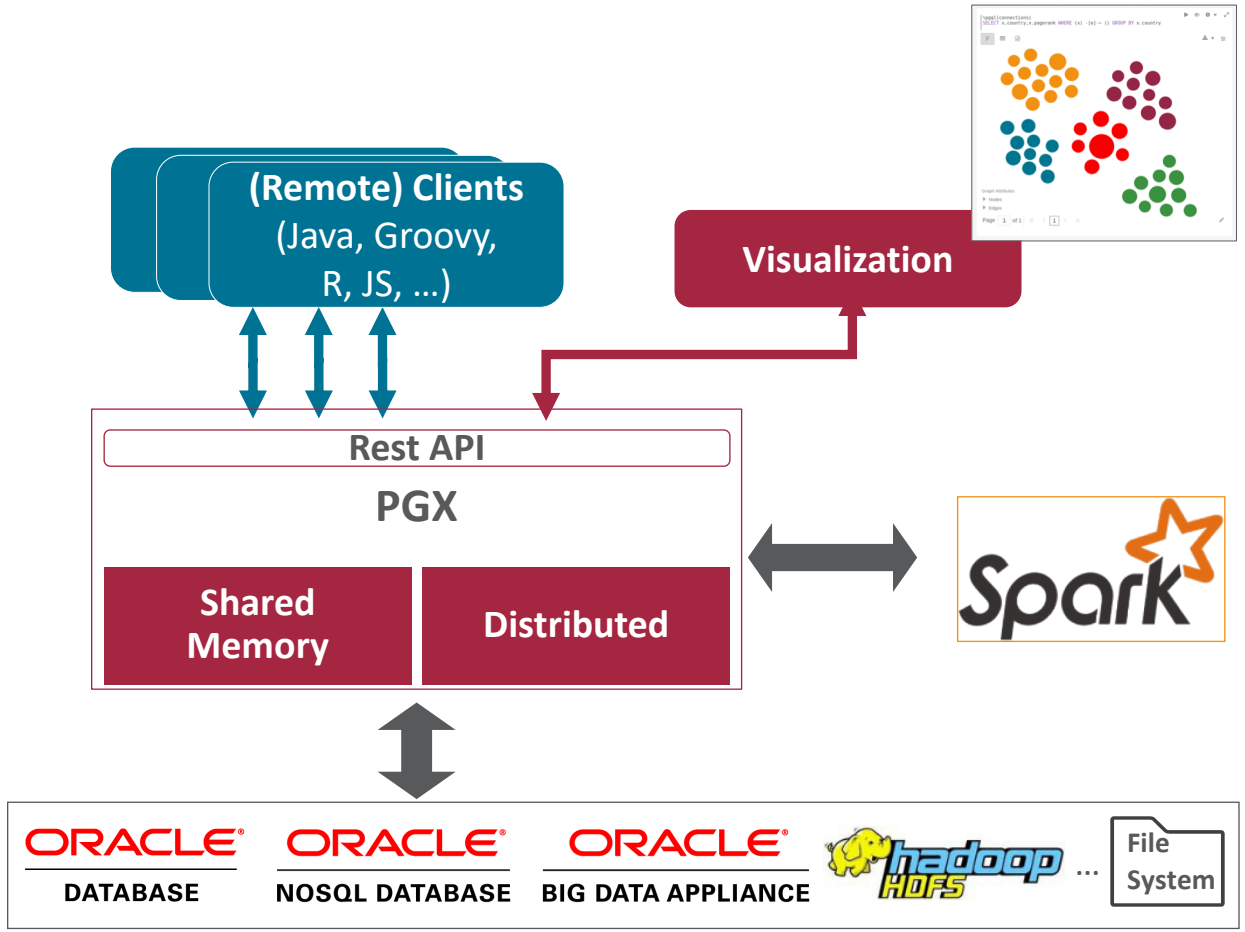


Hop-Distance (BFS) Dijkstra's, Bi-directional Dijkstra's Bellman-Ford's

Link Prediction SALSA (Twitter's Who-to-follow)

Other Classics Vertex Cover Minimum Spanning-Tree (Prim's)

Architecture Overview



- Integrated with various Oracle and open source data storage technologies
- Interoperates with open-source frameworks, e.g. Apache Spark
- In-memory engines for single-machine or distributed execution
- Multiple language bindings for remote execution via REST
- Notebooks and visualization

Graph Workloads

Computational graph analytics

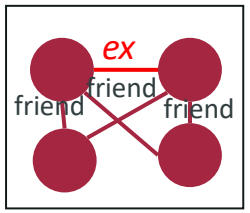
- **Compute** values on vertices and edges
- While **traversing** or **iterating** on the graph
- In **procedural** ways

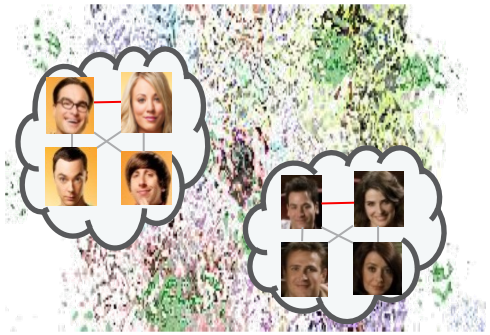
- Connected Components, Shortest Path, Spanning Tree, Pagerank, Centrality

Domain-specific language: Green-Marl

Graph pattern matching

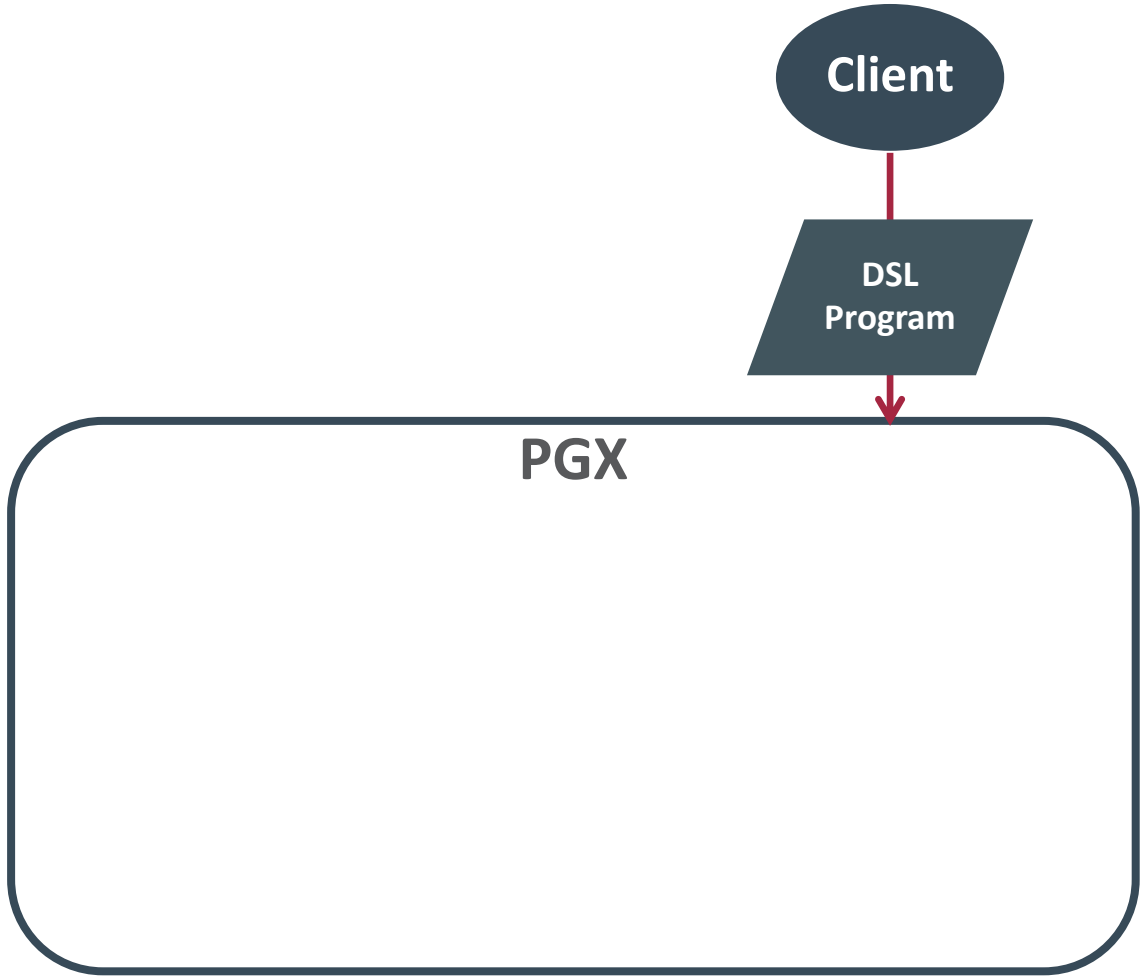
- Given a **description** of a pattern
- Find every subgraph which **matches** the pattern





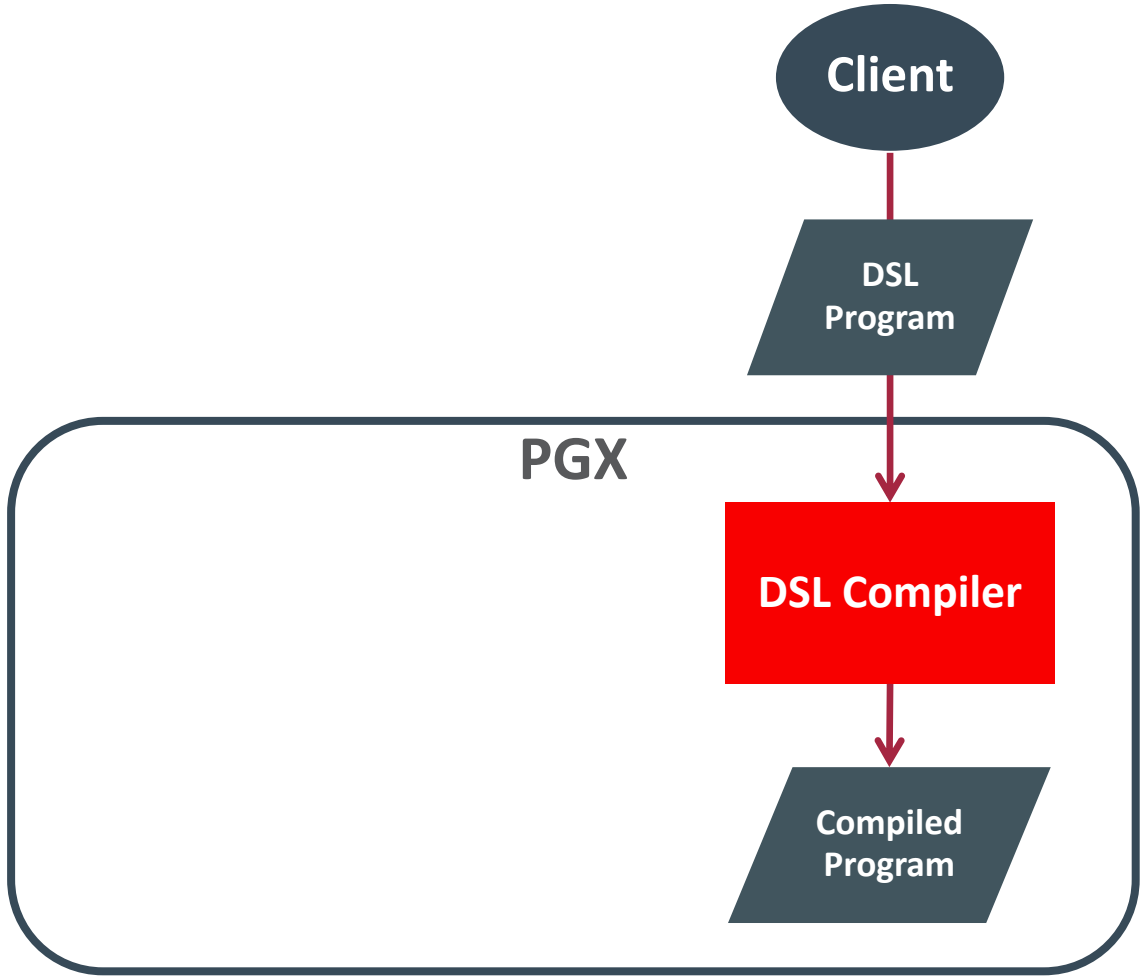
Domain-specific language: PGQL

PGX: Domain-Specific Languages



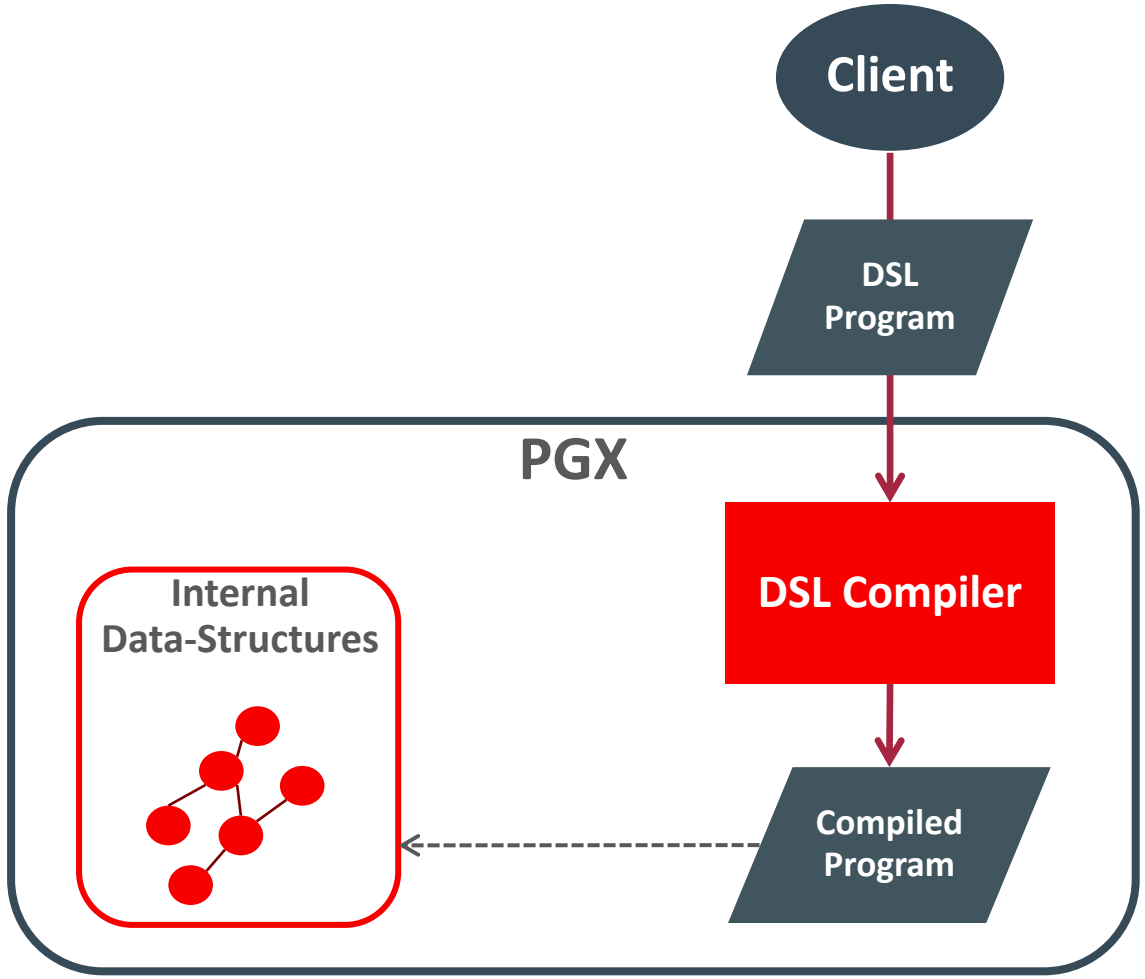
user implements algorithm/query using DSL and submits it to PGX

PGX: Domain-Specific Languages



DSL compiler aggressively optimizes code and generates compiled program

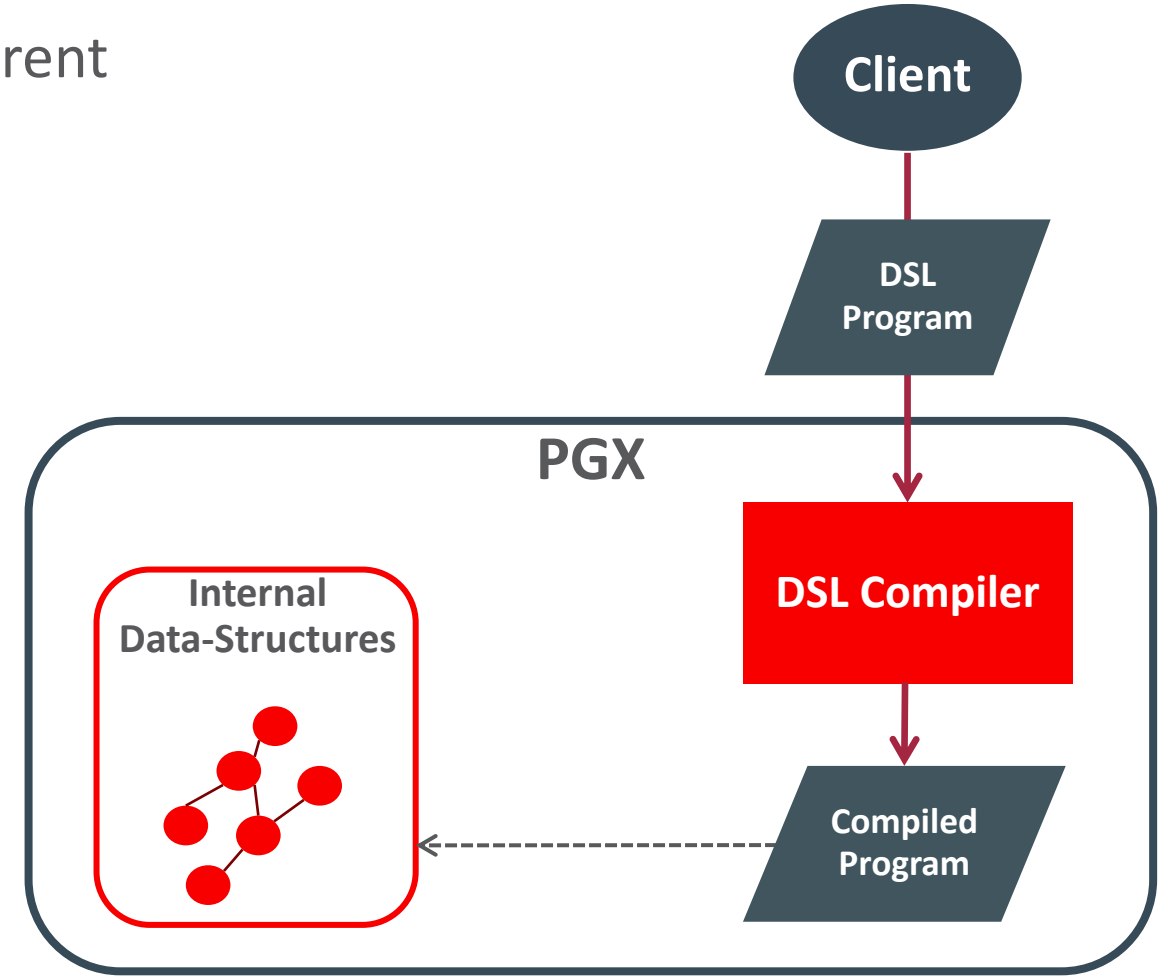
PGX: Domain-Specific Languages



compiled program has direct access to internal data-structures

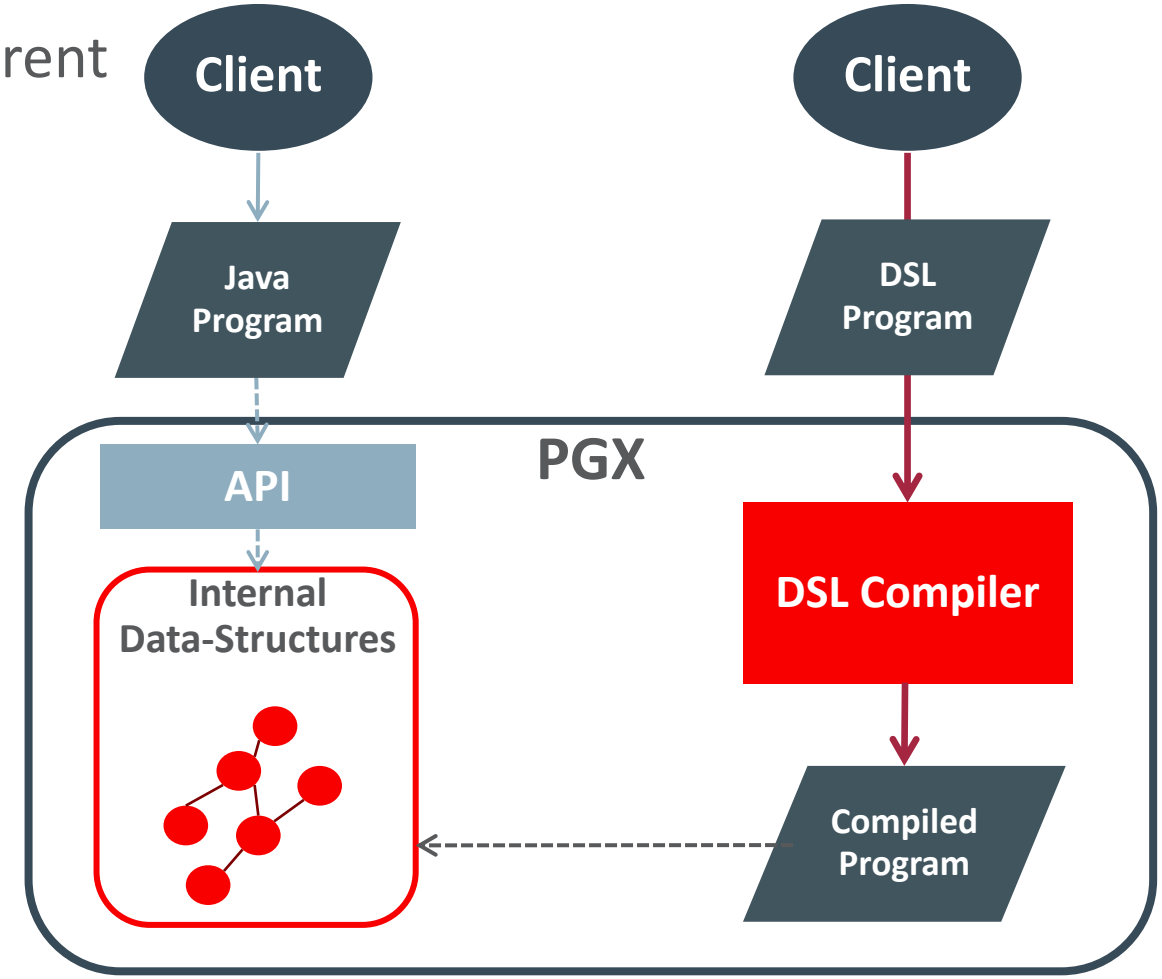
Compiled DSL Benefits

- portability: write once, compile for different backends
  - shared-memory
  - distributed



Compiled DSL Benefits

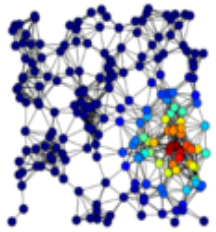
- portability: write once, compile for different backends
  - shared-memory
  - distributed
- performance benefits
  - graph specific optimizations
  - no performance overhead from API



Centrality Algorithms and there Application

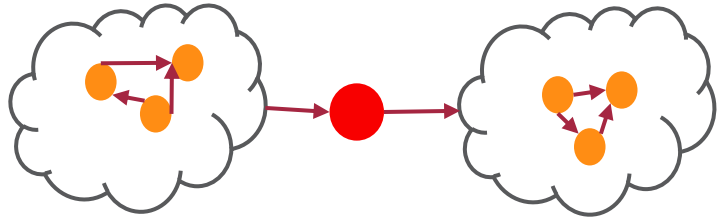
- What is this about:
  - "Find me the most important entity in this graph data set"
  - (but on what account?)
- From graph theory:
  - **Centrality** – a measure of relative importance of vertices in a graph n a graph
  - There are many different centralities defined

: Pagerank, Betweenness Centrality, HITS, Closeness Centrality, Eigenvector, ….

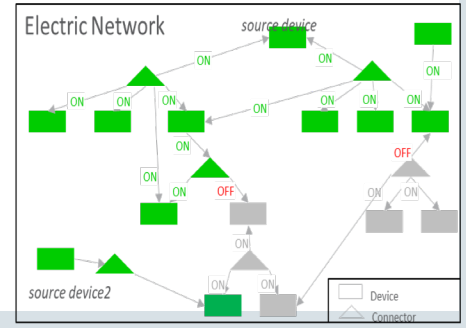


(images from Wikipedia)

- Betweeness Centrality
  - Vertices are considered important, if it is located "in between" other vertices



- Example Application: Asset criticality in Power Grid Network
  - Power lines, transformer, relay, …
  - Failure of which node makes larger damage?
  - In case of storm, which node should be fixed first?



Betweenness Centrality [Brandes 2001]

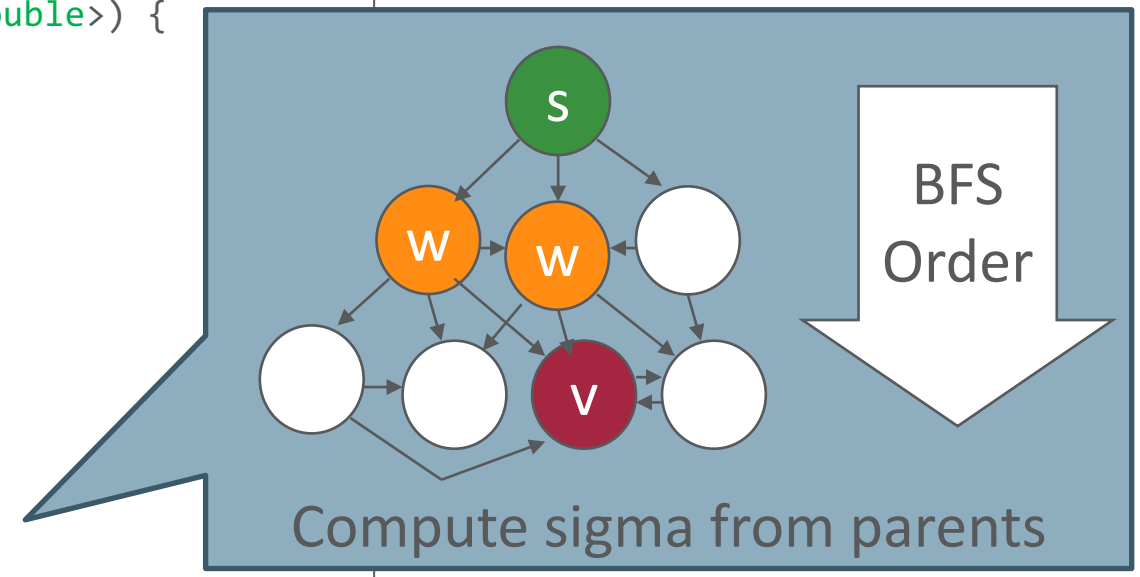
```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + (σ[v]/σ[w]) · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

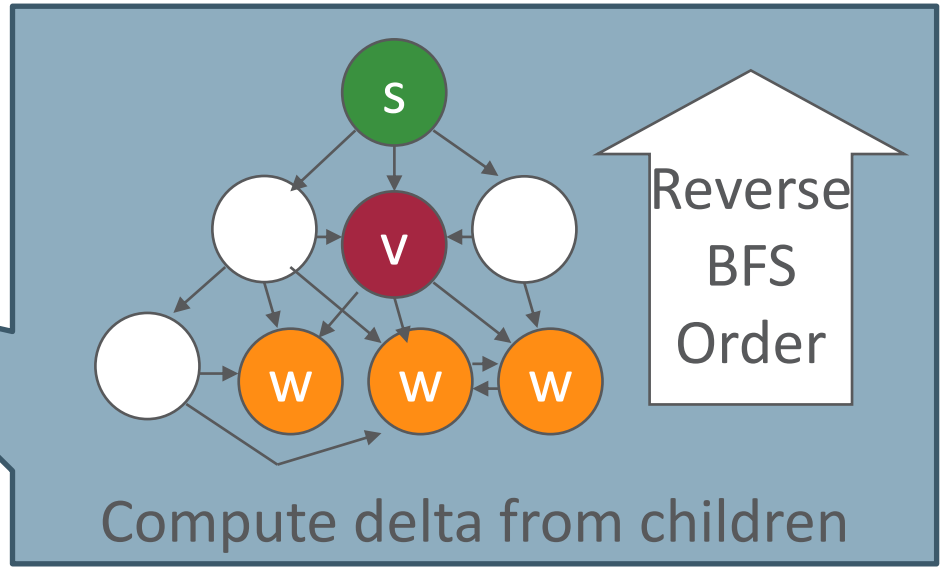
```
procedure BC (G: graph; BC: nodeProperty<double>) {
    G.BC = 0.0; // initialize

    foreach(s: G.nodes) {
        // temporary values for each node
        nodeProperty<double> sigma;
        nodeProperty<double> delta;
        G.sigma = 0.0;
        s.sigma = 1.0;

        // BFS iteration from s
        inBFS(v: G.nodes from s) {
            // summing over BFS parents
            v.sigma = sum(w: v.upNbrs) { w.sigma };
        }
        inReverse(v != s) { //reverse-BFS iteration to s
            // summing over BFS children
            v.delta = sum(w: v.downNbrs) {
                          v.sigma / w.sigma * (1 + w.delta)
                      };

            v.BC += v.delta; // accumulate BC
        }
    }
}
```

Parallel

Parallel

Parallel





Language Workbenches

Opportunities and Challenges

Challenges

usage modes

- ~~editor~~
- command line
- API for language processing

language composition

- open-source frontend
- proprietary backends

language build

- build system integration
- build times

3rd party runtime libraries

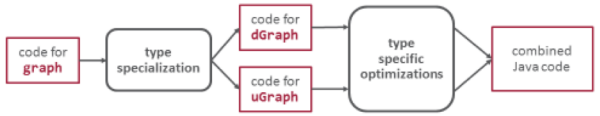
- vulnerabilities
- legal
- memory footprint

Opportunities: Innovation & Competiveness

CY 2017 language extensions for more use cases

Directed & Undirected Graph Support

- directed and undirected graph as explicit types in Green-Marl
- specify for which type of graph an algorithm is defined
- `dGraph:` directed graphs only
  `uGraph:` undirected graphs only
  `graph:` both types
- compiler can generate specialized code for each type



Undirected Traversal

- more control over the traversal direction in BFS & DFS
- traverse the graph in undirected way

```
inBFS (n: G.nodes from root using outEdges) {
  // traverse outgoing edges (default)
}

inBFS (n: G.nodes from root using inEdges) {
  // traverse incoming edges
}

inBFS (n: G.nodes from root using inOutEdges) {
  // traverse undirected (outgoing & incoming) edges
}
```

Ordered Iteration

- iterate over ranges in an ordered way
- only allowed for sequential iterations

```
for (n: G.nodes order by n.prop) {
  …
}

for (n: nSet.items order by 2 * n.prop desc) {
  …
}

for (n: G.nodes order by uniform()) {
  …
}
```

Local Procedures

- helper procedures that are visible only inside the same compilation unit
- allows more modular and cleaner code

```
proc random_walk(graph G, int length; nodeProp<string> walks) {
  foreach (n: G.nodes)
    n.walks = create_walk(G, n, length);
}

local create_walk(graph G, node n, int length) : string {
  string walk = ""; int i = 0;
  node current = n;
  while (i++ < length) {
    walk += current + " ";
    current = current.pickRandomNbr();
  }
  return walk;
}
```

Experimental: Graph Schemas

- define a local graph schema with properties
- use the graph schema as type in procedures
- specify properties once - use them everywhere
- prevents explosion of argument lists with local procedures

```
schema graph myGraph {
  nodeProp<int> size,
  mutable edgeProp<double> weight
}

procedure test(myGraph G, node n) {
  println(n.size);
}
```

specify properties once…

… (re)use them everywhere

Graph Mutation

- create new graphs
- add new nodes/edges
- remove nodes/edges

```
graph G;

mutation(G) {
  node n = G.createNode();
  node n2 = n.createNbr();
}

foreach (n: G.nodes) {
  …
}
```

- Additional data types and collections
- Additional syntactic sugar
- Additional optimisations

Example: MS-BFS Optimization

- **M**ulti-**S**ource **BFS***:
  - bundle multiple independent BFS traversals and run them at the same time
  - benefit from common parts in traversal and SIMD instructions
  - uses cache friendly data access scheme
  - up to 90x speedup compared to traditional BFS implementation
  - but hard to implement **by hand**
    - has to maintain copies for variables for each traversal in one bundle
    - has to manage control flow for different traversals
    - has to differentiate between private and shared variables

➔ **DSL compiler can do this automatically**

* Then, Kaufmann, Chirigati, Hoang-Vu, Pham, Kemper, Neumann, Vo:
The more the Merrier: Efficient Multi-Source Graph Traversal
VLDB 2014

Example: MS-BFS Optimization

```
foreach(s: G.nodes) {

  long foundNodes = 0;
  long levelSum = 0;

  inBFS(v: G.nodes from s) {
    foundNodes++;
    levelSum += currentBFSLevel();
  }

}
```

1. user writes regular Green-Marl code

Example: MS-BFS Optimization

```
foreach(s: G.nodes) {

  long foundNodes = 0;
  long levelSum = 0;

  inBFS(v: G.nodes from s) {
    foundNodes++;
    levelSum += currentBFSLevel();
  }

}
```

2. compiler identifies MS-BFS pattern…

Example: MS-BFS Optimization

```
batchForeach(s: G.nodes) {

  long foundNodes = 0;
  long levelSum = 0;

  inBatchBFS(v: G.nodes from s | index) {
    foundNodes++;
    levelSum += currentBFSLevel();
  }

}
```

3. …and transforms the code to do MS-BFS

Example: MS-BFS Optimization

```
batchForeach(s: G.nodes) {

  long foundNodes = 0;
  long levelSum = 0;

  inBatchBFS(v: G.nodes from s | index) {
    foundNodes++;
    levelSum += currentBFSLevel();
  }

}
```

4. then identifies batch-local variables…

Example: MS-BFS Optimization

```
batchForeach(s: G.nodes) {

  long[] foundNodes = new long[batchSize];
  long[] levelSum = new long[batchSize];

  for(int i = 0; i < batchSize; i++) {
    foundNodes[i] = 0;
    levelSum[i] = 0;
  }

  inBatchBFS(v: G.nodes from s | index) {
    foundNodes[index]++;
    levelSum[index] += currentBFSLevel();
  }

}
```

5. …and creates one instance per iteration in the batch

Example: MS-BFS Optimization

- MS-BFS greatly improves the performance
- but creating copies of each variable can have huge impact on memory consumption (e.g. node/edge properties)
- system might run out of memory for larger graph instances

➔using static analysis the compiler determines memory consumption per thread per BFS iteration

➔compiler generates code that reduces batch-size so that all data fits into memory

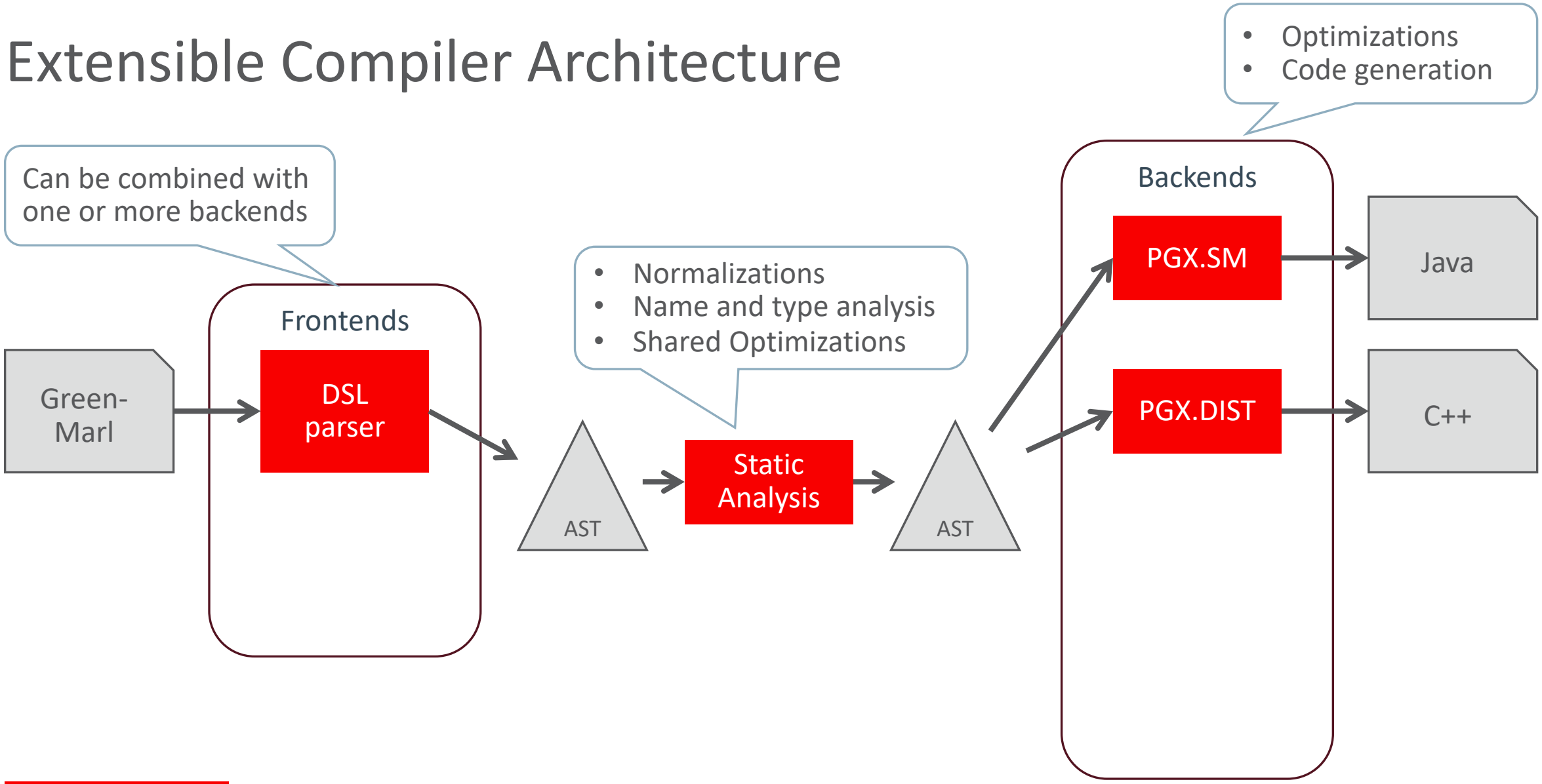
Example: MS-BFS Optimization

Conclusion:

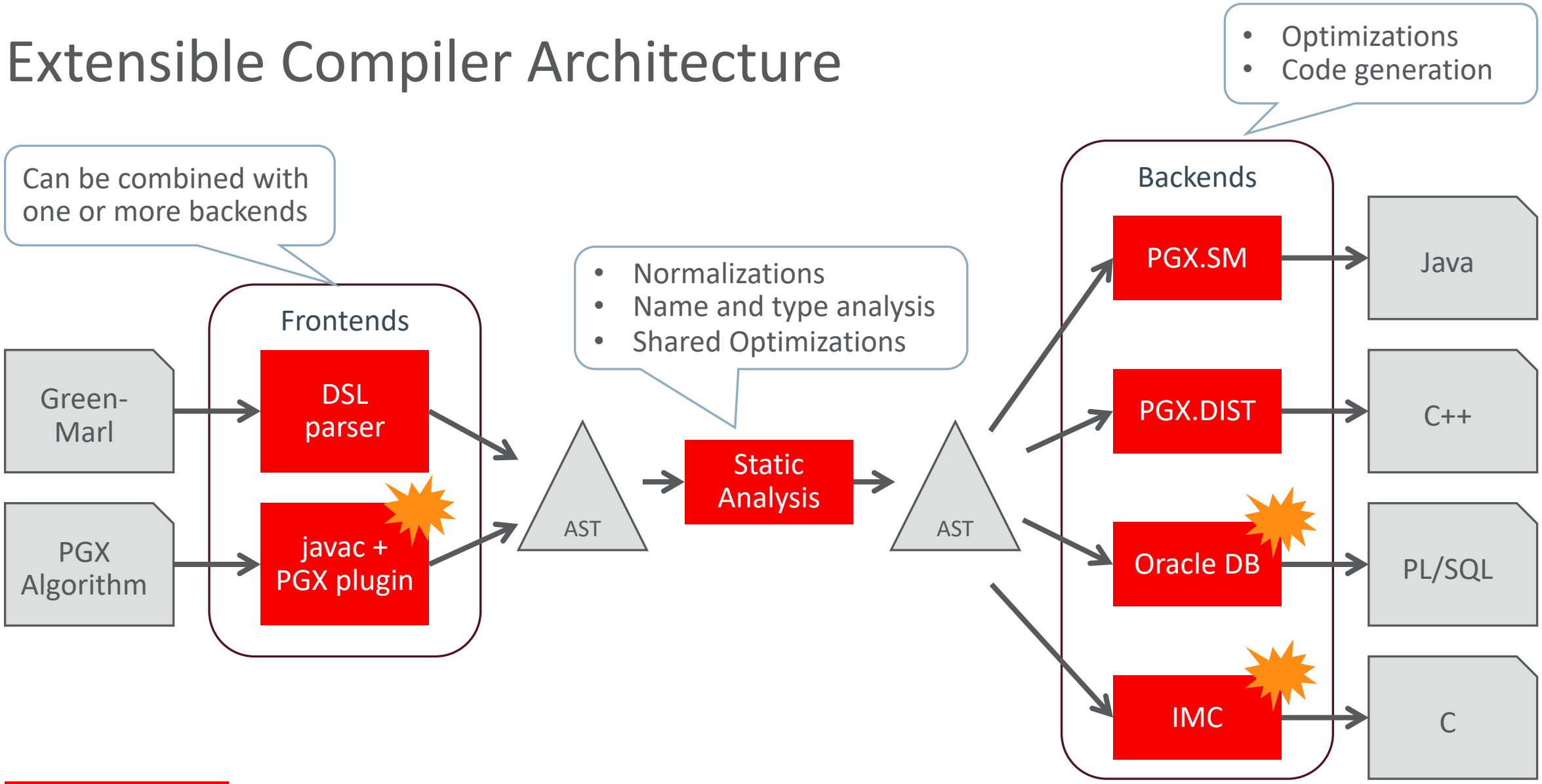
- MS-BFS greatly improves the performance
- auto-transformation fully transparent to the user
- transparent memory handling

➔ Green-Marl code focuses only on the actual problem, everything else is handled by the compiler

Extensible Compiler Architecture



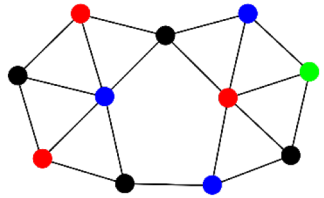
Extensible Compiler Architecture



```
@GraphAlgorithm public class BetweennessCentrality {
  public void bc_full(PgxGraph g, @Out VertexProperty<Double> bc) {
    bc.setAll(0d); // Initialize
    g.getVertices().forEach(s -> {
      VertexProperty<Double> sigma = VertexProperty.create();
      VertexProperty<Double> delta = VertexProperty.create();
      sigma.setAll(0d); sigma.set(s, 1d);
      inBFS(g, s)
        .forward(v -> sigma.set(v, v.getUpNeighbors().sum(sigma)))
        .backwardFilter(v -> v != s)
        .backward(v -> {
          delta.set(v, v.getDownNeighbors().sum(w -> (1 + delta.get(w)) / sigma.get(w)) `1* sigma.get(v));
          bc.reduceAdd(v, delta.get(v));
        });
    });
  }
}
```


ORACLE®

Property graph query languages



input graph

Query

output table

customer	time
"Maria"	2010-12-17T02:42:32
"Iksana"	2011-01-01T20:57:21
"Franklin"	2012-09-29T02:49:09

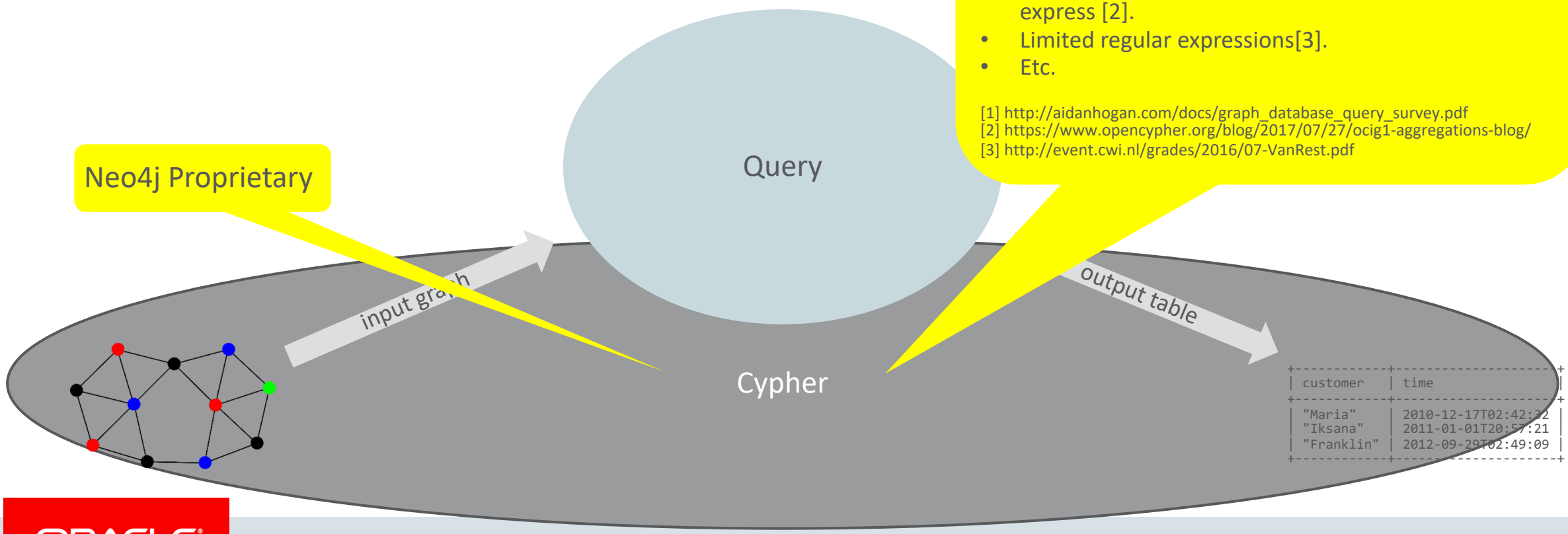
Property graph query languages

Issues with language semantics

- Edges in query must all map to different edges in graph, making query evaluation intractable[1] and making certain queries hard to express.
- GROUP BY and SELECT are merged into a single RETURN clause, making certain queries hard to express [2].
- Limited regular expressions[3].
- Etc.

[1] http://aidanhogan.com/docs/graph_database_query_survey.pdf
[2] https://www.opencypher.org/blog/2017/07/27/ocig1-aggregations-blog/
[3] http://event.cwi.nl/grades/2016/07-VanRest.pdf

Neo4j Proprietary

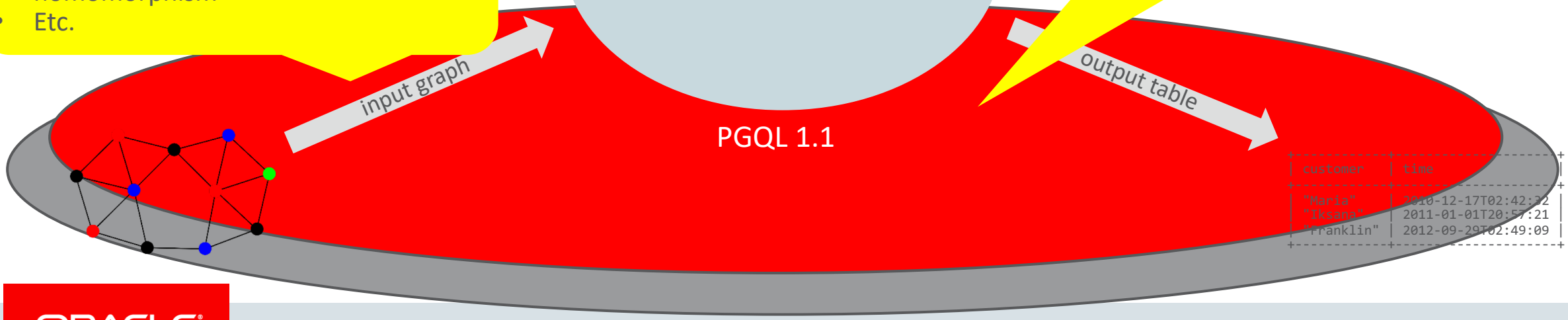


Property graph query languages

Advances in property graph querying
- Full regular expressions
- Pattern matching based on subgraph homomorphism
- Etc.

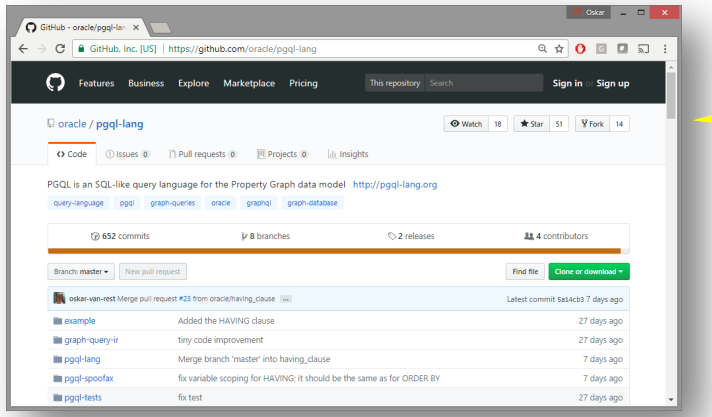
Query

Syntax and semantics aligned with SQL
- Well-established foundation
- Familiarity for SQL users (e.g. GROUP BY, HAVING, ORDER BY)

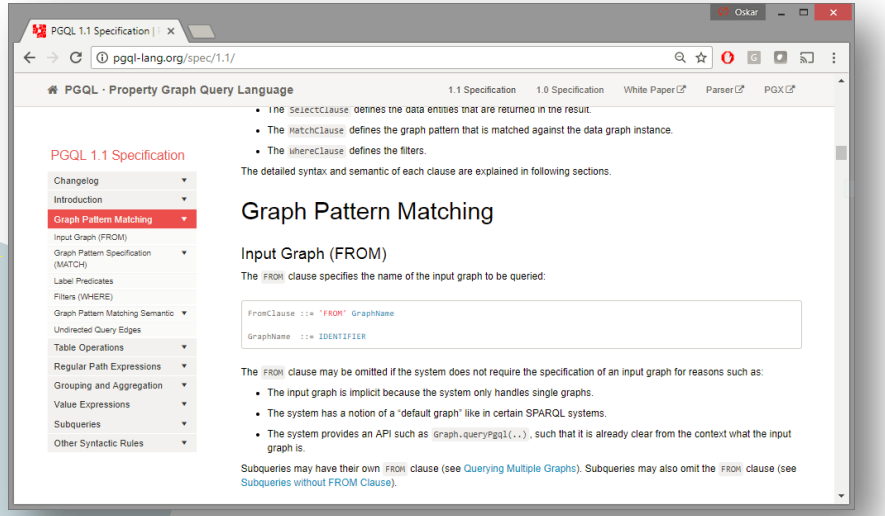


Property graph query languages

Open-sourced parser:
https://github.com/oracle/pgql-lang/



Language specification: http://pgql-lang.org/spec/1.1/



Open-sourced parser

Language specification

Query

input graph

output table

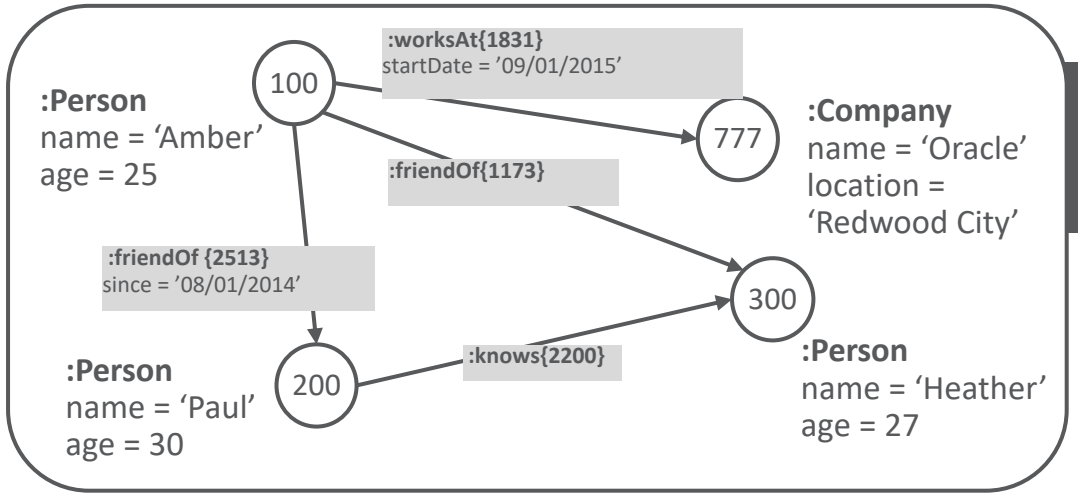
PGQL 1.1

customer	time
"Maria"	2010-12-17T02:42:32
"Ikzarg"	2011-01-01T20:57:21
"Franklin"	2012-09-29T02:49:09

Basic graph pattern matching

- Find all instances of a given pattern/template in the data graph

```
SELECT v3.name, v3.age
  FROM socialNetworkGraph
 MATCH (v1:Person) -[:friendOf]-> (v2:Person) -[:knows]-> (v3:Person)
 WHERE v1.name = 'Amber'
```



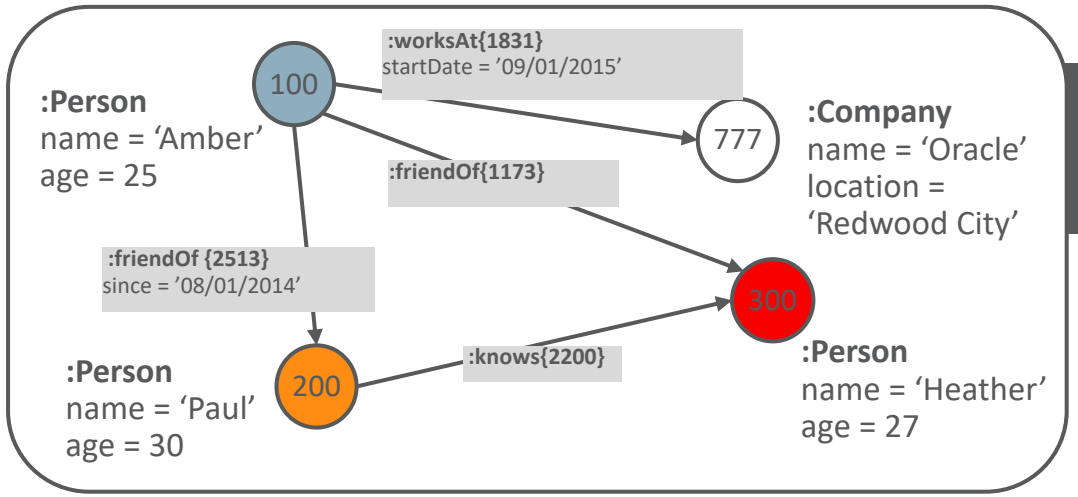
socialNetwork Graph

Query: Find all people who are known by friends of 'Amber'.

Basic graph pattern matching

- Find all instances of a given pattern/template in the data graph

```
SELECT v3.name, v3.age
  FROM socialNetworkGraph
 MATCH (v1:Person) -[:friendOf]-> (v2:Person) -[:knows]-> (v3:Person)
 WHERE v1.name = 'Amber'
```



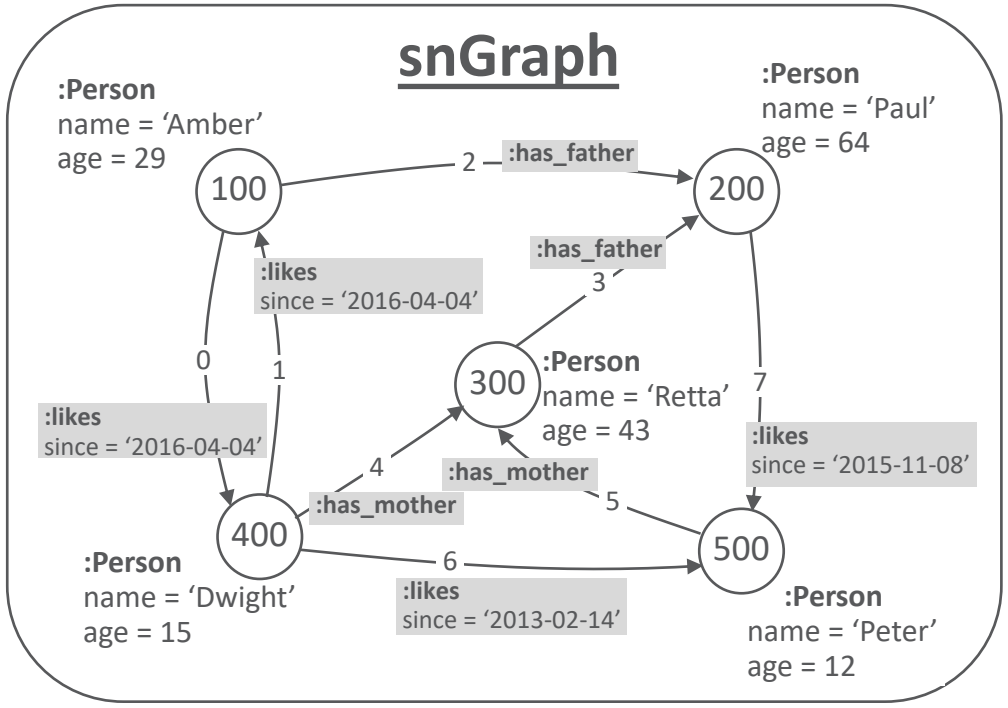
socialNetwork Graph

Query: Find all people who are known by friends of 'Amber'.

Regular path expressions

- Matching a pattern repeatedly
  - Define a **PATH** expression at the top of a query
  - Instantiate the expression in the **MATCH** clause
  - Match **repeatedly**, e.g. zero or more times (*) or one or more times (+)

```
  PATH has_parent AS (child) -[:has_father|has_mother]-> (parent)
SELECT x.name, y.name, ancestor.name
  FROM snGraph
 MATCH (x:Person) -/:has_parent+/-> (ancestor)
     , (y) -/:has_parent+/-> (ancestor)
 WHERE x.name = 'Peter' AND x <> y
```



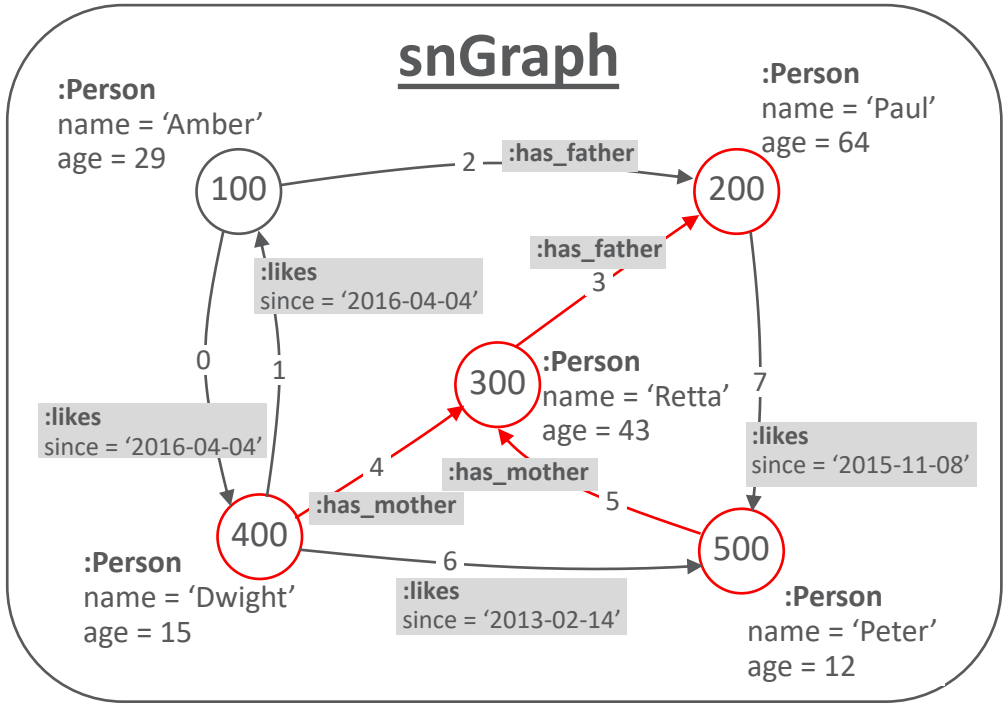
Regular path expressions

- Matching a pattern repeatedly
  - Define a **PATH** expression at the top of a query
  - Instantiate the expression in the **MATCH** clause
  - Match **repeatedly**, e.g. zero or more times (*) or one or more times (+)

```
  PATH has_parent AS (child) -[:has_father|has_mother]-> (parent)
SELECT x.name, y.name, ancestor.name
  FROM snGraph
 MATCH (x:Person) -/:has_parent+/-> (ancestor)
     , (y) -/:has_parent+/-> (ancestor)
 WHERE x.name = 'Peter' AND x <> y
```

Result set

```
+--------+--------+---------------+
| x.name | y.name | ancestor.name |
+--------+--------+---------------+
| Peter  | Retta  | Paul          |
| Peter  | Dwight | Paul          |
| Peter  | Dwight | Retta         |
+--------+--------+---------------+
```



Example: Network Impact Analysis

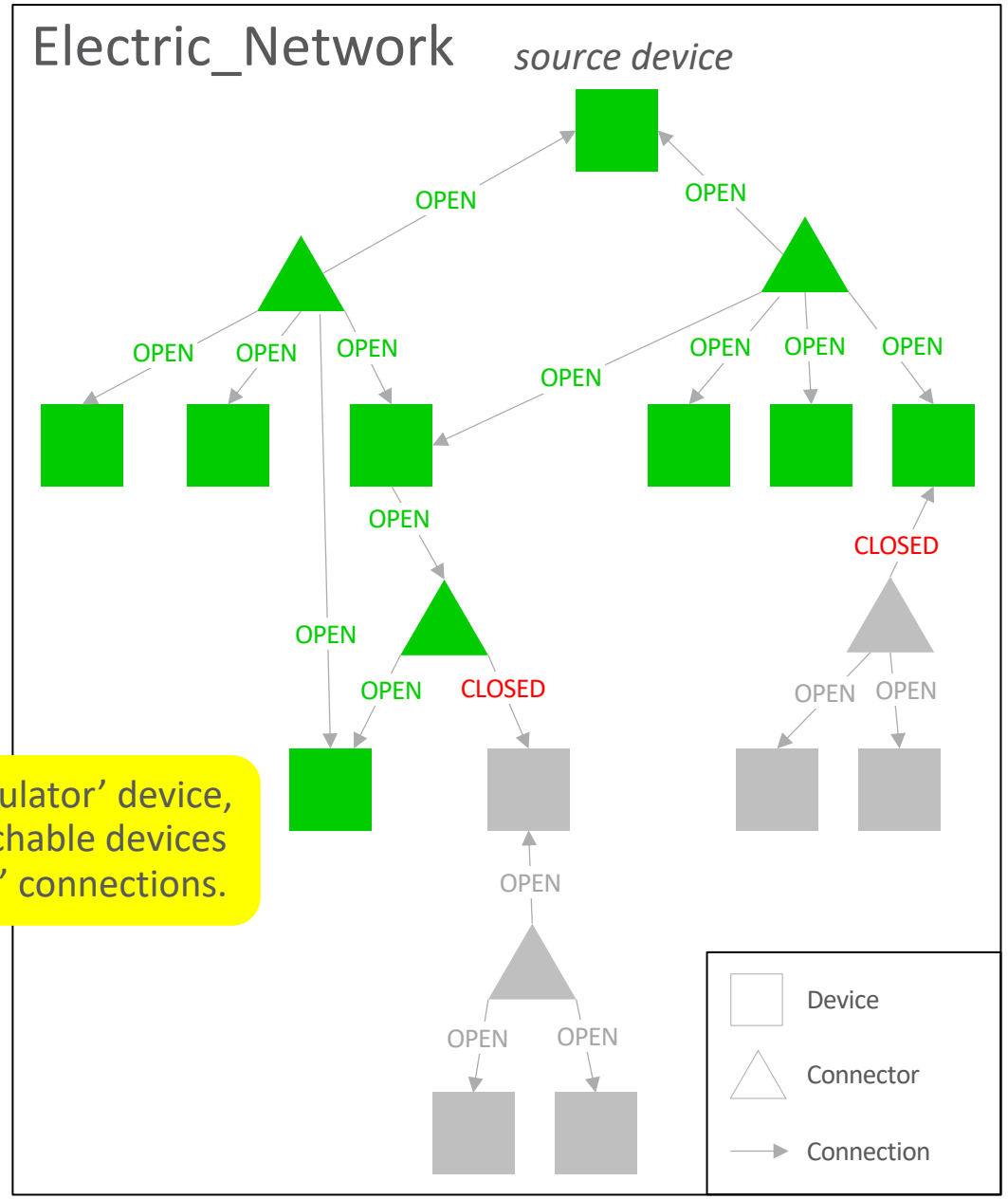
- How does network disruption impacts reachability between electric devices?

```
    PATH connects_to
         AS (from) <-[c1]- (connector) -[c2]-> (to)
         WHERE c1.status = 'OPEN'
           AND c2.status = 'OPEN'
  SELECT n.nickname, COUNT(m)
    FROM Electric_Network
   MATCH (n:Device) -/:connects_to*/-> (m:Device)
   WHERE java_regexp_like(n.nickname, 'Regulator')
     AND n <> m
GROUP BY n
ORDER BY COUNT(m) DESC, n.nickname
```

Query: For each 'Regulator' device, show number of reachable devices following only 'OPEN' connections.

Example result

n.nickname	COUNT(m)
Regulator, VREG2_A	1596
Regulator, VREG4_B	1537
Regulator, VREG4_C	1537
Regulator, HVMV_Sub_RegA	3
Regulator, HVMV_Sub_RegB	3



Patterns, labels, regular expressions

Syntax	Description
`(v1)`	Match vertex v1
`-[e1]->`	Match outgoing edge e1
`<-[e1]-`	Match incoming edge e1
`-[e1]-`	Match incoming or outgoing edge e1
`()`	Match anonymous vertex

`->`
`<-`
`-`
	Match anonymous outgoing/incoming/any-directional edge

`(v1:lbl1|lbl2)`	Match vertex v1 with label lbl1 or lbl2
`-[:lbl1]->`	Match anonymous outgoing edge with label lbl1

Syntax	Description
`-/:lbl/->`	Matches if there exists a path that connects the source and destination by **one** match of the given pattern
`-/:lbl*/->`	Matches if there exists a path that connects the source and destination by **zero or more** matches of the given pattern
`-/:lbl+/->`	Matches if there exists a path that connects the source and destination by **one or more** matches of the given pattern
`-/:lbl?/->`	Matches if there exists a path that connects the source and destination by **zero or one** matches of the given pattern
`-/:lbl{2}/->`	Matches if there exists a path that connects the source and destination by **exactly 2** matches of the given pattern
`-/:lbl{2,}/->`	Matches if there exists a path that connects the source and destination by **at least 2** matches of the given pattern
`-/:lbl{2,3}/->`	Matches if there exists a path that connects the source and destination by **at least 2 and at most 3 (inclusive)** matches of the given pattern
`-/:lbl{,3}/->`	Matches if there exists a path that connects the source and destination by **at least 0 and at most 3 (inclusive)** matches of the given pattern

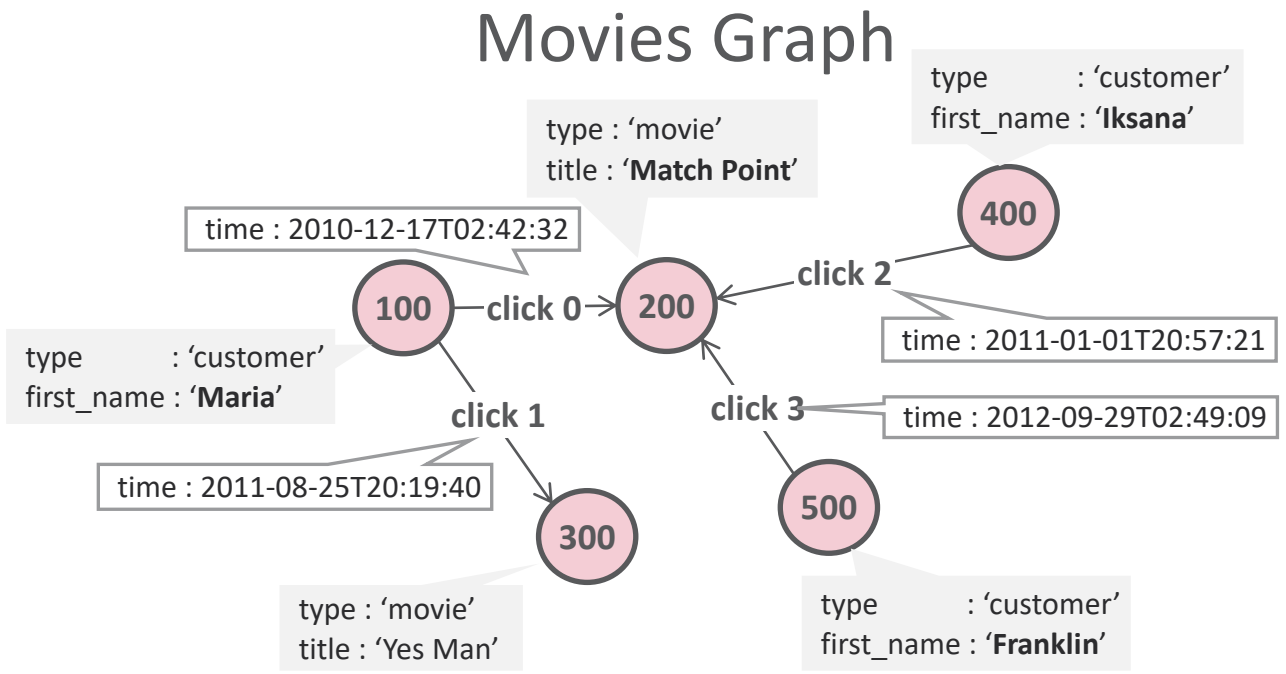
Cartesian product

- Disconnected graph patterns result in Cartesian product

```
SELECT c1.first_name, c2.first_name
  FROM "Movies Graph"
 MATCH (c1), (c2) -[:click]-> (m)
 WHERE c1.first_name = 'Maria'
     , m.title = 'Match Point'
```

```
+---------------+     +---------------+
| c1.first_name |     | c2.first_name |
+---------------+     +---------------+
| Maria         |  X  | Maria         |
+---------------+     | Iksana        |
                      | Franklin      |
                      +---------------+
```

```
=
+---------------+---------------+
| c1.first_name | c2.first_name |
+---------------+---------------+
| Maria         | Maria         |
| Maria         | Iksana        |
| Maria         | Franklin      |
+---------------+---------------+
```



SELECT and WHERE clause

- WHERE (optional) contains a filter expression:
  - The filter expression typically involves properties of vertices/edges defined in the MATCH clause
    - e.g. **WHERE** m.budget > 1000000 **AND** m.budget < 2000000 **AND** …
- SELECT creates tabular projection of results
  - Specify alias names through **SELECT** `exp` **AS** `var`
    - e.g. **SELECT** (m.budget / 1000000.0) **AS** budget_in_millions, …
  - SELECT DISTINCT makes a set out of a multiset (i.e. removes duplicates)
- The expression system is similar to the one in SQL:

Operator type	Operator	Example PGQL
Arithmetic	+, -, *, /, %	1 + 1
Relational	=, <>, <, >, <=, >=	3 < 4
Logical	AND, OR, NOT	true **AND NOT** false

Data types in PGQL on PGX
(note: in PGQL-to-SQL, the available data types correspond to those in Oracle RDBMS)

Data type	Name in PGX config	Example PGQL literal	Implicitly converts to (e.g. integer + long => LONG)
STRING	string	`'Franklin'`	
INTEGER	integer	`-`	LONG, FLOAT, DOUBLE
LONG	long	`2000`	FLOAT, DOUBLE
FLOAT	float	`-`	DOUBLE
DOUBLE	double	`1.95`	
BOOLEAN	boolean	`true`	
DATE	local_date (note: don't use deprecated *date*)	`DATE '2017-08-18'`	
TIME	time	`TIME '20:15:00'`	TIME WITH TIME ZONE
TIMESTAMP	timestamp	`TIMESTAMP '2017-08-18 20:15:00'`	TIMESTAMP WITH TIME ZONE
TIME WITH TIME ZONE	time_with_timezone	`TIME '20:15:00+08'`	TIME
TIMESTAMP WITH TIME ZONE	timestamp_with_timezone	`TIMESTAMP '2017-08-18 20:15:00+08'`	TIMESTAMP

Built-in functions

Function	Description	Return type
id(vertex/edge)	get the identifier of a vertex or edge	exact numeric / string
has_label(vertex/edge, lbl)	returns true if the vertex/edge has the label; false otherwise	boolean
labels(vertex)	returns the labels of a vertex	set<string>
label(edge)	returns the label of an edge	string
all_different(value1, value2, value3, …)	returns true if the values are all different, e.g. all_different(1, 5, 2) ➔ true e.g. all_different(1, 5, 1) ➔ false	boolean
in_degree(vertex)	get outgoing number of edges	exact numeric
out_degree(vertex)	get incoming number of edges	exact numeric
java_regexp_like(string, pattern)	returns whether the string matches the pattern[1]	boolean

Use **all_different** to avoid explosion of non-equality constraints:

```
SELECT COUNT(*)
 MATCH (p1:Person) -[:likes]-> (friend:Person)
     , (p2:Person) -[:likes]-> (friend)
     , (p3:Person) -[:likes]-> (friend)
 WHERE p1 <> p2 AND p1 <> p3 AND p2 <> p3
```

```
SELECT COUNT(*)
 MATCH (p1:Person) -[:likes]-> (friend:Person)
     , (p2:Person) -[:likes]-> (friend)
     , (p3:Person) -[:likes]-> (friend)
 WHERE all_different(p1, p2, p3)
```

[1] see https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

CAST

- Convert from string:
  - **CAST**('true' **AS BOOLEAN**) => true
  - **CAST**('2.50' **AS DOUBLE**) => 2.50
  - **CAST**('2017-08-18' **AS DATE**) => **DATE** '2017-08-18‘
- Convert to string:
  - **CAST**(true **AS STRING**) => 'true'
  - **CAST**(2.50 **AS STRING**) => '2.50'
  - **CAST**(**DATE** '2017-08-18' **AS STRING**) => '2017-08-18'
- Narrowing numeric conversion
  - **CAST**(1.2 **AS INTEGER**) => 1

- Extract date or time portion from a timestamp:
  - **CAST**(**TIMESTAMP** '2017-08-18 20:15:00' **AS DATE**) => **DATE** '2017-08-18'
  - **CAST**(**TIMESTAMP** '2017-08-18 20:15:00' **AS TIME**) => **TIME** '20:15:00'
- Create a timestamp from a date or a time:
  - **CAST**(**DATE** '2017-08-18' **AS TIMESTAMP**) => **TIMESTAMP** '2017-08-18 00:00:00'
  - **CAST**(**TIME** '20:15:00' **AS TIMESTAMP**) => **TIMESTAMP** '2018-03-19 20:15:00'

It takes the current date to create a timestamp

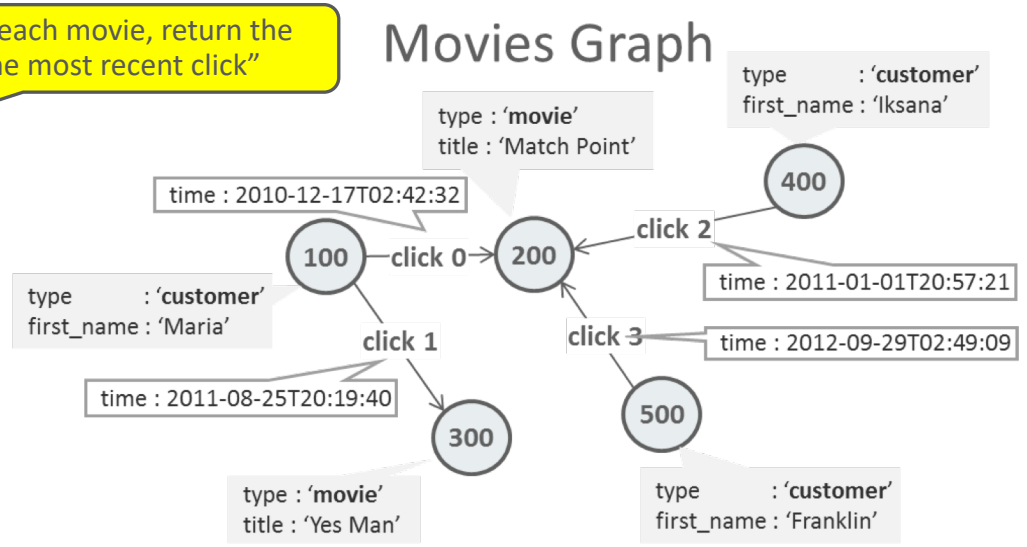
Grouping and aggregation

- `GROUP BY` creates groups of intermediate solutions
- `COUNT, MIN, MAX, SUM, AVG` to compute aggregations over groups of solutions
  - Use `DISTINCT` to eliminate duplicates before aggregation
    - E.g. `COUNT(DISTINCT n.name)`
  - Omit `GROUP BY` to aggregate over the entire set of solutions (returns a single row)
- `HAVING` filters entire groups of solutions
  - E.g. **HAVING COUNT**(*) > 10

Query: "for each movie, return the time of the most recent click"

```
  SELECT m.title, MAX(e.time)
    FROM "Movies Graph"
   MATCH (m) <-[e:click]- ()
   WHERE m.type = 'movie'
GROUP BY m
```

```
+--------------+---------------------+
| m.title      | MAX(e.time)         |
+--------------+---------------------+
| "Yes Man"    | 2011-08-25T20:19:40 |
| "Match Point"| 2010-12-17T02:42:32 |
+--------------+---------------------+
```



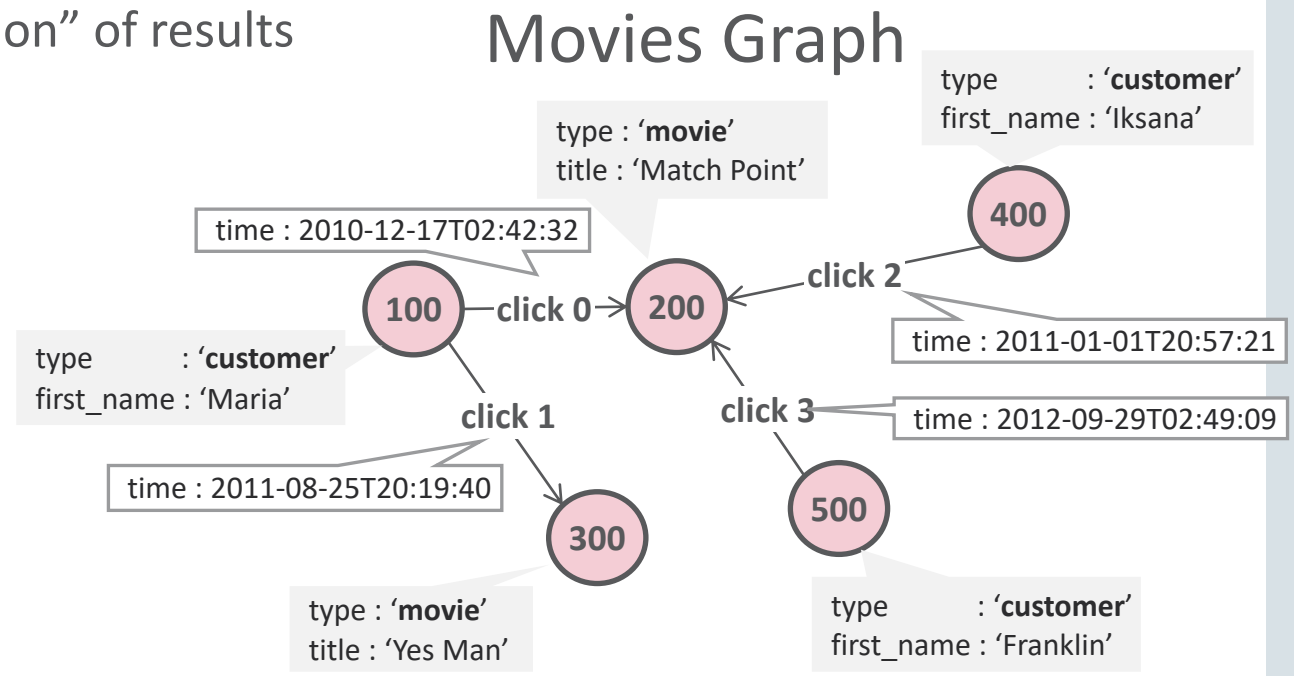
ORDER BY, LIMIT, OFFSET

- ORDER BY
  - Sort result ascendingly (default): **ORDER BY** n.prop **ASC**, …
  - Sort result descendingly: **ORDER BY** n.prop **DESC**, …
- LIMIT and OFFSET
  - For returning the "top-k" results or for "pagination" of results
    - E.g. **OFFSET** 200 **LIMIT** 100 returns results 200 to 300

Query: "order customers by first name and return first two results"

```
  SELECT n.first_name
   MATCH (n)
   WHERE n.type = 'customer'
ORDER BY n.first_name
   LIMIT 2
```

```
+--------------+
| n.first_name |
+--------------+
| Franklin     |
| Iksana       |
+--------------+
```



EXISTS subqueries

- **EXISTS** tests for the existence of a pattern, given the already obtained bindings

Query: "Find friends of friends of Peter"

```
SELECT DISTINCT friendOfFriend.name
 MATCH (a:Person) -[:friend_of]-> () -[:friend_of]-> (friendOfFriend)
 WHERE a.name = 'Peter'
   AND NOT EXISTS (
                    SELECT *
                     MATCH (a) -[:friend_of]-> (friendOfFriend)
                   )
```

Instead of using **SELECT DISTINCT** it's sometimes faster to test for path existence using `-/../->` :

```
SELECT friendOfFriend.name
 MATCH (a:Person) -/:friend_of{2}/-> (friendOfFriend)
 WHERE a.name = 'Peter'
```

More concise way of expressing the query

EXISTS subquery in PATH expression

Query: “In a code base, find functions that are reachable from 'func1' such that no function along the “calls path” transitively writes to a global variable”

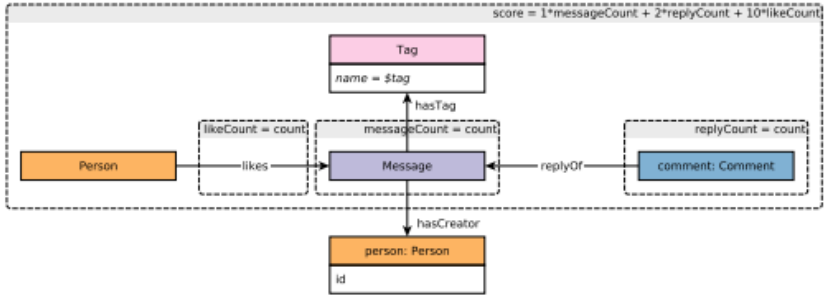
```
  PATH calls_but_no_global_write AS (:Function) -[:calls]-> (dst:Function)
       WHERE NOT EXISTS (
                          SELECT *
                           MATCH (dst) -/:calls*/-> (:Function) -[:writes]-> (:GlobalVariable)
                         )
SELECT f2.name
 MATCH (f1:Function) -/:calls_but_no_global_write+/-> (f2)
 WHERE f1.name = 'func1'
```

Scalar subqueries
(product support upcoming)

LDBC Benchmark query

```
 SELECT person.id
       , SUM((SELECT COUNT(r)
                MATCH (message)<-[r:replyOf]-(comment:comment))
              ) AS replyCount
       , SUM((SELECT COUNT(l)
                MATCH (:person)-[l:likes]->(message))
              ) AS likeCount
       , COUNT(*) AS messageCount
   MATCH (tag:tag) <-[:hasTag]- (message:post|comment)
       , (message) -[:hasCreator]-> (person:person)
   WHERE tag.name = ?
GROUP BY person, tag
ORDER BY 1*messageCount + 2*replyCount + 10*likeCount DESC
       , person.id
   LIMIT 100
```

BI / read / 6

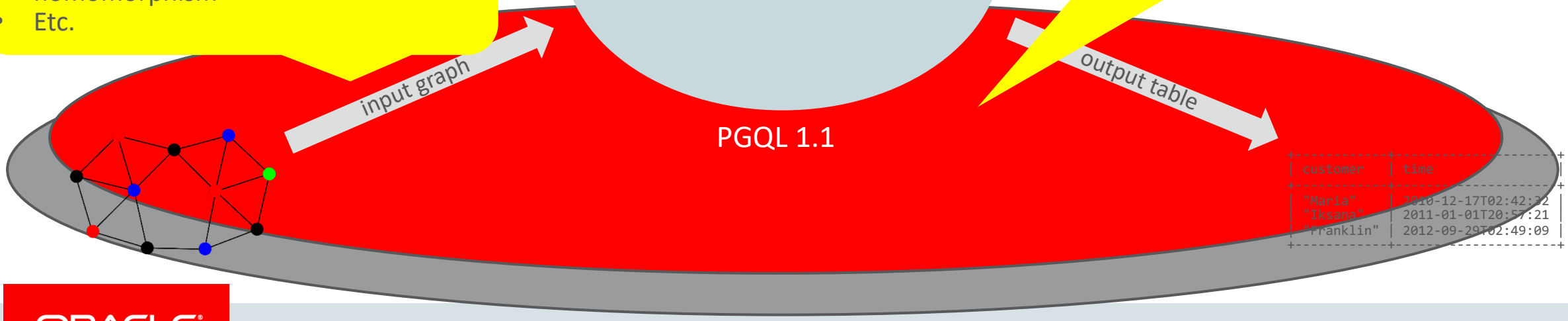
query	BI / read / 6
title	Most active Posters of a given Topic
pattern	
desc.	Get each Person (person) who has created a Message (message) with a given Tag (direct relation, not transitive). Considering only these messages, for each Person node: • Count its messages (messageCount). • Count likes (likeCount) to its messages. • Count Comments (replyCount) in reply to it messages. The score is calculated according to the following formula: 1 * messageCount + 2 * replyCount + 10 * likeCount.
params	1 tag String
result	1 person.id 64-bit Integer R 2 replyCount 32-bit Integer A 3 likeCount 32-bit Integer A 4 messageCount 32-bit Integer A 5 score 32-bit Integer A
sort	1 score ↓ 2 person.id ↑
limit	100
CPs	1.2, 2.3, 8.2

Property graph query languages

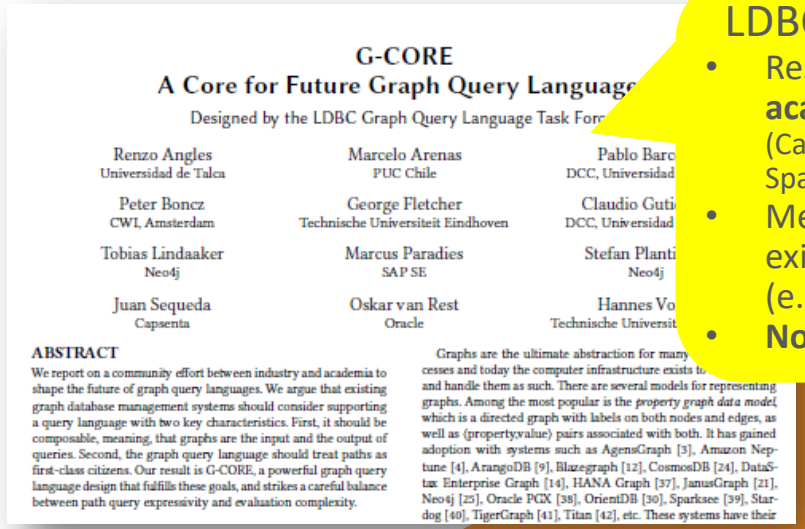
Advances in property graph querying
- Full regular expressions
- Pattern matching based on subgraph homomorphism
- Etc.

Query

Syntax and semantics aligned with SQL
- Well-established foundation
- Familiarity for SQL users (e.g. GROUP BY, HAVING, ORDER BY)



Property graph query languages



G-CORE
A Core for Future Graph Query Languag

Designed by the LDBC Graph Query Language Task For

Renzo Angles, Universidad de Talca
Marcelo Arenas, PUC Chile
Pablo Barc, DCC, Universidad
Peter Boncz, CWI, Amsterdam
George Fletcher, Technische Universiteit Eindhoven
Claudio Guti, DCC, Universidad
Tobias Lindaaker, Neo4j
Marcus Paradies, SAP SE
Stefan Planti, Neo4j
Juan Sequeda, Capsenta
Oskar van Rest, Oracle
Hannes Vo, Technische Universit

ABSTRACT

We report on a community effort between industry and academia to shape the future of graph query languages. We argue that existing graph database management systems should consider supporting a query language with two key characteristics. First, it should be composable, meaning, that graphs are the input and the output of queries. Second, the graph query language should treat paths as first-class citizens. Our result is G-CORE, a powerful graph query language design that fulfills these goals, and strikes a careful balance between path query expressivity and evaluation complexity.

Graphs are the ultimate abstraction for many
cesses and today the computer infrastructure exists to
and handle them as such. There are several models for representing
graphs. Among the most popular is the *property graph data model*,
which is a directed graph with labels on both nodes and edges, as
well as (property,value) pairs associated with both. It has gained
adoption with systems such as AgensGraph [3], Amazon Nep-
tune [4], ArangoDB [9], Blazegraph [12], CosmosDB [24], DataS-
tax Enterprise Graph [14], HANA Graph [37], JanusGraph [21],
Neo4j [25], Oracle PGX [38], OrientDB [30], Sparksee [39], Star-
dog [40], TigerGraph [41], Titan [42], etc. These systems have their

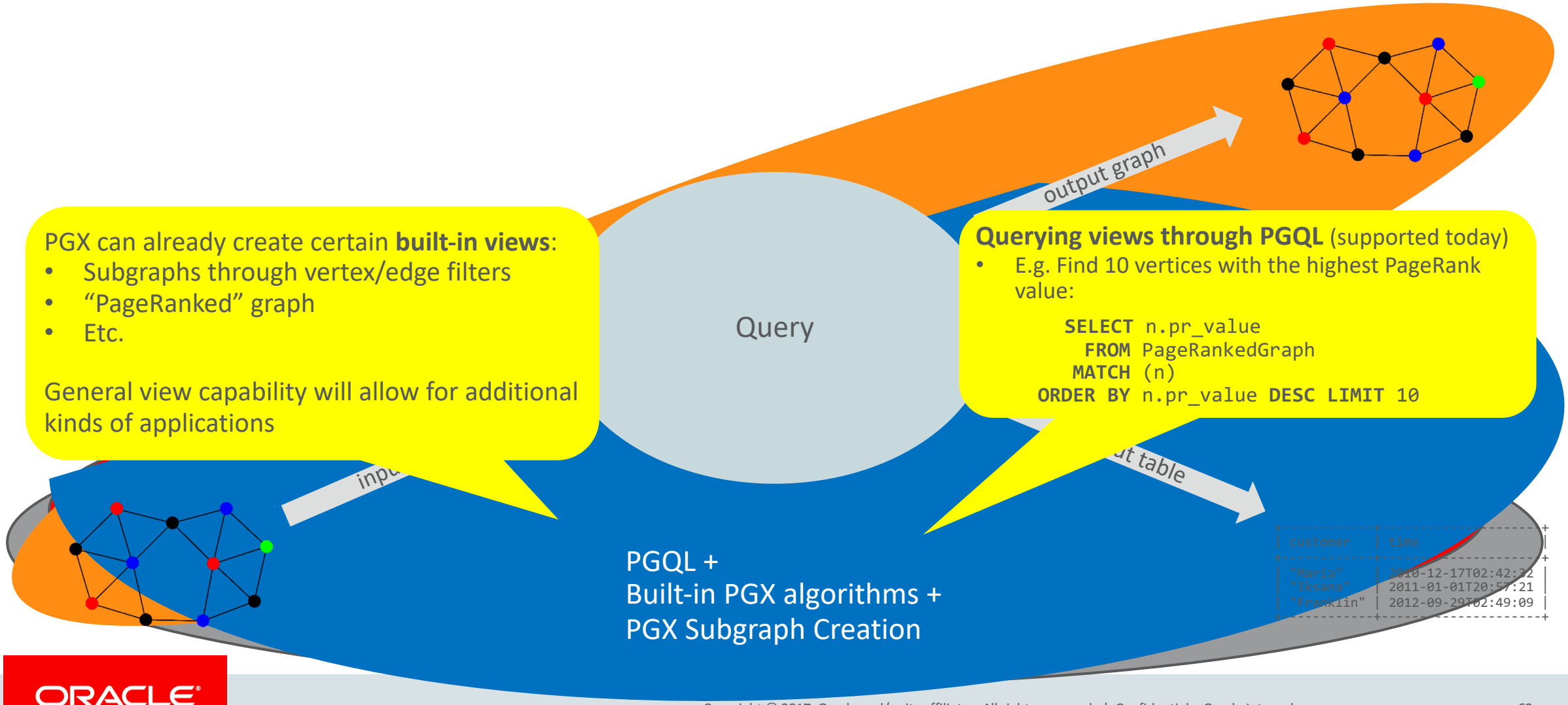
LDBC's G-CORE

- Result of 2.5 years of discussions with **academia** (various universities) **and industry** (Capsenta, HP, Huawei, IBM, Neo4j, Oracle, SAP and Sparsity)
- Meant to **guide** the evolution of both existing and future graph query languages (e.g. SQL/PG, PGQL, Cypher)
- **Not (yet) implemented** by anyone

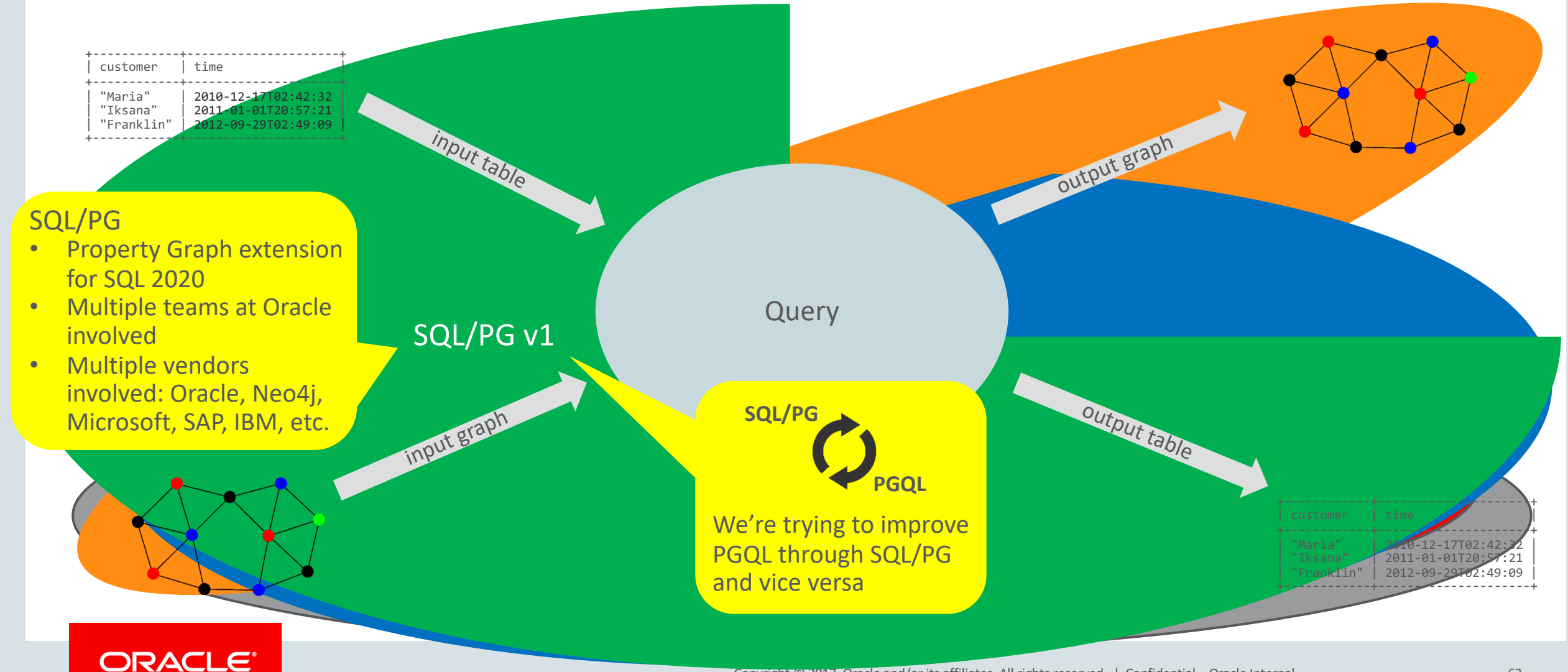
Ability to return graphs from queries allows for:

- **Subqueries**
- **Views**

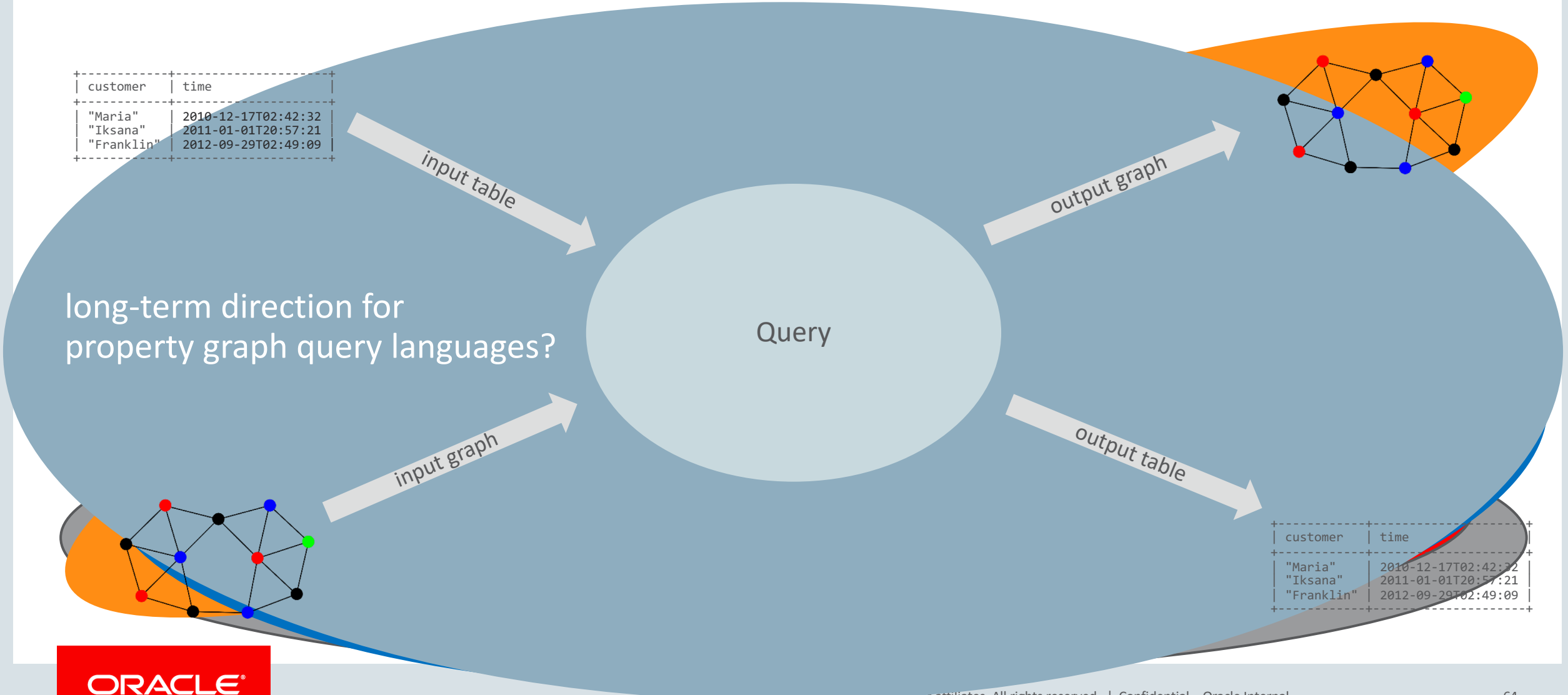
Property graph query languages



Property graph query languages

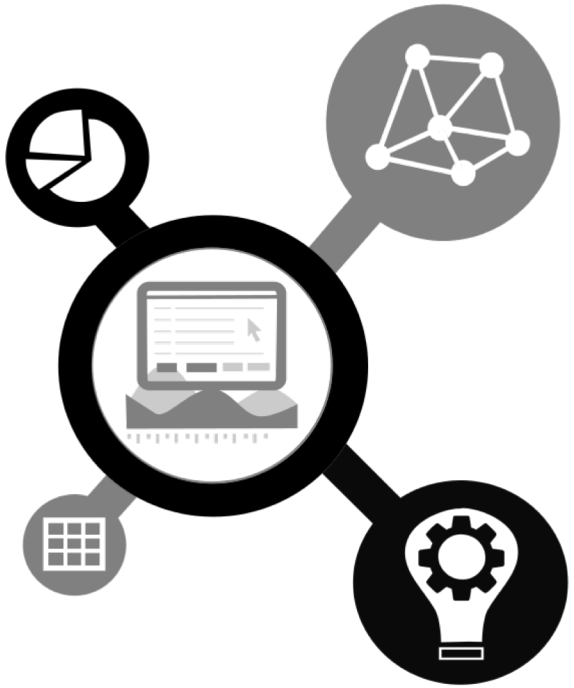


Property graph query languages

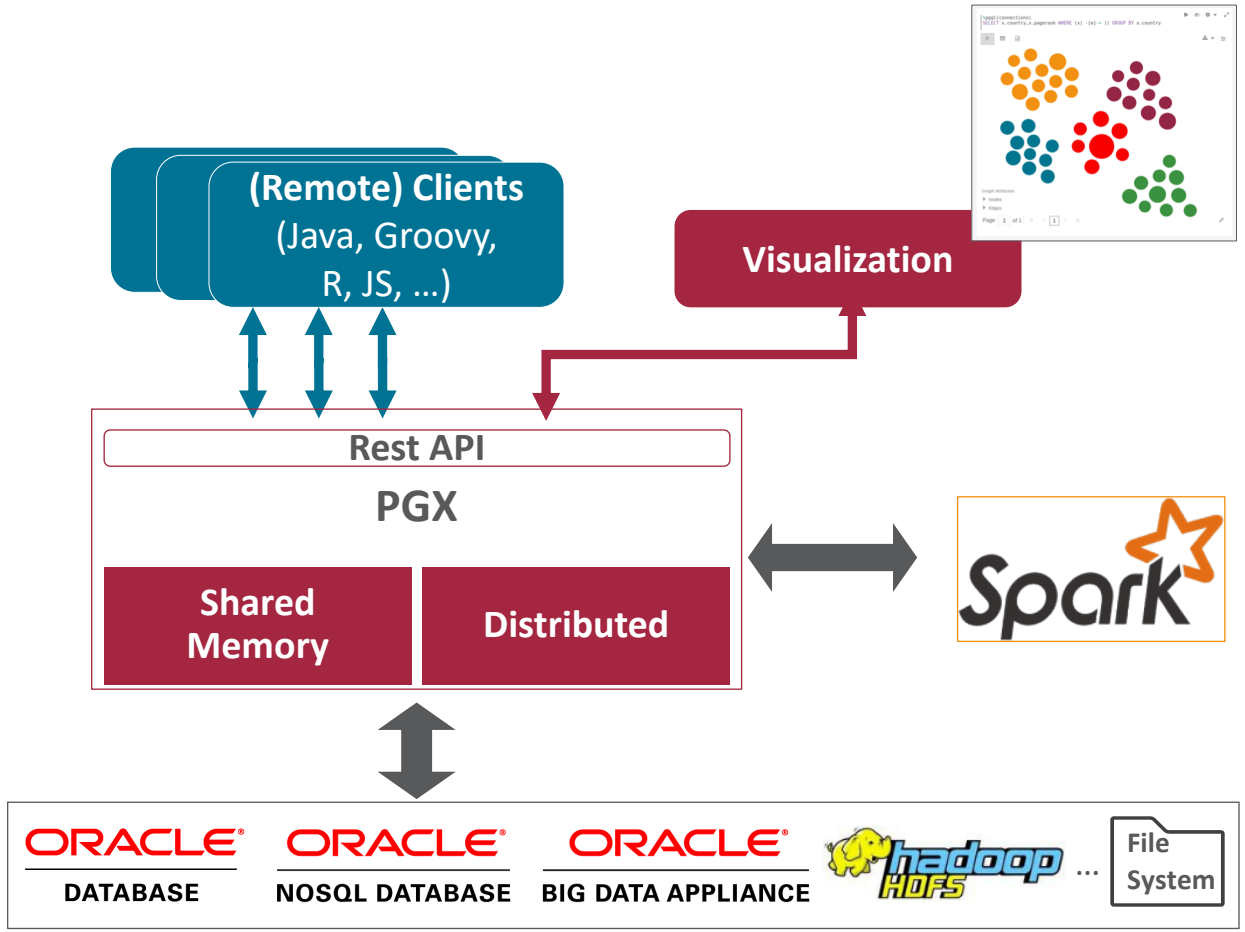


Oracle Labs Data Studio

Data Science with Interactive and Collaborative Notebooks



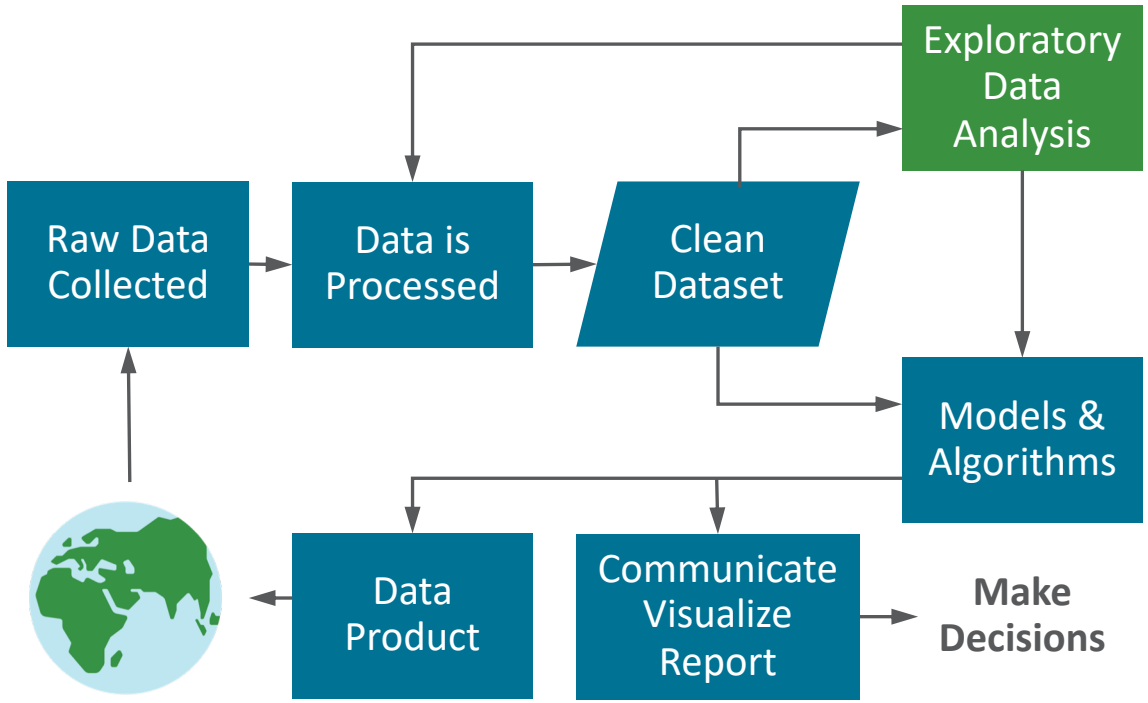
Architecture Overview



- Integrated with various Oracle and open source data storage technologies
- Interoperates with open-source frameworks, e.g. Apache Spark
- In-memory engines for single-machine or distributed execution
- Multiple language bindings for remote execution via REST
- Notebooks and visualization

Data Science: What tools are typically required?

Data Science process



"Doing Data Science", Cathy O'Neil and Rachel Schutt, 2013

Frontend tools

- Quick, interactive execution
- Visualization of results
- Support for modern programming languages
- Remote execution to Cloud

Notebook

IDE for Data Science

Backend tools

- Scalable data manipulation
- Efficient computational engines
- Built-in algorithms and methodologies
- Customization via end-user software engineering

ORACLE DATABASE

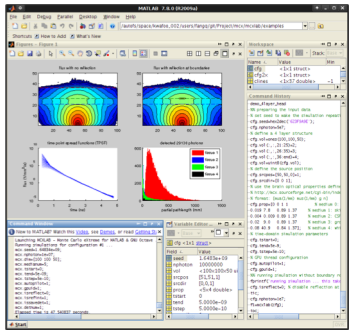
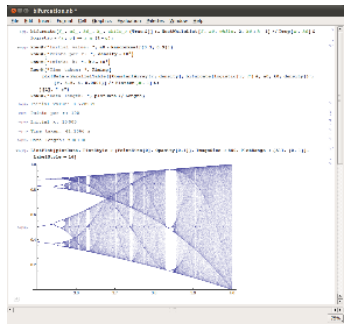
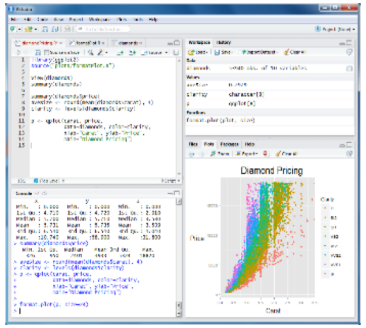
Spark

Rise of Notebook Technology

Early tools

MATLAB · Wolfram Mathematica · RStudio

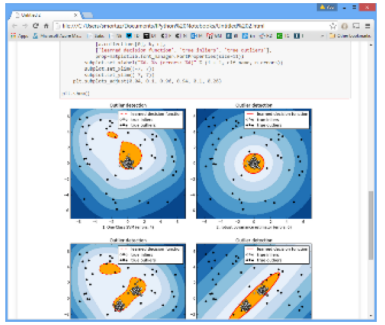
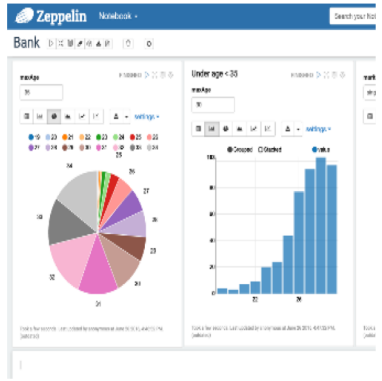
- Single, dedicated programming language
- Stand-alone desktop applications
- Local execution
- REPL-style execution, embedded visualization

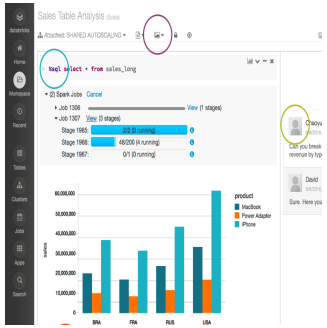




Recent trends

jupyter · Apache Zeppelin · databricks

- Multiple languages, general-purpose languages
- Browser-based
- Execution in Cloud, enables collaborative editing



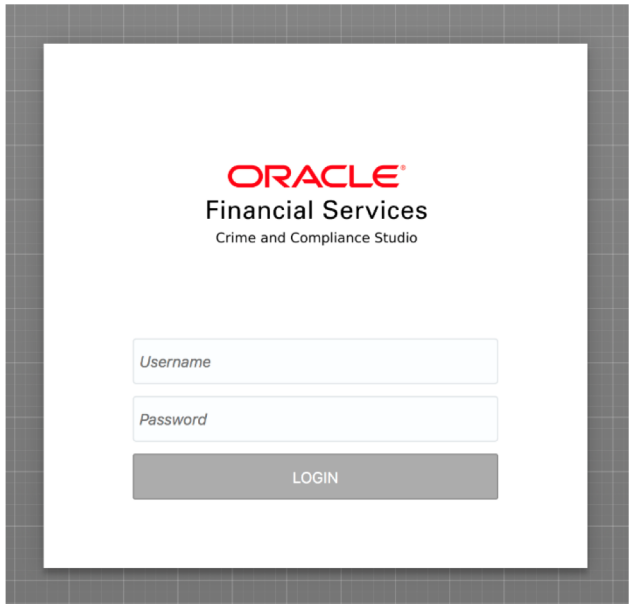
Oracle Labs Data Studio

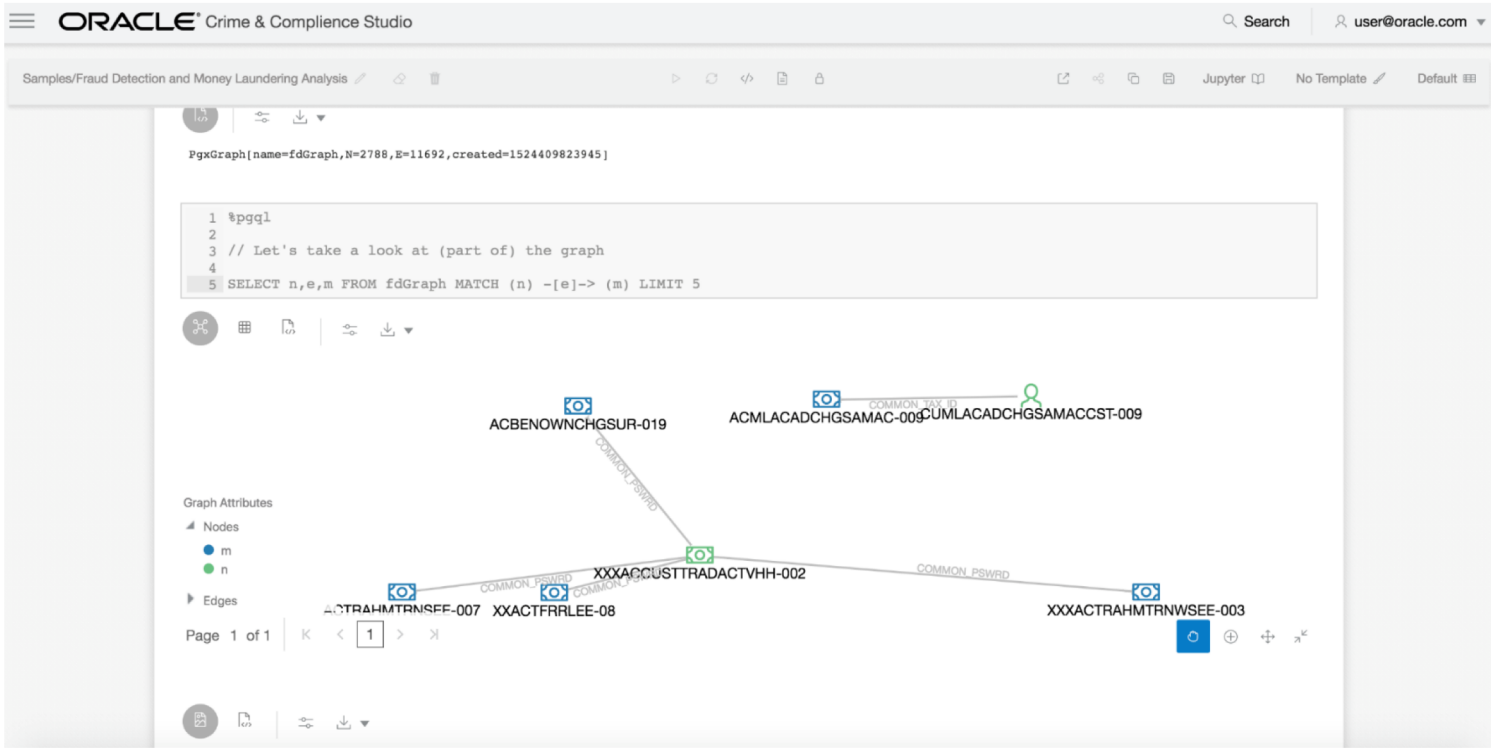
Notebook technology from Oracle Labs

- Built using **Oracle JET**
- Extensible
- Focus on Enterprise
- Integration with Oracle Technologies:
  - Graph Analysis
  - Graph Visualization
  - Advanced polyglot support (Graal)
  - In-database execution (Walnut)

Financial **S**ervices **C**rime and **C**ompliance **Studio**

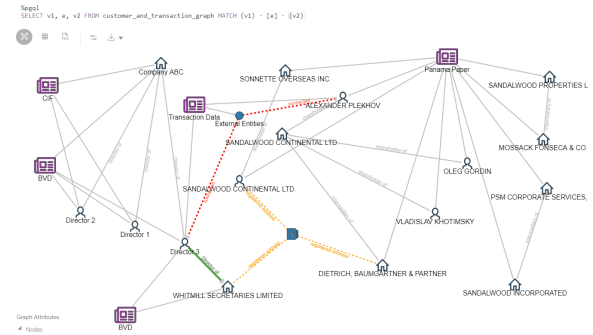
- Large deals already signed in Q4 ranged **200K** to **1M** per deal
- **Graph analytics** for **money laundering prevention** and **fraud detection**

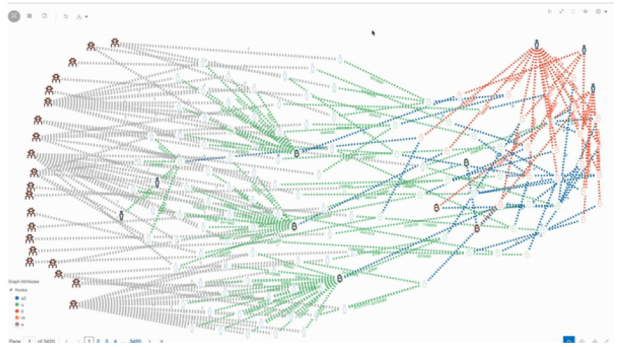


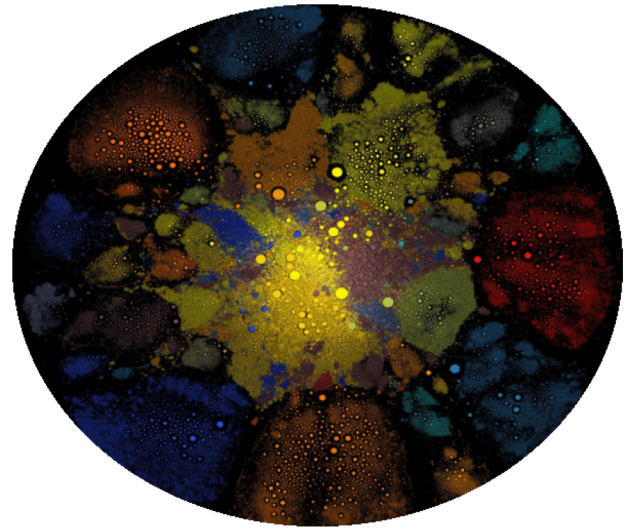


Roadmap and Futures for Data Studio

- Scale the concept from coders (**notebooks**) to business user analysts (**pipelines**) to lightweight data oriented **apps**
- Graph Exploration
- Pipelines/Widgets
- Interpreter-Clients (Jupyter, internal)
- Extension System
- Job Scheduling
- Oracle Graph Training



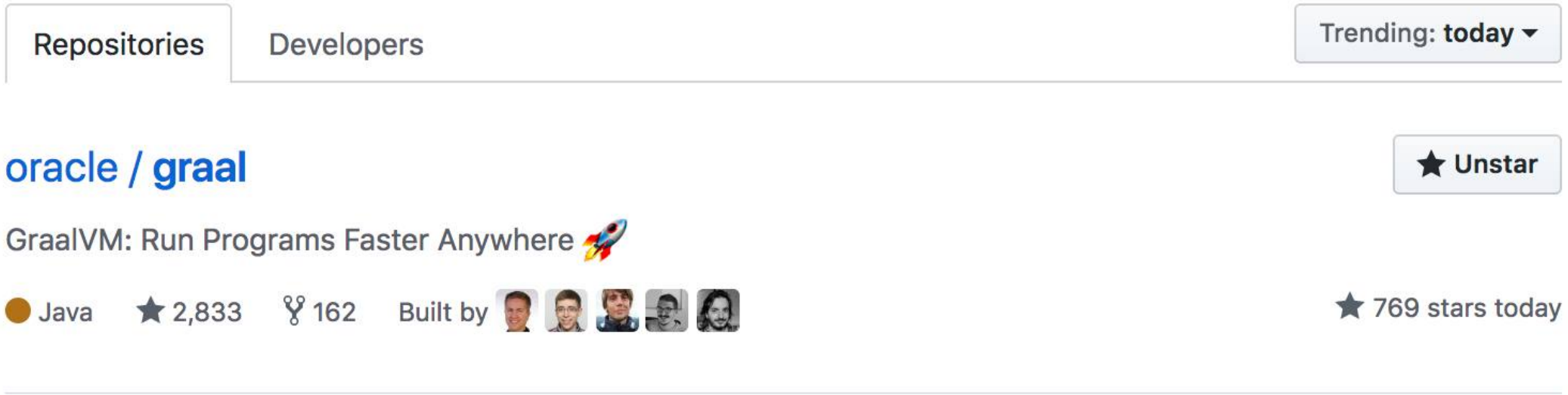




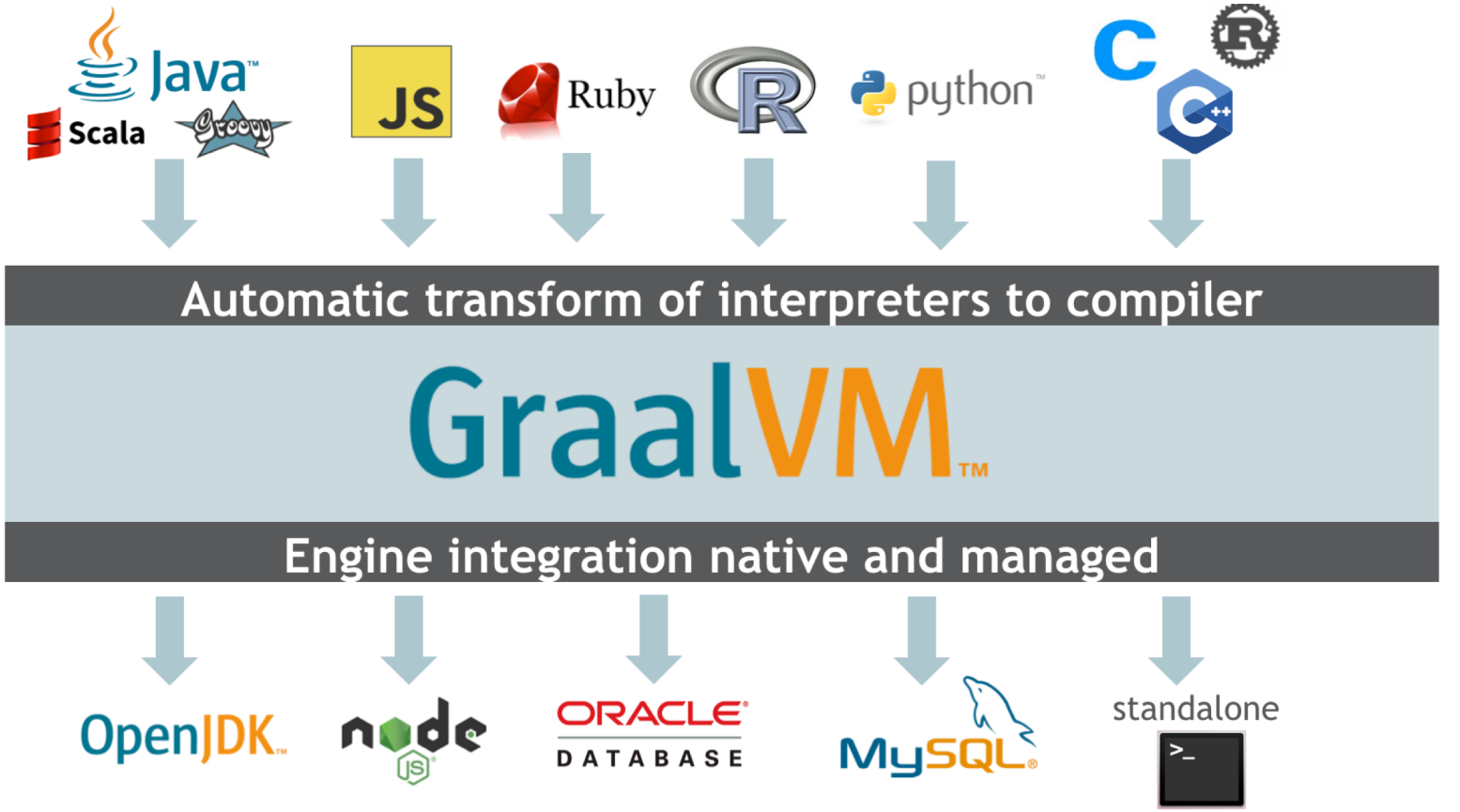
Walnut

Multi-Lingual Data Processing with GraalVM

GraalVM: Oracle Labs Technology on GitHub



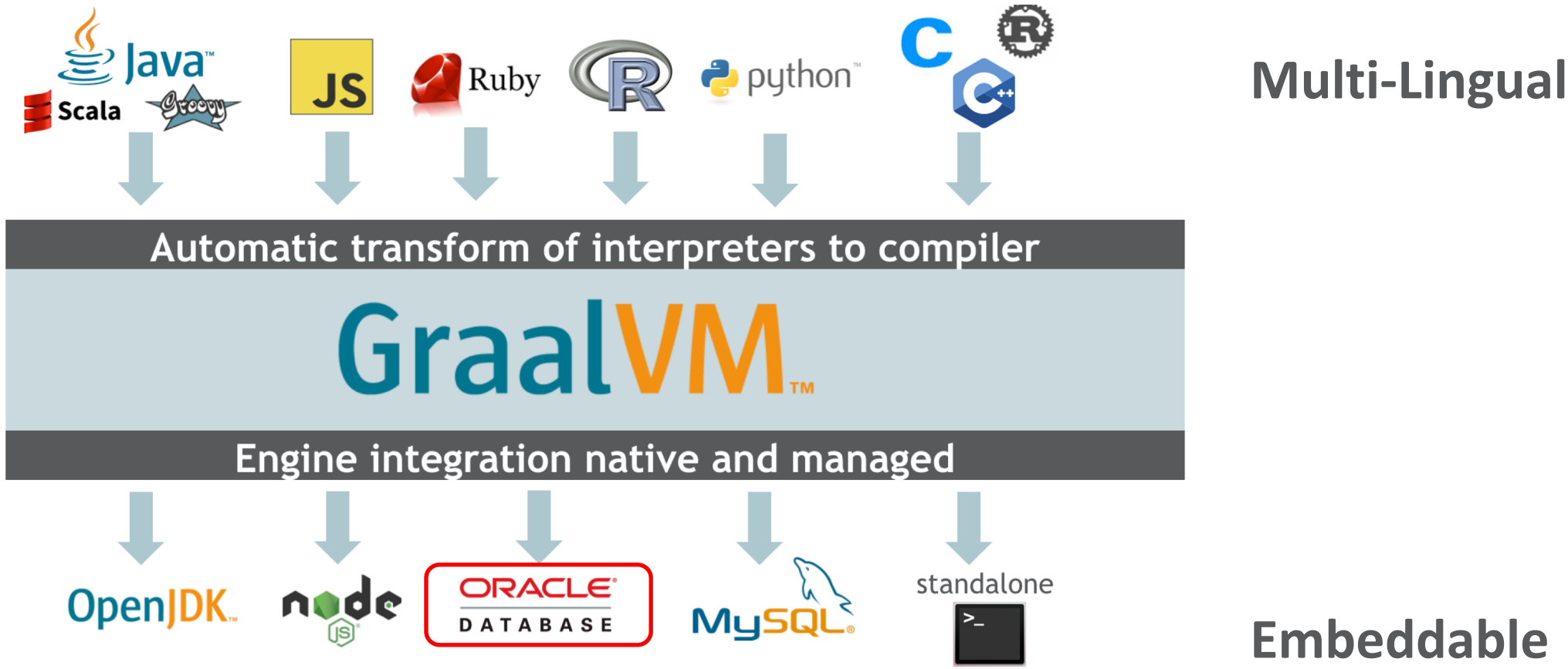
GraalVM: Multi-Lingual, Embeddable Virtual Machine



Multi-Lingual

Embeddable

GraalVM: Multi-Lingual, Embeddable Virtual Machine



Multi-Lingual

Embeddable

Walnut: Multi-Lingual Data Processing with GraalVM

Multi-Lingual Database

- Integrate new programming languages into RDBMS
- Data processing logic in modern languages: JavaScript, Python, Java, …
- Rapid, efficient, eco-system friendly


```
function helloJS(name) {
  return ‘Hello ‘ + name;
}

SELECT helloJS(ENAME) FROM EMP;
```

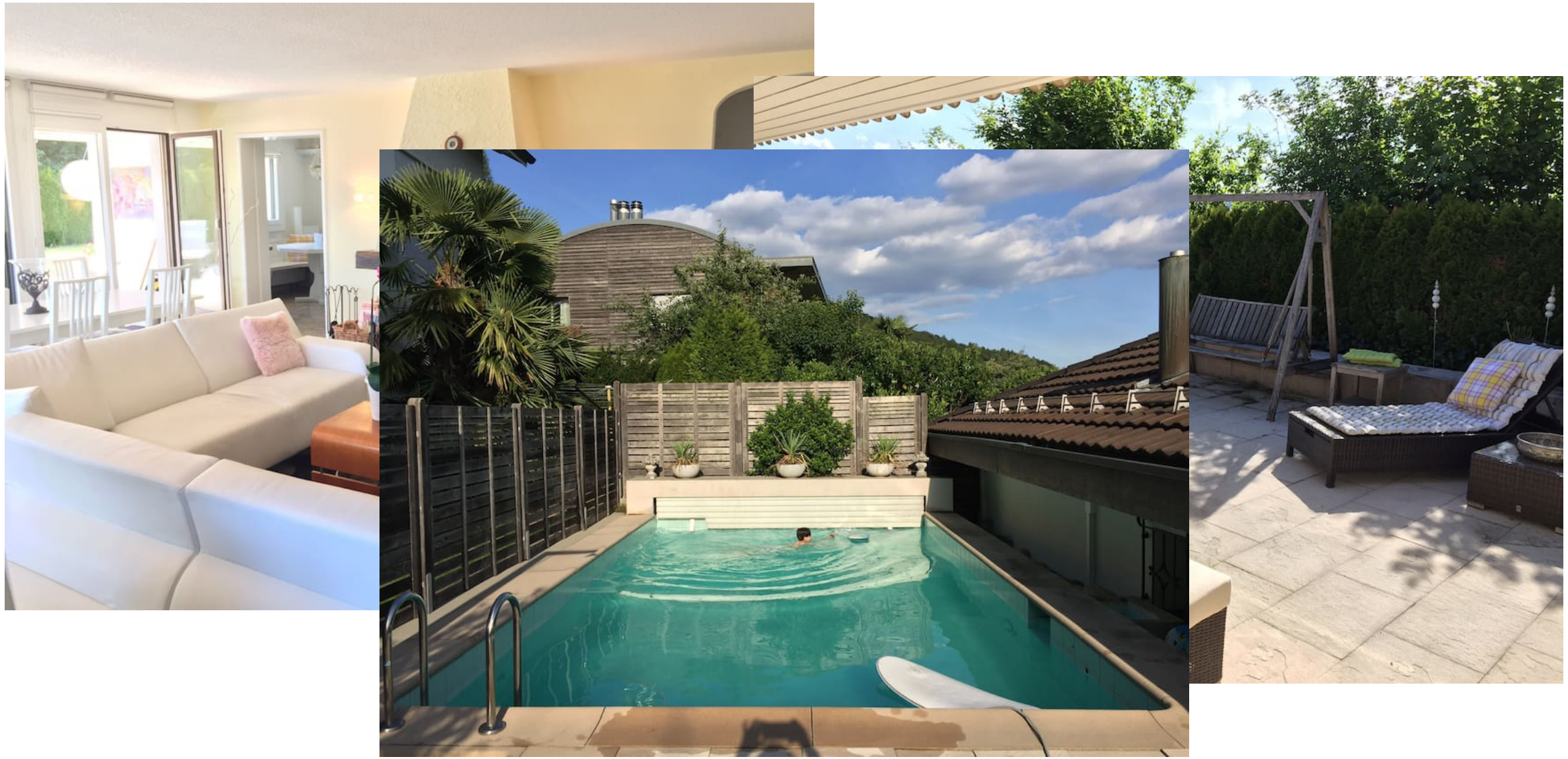
In-Database Data Science

- Familiar ecosystems for Data Science, Machine Learning
- Python: Pandas, scikit-learn, …

- Execute efficiently in database
- “compute close to the data”
- Interoperability between data models (graph, relational, linear algebra)

Internships at Oracle Labs Zurich

- 3-12 months internships
- Regular internships or MSc thesis topics
- Openings at TU Delft Career Centre
- CHF 3700 gross salary



Integrated Cloud

Applications & Platform Services


ORACLE®