W3C GRAPH DATA WORKSHOP

4-6 March 2019, Berlin

Creating Bridges: RDF, Property Graph and SQL

Platinum Sponsors

neo4j

Gold Sponsors

ORACLE

ontotext

Silver Sponsors

ArangoDB

Supporters

LDBC The graph & RDF benchmark reference

Boost 4.0 BIG DATA FOR FACTORIES

W3C WORLD WIDE WEB consortium Deutsch-Österr. Büro

SQL AND GQL

Keith W. Hare, SC32 WG3, JCC Consulting, Inc.
Victor Lee, TigerGraph
Stefan Plantikow, Neo4j
Oskar van Rest, Oracle
Jan Michels, Oracle

Abstract

Since 2017 work has been proceeding on extending SQL with read-only property graph extensions based on the pattern-matching paradigm of Cypher and PGQL. SIGMOD 2017 saw the publication of the future-looking G-CORE paper on fresh directions in PG querying, matched by implementation of compositional queries and graph views in Cypher for Apache Spark. Since spring 2018 the property graph world has been coalescing around the idea of a single GQL language, drawing on all of these precedents, open to other inputs, and closely coordinated with key aspects of SQL and its ecosystem.

In this session, designers and contributors to SQL, Cypher, GSQL and PGQL will describe, discuss and doubtless differ on plans for the new international standard GQL for property graph querying.

Introduction

- SQL – Keith Hare, Convenor, ISO/IEC JTC1 SC32 WG3 Database Languages
  - A brief history
  - SQL 2016
  - SQL Technical Reports
- Property Graphs
  - SQL/PGQ
  - GQL
- GSQL – Victor Lee, TigerGraph
- PGQL – Oskar van Rest, Oracle
- Cypher – Stefan Plantikow, Neo4j
- Summary

Keith Hare
JCC Consulting, Inc.
ISO/IEC JTC1 SC32 WG3

What is SQL?

- SQL is a language for defining databases and manipulating the data in those databases
- SQL Standard uses SQL as a name, not an acronym
  - Might stand for SQL Query Language
- SQL queries are independent of how the data is actually stored – specify what data you want, not how to get it
  - Declarative query language

SQL Standards – a brief history

- ISO/IEC 9075 Database Language SQL
  - SQL-87 – Transactions, Create, Read, Update, Delete
  - SQL-89 – Referential Integrity
  - SQL-92 – Internationalization, etc.
  - SQL:1999 – User Defined Types
  - SQL:2003 – XML
  - SQL:2008 – Expansions and corrections
  - SQL:2011 – Temporal
  - SQL:2016 – JSON, RPR, PTF, MDA (2019)
- 30 years of support and expansion of the standard

SQL:2016 Major Features

- Row Pattern Recognition
  - Regular Expressions across sequences of rows
- Support for Java Script Object Notation (JSON) objects
  - Store, Query, and Retrieve JSON objects
- Polymorphic Table Functions
  - parameters and function return value can be tables whose shape is not known until compile time
- Additional analytics
  - Trigonometric and Logarithm functions
- Multi-dimensional Arrays (2019)

SQL:2016 Parts

Reference	Document title
ISO/IEC 9075-1	Information technology -- Database languages -- SQL -- Part 1: Framework (SQL/Framework)
ISO/IEC 9075-2	Information technology -- Database languages -- SQL -- Part 2: Foundation (SQL/Foundation)
ISO/IEC 9075-3	Information technology -- Database languages -- SQL -- Part 3: Call-Level Interface (SQL/CLI)
ISO/IEC 9075-4	Information technology -- Database languages -- SQL -- Part 4: Persistent stored modules (SQL/PSM)
ISO/IEC 9075-9	Information technology -- Database languages -- SQL -- Part 9: Management of External Data (SQL/MED)
ISO/IEC 9075-10	Information technology -- Database languages -- SQL -- Part 10: Object language bindings (SQL/OLB)
ISO/IEC 9075-11	Information technology -- Database languages -- SQL -- Part 11: Information and definition schemas (SQL/Schemata)
ISO/IEC 9075-13	Information technology -- Database languages -- SQL -- Part 13: SQL Routines and types using the Java programming language (SQL/JRT)
ISO/IEC 9075-14	Information technology -- Database languages -- SQL -- Part 14: XML-Related Specifications (SQL/XML)
ISO/IEC 9075-15	Information technology -- Database languages -- SQL -- Part 15: Multi-dimensional Arrays (SQL/MDA) (2019)

SQL Technical Reports – 19075

- SQL Standards committees have accumulated a great deal of descriptive material
- Useful information (non-normative) but does not belong in the actual standard.
- Started creating Technical Reports from this material
  - First was published in 2011
  - Total of seven are now published
  - Eighth will be published soon
- Available from JTC1 Freely Available Standards page:
  - http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html
  - Search for 19075
  - Must agree to single use license
- The current list of Technical Reports is:

SQL Technical Reports

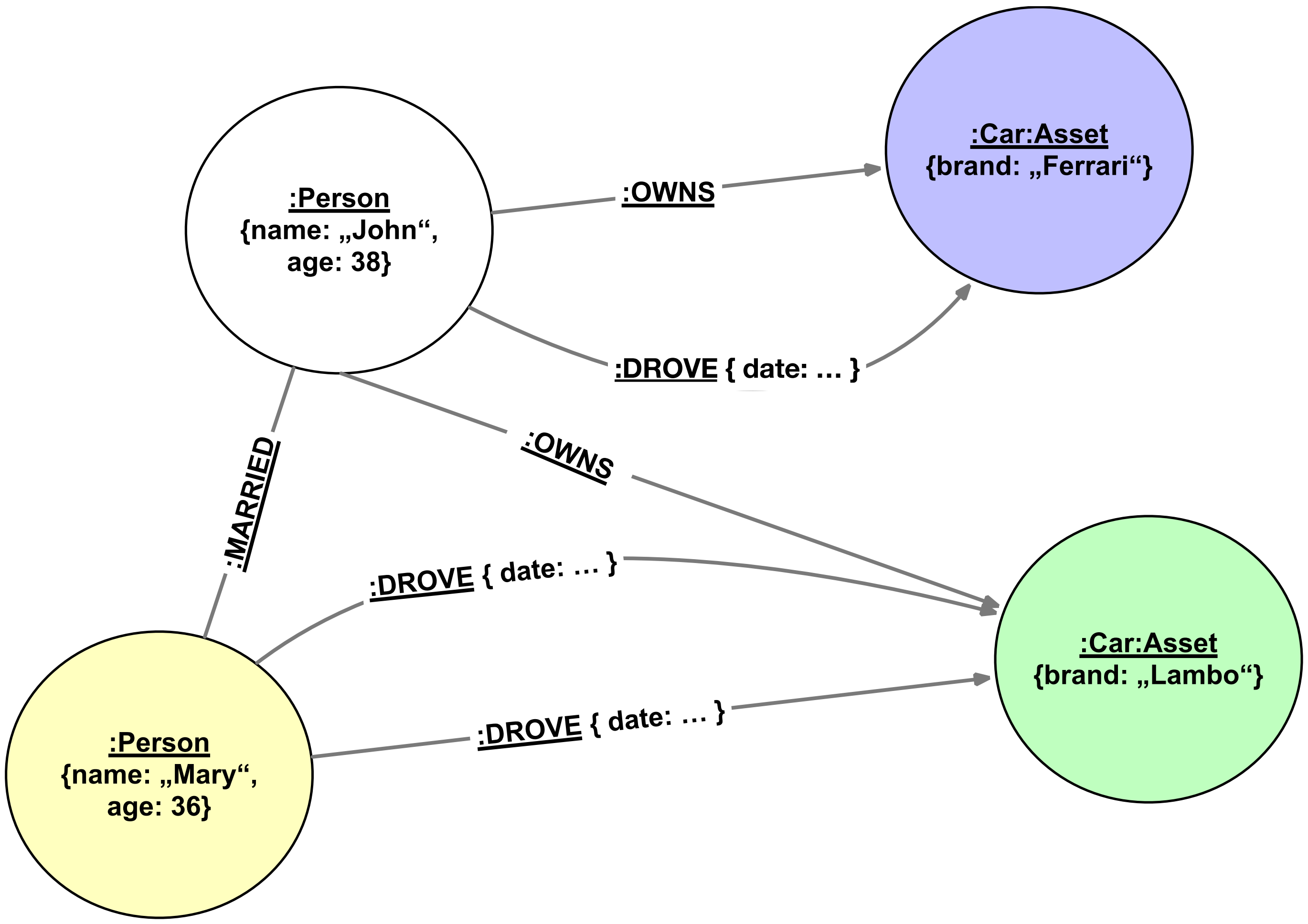
Reference	Document title	Publication Date
ISO/IEC TR 19075-1	Information technology -- Database languages -- SQL Technical Reports -- Part 1: XQuery Regular Expression Support in SQL	2011-07-06
ISO/IEC TR 19075-2	Information technology -- Database languages -- SQL Technical Reports -- Part 2: SQL Support for Time-Related Information	2015-07-01
ISO/IEC TR 19075-3	Information technology -- Database languages -- SQL Technical Reports -- Part 3: SQL Embedded in Programs using the Java ™ programming language	2015-07-01
ISO/IEC TR 19075-4	Information technology -- Database languages -- SQL Technical Reports -- Part 4: SQL with Routines and types using the Java™ programming language	2015-07-01
ISO/IEC TR 19075-5	Information technology -- Database languages -- SQL Technical Reports -- Part 5: Row Pattern Recognition in SQL	2016-12-14
ISO/IEC TR 19075-6	Information technology -- Database languages -- SQL Technical Reports -- Part 6: SQL support for JSON	2017-03-29
ISO/IEC TR 19075-7	Information technology -- Database languages -- SQL Technical Reports - Part 7: SQL Support for Polymorphic Table Functions	2017-03-29
ISO/IEC TR 19075-8	Information technology -- Database languages -- SQL Technical Reports -- Part 8: SQL Support for multi dimensional arrays	2019

What's next?

SC32 WG3 is adding support to the SQL standards in the following areas:

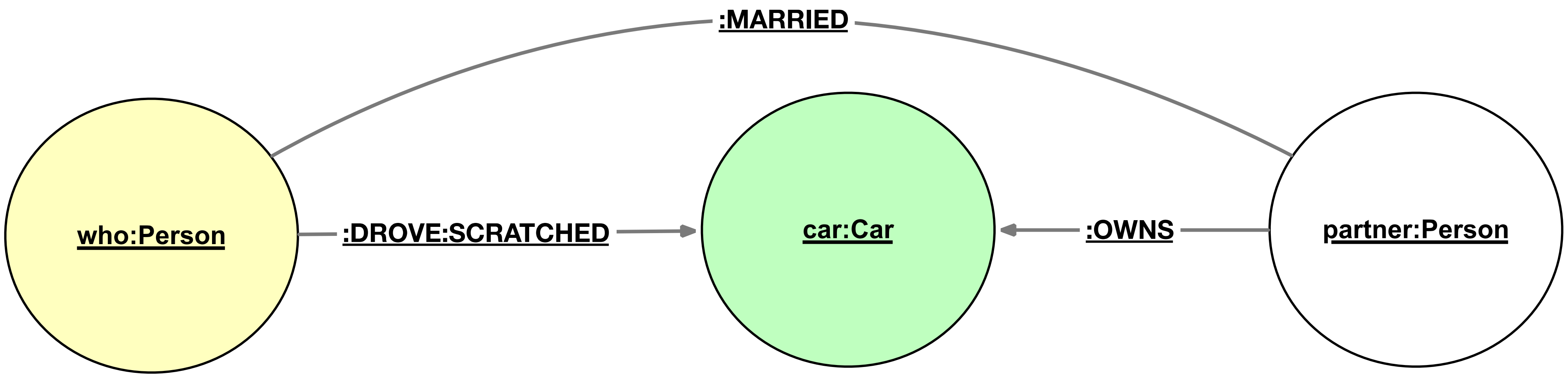
- Property Graph Queries in SQL
- Graph Query Language
- Streaming SQL
- Etc.

Property Graphs



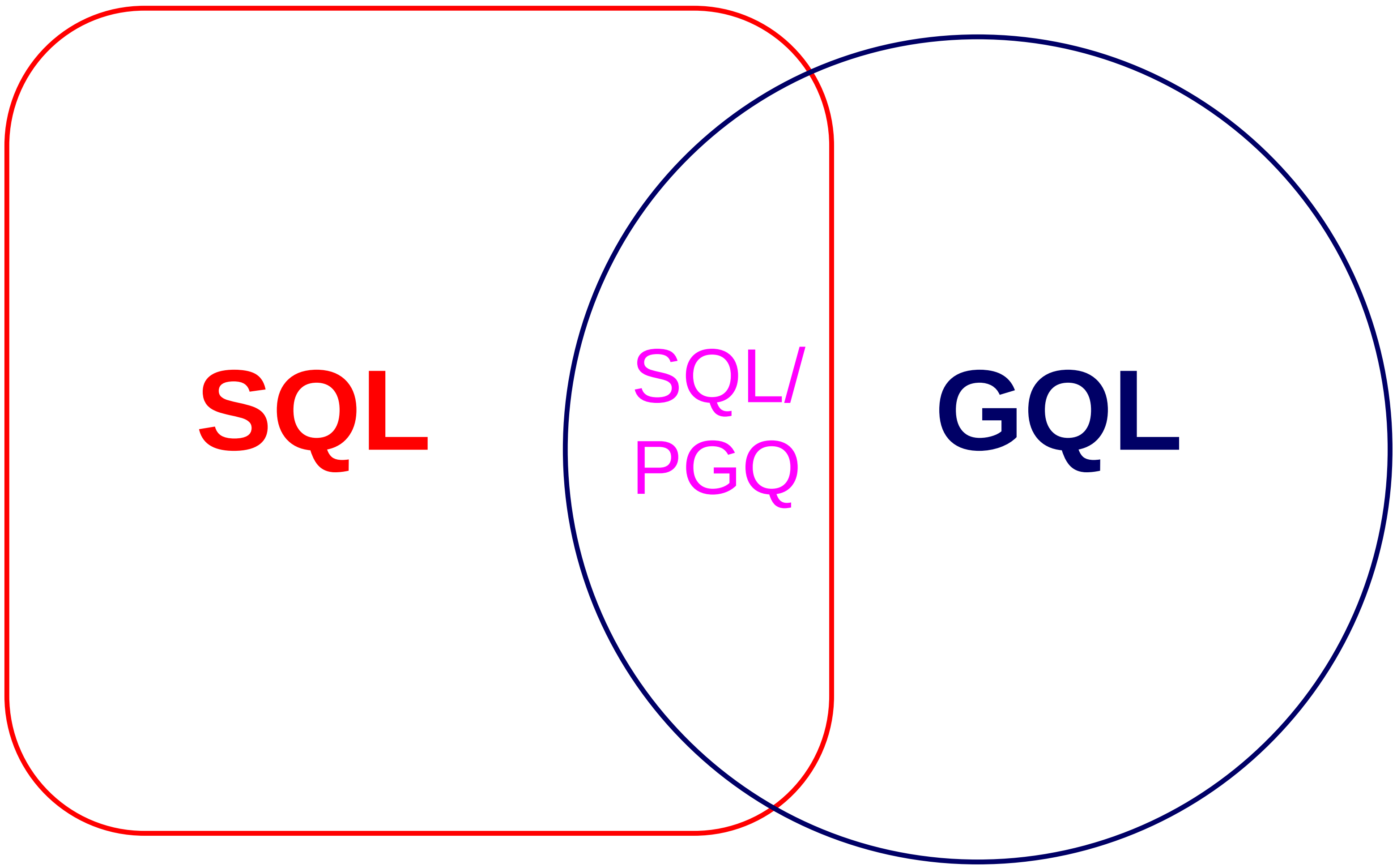
- Nodes/Vertices
- Relationships/Edges
- 0..* Labels
- 0..* Key-Value Properties
- Intrinsic Identity
- Schema:
  Each label defines
  its allowed properties

Property Graph Pattern Matching



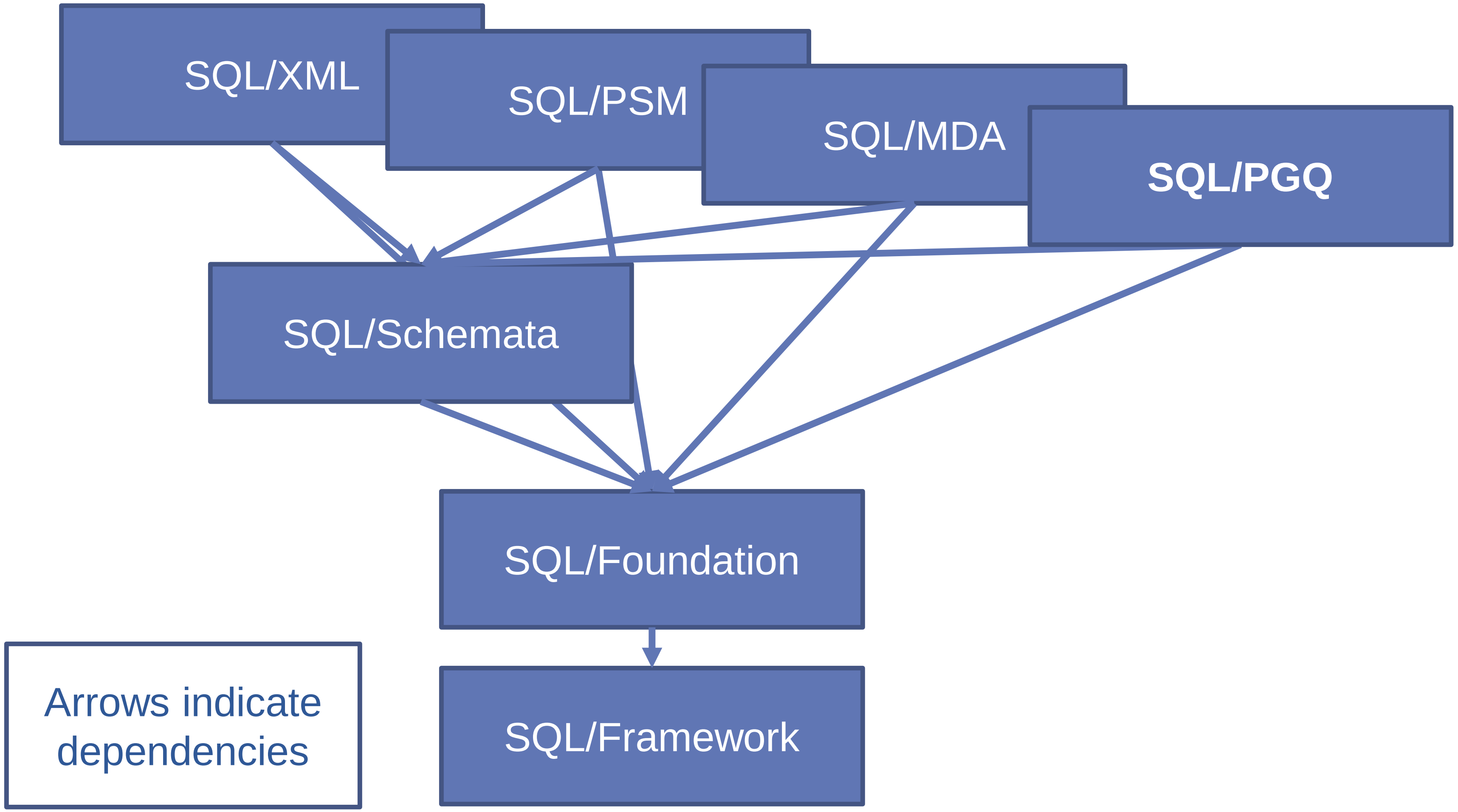

```
SELECT * FROM MyGraph GRAPH_TABLE (

  MATCH (who:Person)-[:DROVE&SCRATCHED]->(car:Car),
        (car)<-[:OWNS]-(partner:Person)
  WHERE EXISTS (who)-[:MARRIED]-(partner)

  COLUMNS ( who.name AS driver, partner.name AS owner )
)
```

SQL, SQL/PGQ, and GQL

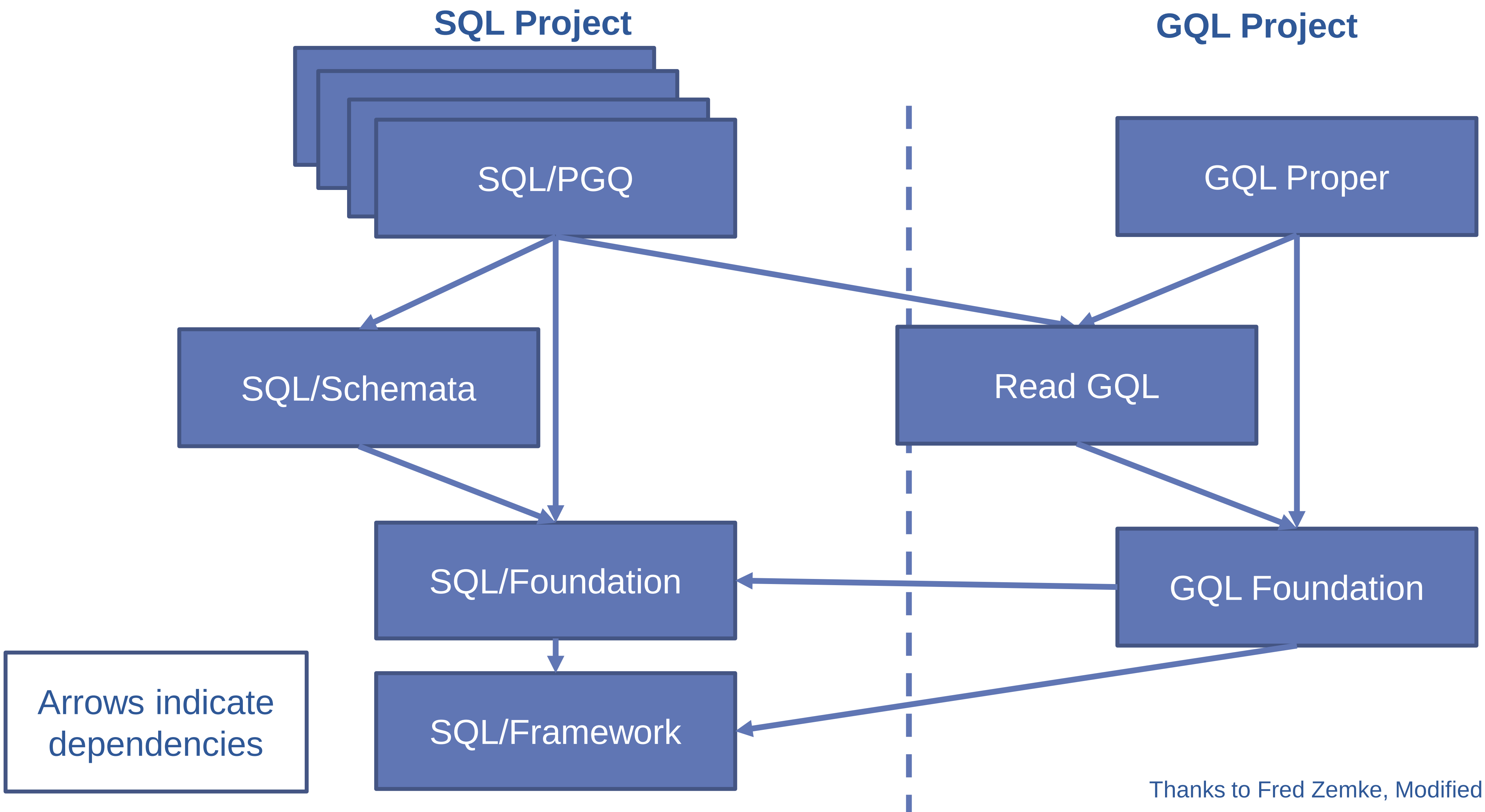


SQL and SQL/PGQ

SQL Project



SQL and GQL Projects



Thanks to Fred Zemke, Modified by WG3

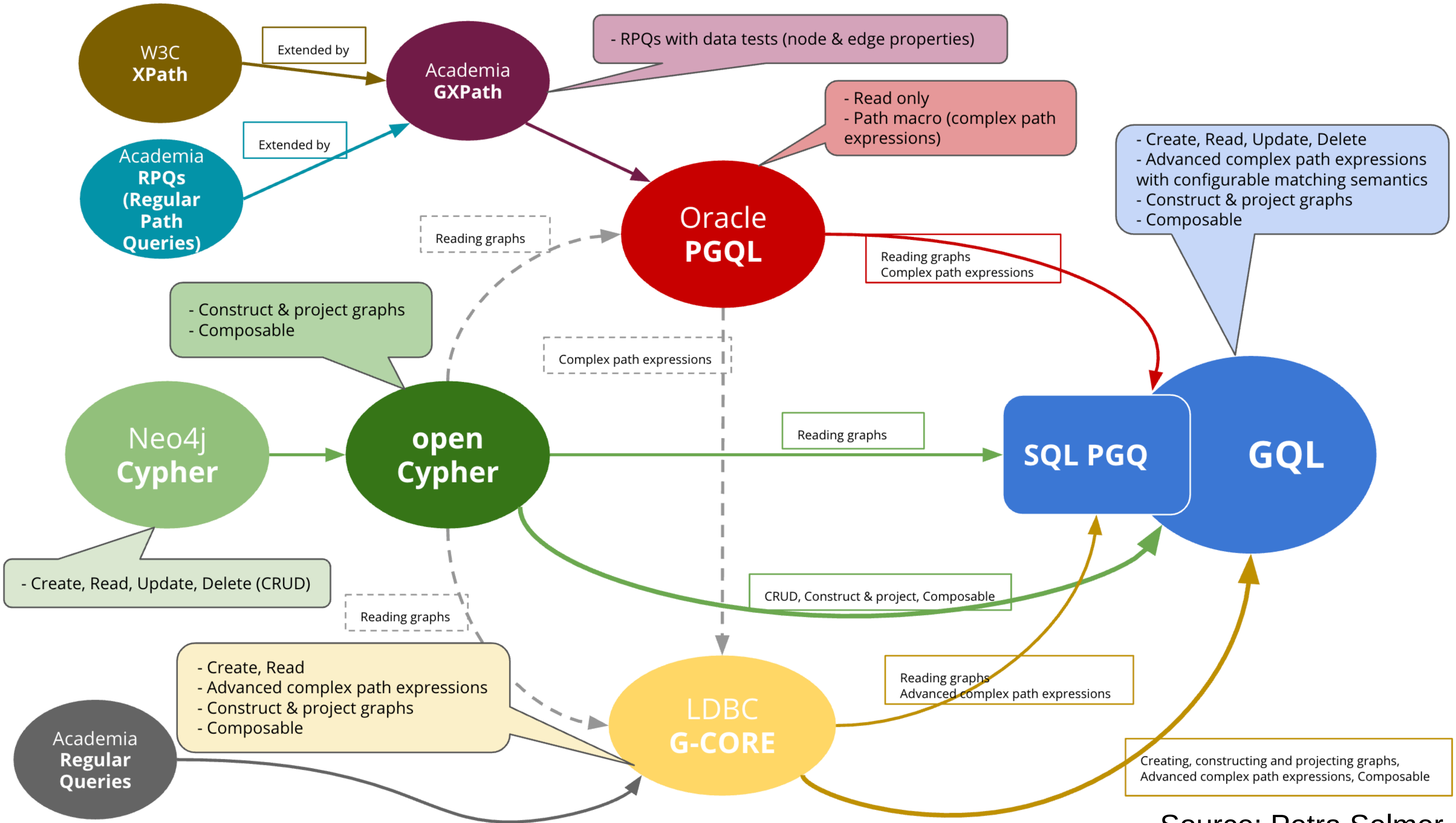
GQL Project Potential Structure

Three parts (at least)

- GQL Foundation (Groundwork, or some other name)
  - Incorporate by reference useful parts of:
    - SQL/Framework
    - SQL/Foundation
- Read GQL (or some other name)
  - Specify graph capabilities needed by both SQL/PGQ and GQL/Proper
  - Graph Pattern Matching…
- GQL/Proper
  - Graph capabilities not needed by SQL/PGQ

What is the input for SQL/PGQ and GQL

- Currently under discussion in various committees
  - ANSI INCITS DM32.2 (Databases) Property Graph Ad Hoc
    - Chaired by Jan Michels, Oracle
    - Participants from
      - Vendors
      - Consultants
      - LDBC Graph QL Task Force
    - Real work happening here
  - ANSI INCITS DM32.2 (Databases)
  - ISO/IEC JTC1 SC32 WG3 – Database Languages
- Current Graph Query efforts



Source: Petra Selmer

Input from Participants

- ANSI INCITS DM32.2 (Databases) Property Graph Ad Hoc
- Chaired by Jan Michels, Oracle
- Participants from
  - Vendors
  - Consultants
  - LDBC Graph QL Task Force
- Vendors Include
  - TigerGraph
  - SAP
  - Oracle
  - Neo4j
  - IBM

Victor Lee

TigerGraph

Property Graph Language for High Performance

Victor Lee, TigerGraph

Origins of GSQL

Design a property graph database for tomorrow's big data and analytics

- Real-time transactions (OLTP) and complex analytics (OLAP)
- Billion- to Trillion- scale graphs

Design Principles

- Native graph - efficient storage and graph traversal
- Parallel processing - speed
- Distributed - scale
- ACID - transactional
- graph "query" language makes it easy to use such a database

GSQL Design Features

Schema-Based

Optimizes storage efficiency and query speed. Supports data-independent app/query development.

Built-in High Performance Parallelism

Achieves fast results while being easy to code

SQL-Like

Familiar to 1 million users

Conventional Control Flow (FOR, WHILE, IF/ELSE)

Makes it easy to implement conventional algorithms

Procedural Queries

Parameterized queries are flexible and can be used to build more complex queries

Transactional Graph Updates

HTAP - Hybrid Transactional / Analytical Processing with real-time data updates

Schema-less vs. Schema-first

- Schema-less: For each access,
  - Machine needs to determine whether a given vertex has the label of interest, has the properties of interest, etc.

Entity1
PropA: val
PropC: val
PropD: val

Entity2
PropC: val
PropE: val

- Schema-first:
  - Machine can read/write property values faster because it already knows which properties exist and where to find them in memory.

	Entity1	Entity2
PropA	val	
PropB		
PropC	val	val
PropD	val	
PropE		val

TigerGraph

Proposals for GQL - Graph Model

- **Schema-first Option**
  - Vertex types and edge types have a defined property schema
  - Vertex instances and edge instances adhere to the schema
  - Option to explicitly name the reverse version of a directed edge
  - Labels can correspond to a type name or be just a tag

SQL-Like

```
CREATE VERTEX Person (ssn int PRIMARY_KEY, firstName string, lastName string, bday date)

CREATE DIRECTED EDGE traveledTo(FROM p Person, TO loc Location, mode string, arrival date)
    WITH REVERSE_EDGE wasVisitedBy
```

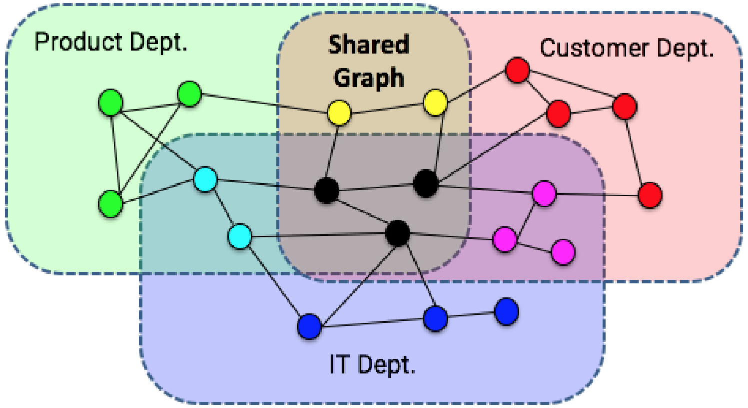
Graphs

- A **graph** is a collection of vertex types and edge types (including all instances of the named types):

```
CREATE GRAPH Travel (Person, Location, TraveledTo, Transportation, TraveledBy)
```

- Can have **multiple graphs**, possibly overlooking/sharing data.
- Each graph is a **domain for access control**, e.g.,

GRANT ROLE admin ON GRAPH Travel TO Victor



TigerGraph

Labels

- A label is associated with a set of zero or properties.
  - Each vertex type name or edge type name is a label, e.g., Person, Location
  - Can create labels with no properties ⇒ tags

- Labels are applied at the instance level.

- When a vertex or edge instance is created, it is given one or more labels ⇒ sets the instance's property schema

Proposals for GQL - Query Language

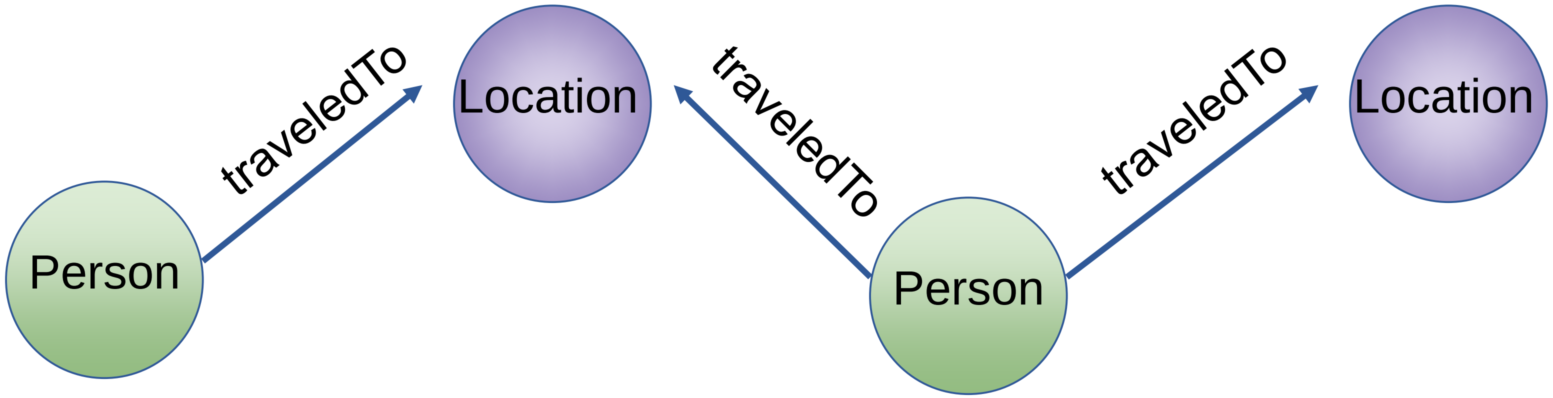
1. Basic Goals
   a. Multi-hop paths — for pattern matching
   b. Composable — can return a graph or a "table"

2. Features for Analytics
   a. Complex data types — list, set, bag(multiset), map, heap
   b. Accumulators — for parallelizable computation
   c. Control flow — Looping, Conditional branching
   d. CRUD, Turing complete — Insert, Update, Delete (SQL-like)
   e. Procedural — Each query can be compiled into a parameterized procedure

TigerGraph

Multi-hop Paths, SELECT Statement

```
SELECT l2
FROM   Person:self -(TraveledTo>:t1)- Location:l1 -(<TraveledTo:t2)- Person:p
                   -(TraveledTo>:t3)- Location:l2
WHERE  self.ssn == mySSN AND p.ssn != self.ssn
                         AND l2.name != l1.name
```



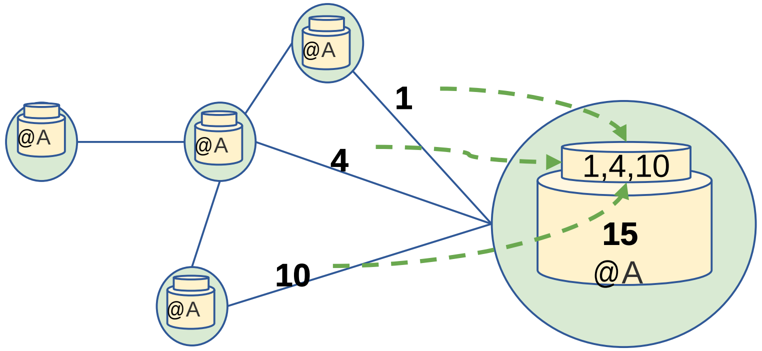
Accumulators

Special types of variables that accumulate information about the graph during traversal.

Local Accumulators:

- Each selected vertex has its own accumulator.
- Local means per vertex. Each vertex does its own processing and considers what it can see/read/write.

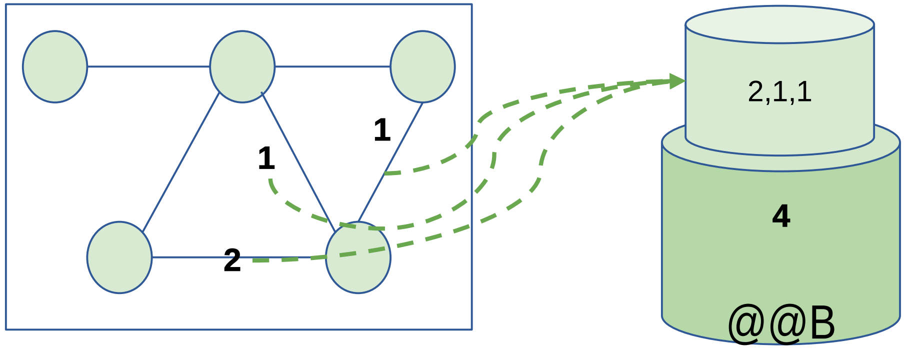
e.x. Accum @A;



Global Accumulators:

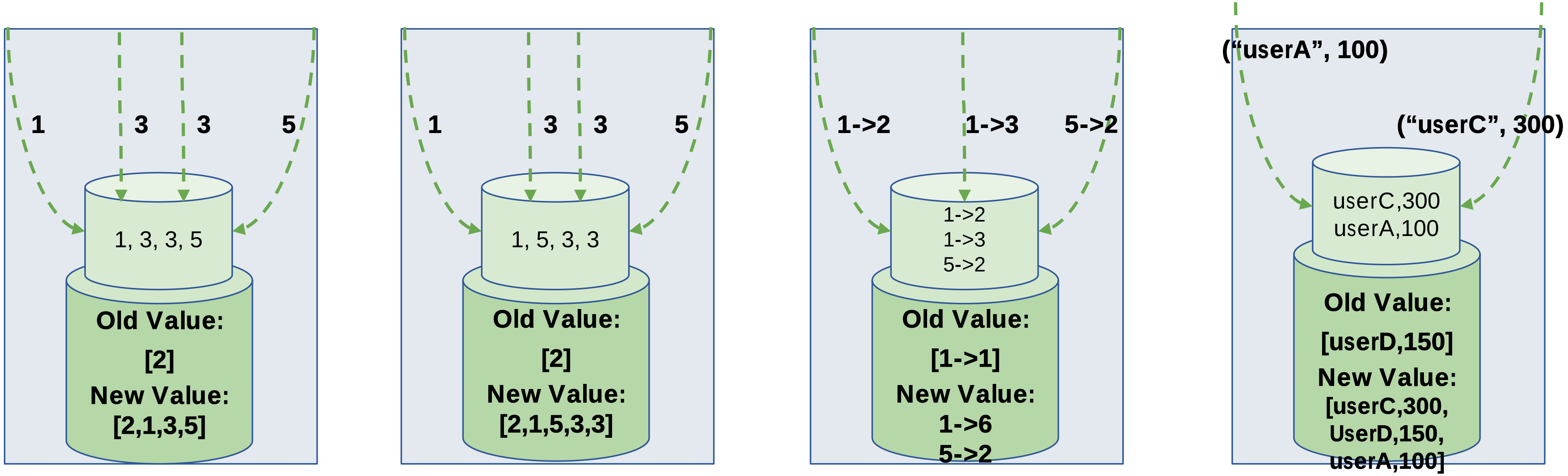
- Stored in globally, visible to all.
- All vertices and edges have access.

e.x. SumAccum @@B;



Accumulators

There are a whole list of accumulators that are supported in GSQL language. They follow the same rules for value assigning and accessing. However each of them has their unique way of **aggregating values.**



SetAccum<int>
Maintains a collection of unique elements.

ListAccum<int>
Maintains a sequential collection of elements.

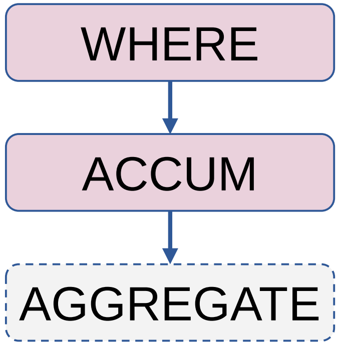
MapAccum<int,SumAccum<int>>
Maintains a collection of (key -> value) pairs.

HeapAccum<Tuple>
Maintains a sorted collection of tuples and enforces a maximum number of tuples in the collection

ACCUM Clause

What is the age distribution of friends that were registered in 2018?

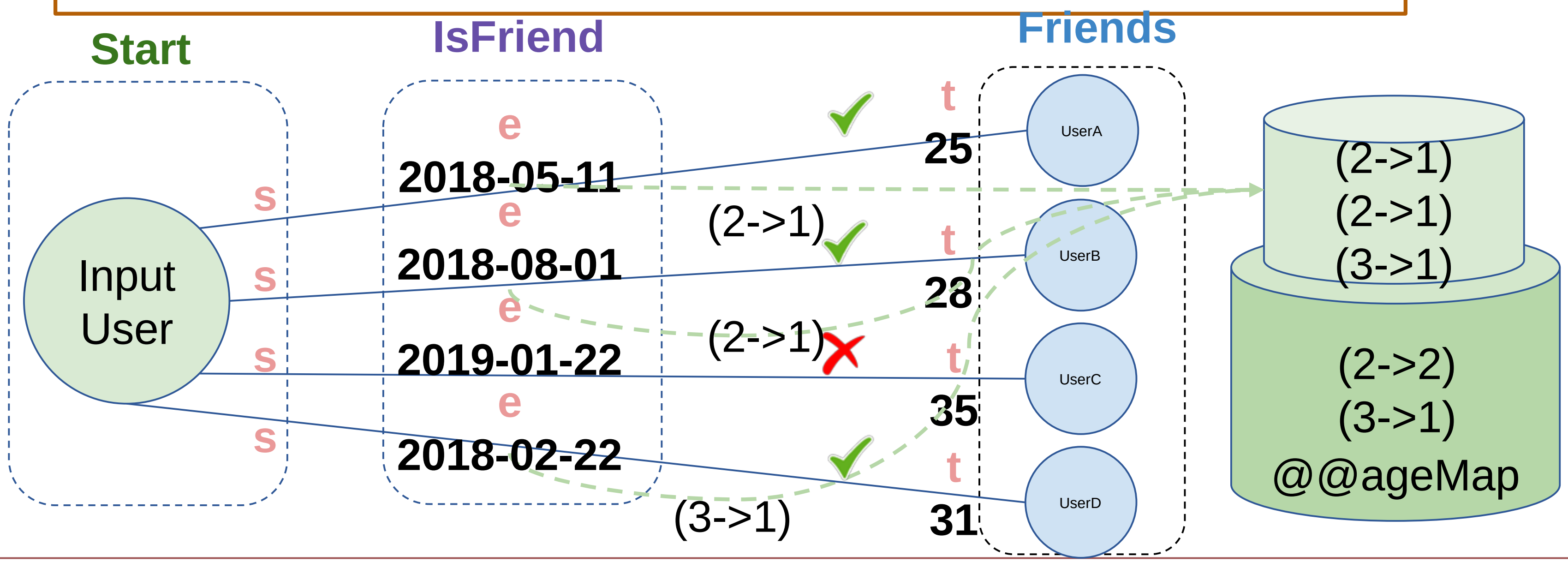
```
CREATE QUERY GetFriends(vertex<User> inputUser) FOR GRAPH Social {
    MapAccum<uint, uint> @@ageMap;
    Start = {inputUser};
     Friends = SELECT t FROM Start:s-(IsFriend:e)-:t
               WHERE e.connectDt BETWEEN to_datetime(“2018-01-01”)
                AND to_datetime(“2019-01-01”)
               ACCUM @@ageMap += (t.age/10->1);
     PRINT @@ageMap;
}
```



Select the matching edges

Local compute + send messages

Aggregate the messages to



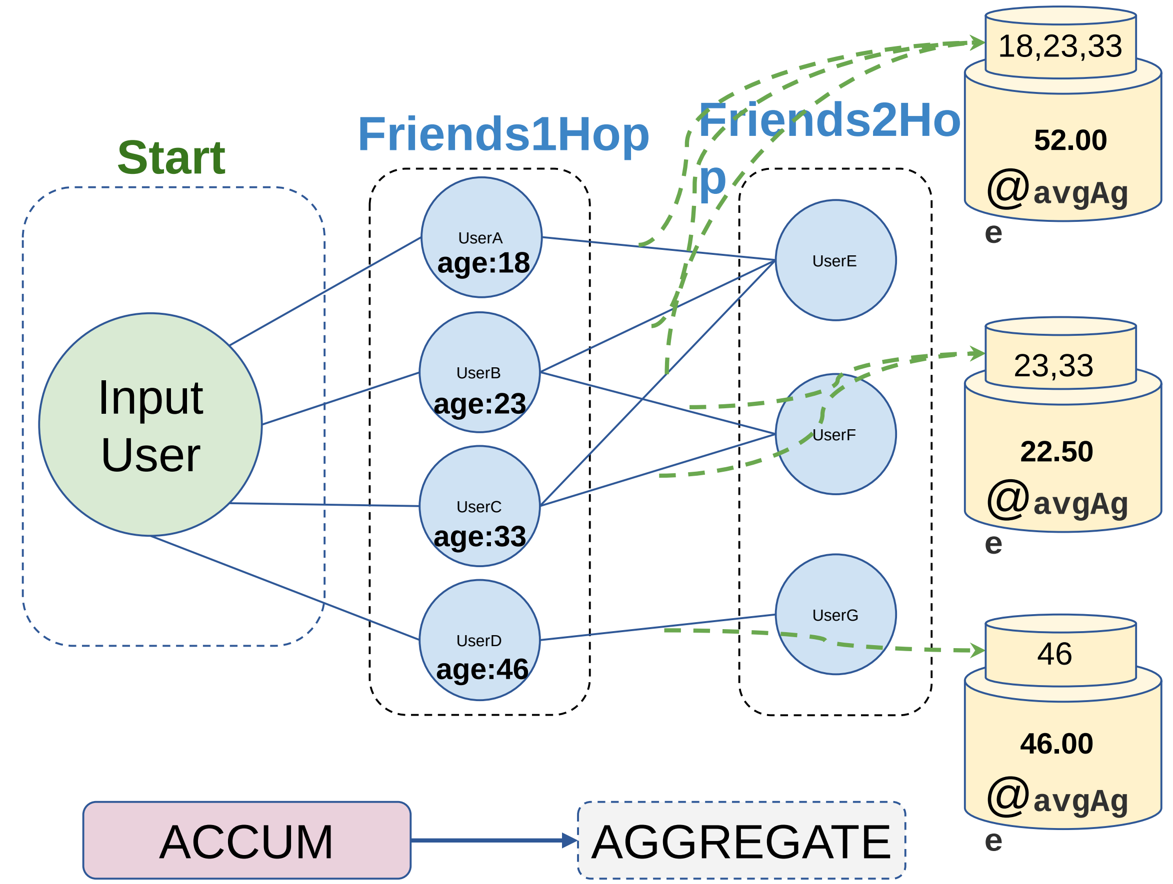
- Only the edges satisfy WHERE do logics in **ACCUM**
- In **ACCUM,** vertices do not see each other's updates b/c updates aren't processed until the AGGREGATE step.
- The **AGGREGATE** phase is done automatically after **ACCUM**. After that, the updated accumulator value can be accessed
- **+=** means sending message to accumulator
- **ACCUM** has access to s, e and t

ACCUM Clause

Output the average age of friends of friends

```
CREATE QUERY GetFriends(vertex<User> inputUser) FOR
GRAPH Social {
    AvgAccum @avgAge;
    Start = {inputUser};
    Friends1Hop = SELECT t FROM Start:s-(IsFriend:e)-:t;
    Friends2Hop = SELECT t
                  FROM Friends1Hop:s-(IsFriend:e)-:t
                  ACCUM t.@avgAge += s.age;
    print Friends2Hop;
}
```

- Update of local accumulator cannot be seen during **ACCUM** phase
- The messages will be aggregated during **AGGREGATE** phase based on accumulator type.



Send the messages to target nodes

Aggregate the messages to accumulator

Other Analytics Features

For use cases for

- complex data types (list, set, map, heap, user-defined tuple)
- control flow
- query-calling-query

See TigerGraph user documentation "GSQL Demo Examples"

https://docs.tigergraph.com/dev/gsql-examples

For TigerGraph's GSQL graph algorithm library, see

https://docs.tigergraph.com/graph-algorithm-library

Oskar van Rest
Oracle

Why Property Graphs with SQL?

- Users are using both SQL data and Property Graph data
- Application development is easier, better, quicker, faster if only one interface

SQL extensions for Property Graphs (PGs)

- Goal: define extensions to query property graphs
  - Agree on one (or possibly more) representation of PGs in SQL
    - Most obvious, in tables
    - Maybe later, some “native” storage format
  - Agree on the way to query PGs in SQL
    - Query PGs “natively” (use the power of pattern matching)
    - Represent result as a table (unleash the power of SQL on the result)
    - Maybe later DML operations on a property graph directly, and graph (view) construction
- Targeted for the next version of SQL (~2020/21)

Property Graph Definition (DDL) – Example

- Example:

> Create a PG w/ two vertex tables and two edge tables.

```
CREATE PROPERTY GRAPH myGraph
    VERTEX TABLES (Person, Message)
    EDGE TABLES (
        Created SOURCE Person DESTINATION Message,
        Commented SOURCE Person DESTINATION Message )
```

- Existing tables (or views): Person, Message, Created, Commented
- We infer keys & connections from primary/foreign keys of underlying tables
  - PK-FK determines connection between vertices via edges (e.g., `person -[created]-> message`)
- All columns of each table are exposed as properties of the corresponding vertex/edge (tables)

DDL – Example (cont.)

Example for optional clauses:

Same PG as before – but fine-grained control over labels, properties, etc.

```
CREATE PROPERTY GRAPH myGraph
    VERTEX TABLES (
        People KEY ( id )
               LABEL Person
               PROPERTIES ( emailAddress AS email ),
        Messages KEY ( id )
                 LABEL Message
                 PROPERTIES ( created AS creationDate, content ) )
    EDGE TABLES (
        CreatedMessage KEY ( id )
                       SOURCE KEY ( creator ) REFERENCES People
                       DESTINATION KEY ( message ) REFERENCES Messages
                       LABEL Created NO PROPERTIES,
        CommentedOnMessage KEY ( id )
                           SOURCE KEY ( commenter ) REFERENCES People
                           DESTINATION KEY ( message ) REFERENCES Messages
                           LABEL Commented NO PROPERTIES )
```

Querying PGs – Example

Get the creationDate and content of the messages created by one person ("email1") and commented on by another person ("email2").

Postfix operator applied to graph, returns table

```
SELECT GT.creationDate, GT.content
FROM myGraph GRAPH_TABLE (
  MATCH
   (Creator IS Person WHERE Creator.email = :email1)
      -[ IS Created ]->
   (M IS Message)
      <-[ IS Commented ]-
   (Commenter IS Person WHERE Commenter.email = :email2)
      WHERE ALL_DIFFERENT (Creator, Commenter)
  ONE ROW PER MATCH
  COLUMNS (
      M.creationDate,
      M.content )
 ) AS GT
```

Vertex pattern enclosed in ( )

Edge pattern enclosed in -[]->

COLUMNS defines the shape of the output table. Properties projected out of the MATCH.

Querying PGs – Example (cont.)

```
SELECT L.Here, GT.GasID, L.There, GT.TotalCost, GT.Eno, GT.Vid GT.Eid
FROM List AS L LEFT OUTER JOIN MyGraph GRAPH_TABLE (
  MATCH CHEAPEST (
      (H IS Place WHERE H.ID = L.Here)
          ( -[R1 IS Route COST R1.Traveltime]-> )*
               (G IS Place WHERE G.HasGas = 1)
           ( -[R2 IS Route COST R2.Traveltime]-> )*
      (T IS Place WHERE T.ID = L.There) )
  ONE ROW PER STEP (V, E)
  COLUMNS ( H.ID AS HID, G.ID AS GasID, T.ID AS TID, TOTAL_COST() AS totalCost,
            ELEMENT_NUMBER (V) AS Eno, V.ID AS Vid, E.ID AS Eid )
) AS GT ON (GT.HID = L.Here AND GT.TID = L.There)
ORDER BY L.Here, L.There, Eno
```

HERE	THERE
Home	HQ
Downtown	Uptown

Given a table with a list of pairs of places called Here and There, for each row in the list, find the cheapest path from Here (H) to There (T), with a stop at a gas station (G) along the way.

Status Update on PGQL

- What is PGQL (Property Graph Query Language)?
  - Query language for PGs with SQL-like syntax
  - Implemented in Oracle Spatial and Graph, Oracle Big Data Spatial and Graph, Oracle Labs' Parallel Graph AnalytiX (PGX)
  - Open-sourced Apache-licensed parser (https://github.com/oracle/pgql-lang)
- Not a standard, but trying to keep closely in sync. with standards
  - Same query structure as **SQL** (SELECT, FROM, WHERE, GROUP BY, ORDER BY, etc.)
  - Same functions and expressions as **SQL** (EXISTS, NOT EXISTS, CASE, CAST, EXTRACT, etc.)
  - Roughly same graph pattern matching capabilities as **SQL/PGQ**

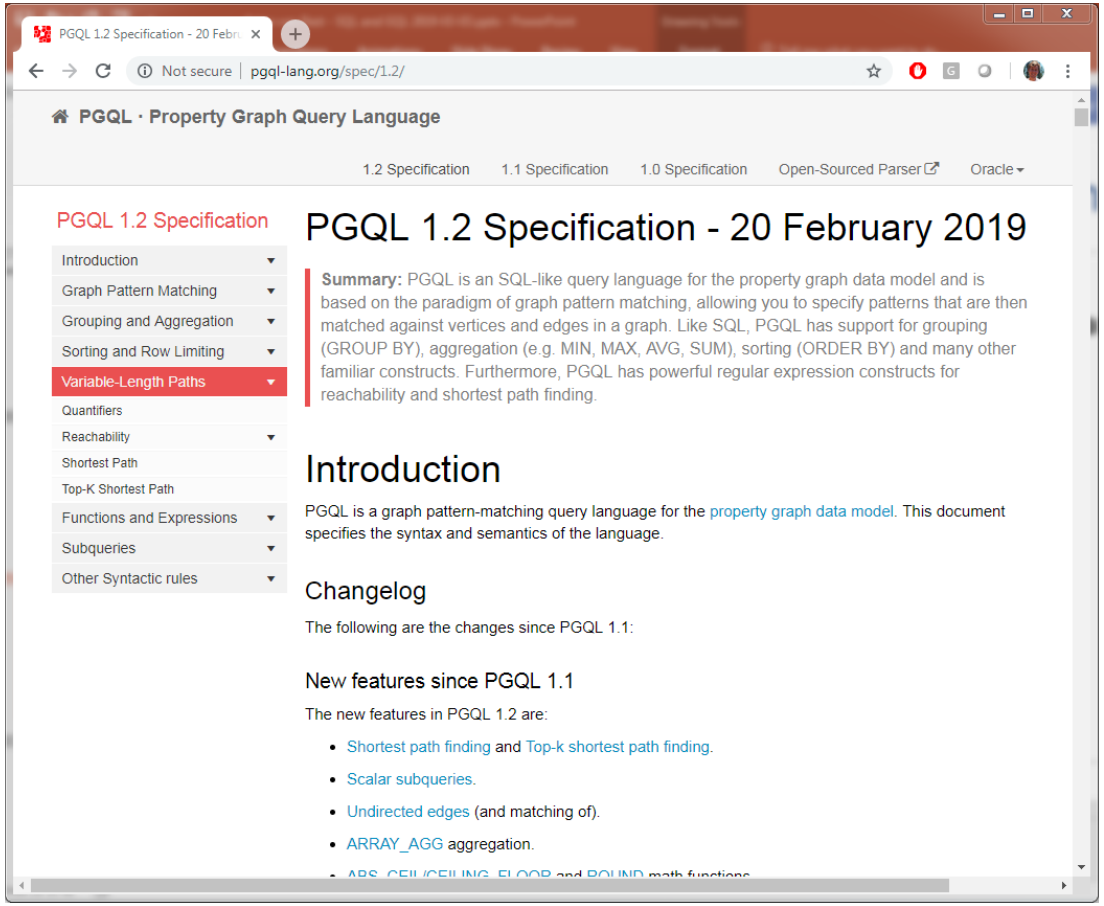
Example PGQL query:

```
  SELECT n.name, m.name,
         SUM(e.distance) AS path_distance
    FROM g MATCH SHORTEST ( (n:Place) –[e]->* (m:Place) )
   WHERE n.name = 'San Francisco' AND m.name = 'Amsterdam'
ORDER BY path_distance
```

Status Update on PGQL (cont.)

- Version 1.2 of PGQL was just released
  - New **graph** features:
    - SHORTEST path
    - TOP k SHORTEST path
    - Group variables and aggregations over them
    - Undirected edges (and matching of)
  - New **SQL** features:
    - Scalar subqueries
    - ABS, CEIL/CEILING, FLOOR and ROUND math functions
    - ARRAY_AGG aggregation
    - EXTRACT function for extracting the year/month/day/hour/minute/second/timezone_hour/timezone_minute from datetime values
    - CASE statement
    - IN and NOT IN predicates

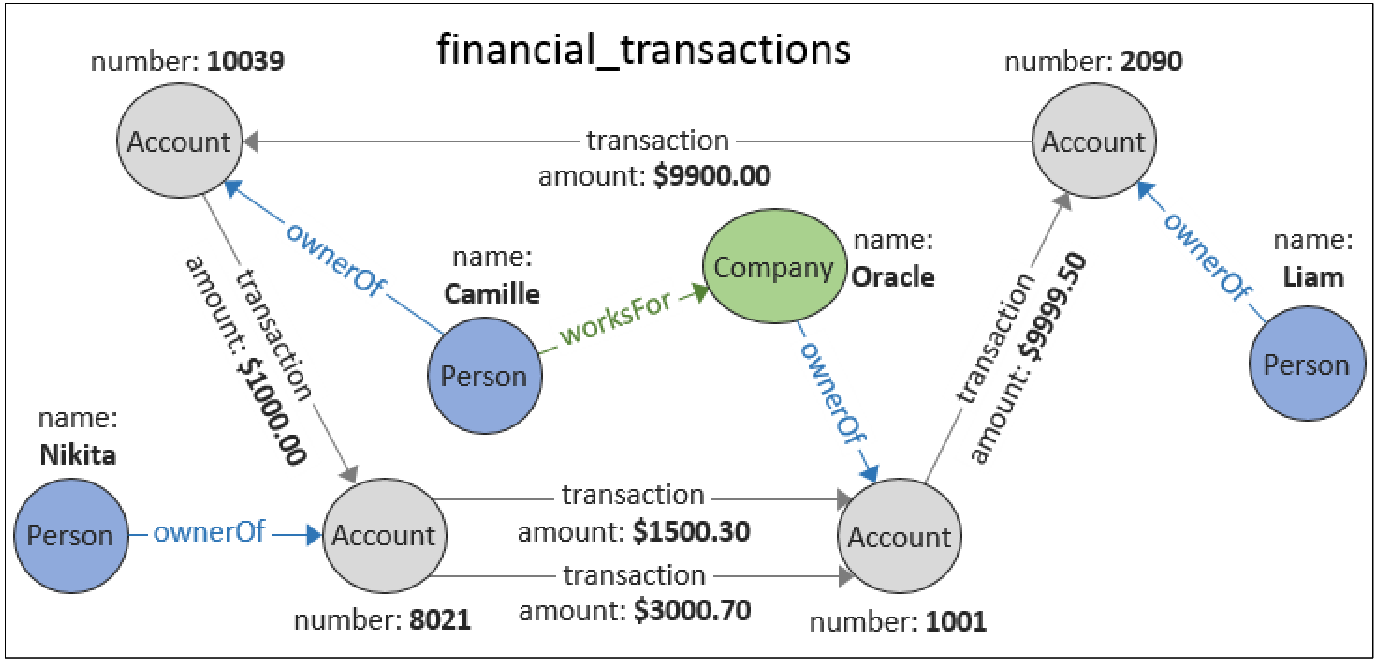
http://pgql-lang.org/spec/1.2/



PGQL – Example

Find 7 shortest paths from Account 10039 back to account 10039, following only “transaction” edges, and select:

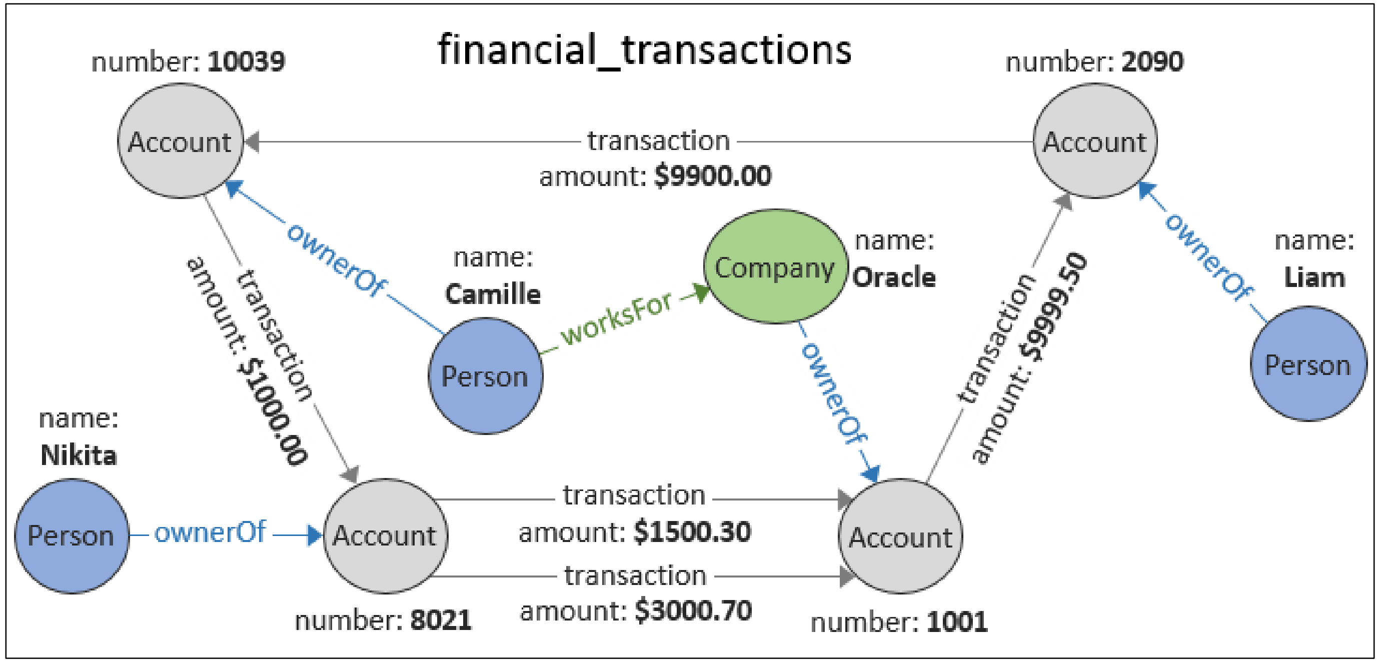
- The length of the path
- The sum of the amounts along the path
- The amounts along the path as an array of values




```
SELECT COUNT(e) AS num_hops,
       SUM(e.amount) AS total_amount,
       ARRAY_AGG(e.amount) AS amounts_along_path
  FROM financial_transactions MATCH TOP 7 SHORTEST (
         (a:Account) -[e:transaction]->* (b:Account) )
 WHERE a.number = 10039 AND a = b
ORDER BY num_hops, total_amount
```

num_hops	total_amount	amounts_along_path
0	<null>	<null>
4	22399.8	[1000.0, 1500.3, 9999.5, 9900.0]
4	23900.2	[1000.0, 3000.7, 9999.5, 9900.0]
8	44799.6	[1000.0, 1500.3, 9999.5, 9900.0, 1000.0, 1500.3, 9999.5, 9900.0]
8	46300.0	[1000.0, 1500.3, 9999.5, 9900.0, 1000.0, 3000.7, 9999.5, 9900.0]
8	46300.0	[1000.0, 3000.7, 9999.5, 9900.0, 1000.0, 1500.3, 9999.5, 9900.0]
8	47800.4	[1000.0, 3000.7, 9999.5, 9900.0, 1000.0, 3000.7, 9999.5, 9900.0]

PGQL – Example (cont.)



Select for each person in the graph:

- The name
- The sum of incoming transactions
- The sum of outgoing transactions
- The number of persons transacted with
- The number of companies transacted with

```
    SELECT p.name AS name,
           ( SELECT SUM(t.amount)      MATCH (a) <-[t:transaction]- (:Account) ) AS sum_incoming,
           ( SELECT SUM(t.amount)      MATCH (a) -[t:transaction]-> (:Account) ) AS sum_outgoing,
           ( SELECT COUNT(DISTINCT p2) MATCH (a) -[t:transaction]- (:Account) <-[:ownerOf]- (p2:Person)
                                       WHERE p2 <> p ) AS num_persons_transacted_with,
           ( SELECT COUNT(DISTINCT c)  MATCH (a) -[t:transaction]- (:Account) <-[:ownerOf]- (c:Company)
                                       ) AS num_companies_transacted_with
     MATCH (p:Person) -[:ownerOf]-> (a:Account)
  ORDER BY sum_outgoing + sum_incoming DESC
```

name	sum_incoming	sum_outgoing	num_persons_transacted_with	num_companies_transacted_with
Liam	9999.5	9900.0	1	1
Camille	9900.0	1000.0	2	0
Nikita	1000.0	4501.0	1	1

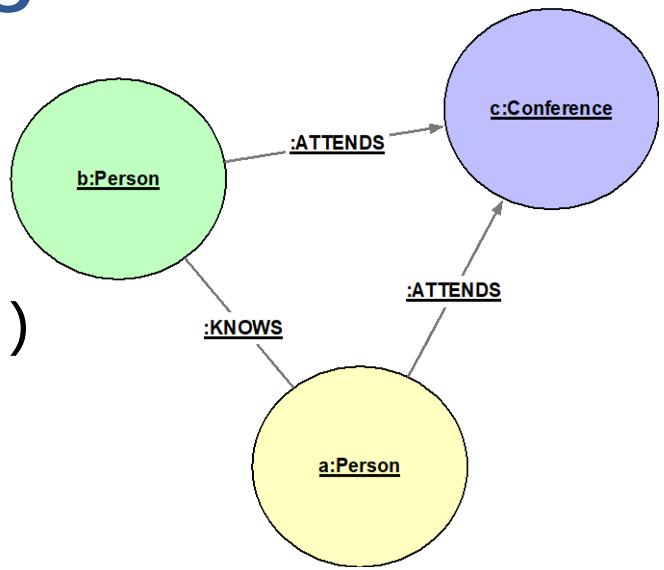
Stefan Plantikow

Neo4j

Declarative Property Graph Querying

- For Neo4j, it started with Cypher in 2011

```
MATCH (a:Person)-[:KNOWS]-(b:Person),
      (a)-[:ATTENDS]->(c:Conf)<-[:ATTENDS]-(b)
RETURN a.name, b.name, count(c)
```



- Since then:
  - New languages (openCypher, PGQL, G-Core, SQL/PGQ, GSQL)
  - New features (RPQs, DML, Views, Indices, Graph construction)
  - Many implementations

Graphs are a Top 10 Data and Analytics Trend for 2019. The application of graph processing and graph DBMSs will grow at 100 percent annually through 2022 to continuously. (Gartner)

From Cypher, PGQL, GSQL, SQL/PGQ to GQL

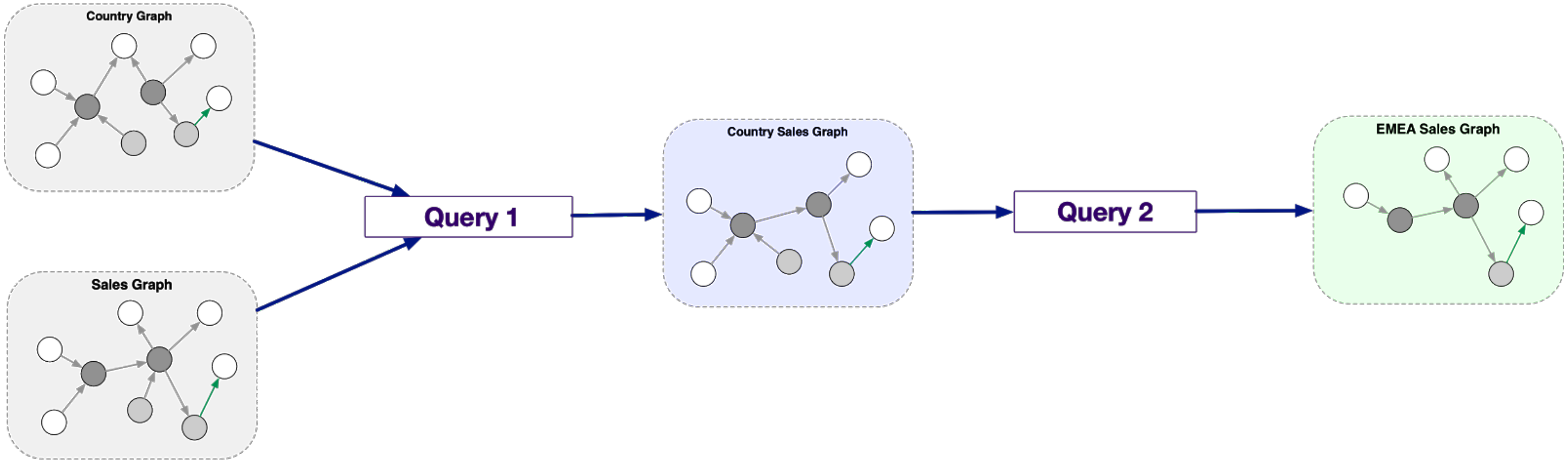
Simple Pattern Matching	**Complex Pattern Matching (RPQs, Shortest/Cheapest Path, Macros)**
Tables out only	**Graphs, tables, scalars in/out**
Single graph only	**Multiple graphs & (parameterized) views**
DML only	**DML, Graph computation, Graph projection**
No schema	**Schema & advanced type system**

All aligned with basic data types, infrastructure, and expressions of the SQL database

Support for basic tabular manipulation (projection, sorting, grouping etc)

http://tiny.cc/gql-scope-and-features

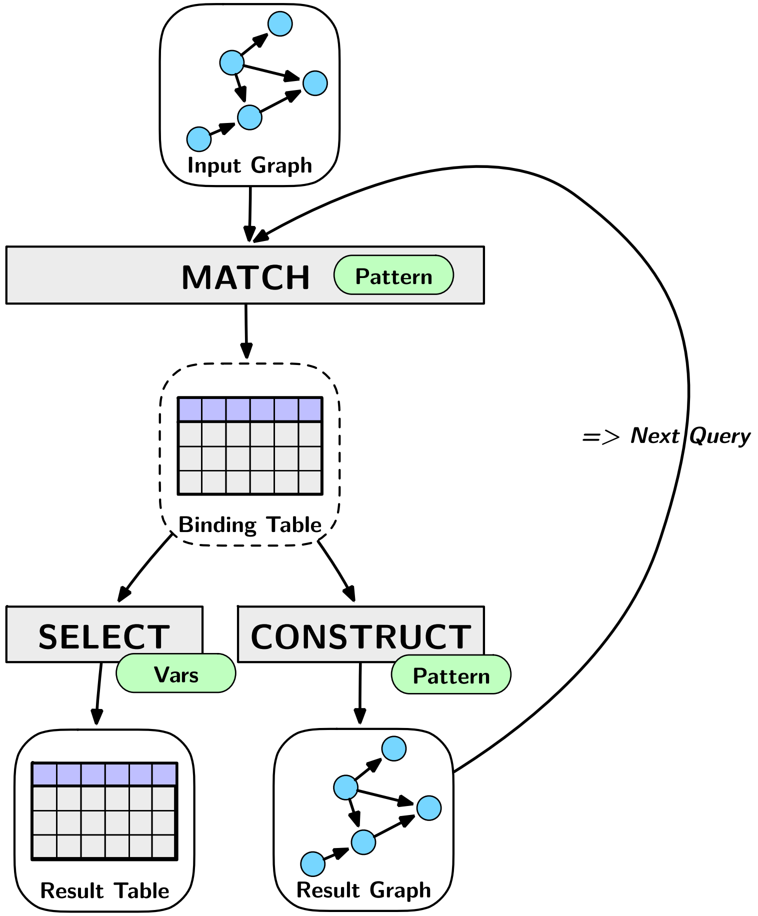
Query Composition



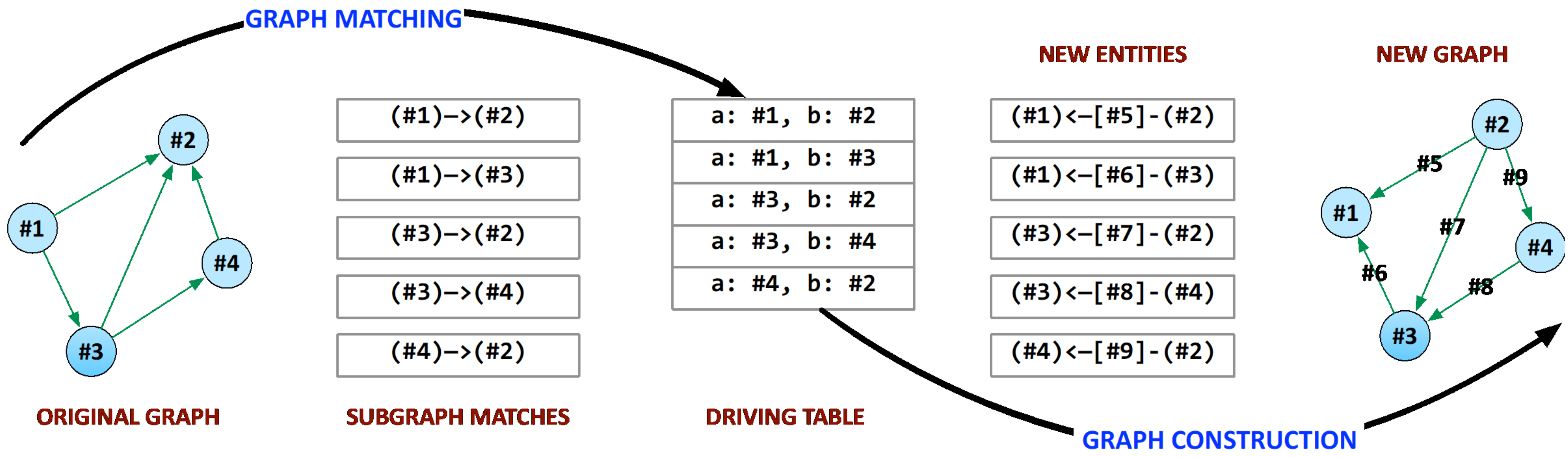
- Use the output of one query as input to another to *enable abstraction and views*
- Both for queries with *tabular* output and *graph* output
- Support for nested queries and procedures, too
- Simple linear composition of tabular output of one query as input to another (Lateral Join)

Query Composition Operators

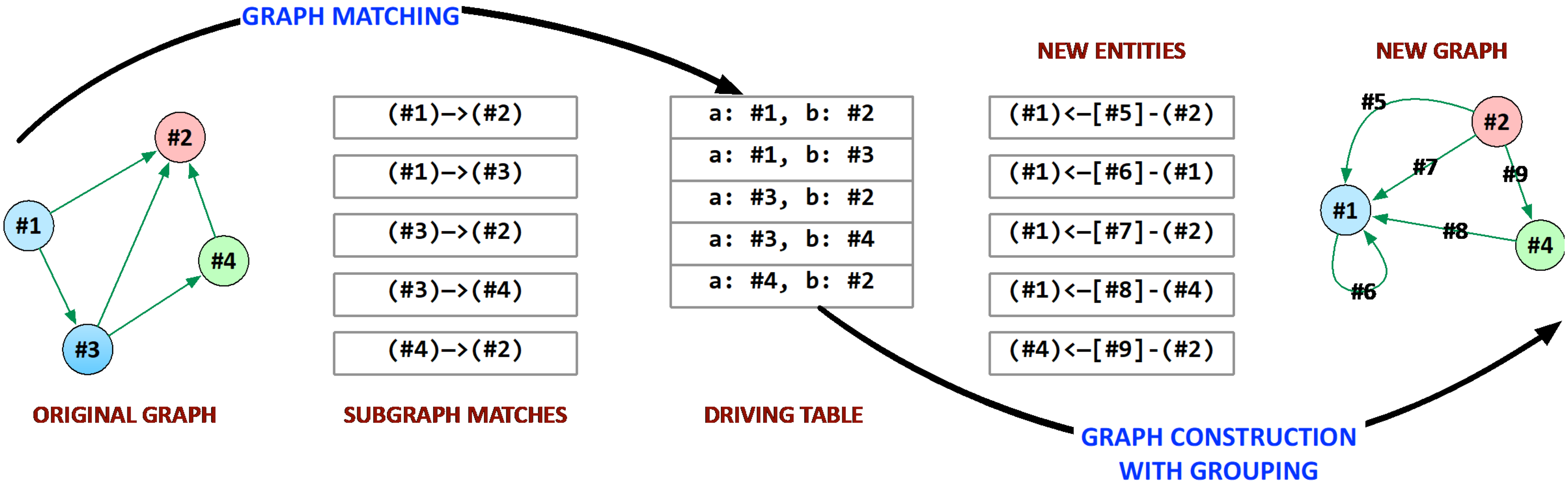
- Graph in => Graph out
- Gradually build up the right graph
  - Aggregate nodes and edges
  - Transform properties
  - Derive graph structure
- Match – (Construct – Match)* - Select?
- Graph operators: Union, Intersect etc.



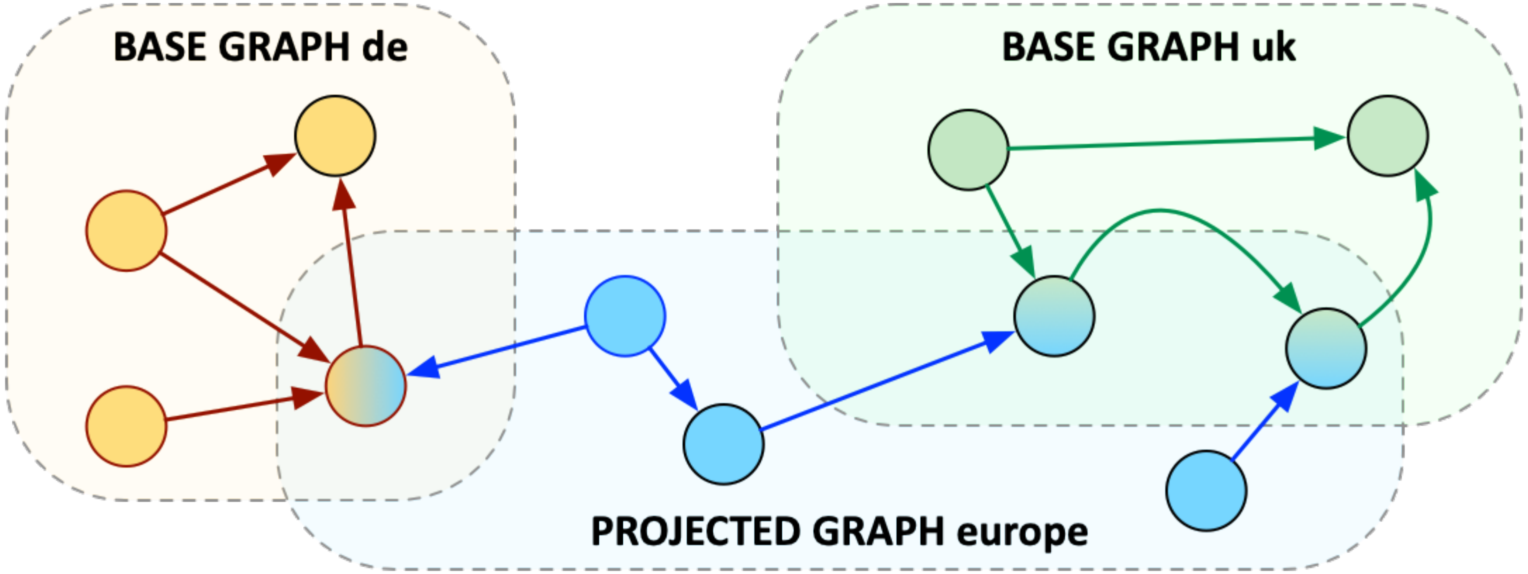
Graph Construction



Graph Construction with Grouping

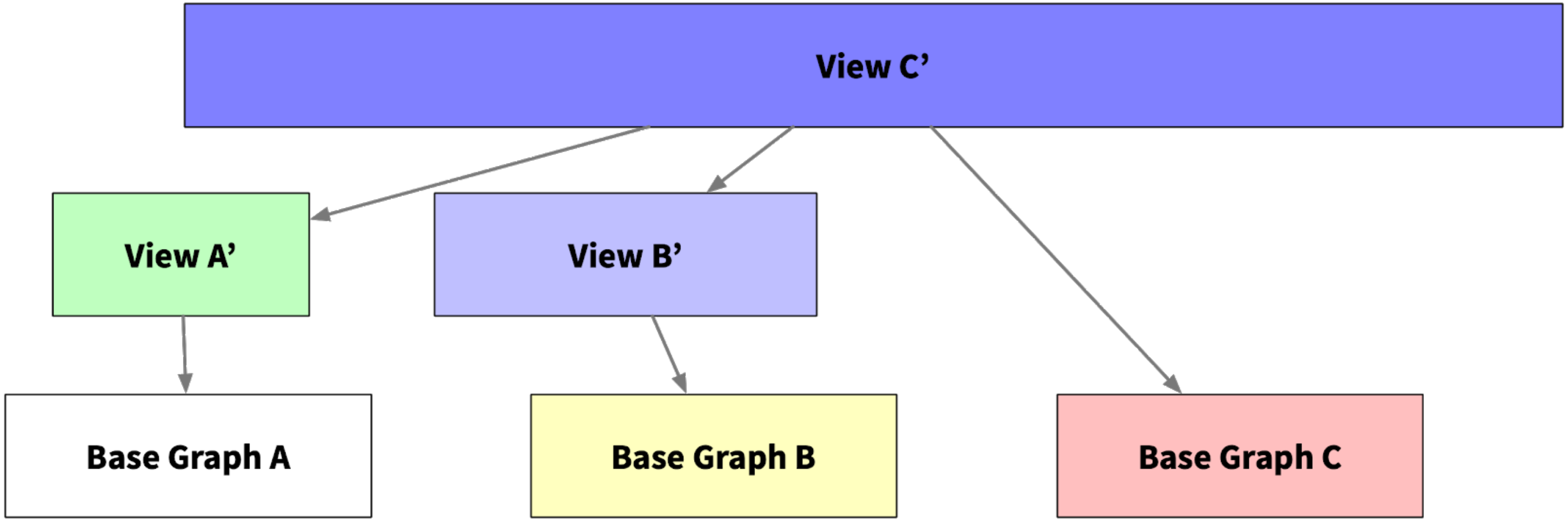


Projected graphs



- Sharing existing elements in the projected graph
- Deriving new elements in the projected graph
- Shared edges always point to the same (shared) endpoints in the projected graph

Views



- Graph elements are shared between graphs and views
- Graph elements are "owned" by their base graph or introducing views
- Sharing graph must form a DAG

Example Query

```
QUERY same_city_friends($year: INT) {
  FROM social_network
  MATCH (a)-[e1:LIVED_IN]->(c:City)<-[e2:LIVED_ID]-(b)-[:KNOWS]-(a)
  WHERE a <> b AND e1.year = $year AND e2.year = $year
  CONSTRUCT
    MERGE (a), (b)
    INSERT (a)<-[:SAME_CITY_FRIEND]->(b)
  RETURN GRAPH
}

FROM same_city_friends(1978)
MATCH SHORTEST SIMPLE PATH p=(a) (()-[:SAME_CITY_FRIEND]-())* (b)
RETURN size(p), count(p) GROUP BY size(p)
```

Schema & Graph Types

```
CREATE GRAPH TYPE Uni (
  -- Abstract element types
  University (),
  Course (name: STRING!),
  Person (birthday: DATE?, name: STRING!),
  Student <: Person (birthday: DATE?, name: STRING!, student_id: INT!),
  VISITS (term: STRING!),
  STUDIES_AT (),

  -- Allowed node and edge types in the graph
  (Student),
  (Course),
  (University),
  (Student)-[VISITS]->(Course),
  (Student)-[STUDIES_AT]->(University)
)
```

Type System

- Base data types from SQL (with modifications, i.e. only Unicode) `“abc”, 12.34`
- Support for nested data / documents `{ name: …, sizes: [ 1, 2 ] }`
- Dynamic typing <u>and</u> optional static typing
- Graph types

Towards GQL

- *More topics to come*
  Graph computation, Environment, Tabular features, DML, …

- *Editing*
  How to share data types between SQL Foundation and GQL?

- *Community engagement*
  gqlstandards.org => community call
  openCypher => openGQL

http://tiny.cc/gql-scope-and-features

GQL Scope and Features

A new *and independent*

Declarative,

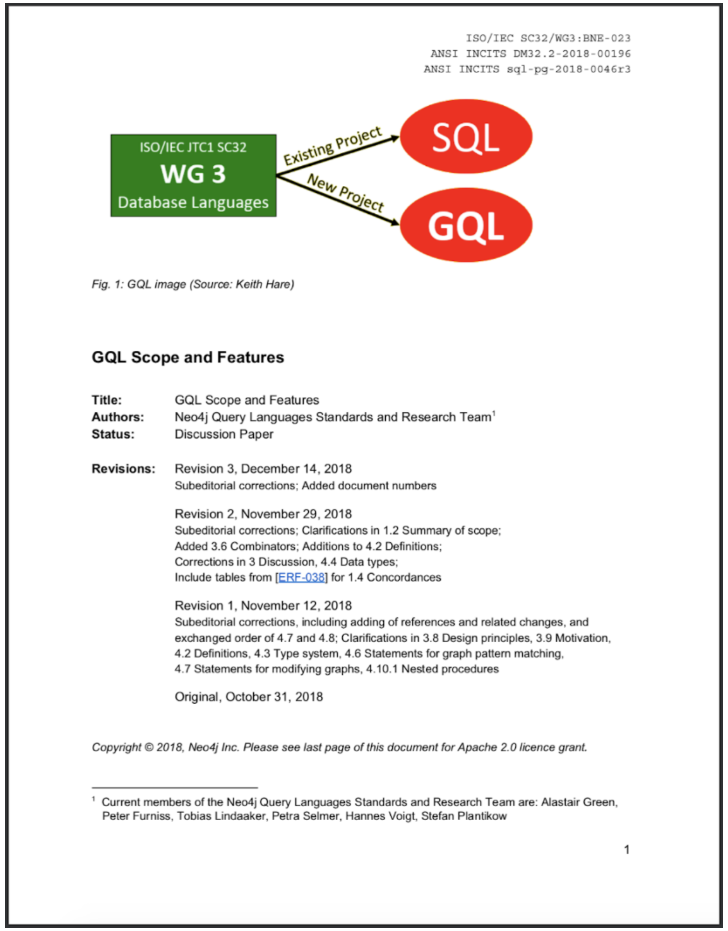
Composable,

Compatible,

Modern,

Intuitive

Property Graph Query Language



ISO/IEC SC32/WG3:BNE-023
ANSI INCITS DM32.2-2018-00196
ANSI INCITS sql-pg-2018-0046r3

Fig. 1: GQL image (Source: Keith Hare)

GQL Scope and Features

Title: GQL Scope and Features
Authors: Neo4j Query Languages Standards and Research Team[1]
Status: Discussion Paper

Revisions: Revision 3, December 14, 2018
Subeditorial corrections; Added document numbers

Revision 2, November 29, 2018
Subeditorial corrections; Clarifications in 1.2 Summary of scope;
Added 3.6 Combinators; Additions to 4.2 Definitions;
Corrections in 3 Discussion, 4.4 Data types;
Include tables from [ERF-038] for 1.4 Concordances

Revision 1, November 12, 2018
Subeditorial corrections, including adding of references and related changes, and exchanged order of 4.7 and 4.8; Clarifications in 3.8 Design principles, 3.9 Motivation, 4.2 Definitions, 4.3 Type system, 4.6 Statements for graph pattern matching, 4.7 Statements for modifying graphs, 4.10.1 Nested procedures

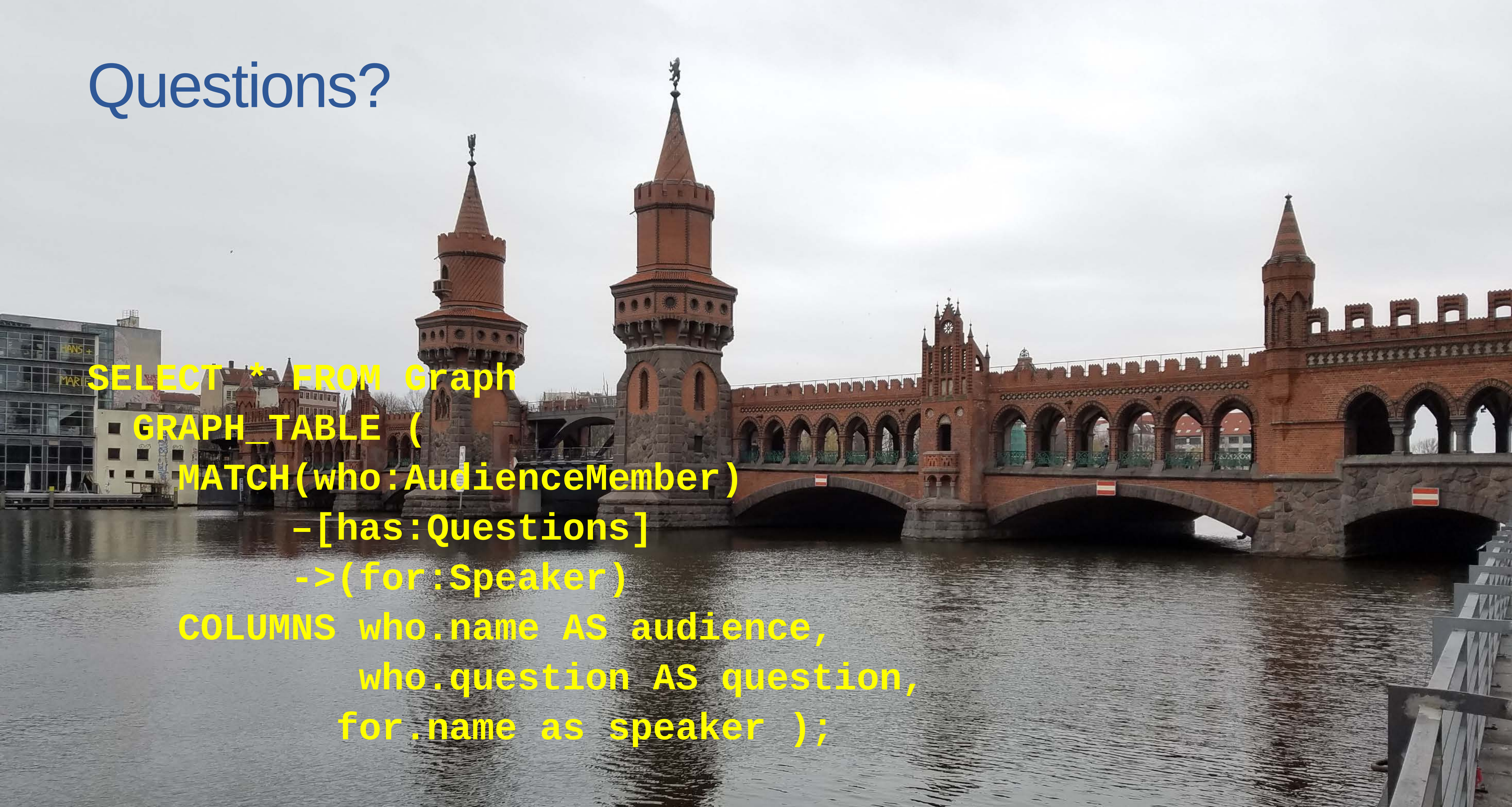
Original, October 31, 2018

[1] Current members of the Neo4j Query Languages Standards and Research Team are: Alastair Green, Peter Furniss, Tobias Lindaaker, Petra Selmer, Hannes Voigt, Stefan Plantikow

Summary

- SQL Standards have a long history
  - 30 years of experience integrating new technologies, including
    - Row Pattern Recognition
    - JSON
    - Polymorphic Table Functions
    - Additional analytics
    - Multi Dimensional Arrays – SQL/MDA
  - Property Graph queries in SQL
- New database language standard – Graph Query Language

Questions?



```
SELECT * FROM Graph
  GRAPH_TABLE (
    MATCH(who:AudienceMember)
         -[has:Questions]
         ->(for:Speaker)
    COLUMNS who.name AS audience,
            who.question AS question,
           for.name as speaker );
```

References

- ISO/IEC JTC1 SC32/WG3:ERF-037r1, “Relating GQL and SQL”, Fred Zemke, September 26, 2018.
- ISO/IEC JTC1 SC32/WG3:ERF-034 “GRAPH_TABLE Proposal”, Fred Zemke, September 14, 2018
- ISO/IEC JTC1 SC32/WG3:BNE-027r1 “Property Graph Data Model – The Proposal”, Jan Michels, January 16, 2019
- GQL Standards Web site: https://www.gqlstandards.org/