CIKM 2020 Tutorial

Multi-Model Data Query Languages and Processing Paradigms

Qingsong Guo, Jiaheng Lu, Chao Zhang
University of Helsinki

Calvin Sun, Steven Yuan
Huawei Technologies Co. Ltd

UNIVERSITY OF HELSINKI

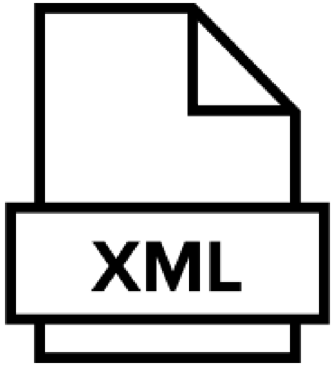
HUAWEI

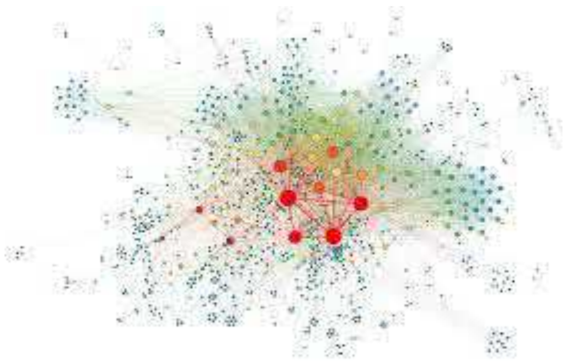
Agenda

1. Introduction
2. Data models
3. Multi-model data query languages
4. Comparison of the query languages
5. Open problem and challenges
6. Hands-on section

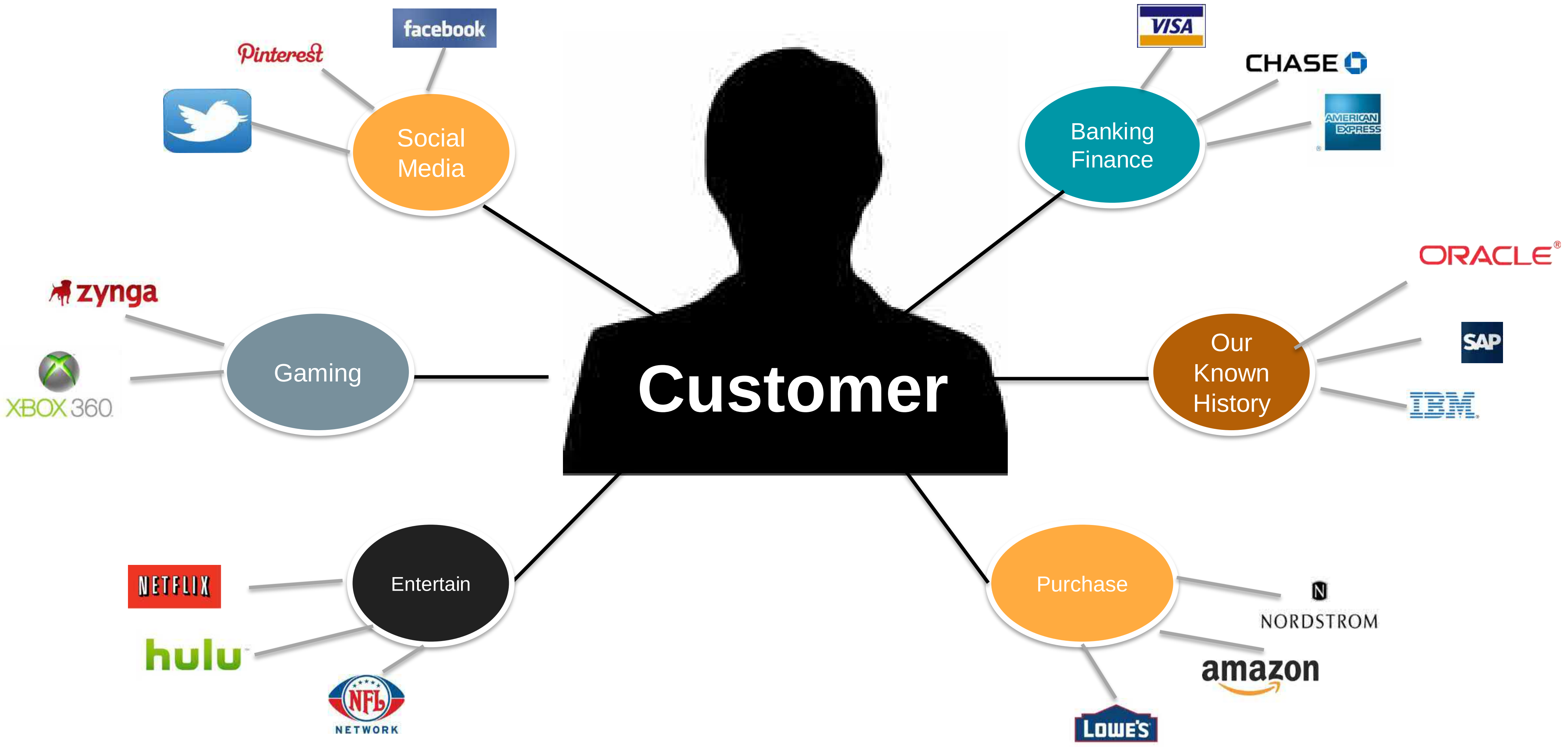
A grand challenge on variety

- Big data: Volume, Variety, Velocity, Veracity
- Variety: hierarchical data (XML, JSON), graph data (RDF, property graphs, networks), tabular data (CSV), etc.

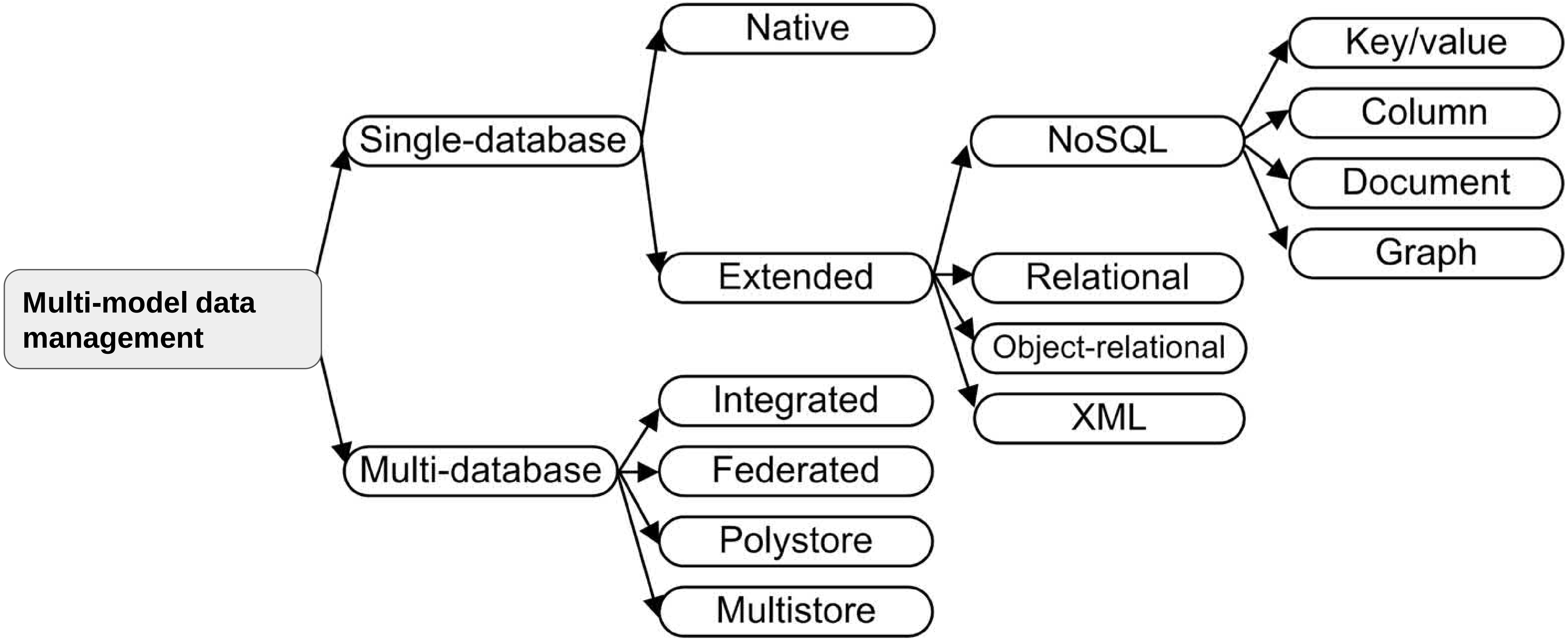




Motivation: E-commerce



Classification of approaches for multi-model data management



Single database: A Multi-model DB

- A multi-model database is designed to support multiple data models against a single, integrated backend.

Multi-model DB

Multi-model DBMSs

359 systems in ranking, October 2020

Rank			DBMS	Database Model	Score		
Oct 2020	Sep 2020	Oct 2019			Oct 2020	Sep 2020	Oct 2019
1.	1.	1.	Oracle	Relational, Multi-model	1368.77	-0.59	+12.89
2.	2.	2.	MySQL	Relational, Multi-model	1256.38	-7.87	-26.69
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1043.12	-19.64	-51.60
4.	4.	4.	PostgreSQL	Relational, Multi-model	542.40	+0.12	+58.49
5.	5.	5.	MongoDB	Document, Multi-model	448.02	+1.54	+35.93
6.	6.	6.	IBM Db2	Relational, Multi-model	161.90	+0.66	-8.87
7.	↑8.	7.	Elasticsearch	Search engine, Multi-model	153.84	+3.35	+3.67
8.	↓7.	8.	Redis	Key-value, Multi-model	153.28	+1.43	+10.37
9.	9.	↑11.	SQLite	Relational	125.43	-1.25	+2.80
10.	10.	10.	Cassandra	Wide column	119.10	-0.08	-4.12
11.	11.	↓9.	Microsoft Access	Relational	118.25	-0.20	-12.93
12.	12.	↑13.	MariaDB	Relational, Multi-model	91.77	+0.16	+5.00
13.	13.	↓12.	Splunk	Search engine	89.40	+1.51	+2.57

DB-engineers ranking ranks database according to their popularity. The ranking is updated monthly.

There are 8 multi-model database in top-10.

There are 85 multi-model database among 359 in total.

A true multi-model database can do :

Although many databases claimed that they are multi-model, they are not true multi-model databases.

A true multi-model database is expected to do:

- Provide a unified query language that not only query the individual data models, but also query across multiple data models,
- Index data with different models,
- Load multi-model data as is (no schema required before the loading),
- Provide ACID, scalability and security over multi-model data seamlessly.

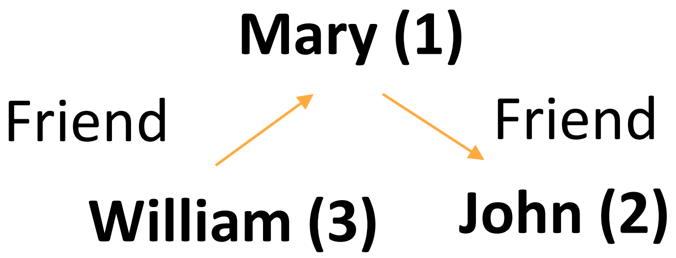
Two examples of open-source databases:

ArangoDB is designed as a native multi-model database, supporting key/value, document and graph models.

Orient DB supports graph, document, key/value and object models.

Both are open-source databases.

An example of multi-model data and query

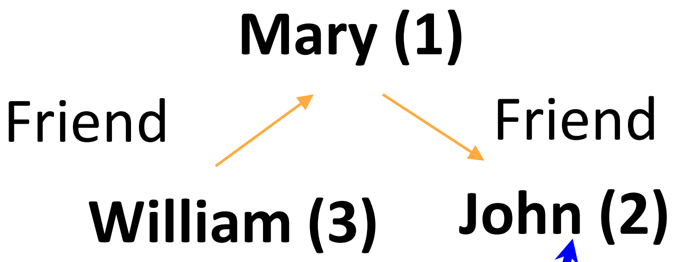



```
"1" -- > "34e5e759"
"2"-- > "0c6df508"
```

Customer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000

```
{"Order_no":"0c6df508",
 "Orderlines": [
   { "Product_no":"2724f"
     "Product_Name":"Toy",
     "Price":66 },
   { "Product_no":"3424g",
     "Product_Name":"Book",
     "Price":40 } ]
}
```

An example of multi-model data and query

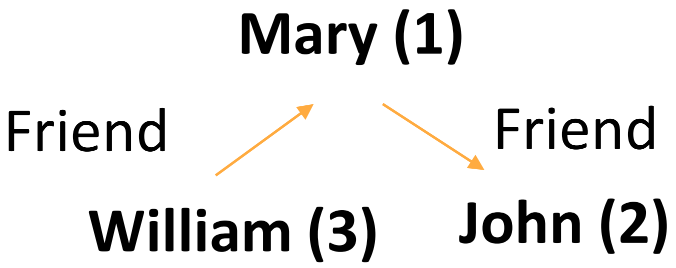


"1" -- > "34e5e759"
"2"-- > "0c6df508"

Customer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000

```
{"Order_no":"0c6df508",
 "Orderlines": [
   { "Product_no":"2724f"
     "Product_Name":"Toy",
     "Price":66 },
   { "Product_no":"3424g",
     "Product_Name":"Book",
     "Price":40 } ]
}
```

Q: Return all products which are ordered by a friend of a customer whose credit limit is over 3000



Customer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000

```
"1" -- > "34e5e759"
"2"-- > "0c6df508"
```

```
{"Order_no":"0c6df508",
 “Orderlines": [
    { "Product_no":"2724f”
      “Product_Name":“Toy",
      "Price":66 },
    { "Product_no":“3424g”,
      "Product_Name":“Book",
      "Price":40 } ]
}
```

An example of multi-model query (ArangoDB)

```
Let CustomerIDs =(FOR Customer IN Customers FILTER
Customer.CreditLimit > 3000 RETURN Customer.id)

Let FriendIDs=(FOR CustomerID in CustomerIDs FOR
Friend IN 1..1 OUTBOUND CustomerID Knows return
Friend.id)

For Friend in FriendIDs

For Order in 1..1 OUTBOUND Friend Customer2Order

Return Order.orderlines[*].Product_no
```

Recommendation query:

Return all products which are ordered by a friend of a customer whose credit limit is over 3000.

```
Select
expand(out("Knows").Orders.orderlines.Produ
ct_no) from Customers where CreditLimit >
3000
```

Recommendation query:

Return all products which are ordered by any friend of a customer whose credit limit is over 3000.

Summary for Introduction part

- Multi-model data management emerges to handle the Variety challenge of big data.
- There is no standard for multi-model query languages so far.
- Existing multi-model query languages are extended from SQL, XQuery or graph query languages.

Main references about Introduction to multi-model databases

- Pete Aven Building on Multi-Model Databases Released July 2017 Publisher(s): O'Reilly Media, Inc. ISBN: 9781491977903
- J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The bigdawg polystore system. SIGMOD Rec., 44(2):11–16, 2015.
- J. Lu and I. Holubová. Multi-model data management: What’s new and what’s next? In Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017, pages 602–605. OpenProceedings.org, 2017.
- J. Lu and I. Holubová. Multi-model Databases: A new journey to handle the variety of data. ACM Computing Surveys, 52(3), 2019.

02 Data models

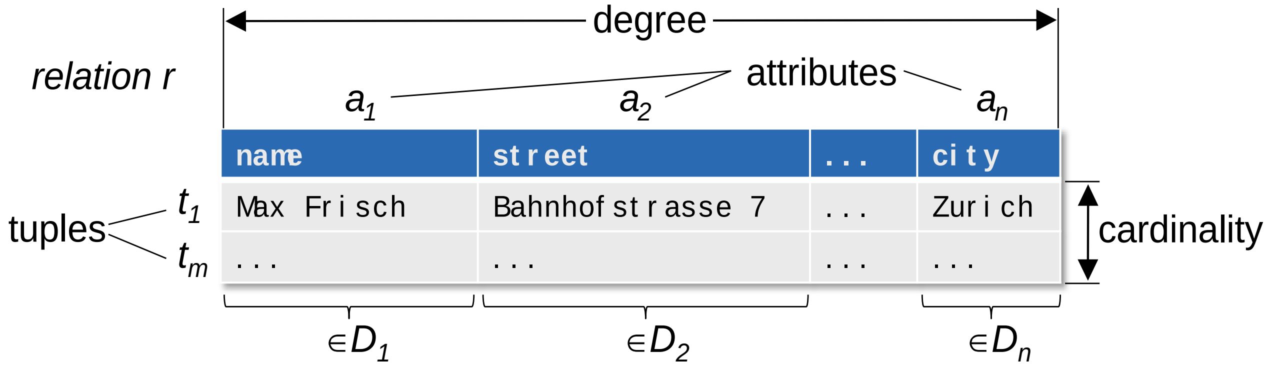
We will briefly discuss the major data models adopted by database systems and a benchmark for multi-model data.

- The relational model and its extensions
- The semi-structured data models, e.g. XML and JSON
- The graph data models
- …

The Relational Model

The dominant data model of last 5 decades

- A relation is a subset of Cartesian product and logically represented as un-ordered tuples and each record is uniquely identified by a key
- Table, column, rows
- Cannot nest one tuple within another



A relational model can be described by 3 components:

- Primitive types: integer, char, string, date, etc.
- Relational constructor used on the primitive types
- A set of operators that can be used to each primitive type and type constructor

Extensions for the Relational Model

The relational model can be extended by modifying these components

- **Nested relational model (NRM)**
  - Remove the restriction of 1NF
  - Contains nested type constructors that allow building nested relations from atomic types by using tuple constructors and set constructors
- **Object-relation model (ORM)**
  - separates set and tuple of the relational constructor and support object
- **JSON**
  - includes other type constructors such as lists, multisets, arrays, etc.

Semi-Structured Data: XML/JSON

- Self-describing by associating semantic tags or markers and enforce hierarchies of records and fields by nesting elements within the data.
- Enable more flexible processing and exchanging of the data.

- Richer (than relational) type systems
  - Object-Oriented data model
  - Nested Relational data model
- Schemaless and Schema-Optional data
  - XML as labelled tree
  - Schemaless Labelled Graphs
- Scalable nested & semistructured formats
  - (schemaless) JSON
  - (machine-oriented, columnar) Parquet, ORC, …
  - Google's buffer protocols …

Can be thought of as SQL data model extension and restriction removal

- complex types: arrays, (nested) tuples, maps
- rigid schema is not necessary

JSON as an example

Primitive values

- A string, which looks like "Hello"
- A number, which looks like 42 or -3.14159
- true or false
- null

Structured values

- Object: a list of name-value pairs (i.e., fields)

```
{ "partno": 461,
  "description": "Wrench"
}
```

- Array: an ordered list of items
  - `[1, 2.5, "Hello", true, null]`

Order JSON document

```
{"Order_no":"0c6df508",
    “Orderlines": [
        { “Product_no”: "2724f”
          “Product_Name”: “Toy",
          "Price":66 },
        { “Product_no”: “3424g”,
          “Product_Name”: “Book",
          "Price":40 } ]
}
```

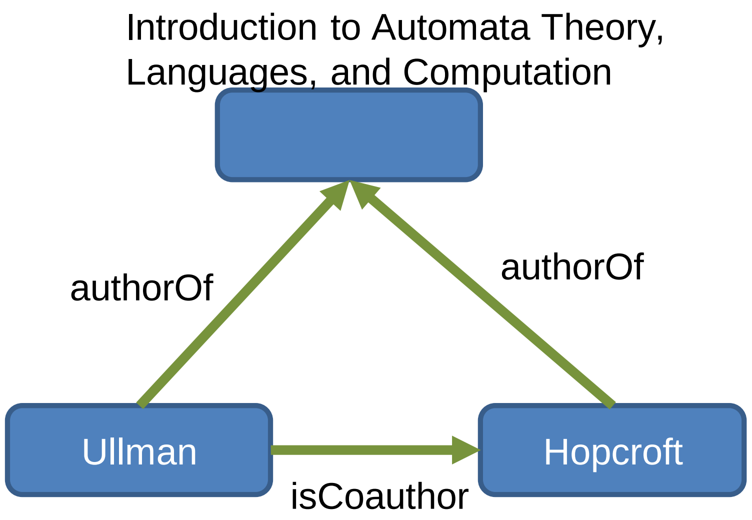
The items in an array and the values in the fields of an object can be any JSON values, arrays and objects

Graph Data Models

- A generalization of the relational model and semi-structured model
- It consists of a set of vertices V and edges E connecting the vertices from V

- Edge-labeled graph (N, E, L)
  – RDF <subject, predicate, object>, knowledge graph
  – SPARQL

- Property graph model (PGM)
  – Represents data as a directed, attributed multi-graph. Vertices and edges are rich objects with a set of labels and a set of key-value pairs, so-called properties
  – Cypher, openCypher, Gremlin, etc.



<Ullman, authorOf, Introduction to Automata Theory, Languages, and Computation>

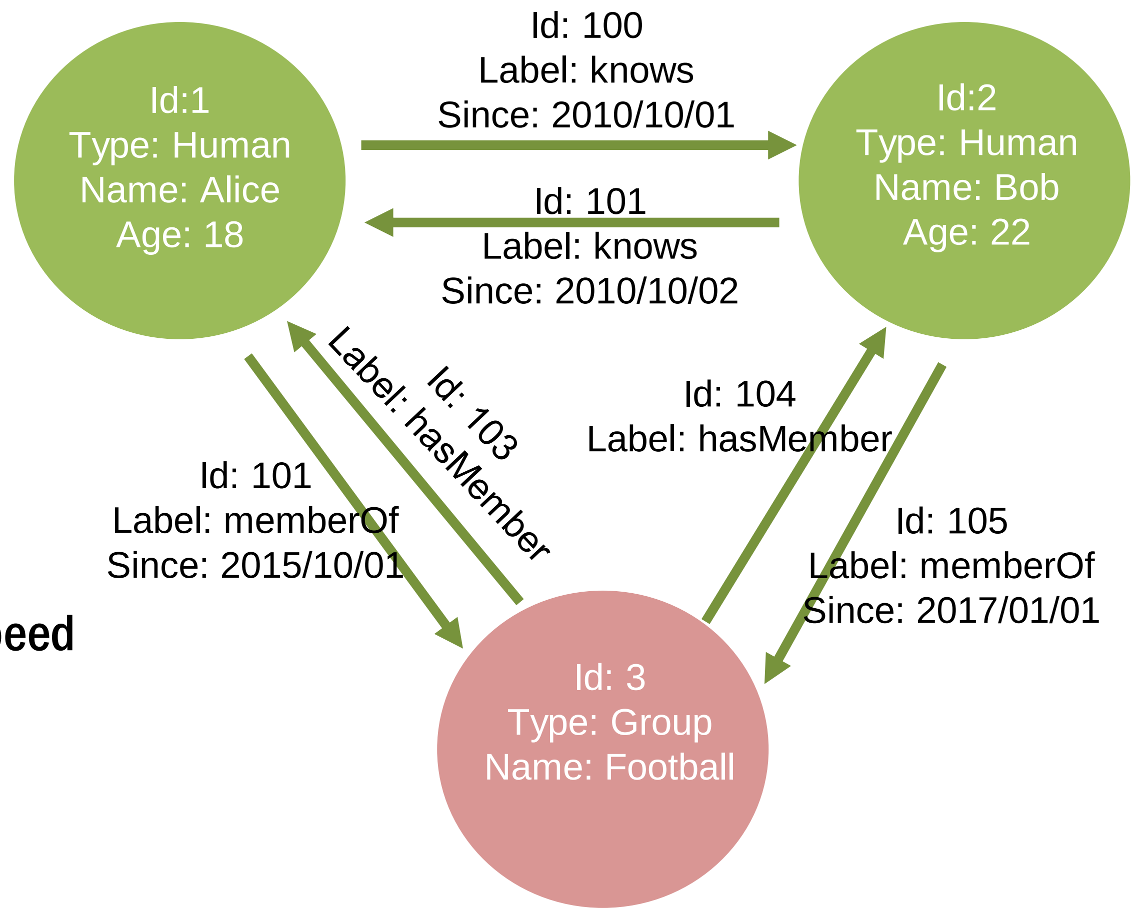
< Hopcroft, authorOf, Introduction to Automata Theory, Languages, and Computation>

<Ullman, isCoauthor, Hopcroft>

Property Graph Model

Key features:

- Nodes have labels, *Type:Human*
- Nodes have key-value properties
- Relationships between nodes
- Relationships have labels
- Relationships have key / value properties
- Relationships are directed but transversal at equals speed in both directions
- Semantics of the directions is up to the applications



Key-Value Data

- The simplest data model consists of a collection of <key, value> mappings

KEY1 → Value1
KEY2 → Value2
KEY3 → Value3
KEY4 → Value4
…

(a)

Key	Value
User1: employee	{65, 865, 9634}
User2: employee	{34, 85, 76, 94}
User3: employee	{name: mark, empid:346}
User4: employee	{desg:manager, branchcode: 345}
…	….

(b)

Formal Relational Query Languages

Two mathematical Query Languages form the basis for “real” languages (e.g. SQL), and for implementation

- Relational algebra
  - More operational(procedural), and always used as an internal representation for query evaluation plans
  - Select, Project, Union, Set different, Cartesian product, Rename

- Relational calculus
  - Tuple Relational Calculus: filtering variable ranges over tuples {T | Condition}
    - Alpha: proposed by Codd in 1971; QUEL: INGRES 1975
    - { T.name | Author(T) AND T.article = 'database' }
  - Domain Relational Calculus: the filtering variable uses the domain of attributes instead of entire tuple values, $\{ a_1, a_2, a_3, \ldots, a_n \mid P(a_1, a_2, a_3, \ldots ,a_n)\}$
    - {< article, page, subject > | ∈ TutorialsPoint ∧ subject = 'database'}

Relational Query Language: SQL

- SQL is a standard language for querying and manipulating data

- SQL is a very high-level (declarative) programming language
  - **SELECT, WHERE, FROM** syntax
  - This works because it is optimized well!

- Many standards out there:
  - ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ...
  - Vendors support various subsets
  - Recursive common table expression (CTE)

SQL stands for
Structured **Q**uery **L**anguage

Path Queries for Document Data

- **Path query** P = $x \xrightarrow{\alpha} y$
  - Regular path query (RPQ)
  - Conjunctive regular path query (CRPQ)
  - Context-free path query (CFPQ, replacing regular expressions with context-free grammar)
- XML documents
  - XPath, XQuery
  - .//x[@knows]/y
- JSON (JavaScript Object Notation)
  - SQL++, JSONiq (based on XQuery), UNQL (like SQL), JsonPath (XPath-like), GraphQL, etc.

RPQ P: The (transitive) friend-of-a-friend relationship in social network

P := $x \xrightarrow{knows^+} y$

CRPQ C = (P_1 ^ P_2 ^ ^ P_n), where R1, ..., Rn are RPQs

P1 := $x \xrightarrow{a^+} y$

P2 := $x \xrightarrow{c^+} z$

P3 := $y \xrightarrow{b^+} z$

Ingredients for Graph Query Languages

Pioneered by academic work on Conjunctive Query (CQ) extensions for graphs (in the 90's)

- SPARQL, Cypher, Gremlin
- Path expressions (PEs) for navigation
- Variables for manipulating data found during navigation
- Stitching multiple PEs into complex graph patterns → conjunctive regular path queries (CRPQs)

A RPQ: citizenOf | ((bornIn | livesIn) locatedIn*)

A simple graph pattern: (x, hasWon, Nobel), (x, hasWon, Booker)

A complex graph patterns: Ans(x,y)=← (x, hasWon, Nobel), (x, hasWon, Booker), (x, (citizenOf | ((bornIn | livesIn) locatedIn*)), y)

Running Example

Running example in CRPQ form

- count toys bought in common per customer pair

```
Q(c1, c2, count (p)) :- c1 –Bought-> p,
                        c2 –Bought-> p,
                     p.category = "toys",
                        c1 < c2
```

Product-Customer Graph

Vertex types:

- Product (name, category, price)
- Customer (ssn, name, address)

Edge types:

- Bought (discount, quantity)
- Customer c bought 100 units of product p at discount 5%:

modeled by edge

c -- (Bought {discount=5%, quantity=100})→ p

Running Example in SPARQL/Cypher

- Querying with SPARQL

```
SELECT  ?c1, ?c2, count (?p)
WHERE  { ?c1 bought ?p.
         ?c2 bought ?p.
           ?p  category ?cat.
           FILTER (?cat == “toys” && ?c1 < ?c2) }
GROUP BY ?c1, ?c2
```

- Querying with Cypher

```
MATCH   (c1:Customer) –[:Bought]-> (p:Product)
          <-[:Bought]- (c2:Customer)

WHERE   p.category = “Toys” AND c1.name < c2.name

RETURN  c1.name AS cust1,
          c2.name AS cust2,
          COUNT (p)  AS inCommon
```

c1.name, c2.name are composite group key
– no explicit group-by clause, just like CQ

Running Example in Gremlin

place one traverser on each vertex

filter traversers by label

extend each traverser t: bind variable 'c1' to the vertex where t resides

```
V().hasLabel('Customer').as('c1')
   .out('Bought').hasLabel('Product').has('category','Toys').as('p')
   .in('Bought').hasLabel('Customer').as('c2')
```

Traversers flow along out-edges/in-edges of type 'Bought'

```
.select ('c1', 'c2','p').by('name')
.where ('c1', lt('c2'))
```

for each traverser extract the tuple of bindings for variables c1,c2,p, return its projection on 'name' property.

```
.group().by(select('c1','c2')).by(count())
```

filter these tuples according to where condition

group tuples

first by() specifies group key

second by() specifies group aggregation

Reference

- S. Abiteboul and C. Beeri. On The Power Of Languages For The Manipulation Of Complex Objects. Technical Report 846, INRIA, Paris, May 1988.
- R. Angles, M. Arenas, P. Barceló, P. A. Boncz, G. H. L. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. F. Sequeda, O. van Rest, and H. Voigt.
- G-CORE: A core for future graph query languages. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, pages 1421–1432. ACM, 2018.
- E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. Research Report / RJ / IBM / San Jose, California, RJ599, August 1969.
- E. F. Codd. A relational model of data for large shared data banks. Commun. ACM, 13(6):377–387, 1970.
- E. F. Codd. Extending the database relational model to capture more meaning. ACM Trans. Database Syst., 4(4):397–434, Dec. 1979.
- A. Deutsch, Y. Xu, M. Wu, and V. Lee. Tigergraph: A native MPP graph database. CoRR, abs/1901.08248, 2019.
- O. Hartig and J. Pérez. Semantics and complexity of GraphQL. In Proceedings of the 2018 World Wide Web Conference, WWW ’18, pages 1155–1164, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.
- I. Robinson, J. Webber, and E. Eifrem. Graph Databases: New Opportunities for Connected Data. O’Reilly Media, Inc., 2nd edition, 2015.
- M. A. Rodriguez. The Gremlin Graph Traversal Machine and Language. CoRR, abs/1508.03843, 2015.
- M. H. Scholl. Extensions to the Relational Data Model. In Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development. Jon.Wiley & Sons, 1992.
- M. H. Scholl, H. Paul, and H. Schek. Supporting flat relations by a nested relational kernel. In VLDB’87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England, pages 137–146. Morgan Kaufmann, 1987.

03 Multi-model data query languages

We will discuss the syntax of 6 well-known multi-model data query languages, which fall into three categories:

- Relation-extensions: Asterix SQL++, Oracle PL/SQL
- Document-extensions: Marklogic XQuery, ArangoDB AQL
- Graph-extensions: OrientDB, AgensGraph

AsterixDB SQL++

- SQL++ : A Backwards-Compatible SQL , which can access a SQL extension with nested and semi-structured data
- Queries exhibit XQuery and OQL abilities, yet backwards compatible with SQL-92
- Supports relation and JSON

- Simpler than XML and the XQuery data model
- Unlike labeled trees (the favorite XML abstraction of XPath and XQuery research) makes the distinction between tuple constructor and list/array/bag constructor

SQL++: http://arxiv.org/abs/1405.3631
http://db.ucsd.edu/wp-content/uploads/pdfs/375.pdf

SQL++ Data Model

```
{
  location: 'Alpine',
  readings: {{
    {
      time: timestamp('2014-03-12T20:00:00'),
      ozone: 0.035,
      no2: 0.0050
    },
    {
      time: timestamp('2014-03-12T22:00:00'),
      ozone: 'm',
      co: 0.4
    } }}
}
```

- With schema
- Or, schemaless
- Or, **partial schema**

```
{
  location: string,
  readings: {{
    {
      time:  timestamp,
      ozone: any,
      *
    }
  }}
}
```

SQL++ Data Model

Can think of as extension of SQL

- Extend with arrays + nesting + heterogeneity by following JSON's notation

Heterogeneous tuples in collections

Array nested inside a tuple

Arbitrary compositions of array, bag, tuple

```
{
 'location': 'Alpine',
 'readings': [
  {
   'time': timestamp('2014-03-12T20:00:00'),
   'ozone': 0.035,
   'no2': 0.0050
  },
  {
   'time': timestamp('2014-03-12T22:00:00'),
   'ozone': 'm',
   'co': 0.4
  } ]
}
```

Can also think of as extension of JSON

- Use single quotes for literals
- Extended with bags and enriched types

Bags {{ … }}

Enriched types

```
{
 'location': 'Alpine',
 'readings': {{
  {
   'time': timestamp('2014-03-12T20:00:00'),
   'ozone': 0.035,
   'no2': 0.0050
  },
  {
   'time': timestamp('2014-03-12T22:00:00'),
   'ozone': 'm',
   'co': 0.4
  } }}
}
```

SQL++ Queries

BNF Grammar for SQL++ queries

- Semi-structured query
- Composability*:*
  - **SELECT-FROM-WHERE** (**SFW**)
  - Complex: tuple, collection *or* map
- Configuration parameters
- A map contains mappings of value pairs

query	→	*sfw_query*
	|	*expr_query*
sfw_query	→	*config*...(*sfw_query*)
	|	SELECT [DISTINCT] *select_clause*
		[FROM *from_item* ...]
		[WHERE *expr_query*]
		[GROUP BY *group_item* ...]
		[HAVING *expr_query* ...]
		[(UNION|INTERSECT|EXCEPT)
		[ALL] *sfw_query*]
		[ORDER BY *order_item* ...]
		[LIMIT *expr_query*]
		[OFFSET *expr_query*]
select_clause	→	[TUPLE] *select_item* ...
	|	ELEMENT *expr_query*
select_item	→	*expr_query* [AS *attribute*]
from_item	→	*from_single*

SQL++ Expressions

Query ::= (Expression | SelectStatement) ";"

Expression ::= OperatorExpression | QuantifiedExpression

Operator Expression

OperatorExpression ::=
PathExpression | Operator OperatorExpression |
OperatorExpression Operator (OperatorExpression)? |
OperatorExpression <BETWEEN> OperatorExpression <AND> OperatorExpression

1. Arithmetic Operators, to perform basic mathematical operations;
2. Collection Operators, to evaluate expressions on collections or objects;
3. Comparison Operators, to compare two expressions;
4. Logical Operators, to combine operators using Boolean logic.

Quantified Expressions

QuantifiedExpression ::=
((<ANY>|<SOME>) | <EVERY>) Variable <IN> Expression ("," Variable "in" Expression)* <SATISFIES> Expression (<END>)?

A SQL++ Query

$\Gamma_1 = \langle$ readings: {{
```
  {co: 2.2},
  {co: 1.2, no2: [0.5, 2]},
  {co: 1.8, no2: 0.7} }},
 max: 2 ⟩
```

```
FROM    readings AS r
WHERE   r.co < max
ORDER BY r.no2
LIMIT 2
SELECT  l.co AS co
```

```
[0.7, [0.5, 2]]
```

$B^{out}{}_{FROM} = B^{in}{}_{WHERE} = $ {{
⟨r : {co: 2.2}⟩,
⟨r : {co:1.2, no2:[0.5, 2]}⟩,
⟨r : {co: 1.8, no2:0.7}⟩ }}

$B^{out}{}_{WHERE} = B^{in}{}_{ORDERBY} = $ {{
⟨r : {co:1.2, no2:[0.5, 2]}⟩,
⟨r : {co:1.8, no2:0.7}⟩
}}

$B^{out}{}_{ORDERBY} = B^{in}{}_{LIMIT} = $ [
⟨r : {co:1.8, no2: 0.7}⟩,
⟨r : {co:1.2, no2:[0.5, 2]}⟩]

$B^{out}{}_{LIMIT} = B^{in}{}_{SELECT} = $ [
⟨r : {co:1.8, no2:0.7}⟩,
⟨r : {co:1.2, no2:[0.5,2]}⟩]

Bindings From Tuple Variables to Element Variables

- Find the highest two sensor readings that are below 1.0

```
readings:
  [1.3, 0.7, 0.3, 0.8]
```

```
SELECT   r AS co
FROM     readings AS r
WHERE    r < 1.0
ORDER BY r DESC
LIMIT    2
```

```
[
 { co: 0.8 },
 { co: 0.7 }
]
```

`FROM readings AS r`

$$B^{out}_{FROM} = B^{in}_{WHERE} = \{\{ \langle r : 1.3 \rangle, \langle r : 0.7 \rangle, \langle r : 0.3 \rangle, \langle r : 0.8 \rangle \}\}$$

`WHERE r < 1.0`

$$B^{out}_{WHERE} = B^{in}_{ORDERBY} = \{\{ \langle r : 0.7 \rangle, \langle r : 0.3 \rangle, \langle r : 0.8 \rangle \}\}$$

`ORDER BY r DESC`

$$B^{out}_{ORDERBY} = B^{in}_{LIMIT} = [\langle r : 0.8 \rangle, \langle r : 0.7 \rangle, \langle r : 0.3 \rangle]$$

`LIMIT 2`

$$B^{out}_{LIMIT} = B^{in}_{SELECT} = [\langle r : 0.8 \rangle, \langle r : 0.7 \rangle]$$

`SELECT r AS co`

Backwards Compatibility with SQL

Find sensors that recorded a temperature below 50

```
readings : {{
  { sid: 2, temp: 70.1 },
  { sid: 2, temp: 49.2 },
  { sid: 1, temp: null }
}}
```

```
{{
   { sid : 2 }
}}
```

```
SELECT DISTINCT r.sid
FROM   readings AS r
WHERE  r.temp < 50
```

sid	temp
2	70.1
2	49.2
1	null

sid
2

Path Navigation

Two types path navigations

1. **Tuple path navigation** t.a from the tuple t to its **attribute a** returns the value of a
2. **Array path navigation** a[i] returns the **i-th element** of the array a

```
<r:{ ci: 1.2, no: [0.5, 2] }>
```

```
@tuple_nav {absent: missing, type_mismatch: null}
@array_nav {absent: missing, type_mismatch: null}

([r.co, r.so, 7.co, r.no[1], r.no[3], r.co[1]])
```

Oracle PL/SQL

A relational DBMS extended to support multi-model data

- Relational: SQL
- XML document: XML is a special data type and use XMLExists to replace the where clause
- Graph: SPARQL-in-SQL query
- RDF: SPARQL-in-SQL query

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX fn: <http://www.w3.org/2005/xpath-functions#>
PREFIX afn: <http://jena.hpl.hp.com/ARQ/function#>
SELECT (fn:upper-case(?object) as ?object1)
WHERE { ?subject dc:title ?object }

PREFIX fn: <http://www.w3.org/2005/xpath-functions#>
PREFIX afn: <http://jena.hpl.hp.com/ARQ/function#>
SELECT ?subject (afn:namespace(?object) as ?object1)
WHERE { ?subject <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?object }
```

Oracle PL/SQL Program

PL/SQL Block consists of three sections:

- The Declaration section (optional).
- The Execution section (mandatory)
  - SQL commands are embedded here
- The Exception Handling (or Error) section (optional)

```
DECLARE
   Variable declaration
BEGIN
   Program Execution
EXCEPTION
   Exception handling
END;
```

Query: get the salary of an employee with id '16' and display it on the screen

```
DECLARE
     var_salary number(6);
     var_emp_id number(6) = 16;
BEGIN
     SELECT salary
      INTO var_salary
      FROM employee
      WHERE emp_id = var_emp_id;
      dbms_output.put_line(var_salary);
      dbms_output.put_line('The employee ' || var_emp_id || ' has salary ' || var_salary);
END;
```

```
LOOP
   statements;
   EXIT;
   {or EXIT WHEN condition;}
END LOOP;
```

Oracle PL/SQL XQuery

PL/SQL using XQuery to query XML data

- SQL/XML functions **XMLQuery**, **XMLTable**, **XMLExists**, and **XMLCast** combine that power of expression and computation with the strengths of SQL

We can query relational data as XML using XMLQuery, but we can not join relational data and XML data in a single query together

```
DEFINE REGION = 'Asia'
SELECT XMLQuery('for $i in fn:collection("oradb:/HR/REGIONS"),
                 $j in fn:collection("oradb:/HR/COUNTRIES")
               where $i/ROW/REGION_ID = $j/ROW/REGION_ID
                   and $i/ROW/REGION_NAME = $regionname
                return $j'
               PASSING CAST('&REGION' AS VARCHAR2(40)) AS "regionname"
               RETURNING CONTENT) AS asian_countries
FROM DUAL;
```

MarkLogic Data Models

A document DBMS extended to support multi-model data

- XML, RDF, Full-text search

MarkLogic Data Models

- Triples in Documents

```
<?xml version="1.0" encoding="UTF-8"?>
<sem:triples xmlns:sem="http://marklogic.com/semantics">
  <sem:origin>file:///home/cgreer/source/sasquatch-data/dbp-tmp/person-sample-1.nt</sem:
  <sem:triple>
    <sem:subject>http://dbpedia.org/resource/David_Cronenberg</sem:subject>
    <sem:predicate>http://dbpedia.org/ontology/birthPlace</sem:predicate>
    <sem:object>http://dbpedia.org/resource/Toronto</sem:object>
  </sem:triple>
  <sem:triple>
    <sem:subject>http://dbpedia.org/resource/David_Cronenberg</sem:subject>
    <sem:predicate>http://dbpedia.org/ontology/birthDate</sem:predicate>
    <sem:object datatype=" http://www.w3.org/2001/XMLSchema#date ">1943-03-15</sem:object>
  </sem:triple>
  <sem:triple>
    <sem:subject>http://dbpedia.org/resource/David_Cronenberg</sem:subject>
    <sem:predicate>http://purl.org/dc/elements/1.1/description</sem:predicate>
    <sem:object xml:lang=" en ">Director</sem:object>
  </sem:triple>
```

- Extending Triples with Context

subject	predicate	object	doc ID	position
:person4	:first-name	"John"	11	5 - 9
:person5	:alma-mater	:Brown	4	25 - 40
:person5	:birth-year	1929	9	13 - 17

MarkLogic XQuery Queries

FLWOR expressions

- For, Let, Where, Order by, Return
- XPath expressions

"a FLWOR expression ... supports iteration and binding of variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents and for restructuring data."

Extracting subsets: XPath vs. FLWOR approach

- Get the title element for each recipe whose yield is greater than 20:

```
collection(‘recipeml/docs.xml’)/recipeml/ recipe/head/title[../yield > 20]
```

- Go through all the documents in the collection, and for any with a yield of more than 20, get the title:

```
for $doc in collection('recipeml/docs.xml')/recipeml
where $doc/recipe/head/yield > 20
return $doc/recipe/head/title
```

MarkLogic Querying Triples

Which person born in Brooklyn

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT ?person, ?name
WHERE { ?person onto:birthPlacedb:Brooklyn;
                foaf:name ?name .}
```

Which countries did Nixon visit?

```
sem:sparql("
      select ?country {
           <http://example.org/news/Nixon>
                 <http://example.org/wentTo> ?country
       } ",(),(),
cts:and-query( (
      cts:path-range-query("//sem:triple/@confidence",">",80) ,
      cts:path-range-query("//sem:triple/@date","<",xs:date("1974-
01-01")),
       cts:or-query( (
             cts:element-value-query(xs:QName("source"),"AP
Newswire"),
             cts:element-value-query(xs:QName("source"),"BBC")
             ) )
      ) )
)
```

ArangoDB Query Language AQL

A native multi-model DBMS that supports

- Graph
- Key-value
- Json

Doing queries with AQL

- Data retrieval with filtering, sorting and more
- Simple graph queries
- Traversing through a graph with different options
- Shortest path queries

SQL	AQL
database	database
table	collection
row	document
column	attribute
table joins	collection joins
primary key	primary key (automatically present on _key attribute)
index	index

ArangoDB AQL

- Selecting all rows / documents from a table / collection, with all columns / attributes

```
FOR user IN users
  RETURN user
```

- Filtering rows / documents from a table / collection, with projection

```
FOR user IN users
  FILTER user.active == 1
  RETURN {
    name: CONCAT(user.firstName,
" ",
                 user.lastName),
    gender: user.gender
  }
```

- Sorting rows / documents from a table / collection

```
FOR user IN users
  FILTER user.active == 1
  SORT user.name, user.gender
  RETURN user
```

AQL JOINS

Similar to joins in relational databases, ArangoDB has its own implementation of JOINS. Coming from an SQL background, you may find the AQL syntax very different from your expectations.

- Inner join can be expressed easily in AQL by nesting FOR loops and using FILTER statements:

```
FOR user IN users
  FOR friend IN friends
    FILTER friend.user ==
user._key
    RETURN MERGE(user, friend)
```

- Outer join: Outer joins are not directly supported in AQL, but can be implemented using subqueries:

```
FOR user IN users
  LET friends = (
    FOR friend IN friends
      FILTER friend.user == user._key
      RETURN friend
  )
  FOR friendToJoin IN (
     LENGTH(friends) > 0 ? friends :[ { /* no match exists
*/ } ]
   )
    RETURN { user: user, friend: friend
    }
```

AQL Graph Traversal

- **Traverse to the parents**

FOR v **IN** 1..1 **OUTBOUND** "Characters/2901776" ChildOf RETURN v.name

- **Traverse to the children**

FOR c **IN** Characters **FILTER** c.name == "Ned" FOR v IN 1..1 INBOUND c ChildOf RETURN v.name

- **Traverse to the grandchildren**

FOR c **IN** Characters **FILTER** c.name == "Tywin" **FOR** v IN 2..2 INBOUND c ChildOf RETURN v.name

- **Traverse with variable depth**

FOR c IN Characters FILTER c.name == "Joffrey" FOR v IN 1..2 OUTBOUND c ChildOf RETURN DISTINCT v.name

This FOR loop doesn’t iterate over a collection or an array, it walks the graph and iterates over the connected vertices it finds, with the vertex document assigned to a variable (here: v).

OrientDB

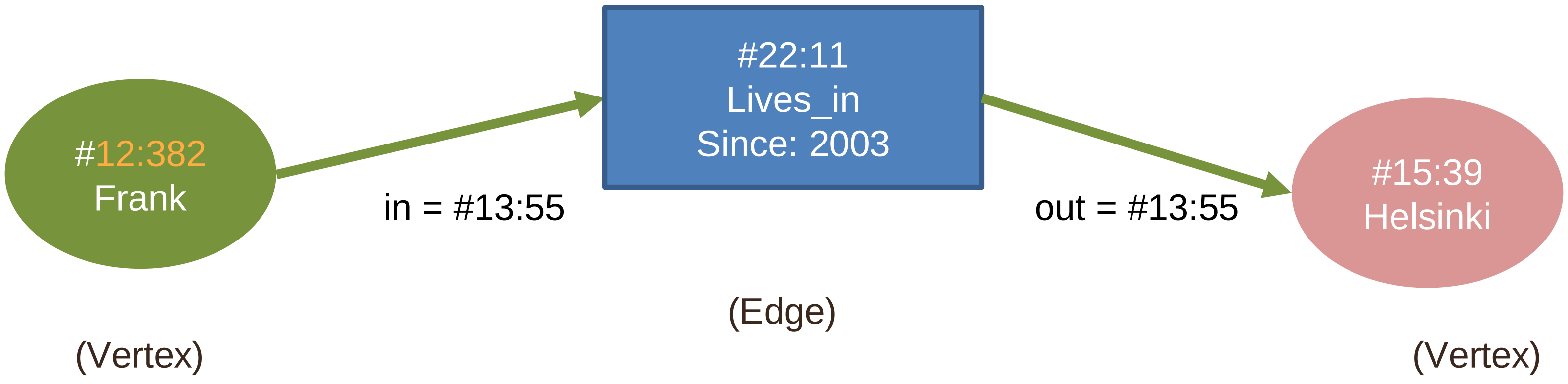
A Multi-Model Database

- Document, Graph, Spatial, FullText
- Tables -> Classes
- Extended SQL

Each element in the Graph has own immutable Record ID, such as #13:55, #22:11

Data models

Connections use persistent pointers



OrientDB: Data Model


```
{ “@rid”: “12:382”,
  “@class”: “Customer”,
  “name”: “Frank”,
  “surname” : “Raggio”,
  “phone” : “+358 0402678479”,
  “details”: {
      “city”:”London",
      “tags”:”millennial” }
}
```

Vertices and edges are JSON documents

- Schema-less
- Schema-full
- Schema-hybrid
- Nested documents
- Rich indexing and querying

OrientDB Query Language

OrientDB supports SQL as a query language with some differences

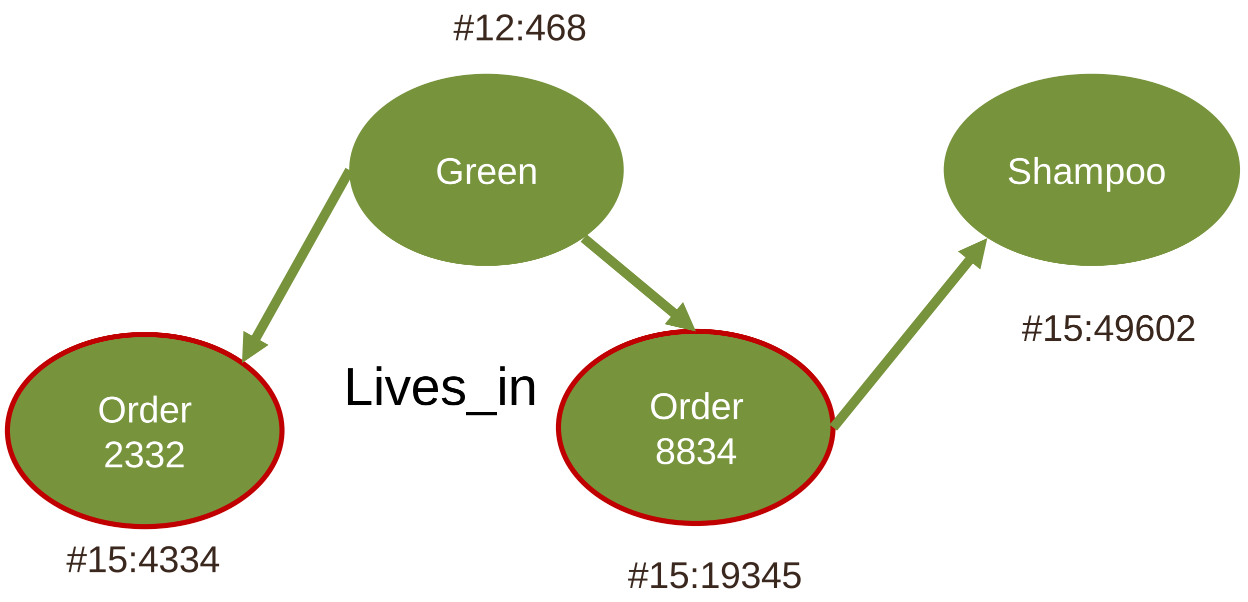
- **SELECT** city, sum(salary) AS salary
- **FROM** Employee
- **GROUP BY** city
- **HAVING salary** > 1000

Get all the outgoing vertices connected with edges with label (class) “Eats” and "Favourited" from all the Restaurant vertices in Rome

```
SELECT out('Eats', 'Favorited')
FROM Restaurant
WHERE city = 'Rome'
```

OrientDB: Graph Traversal

SELECT expand(out())
FROM #12:468



SELECT expand(out())
FROM Customer
WHERE name = ‘Green’

This uses an index to retrieve the starting vertex (#12:468) vertex

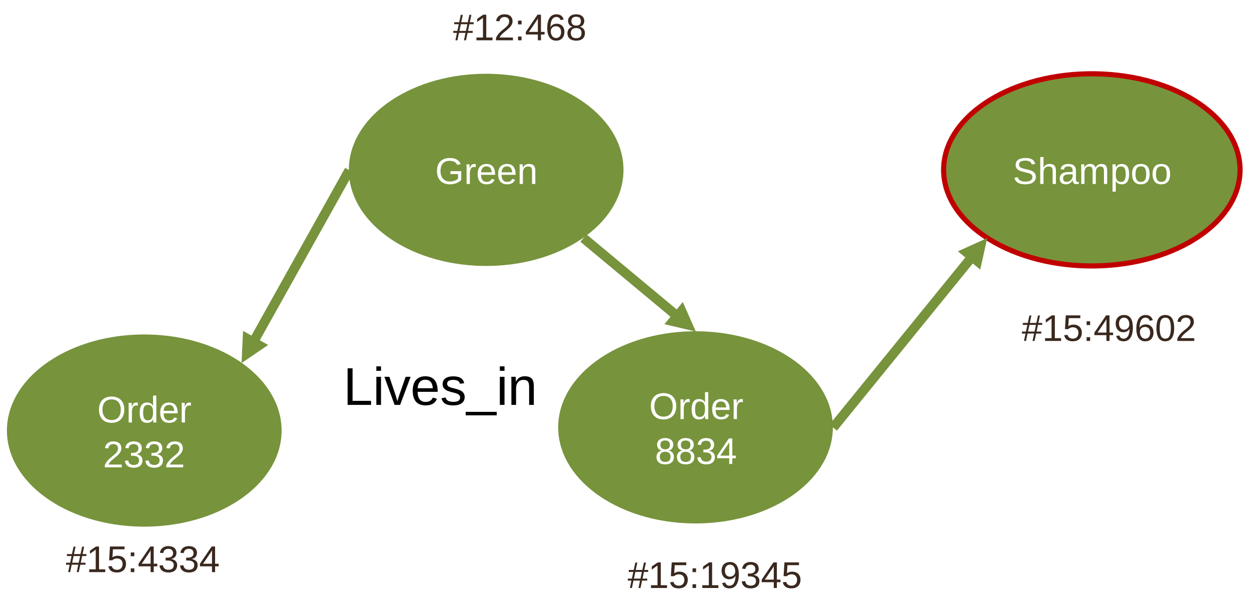
OrientDB: Graph Traversal

```
SELECT expand( out().out() )
FROM #12:468
```

```
SELECT expand( in().in() )
FROM #15:49602
```

```
SELECT expand( out().out() )
FROM Customer
WHERE name = 'Green'
```

```
SELECT expand( in().in() )
FROM Product
WHERE name = 'White Soap'
```



OrientDB Traverse and Pattern Matching

Traverse

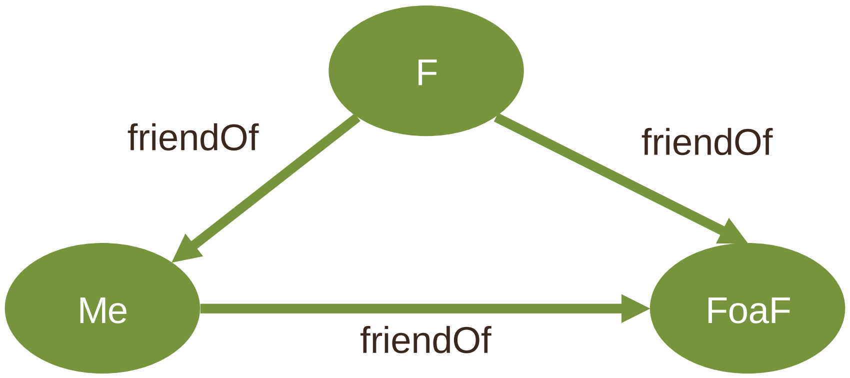
In a social network-like domain, a user profile is connected to friends through links.

- TRAVERSE out("Friend")
- FROM #10:1234 WHILE $depth <= 3
- STRATEGY BREADTH_FIRST

Pattern Matching

MATCH {class: Person, WHERE: (name = ‘Abel’), AS: me} -friendOf->{}-friendOf>{AS: foaf}, {AS: me}-friendOf->{AS: foaf}

RETURN me.name AS myName, foaf.name AS foafName



AgensGraph

A forked project of PostgreSQL (v9.6.2) supports

- Relational data, property graph, and JSON documents

Features

- Integrated querying using SQL (Relational data) and Cypher (Graph data)
- SQL for relational data and Cypher for Graph data
- JSON is a special data type
- Graph data object management
- Hierarchical graph label organization
- Property indexes on both vertexes and edges

AgensGraph Data Model

- Extended property graph model
- Data objects
  - Graph
  - Vertex and edge
  - Each vertex and edge can have a JSON document as its property
- **Label hierarchy**
  - Vertexes and edges can be grouped into labels (e.g. person, student, teacher, ...)
  - Labels are organized as a hierarchy

RPQ with AgensGraph

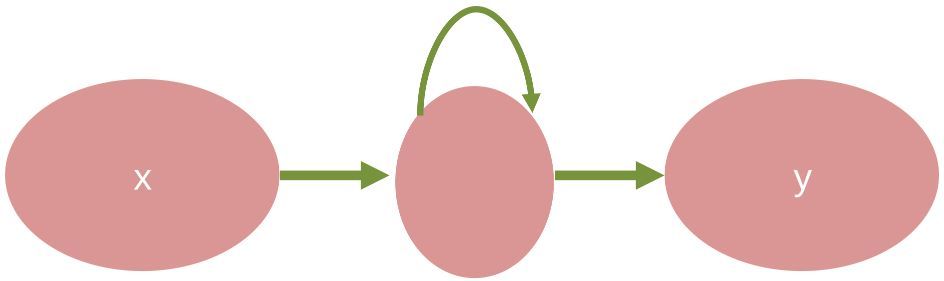
RPQ can be written as Variable-length Edge (VLE) Query

- Can be implemented using recursive common table expression (CTE) in SQL
- But CTE is inefficient for VLE query
  - Using CTE is BFS (Breadth First Search)-style processing
  - BFS processing needs to buffer intermediate results

VLE with Cypher

```
MATCH
    p=(x)-[:Parent*]->(y)
RETURN
    (x), (y), length(p)
ORDER BY (y), (x),length(p)
```

match (x)-[*1..5]->(y) return x, y;



Reference

- ArangoDB Query Language(AQL). https://www.arangodb.com/docs/stable/aql/index.html.
- C. Zhang and J. Lu. Holistic evaluation in multi-model databases benchmarking. Distributed and Parallel Databases, pages 1–33, 2019.
- C. Zhang, J. Lu, P. Xu, and Y. Chen. UniBench: A Benchmark for Multi-model Database Management Systems. In TPCTC ’18, Rio de Janeiro, Brazil, August 27-31, 2018, Revised Selected Papers, volume 11135 of Lecture Notes in Computer Science, pages 7–23. Springer, 2018.
- S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source BDMS. Proc. VLDB Endow., 7(14):1905–1916, 2014.
- R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, andD. Vrgoc. Foundations of modern query languages for graph databases. ACM Comput. Surv., 50(5):68:1–68:40, 2017.
- K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases. CoRR, abs/1405.3631, 2014.
- P. T. Wood. Query languages for graph databases. SIGMOD Rec., 41(1):50–60, 2012.

04 Comparison of the query languages

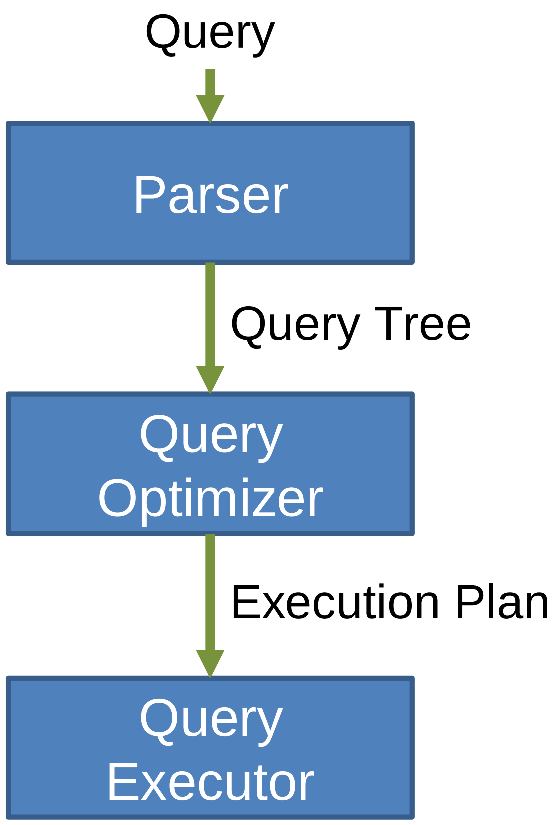
A comparative study of the query languages from 4 perspectives.

- The semantic difference
- The internal representations
- The expressive power
- The manner of query evaluation

Processing Paradigm

In general, the evaluation of a multi-model query consists of the following stages:

- The parser transforms the query into an **internal representation**, e.g., relational algebra expression for SQL;
- By using heuristic rules, the **optimizer** rewrites the expression into one that promises a more efficient evaluation;
- Different query evaluation plans are constructed for the optimized expression, (e.g., taking into account access paths for the data);
- The engine executes the evaluation plan and return results to the user.

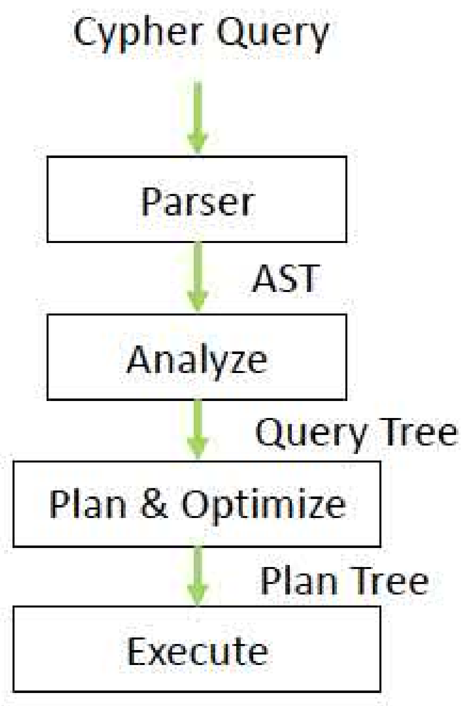


Query processing in AgensGraph

- Cypher query is processed by the same process with SQL
- We integrate Cypher query processing with SQL query engine from the parser to the executor
  - So you can use any PostgreSQL's expressions and functions in Cypher

- Cypher query's results is a relation
  - We treat Cypher query as a subquery
  - Existing query optimizations can be applied to Cypher query too(e.g. rolling up subquery, predicate push-down, join ordering, ...)

- Can make a query by combining SQL and Cypher as a subquery



Query processing in AgensGraph

- Cypher query is a chain of Cypher clauses
  - Each clause produces its results as a relation
- Chained execution
  - The results from the former clause are provided to the next clause
- Transform a Cypher query to a query tree
  - Each clause is transformed to a query structure
  - A MATCH clause is transformed to a query structure with joins
  - The chained clauses are combined as subqueries

Comparisons

Query languages can be compared W.R.T the following perspectives:

- **Semantics**: precisely defines the computation for each expression
- **Internal representation**: the internal representation of a parsed query
- **Expressive power**: what can and what cannot be expressed in a given query language?
- **Complexity of evaluation:** how complex is it to actually evaluate the queries expressible in the query language?
- **Complexity of static analysis**: how difficult is it to analyze and optimize queries to ensure a good evaluation performance?

Formal semantics for declarative query languages

```
SELECT * FROM ( SELECT  R.A, R.A   FROM  R )  S
```

- PostgreSQL outputs a table with two columns named “A”
- Oracle throws an ERROR: reference to column “A” is ambiguous

```
SELECT * FROM  R  WHERE EXISTS (
          SELECT *  FROM (
                    SELECT R.A, R.A  FROM  R )  S )
```

- Both PostgreSQL and Oracle output R

Who is right?

Let’s have a look at the SQL standard!

- If the <select list>* is simply contained in a <subquery> that is immediately contained in an <exists predicate>, then the <select list> is equivalent to a <value expression> that is an arbitrary <literal>.
- Otherwise, the <select list> * is equivalent to a <value expression> sequence in which each <value expression> is a column reference that references a column of T and each column of T is referenced exactly once. The columns are referenced in the ascending sequence of their ordinal position within T

The need of formal semantics for query languages

- Avoid ambiguity of natural language
- Clearly defined and not subject to interpretation
- Easy to understand and implement

Previous attempts

- Many simplifying assumptions: no bags, no nulls
- No justification of correctness

Three kinds of semantics

- Operational semantics
  - describing the meaning of a programming language by specifying how it executes on an abstract machine.
  - very helpful in implementation
  - Relational algebra
- Denotational semantics
  - defining the meaning of programming languages by mathematical concepts.
  - Relational calculus (Two-valued logic)
- Axiomatic semantics
  - giving the meaning of a programming construct by axioms or proof rules in a program logic
  - useful in developing and verifying programs

SQL++ Semantics: BNF Grammar for FROM Clause

SQL++ FROM clause allows variables to range over any data (SQL FROM clause tuple variables range over tuples only)

Semantics of SQL++ FROM are defined inductively

- (1) Lines 4-5: the SQL++ core base case where the FROM item ranges over a single collection or tuple
- (2) Lines 6-7: the SQL++ core inductive case where the FROM item comprises correlation between two other FROM items
- (3) Lines 8-12: the “syntactic sugar” cases, where the grammar introduces well known SQL constructs (e.g., joins, outer joins) as well as the unnesting constructs of NoSQL databases

```
1  from_clause
2    → FROM from_item
3  from_item
4    → expr_query AS var (AT var)?
5    | expr_query AS { var : var }
6    | from_item (INNER|LEFT OUTER?) CORRELATE? from_item
7    | from_item FULL OUTER? CORRELATE? from_item ON expr_query
8    | from_item , from_item
9    | from_item (INNER|LEFT|RIGHT|FULL) JOIN from_item
10         ON expr_query
11   | (INNER|OUTER) FLATTEN(expr_query AS var,
12         expr_query AS var)
13 from_param
14   → bag_order: (counter|null|missing|error)
15   | coerce_null_to_collection    : (singleton|empty|error)
16   | coerce_missing_to_collection: (singleton|empty|error)
17   | coerce_value_to_collection   : (singleton|error)
18   | coerce_null_to_tuple    : (empty|error)
19   | coerce_missing_to_tuple: (empty|error)
20   | no_match: (null|missing)
```

SQL++ Path Navigation Semantics

Semantics for **path navigations**(t:a and a[i])

Utilize parameters from the @tuple_nav
and array_nav parameter groups

- The absent parameter specifies the returned value when an attribute/array element is absent: null, missing, or throw an error.
- The type mismatch parameter specifies whether to return null, missing, or throw an error when a tuple/array navigation is invoked on a non-tuple/array.

$$t.a \rightarrow \begin{cases} v & \text{if } t \text{ is a tuple that maps } a \text{ to } v \\ \texttt{@tuple\_nav.absent} & \text{if } t \text{ is a tuple that does not map } a \\ \texttt{@tuple\_nav.type\_mismatch} & \text{otherwise} \end{cases}$$

$$a[i] \rightarrow \begin{cases} v & \text{if } a \text{ is an array with } i\text{-th element as } v \\ \texttt{@array\_nav.absent} & \text{if } a \text{ is an array with } n \text{ elements} \wedge (i < 1 \vee i > n) \\ \texttt{@array\_nav.type\_mismatch} & \text{otherwise} \end{cases}$$

Variable binding semantics: from Tuple Variables to Element Variables

- Find the highest two sensor readings that are below 1.0

```
readings:
  [1.3, 0.7, 0.3, 0.8]
```

```
SELECT    r AS co
FROM      readings AS r
WHERE     r < 1.0
ORDER BY  r DESC
LIMIT     2
```

```
[
 { co: 0.8 },
 { co: 0.7 }
]
```

```
FROM readings AS r
```

$$B^{out}_{FROM} = B^{in}_{WHERE} = \{\{ \langle r : 1.3 \rangle, \langle r : 0.7 \rangle, \langle r : 0.3 \rangle, \langle r : 0.8 \rangle \}\}$$

```
WHERE r < 1.0
```

$$B^{out}_{WHERE} = B^{in}_{ORDERBY} = \{\{ \langle r : 0.7 \rangle, \langle r : 0.3 \rangle, \langle r : 0.8 \rangle \}\}$$

```
ORDER BY r DESC
```

$$B^{out}_{ORDERBY} = B^{in}_{LIMIT} = [\langle r : 0.8 \rangle, \langle r : 0.7 \rangle, \langle r : 0.3 \rangle]$$

```
LIMIT 2
```

$$B^{out}_{LIMIT} = B^{in}_{SELECT} = [\langle r : 0.8 \rangle, \langle r : 0.7 \rangle]$$

```
SELECT r AS co
```

Semantics of SQL++ Values

BNF Grammar for SQL++ Values

- Missing value
- Defined value
  - scalar, complex or null
  - Complex: tuple, collection or map
- A collection is an array or a bag
- A map contains mappings of value pairs

value	→	*defined_value*
	|	**missing**
defined_value	→	[*id* ::] *scalar_value*
	|	[*id* ::] *complex_value*
	|	[*id* ::] **null**
complex_value	|	*tuple_value*
	|	*collection_value*
	|	*map_value*
scalar_value	→	*primitive_value*
	|	*enriched_value*
primitive_value	→	' *string* '
	|	*number*
	|	**true**
	|	**false**
enriched_value	→	*type* (*primitive_value* , ...)
tuple_value	→	{ *name* : *defined_value* , ...}
collection_value	→	*array_value*
	|	*bag_value*
array_value	→	[*value* , ...]
bag_value	→	{{ *value* , ...}}
map_value	→	**map**(*value* : *defined_value* , ...)

Semantics for RPQ

RPQ definition

The regular path queries are all and only those expressions recursively generated as follows.

- If $a \in L$, then $a \in RPQ$.
- If $e \in RPQ$, then $(e)^- \in RPQ$.
- If $e, f \in RPQ$, then $(e) / (f) \in RPQ$.
- If $e, f \in RPQ$, then $e+f \in RPQ$.
- If $e \in RPQ$, then $e^+ \in RPQ$.

Semantics

As a query algebra, RPQ allows us to: select all edges (i.e., paths of length 1) sharing an edge label, take the inverse of a set of paths, concatenate paths from two sets of paths, take the union of two sets of paths, and to take the transitive closure of a set of paths.

Let $G= (V, E, \eta, \lambda, v)$ be a property graph. The semantics of evaluating an RPQ $p \in RPQ$ over G is the set of vertex pairs $\langle p\rangle_G = V \times V$, recursively defined as follows.

- If $p=a \in L$, then $\langle p\rangle_G = \{s, t) \mid \exists edge \in E$ such that $\eta(edge) = (s, t)$ and $a \in \lambda(edge)\}$.
- If $p= (e)^- \in RPQ$, then $\langle p\rangle_G =\{(t, s) \mid (s, t) \in \langle e\rangle_G\}$.
- If $p= e/f \in RPQ$, then then $\langle p\rangle_G =\{(t, s) \mid \exists u \in V$ such that $(s,u) \in \langle e\rangle_G$ and $(u, t) \in \langle f\rangle_G$.
- If $p=e+f \in RPQ$, then $\langle p\rangle_G = \langle e\rangle_G + \langle f\rangle_G$
- If $g = (e)^+ \in RPQ$, then $\langle p\rangle_G = \{(s, t) \mid (s, t) \in TC(\langle e\rangle_G)\}$, where $TC(\langle e\rangle_G)$ denotes the transitive closure of binary relation $\langle e\rangle_G$

CRPQ Examples

- Pairs of customers who have bought same product (do not list a customer with herself):

Q1(c1,c2) :- c1 –Bought.^Bought-> c2, c1 != c2

- Customers who have bought and also reviewed a product:

Q2(c) :- c –Bought-> p, c –Reviewed-> p

CRPQ Semantics

- Naturally extended from single path expressions, following model of CQs

- Declarative
  - lifting the notion of satisfaction of a path expression atom by a source-target node pair to the notion of satisfaction of a conjunction of atoms by a tuple

- Procedural
  - based on SPRJ manipulation of the binary relations

  yielded by the individual path expression atoms

Internal representations

An algebra is always used as an internal representation to support query optimization (a set of equivalent rules):

1. SQL++: the nested relational algebra
2. Oracle PL/SQL: relational algebra
3. Marklogic XQuery: XQuery algebra,
4. ArangoDB AQL: no algebraic implementation
5. OrientDB: no defined algebra
6. AgensGraph: extend the relational algebra (PostgreSQL)

Expressive Powers

Three important expressive powers

- Conjunctive queries
  - Defined by Select, Project, Join algebra
- Relational completeness
  - Relational algebra, relational calculus
  - SQL92, AQL
- Turing completeness
  - Simulate the Turing machine
  - Oracle PL/SQL
  - Gremlin is the only one in graph languages

Recursions (Reference to their own)

- recursive Common Table Expressions (CTEs)
- SQL99

syntax of a recursive CTE:

```
WITH expression_name (column_list)
AS
(   -- Anchor member
   initial_query
   UNION ALL
   -- Recursive member that references
      expression_name.
   recursive_query
)
-- references expression name
SELECT *
FROM expression_name
```

CQ< SQL92 < ArangoDB QL, SQL++, AgensGraph, OrientDB < Oracle PL/SQL = MarkLogic XQuery

Query evaluation complexity

- How complex is it to actually evaluate the queries expressible in the language?
- There is a trade-off between the expressive power and evaluation complexity

Three types of complexity of evaluating a query

- Data complexity:
  - Make the query as a fixed entity and to measure the complexity in terms of the size of the database only.
- Query complexity :
  - Measure the cost in terms of the size of the query by assuming the database never changes
- Combined complexity
  - A general scenario both the database changes and many different queries are asked

Query evaluation complexity

- Standard complexity classes
  - LogSpace, Ptime, NP, Pspace, ExpTime

Two parallel complexity classes AC^0 and TC^0

- AC^0 :
  - the class of all problems solvable by uniform constant depth, polynomial size circuits with not , and and or gates of unbounded fan-in
- TC^0:
  - An analog of AC^0 where also threshold gates are available
- LogCFL
  - The class of all problems that are logspace-reducible to a context-free language

$AC^0 \subset TC^0 \subseteq \text{LogSpace} \subseteq \text{LogCFL} \subseteq \text{Ptime} \subseteq \text{NP} \subseteq \text{Pspace} \subseteq \text{ExpTime}$

Query evaluation complexity

- Theorem
  - The data complexity of Evaluation(CQ) is in AC^0
  - The combined complexity of Evaluation(CQ) is NP-complete
  - The combined complexity of Evaluation(Acyclic CQ) is LogCFL
  - The containment problem Evaluation(Acyclic CQ) is LogCFL

For the multi-model query languages

- AC^0 :
  - LogCFL ⊆ SQL++ ⊆ ArangoDB QL ⊆ AgensGraph ⊆ OrientDB ⊆ NP
  - NP ⊆ MarkLogic XQuery ⊆ Oracle PL/SQL ⊆ Pspace
  - SQL++ ⊆ ArangoDB QL ⊆ AgensGraph ⊆ OrientDB ⊆ Oracle PL/SQL ⊆ MarkLogic XQuery

Reference

- E. F. Codd. A Data Base Sublanguage Founded on the Relational Calculus. In Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET ’71, pages 35–68, New York, NY, USA, 1971. Association for Computing Machinery.
- E. F. Codd. A data base sublanguage founded on the relational calculus. In Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET ’71, pages 35–68, New York, NY, USA, 1971.
- E. F. Codd. Relational completeness of data base sublanguages. Research Report /RJ / IBM / San Jose, California, RJ987, 1972.
- J. Marton, G. Szárnyas, and D. Varró. Formalising openCypher Graph Queries in Relational Algebra. In Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings, volume 10509 of Lecture Notes in Computer Science, pages 182–196. Springer, 2017.
- V. Z. Moffitt and J. Stoyanovich. Temporal graph algebra. In Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017, pages 10:1–10:12. ACM, 2017.
- A. Mokhov. Algebraic graphs with class (functional pearl). In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017, pages 2–13. ACM, 2017.
- M. Negri, G. Pelagatti, and L. Sbattella. Formal Semantics of SQL Queries. ACM Trans. Database Syst., 16(3):513–534, Sept. 1991.
- K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases. CoRR, abs/1405.3631, 2014.
- M. A. Rodriguez. The Gremlin Graph Traversal Machine and Language. CoRR, abs/1508.03843, 2015.
- H. Thakkar, D. Punjani, S. Auer, and M. Vidal. Towards an integrated graph algebra for graph pattern matching with Gremlin. In Database and Expert Systems Applications - 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I, volume 10438 of Lecture Notes in Computer Science, pages 81–91. Springer, 2017.
- A. M. Turing. On computable numbers, with an application to the Entscheidungs problem. Proceedings of the London Mathematical Society, 2(42):230–265, 1936.

05 Open problem and challenges

Open problems and challenges in designing multi-model data query languages

- Design an algebra for a multi-model query language
- General approaches for cross-model query optimization

Defining a formal semantics for MMQL

Cross-model query involves many types of join operators

- Relation-Graph join
- Relation-JSON join
- Graph-Graph join
- Graph-JSON join
- …

How to define an algebra or logic?

- Typically we are to define an algebra or logic that capture the semantics for each join operation.

Cross-model query optimization

Many challenges in query evaluation: query optimization, query execution, self-tuning, data placement/migration

Cross-model query optimization

- Query-based vs. workload-based optimization
- View-based query rewriting
- Cost-based optimizations (cost model precisely capture the query cost)
- An algebra for query rewriting

Summary

- We discussed 6 representative multi-model data query languages
  - from essential syntax

- We have made a comparison of these query languages
  - from point of view of expressive power and semantics

- The existing multi-model data query languages is far beyond perfect
  - Both semantics and cross-model query evaluation

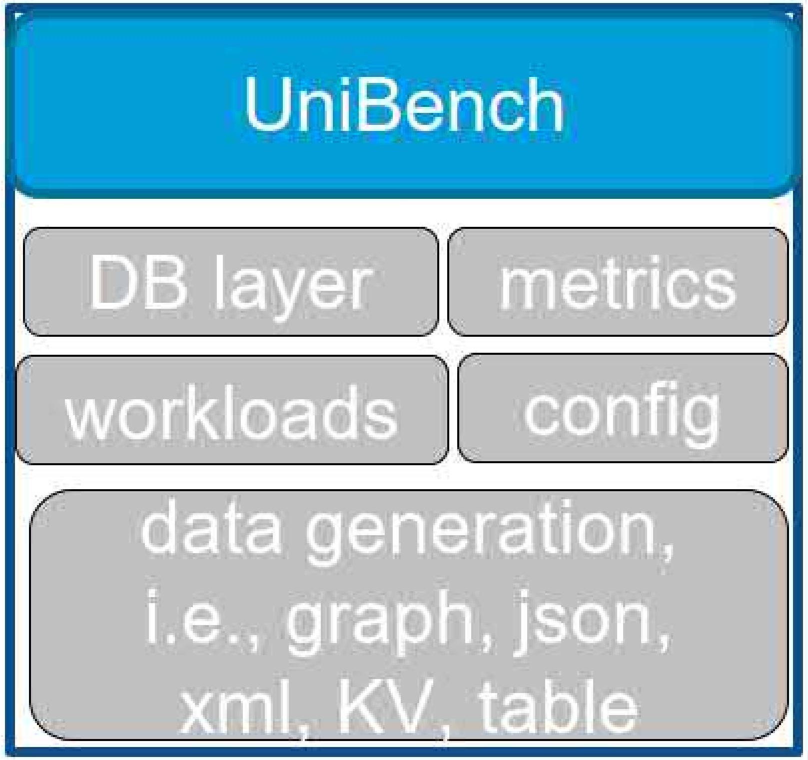
06 Hands-on section

We will invite the participants to learn, write and run some multi-model queries of UniBench by using a native multi-model database, ArangoDB (please install it in advance)

1. A brief introduction to UniBench and ArangoDB (5 mins)
2. Hands-on experience for multi-model queries with ArangoDB (20 mins)
3. Hands-on exercises for participants (10 mins)
4. Q&A (10 mins)

UniBench: A benchmark for multi-model databases

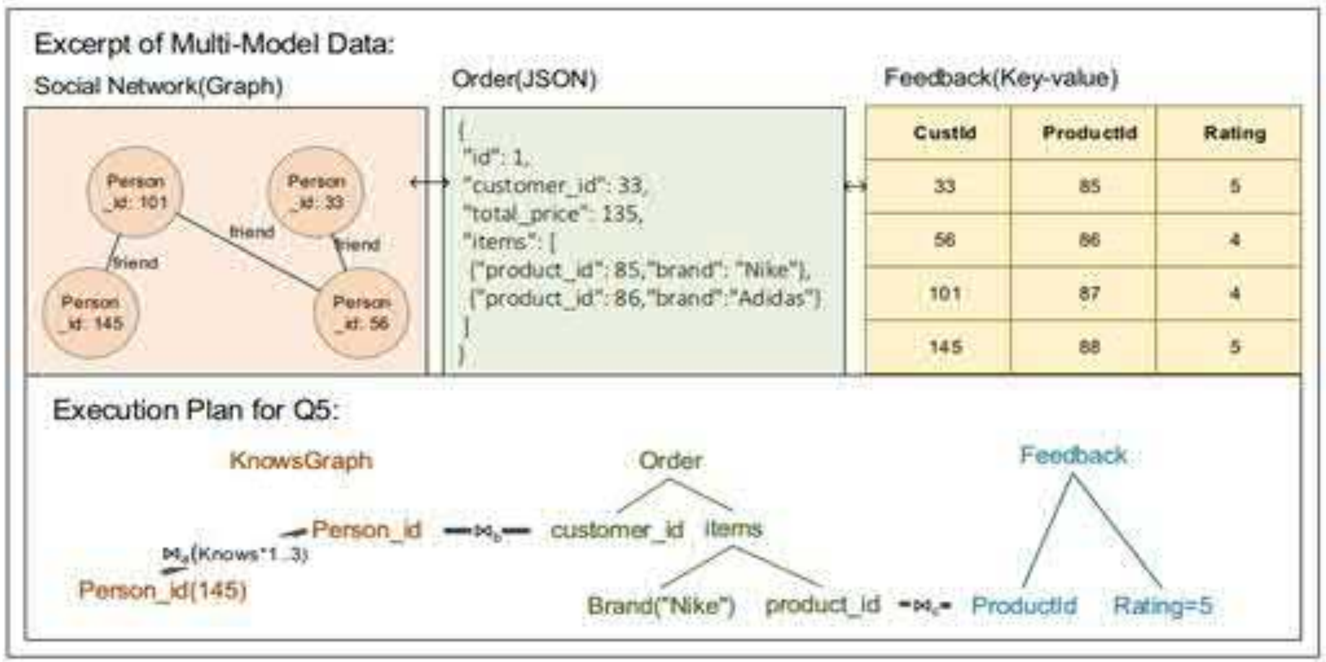
- A mixed data model: a scenario related to social network and e-commerce
- Multi-model data generation: scalable generation of 5 types of data
- Multi-model workloads: 10 multi-model queries, 2 multi-model transactions



Project website: https://www.helsinki.fi/en/researchgroups/unified-database-management-systems-udbms/unibench-towards-benchmarking-multi-model-dbms

A query example of UniBench

Given a start customer *c* and a product category *b*, find persons who are *c*'s friends within 3-hop friendships in Knows graph, return their bought products in the given category *b*, as well as the products' feedback with the 5-score rating.



ArangoDB

- Native multi-model NoSQL database (JSON, Key-value, and Property Graph, Spatial, Text), Schema-less
- Query language: AQL (For, Let, Filter, Return, FLFR expressions)
- ACID transaction and Auto Sharding
- Open source (Apache 2.0)

Link for the hands-on session:

https://version.helsinki.fi/chzhang/cikm-2020-hands-on-session-for-multi-model-queries/-/blob/master/hands-on.ipynb

Reference

- ArangoDB. https://www.arangodb.com/.
- ArangoDB Query Language(AQL). https://www.arangodb.com/docs/stable/aql/index.html.
- Chao Zhang and Jiaheng Lu. Holistic evaluation in multi-model databases benchmarking. Distributed and Parallel Databases, 2019.
- Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. UniBench: A Benchmark for Multi-model Database Management Systems. In TPCTC '18, Rio de Janeiro, Brazil, August 27-31, 2018, Revised Selected Papers, volume 11135 of Lecture Notes in Computer Science, pages 7–23. Springer, 2018.

THANKS

Does anyone have any questions?