Graph Normal Form

June 2022
Molham Aref on behalf of RelationalAI

Challenge - meeting people where they are…

Graphs + Navigational queries + Conceptual modelers are preferred by graph community **(LPG + triple stores)**

Tables + SQL + BI Tools are preferred by business analyst community

Tensors + Linear Algebra + Notebooks are preferred by data science and ML community

JSON + GraphQL + IDE/Editors are preferred by the developer community

Can we implement these abstractions as views on common internal representation? Can we have these abstractions **and** high performance?

Key Insight

The Table, Tensor, Graph, and JSON abstractions are just views on **Graph Normal Form** (aka 6NF + “things”) relational schema.

A **GNF** relation is a key plus at most one other value. It is irreducible.

Using **GNF** in traditional SQL RDBMS’s is performance suicide!

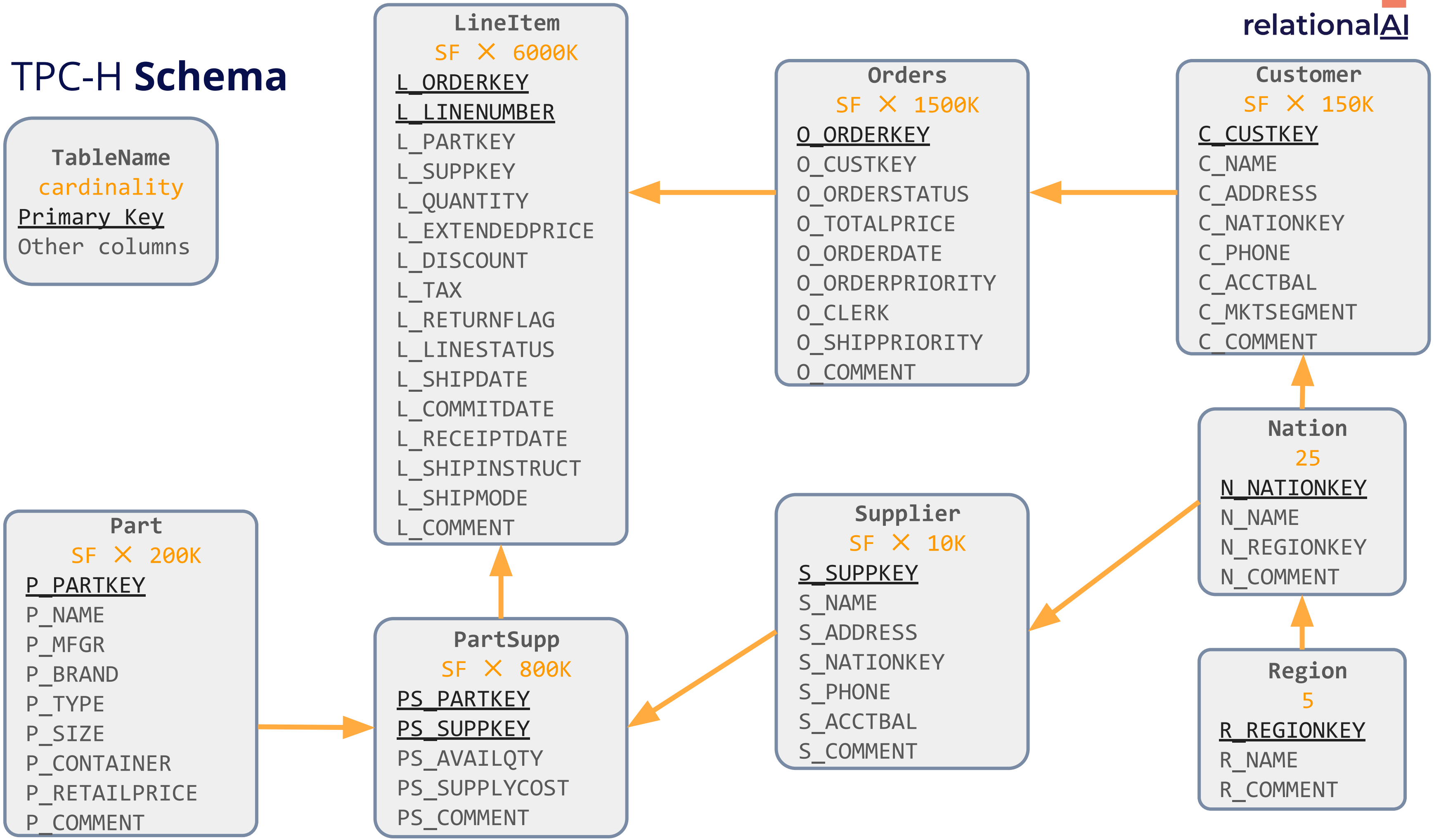
Recent advances in worst-case optimal joins and semantic optimization make it possible to support **GNF**.

relationalAI

Use Case: Business Intelligence

relationalAI

TPC-H **Schema**



TPC-H **Schema Mapping**

Graph normal form (GNF) decomposes relations to irreducible components.

For example, for the lineitem table, all the value columns become separate relations.

LineItem
L_ORDERKEY
L_LINENUMBER
L_PARTKEY
L_SUPPKEY
L_QUANTITY
L_EXTENDEDPRICE
L_DISCOUNT
L_TAX
L_RETURNFLAG
L_LINESTATUS
L_SHIPDATE
L_COMMITDATE
L_RECEIPTDATE
L_SHIPINSTRUCT
L_SHIPMODE
L_COMMENT

⋈

L_PARTKEY
L_ORDERKEY
L_LINENUMBER
L_PARTKEY

L_TAX
L_ORDERKEY
L_LINENUMBER
L_TAX

L_RECEIPTDATE
L_ORDERKEY
L_LINENUMBER
L_RECEIPTDATE

L_SUPPKEY
L_ORDERKEY
L_LINENUMBER
L_SUPPKEY

L_RETURNFLAG
L_ORDERKEY
L_LINENUMBER
L_RETURNFLAG

L_SHIPINSTRUCT
L_ORDERKEY
L_LINENUMBER
L_SHIPINSTRUCT

L_QUANTITY
L_ORDERKEY
L_LINENUMBER
L_QUANTITY

L_LINESTATUS
L_ORDERKEY
L_LINENUMBER
L_LINESTATUS

L_SHIPMODE
L_ORDERKEY
L_LINENUMBER
L_SHIPMODE

L_EXTENDEDPRICE
L_ORDERKEY
L_LINENUMBER
L_EXTENDEDPRICE

L_SHIPDATE
L_ORDERKEY
L_LINENUMBER
L_SHIPDATE

L_COMMENT
L_ORDERKEY
L_LINENUMBER
L_COMMENT

L_DISCOUNT
L_ORDERKEY
L_LINENUMBER
L_DISCOUNT

L_COMMITDATE
L_ORDERKEY
L_LINENUMBER
L_COMMITDATE

SQL

Tensor Notation

Q1-b

Total extended price

```
select
    sum(l_extendedprice)
from
    Lineitem
```

```
sum[extendedprice]



// The auto-generated RAI TPC-H schema
// uses the SQL column names

// As RAI supports types and overloading,
// it's not necessary to use Hungarian
// notation (i.e. the letter/underscore prefix)


// Names are easier to read without the
// Hungarian prefix so we’ll omit them here
```

SQL

Tensor Notation

(point-free)

Q1-d

Sum of all charges

```
select
     sum(l_extendedprice *
        (1 - l_discount) *
        (1 + l_tax))
from
     lineitem
```

```
sum[extendedprice *
     (1 - discount) *
     (1 + tax)
]
```

SQL

Tensor Notation
(point-wise)

Q1-d

Sum of all charges

```
select
     sum(l_extendedprice *
        (1 - l_discount) *
        (1 + l_tax))
from
     lineitem
```

```
sum[extendedprice[o, num] *
     (1 - discount[o, num]) *
     (1 + tax[o, num])
     for o, num
]
```

SQL

Q1-d

Sum of all charges

```
select
    sum(l_extendedprice *
        (1 - l_discount) *
        (1 + l_tax))
from
    lineitem
```

Tensor Notation

(composition)

```
def result = sum[charge]

def charge =
    extendedprice *
    (1 - discount) *
    (1 + tax)
```

SQL

Relational Notation

Q5-u

All customers in Asia (Join)

```
select    c_custkey
from      customer, nation, region
where     c_nationkey = n_nationkey and
          n_regionkey = r_regionkey and
          r_name = 'ASIA'
```

```
def result(c) =
    nationkey(c, n) and
    regionkey(n, r) and
    name[r] = "ASIA"
    forany n, r
```

SQL

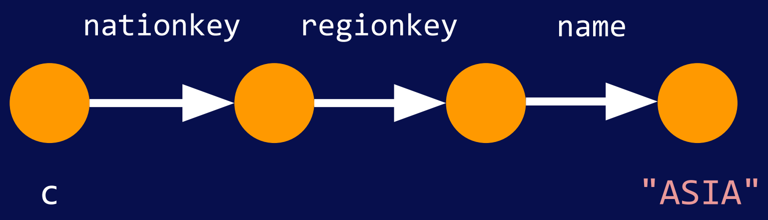
Navigational Notation

Q5-u

All customers in Asia (Join)

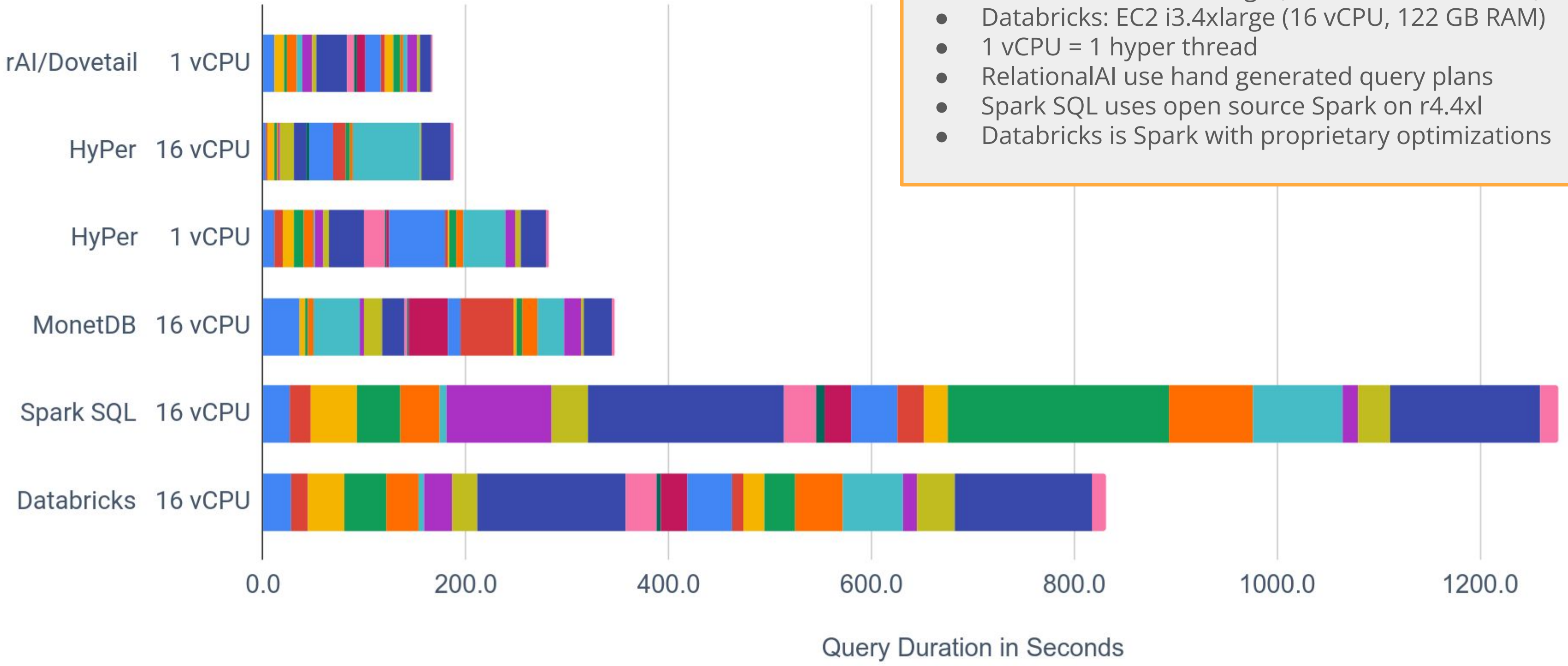
```
select    c_custkey
from      customer, nation, region
where     c_nationkey = n_nationkey and
          n_regionkey = r_regionkey and
          r_name = 'ASIA'
```

```
def result(c) =
    c.nationkey.regionkey.name = "ASIA"
```



relationalAI

TPC-H Stacked Query Duration SF100

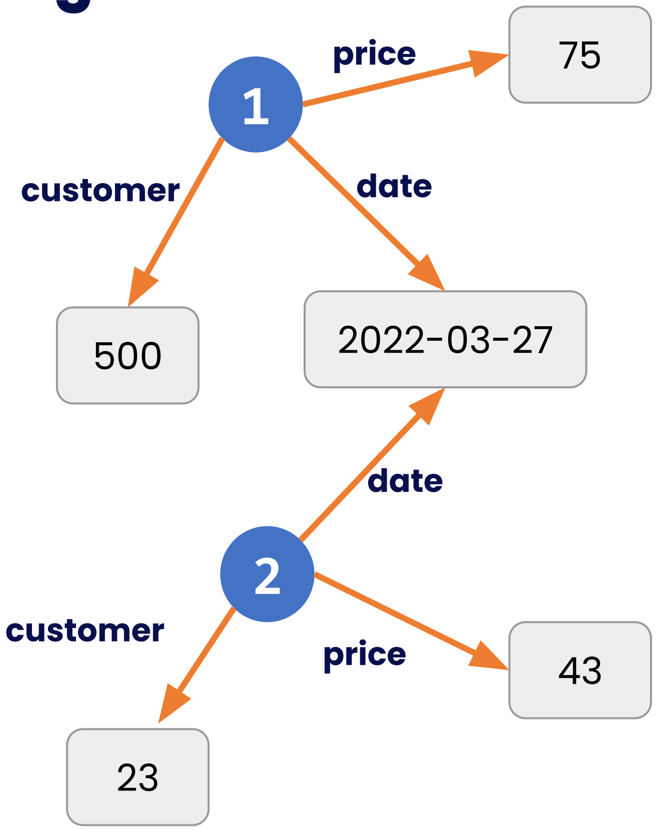


- SF100 ≈ 100GB of CSV data
- Hardware: EC2 r4.4xlarge (16 vCPU, 122 GB RAM)
- Databricks: EC2 i3.4xlarge (16 vCPU, 122 GB RAM)
- 1 vCPU = 1 hyper thread
- RelationalAI use hand generated query plans
- Spark SQL uses open source Spark on r4.4xl
- Databricks is Spark with proprietary optimizations

Tables as a Collection of (Hyper)Edge Relations

orderkey	customer	date	price
1	500	2022-03-27	75
2	23	2022-03-27	43

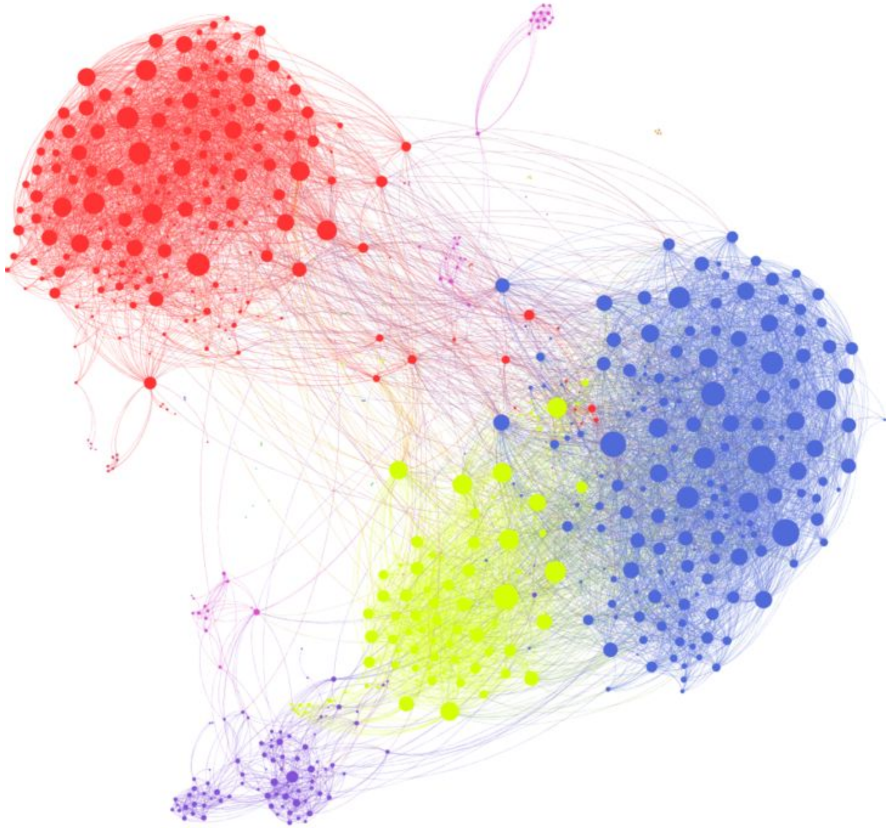
```
customer(1, 500)
customer(2, 23)

date(1, 2022-03-27)
date(2, 2022-03-27)

price(1, 75)
price(2, 43)
```



SQL tables are in a sense a modularity construct, grouping relations with the same primary key.

relationalAI

Use Case: Graph Intelligence



Challenge

Business Intelligence was easy, but how about Graph Intelligence?

The good news is we can express (hyper-)graph use cases using an “edge” relation and:

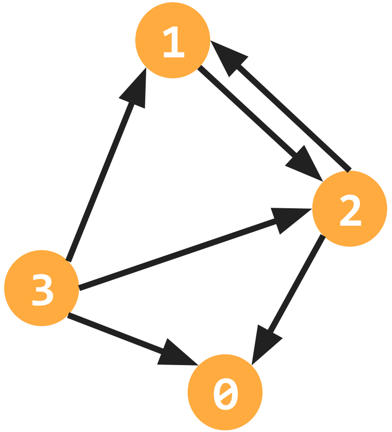
- Self-joins
- Aggregation
- Recursion (through aggregation)

Using self-joins and recursion in traditional SQL RDBMS’s is performance suicide!

Degree Query

SQL	`SELECT source AS id, COUNT(*)`
	`FROM edge`
	`GROUP BY id`

Spark Dataframes	`result = edges.groupBy("src").agg(count("*"))`
Spark GraphFrames	`g = GraphFrame(nodes, edges)` `result = g.outDegrees`
Neo4J Cypher	`MATCH (n:node)-[r]->()` `RETURN n.id, COUNT(DISTINCT r) as degree`
Tensor Notation	`def degree[x] = count[edge[x]]`



Sample graph where every node is labelled with its degree: the number of outgoing edges for that node.

Triangle Count for Entire Graph

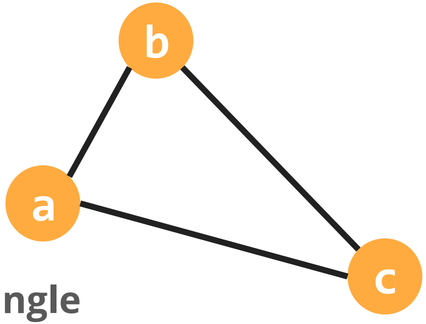
Neo4J Cypher	`MATCH( (a:node)-[:POINTSTO]->(b:node) )`
	`MATCH( (b:node)-[:POINTSTO]->(c:node) )`
	`MATCH( (a:node)-[:POINTSTO]->(c:node) )`
	`WHERE a.id < b.id < c.id`
	`RETURN COUNT(*);`

SQL	`SELECT COUNT(*)`
	`FROM edge e1, edge e2, edge e3`
	`WHERE`
	`  e1.source = e2.source AND`
	`  e1.dest   = e3.source AND`
	`  e2.dest   = e3.dest   AND`
	`  e1.source < e3.source AND`
	`  e3.source < e2.dest`

Relational Notation	`def distinct_triangle(a, b, c) =`
	`    edge(a, b) and`
	`    edge(a, c) and`
	`    edge(b, c) and`
	`    a < b and b < c`
	
	`def result = count[distinct_triangle]`

Triangle count is one of the most studied graph analytical queries. One of its uses is to compute the clustering coefficient, which is a useful descriptive statistics of a graph.

Triangle count has been applied for spam detection, and in random graph models.



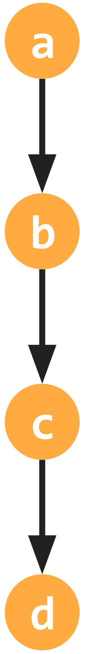
Triangle pattern

relationalAI

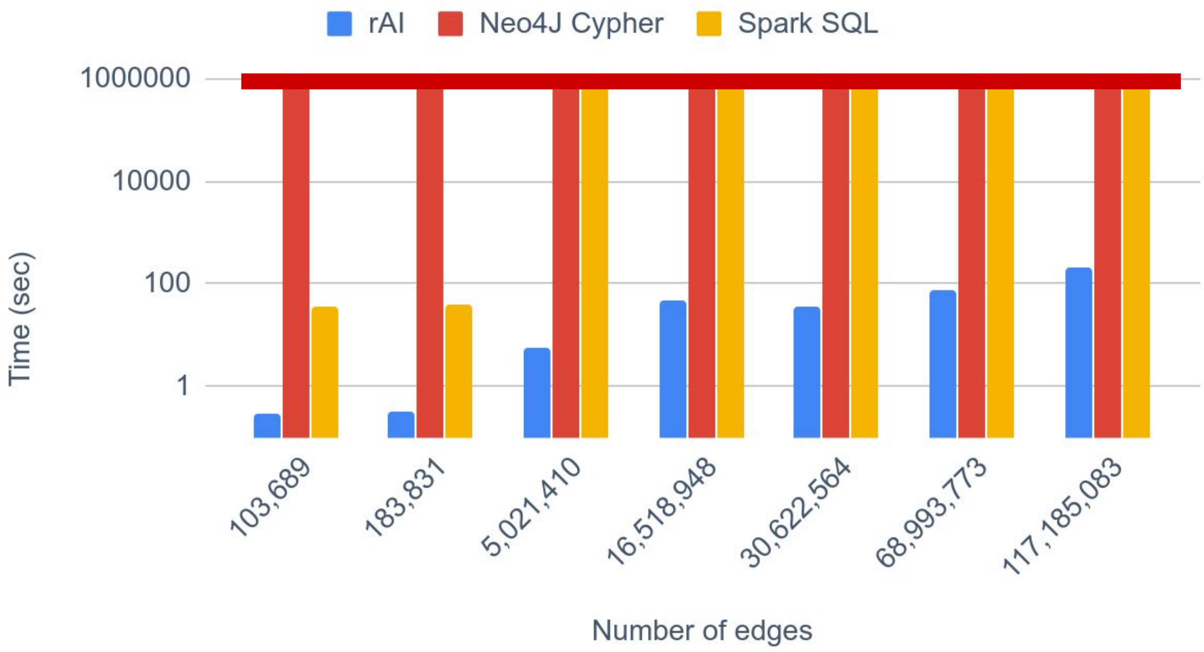
Path Count per Node (3 hops)

<table>
<tr><td>Neo4J Cypher</td><td><pre>MATCH((a:node)-[:POINTSTO]->(b:node))
MATCH((b:node)-[:POINTSTO]->(c:node))
MATCH((c:node)-[:POINTSTO]->(d:node))
RETURN a.id, COUNT(*);</pre></td></tr>
<tr><td>SQL</td><td><pre>SELECT e1.source, COUNT(*)
FROM edge e1, edge e2, edge e3
WHERE
  e1.dest = e2.source AND
  e2.dest = e3.source AND
GROUP BY e1.source</pre></td></tr>
<tr><td>Tensor Notation</td><td><pre>def path3(a, b, c, d) =
    edge(a, b) and
    edge(b, c) and
    edge(c, d)

def result[a] = count[path3[a]]</pre></td></tr>
</table>



Results: Path Count per Node (3 hops)

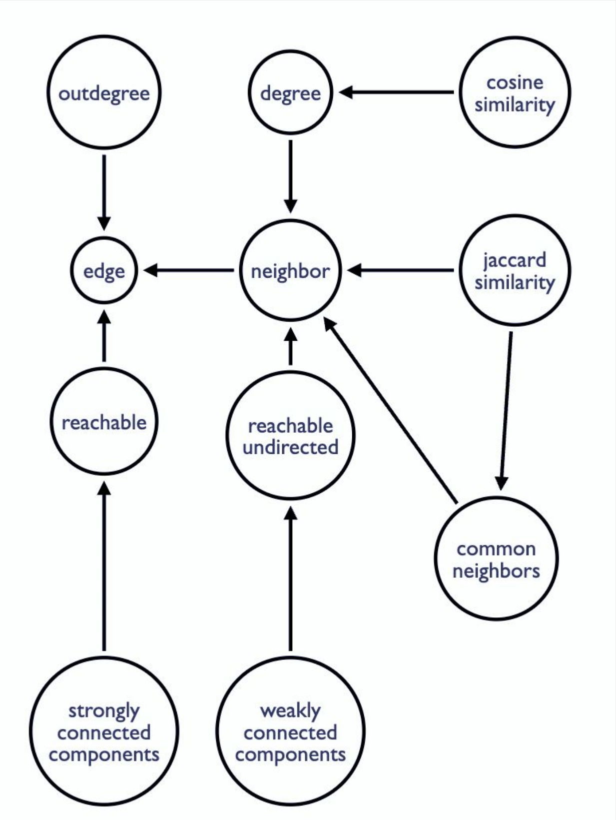


Graph Analytics

No explicit syntax for graphs

```
module graph_analytics[G]
  with G use node, edge

  def neighbor(x, y) = edge(x, y) or edge(y, x)
  def outdegree[x] = count[edge[x]]
  def degree[x] = count[neighbor[x]]
  def cn[x, y] = count[intersect[neighbor[x], neighbor[y]]]    // Count of Common Neighbors

  def reachable = edge; reachable.edge                                       // Recursive!
  def reachable_undirected = neighbor; reachable_undirected.neighbor        // Recursive!

  def scc[x] = min[v: reachable(x, v) and reachable(v, x)]      // Strongly Connected Component
  def wcc[x] = min[reachable_undirected[x]]                     // Weakly Connected Component

  def cosine_sim[x, y]  = cn[x, y] / sqrt[degree[x] * degree[y]]
  def jaccard_sim[x, y] = cn[x, y] / count[neighbor[x]] + count[neighbor[y]] - cn[x, y]
  …
end
```

Dependencies



Graph Analytics

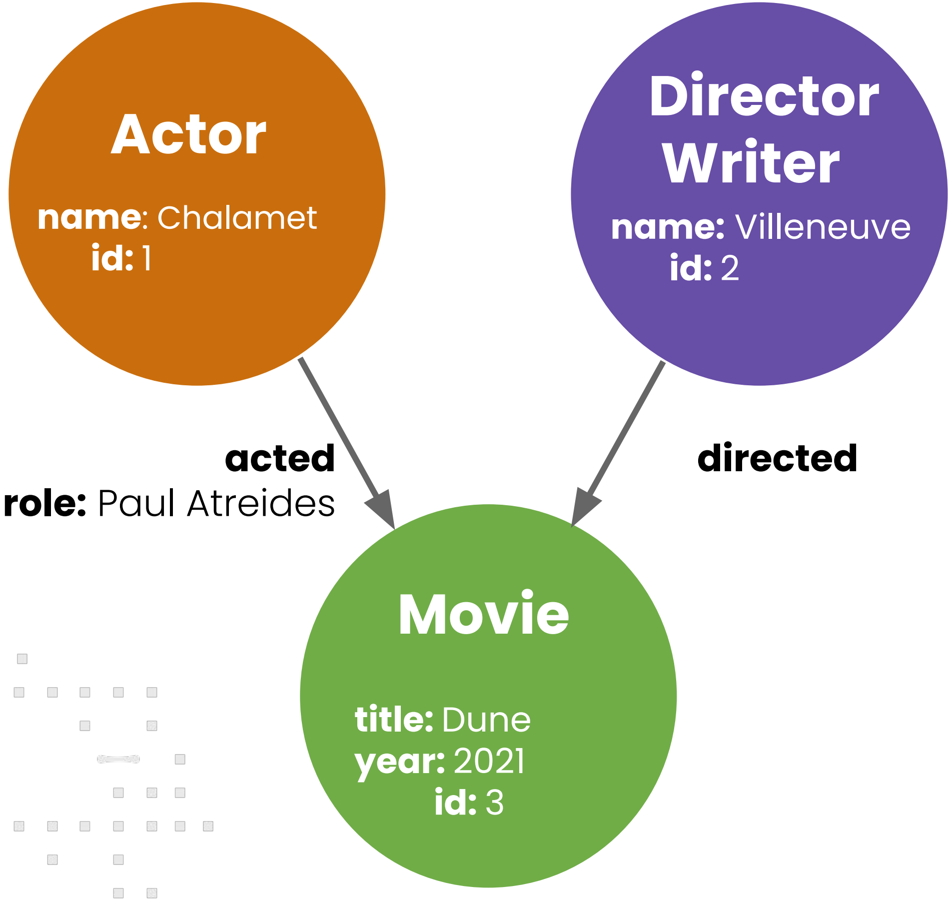
From the definition of edge, we build neighbor, from there we can build reachable undirected and that gives us the ability to build weakly connected components.

From neighbor we can build common neighbors and then jaccard similarity which depends on both.

```
module graph_analytics[G]
  with G use node, edge

  def neighbor(x, y) = edge(x, y) or edge(y, x)
  def outdegree[x] = count[edge[x]]
  def degree[x] = count[neighbor[x]]
  def cn[x, y] = count[intersect[neighbor[x], neighbor[y]]]

  def reachable = edge; reachable.edge

  def reachable_undirected = neighbor; reachable_undirected.neighbor

  def scc[x] = min[v: reachable(x, v) and reachable(v, x)]
  def wcc[x] = min[reachable_undirected[x]]

  def cosine_sim[x, y]  = cn[x, y] / sqrt[degree[x] * degree[y]]
  def jaccard_sim[x, y] = cn[x, y] / count[neighbor[x]] + count[neighbor[y]] - cn[x, y]
  …
end
```

RelationalAI

Labelled Property Graphs as Relational Graphs




```
movie(3)
title(3, "Dune")
year(3, 2021)

director(2)
writer(2)
name(2, "Villeneuve")

directed(2, 3)

actor(1)
name(1, "Chalamet")

acted(1, 3)
role(1, 3, "Paul Atreides")
```

Conclusion

We can have relational representation of graphs in a system with...

- **Indexes and index organized relations**
  - To store adjacency lists
- **Materialized views based on the full-query language**
  - To store precomputed links between nodes (e.g. c.nation.region.name)
- **Worst-case optimal multi-way join algorithms**
  - For efficient evaluation of queries with many joins (like the kind you would seen with in GNF schema)
  - For self-joins
- **Semantic query optimizer**
  - To take advantage of graph structure to eliminate exponential amount of redundant work
  - To speedup aggregations
  - To take advantage of materialized views
- **Recursion (implemented with double differencing and demand transformation)**
  - To optimize fixpoint queries
- **Higher order syntax**
  - To quantify over relation names
  - To support property graph and triple-store abstractions (the latter is a view on the former)

relationalAI

Conclusion (cont.)

For the first time we can have a **relational** graph management system that supports

- expressive **reasoning**
- **hyper graphs**
- **temporal** features
- **performance**: JIT, Worst-case optimal joins, semantic query optimization
- **scalability**: Cloud-native (i.e. separation of compute & storage)
- **derived** and **materialized views**
- **streaming** support with expressive incremental view maintenance
- **versioning**
- **integrity** constraints
- **BI** - with SQL/Table abstraction
- **(Auto)ML** - with LA/Tensor abstraction

relationalAI

Use Case: Linear Algebra

relationalAI

Challenge

How about linear algebra? Can we handle sparse and dense use cases?

Again, the good news is that we can express Linear Algebra operations using:

- Joins
- Aggregation
- Recursion

Using joins and recursion in traditional SQL RDBMS's is performance suicide!

Tensor Notation for TPC-H Schema

lineitem.csv

```
1 1552 93 1 17 24710.35 0.04 0.02 N O 1996-03-13 1996-02-12 1996-03-22 DELIVER IN PERSON TRUCK
1  674 75 2 36 56688.12 0.09 0.06 N O 1996-04-12 1996-02-28 1996-04-20 TAKE BACK RETURN  MAIL
1  637 38 3  8 12301.04 0.10 0.02 N O 1996-01-29 1996-03-05 1996-01-31 TAKE BACK RETURN  REG AIR
...
```

SQL Table

LineItem
- <u>L_ORDERKEY</u>
- <u>L_LINENUMBER</u>
- L_PARTKEY
- L_SUPPKEY
- L_QUANTITY
- L_EXTENDEDPRICE
- L_DISCOUNT
- L_TAX
- L_RETURNFLAG
- L_LINESTATUS
- L_SHIPDATE
- L_COMMITDATE
- L_RECEIPTDATE
- L_SHIPINSTRUCT
- L_SHIPMODE
- L_COMMENT

Tensor Notation

...

l_extendedprice[o, num]

```
1 1552  24710.35
1  674  56688.12
1  637  12301.04
...
```

...

l_shipdate[o, num]

```
1 1552 1996-03-13
1  674 1996-04-12
1  637 1996-01-29
...
```

...

l_shipmode[o, num]

```
1 1552  TRUCK
1  674  MAIL
1  637  REG AIR
...
```

...

Tensors as Relations

vector

$$\begin{bmatrix} 4 \\ 1 \\ 8 \end{bmatrix}$$

```
(1, 4)
(2, 1)
(3, 8)
```

binary relation

matrix

$$\begin{bmatrix} -1.3 & 0.6 \\ 20.4 & 5.5 \\ 9.7 & -6.2 \end{bmatrix}$$

```
(1, 1, -1.3)
(1, 2,  0.6)
(2, 1, 20.4)
(2, 2,  5.5)
(3, 1,  9.7)
(3, 2, -6.2)
```

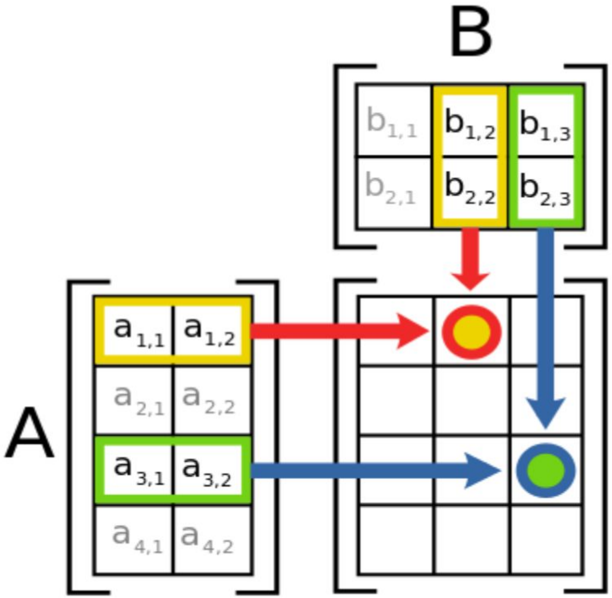
ternary relation

A relational database system that is effective for tensors would be an outstanding proof-point for the relational model.

(and imagine the data management benefits this would have for ML systems!)

Tensors as Relations: Matrix Multiplication

[Deep Learning with Relations at NeurIPS](#)



Matrix multiplication diagram.svg, CC BY-SA 3.0, User:Bilou

Math

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Rel *Our new relational language*

```
def C[i, j] = sum[k: A[i, k] * B[k, j]]
```

SQL

```
SELECT A.row, B.col, SUM(A.val * B.val)
FROM A INNER JOIN B ON A.col = B.row
GROUP BY A.row, B.col
```

relationalAI

Use Case: JSON & semi-structured data


```
{
  "first_name": "John",
  "last_name": "Smith",
  "address": { "city": "Seattle",
               "state": "WA" },
  "phone": [
    { "type": "home",
      "number": "206-456" },
    { "type": "work",
      "number": "206-123" }
  ]
}
```

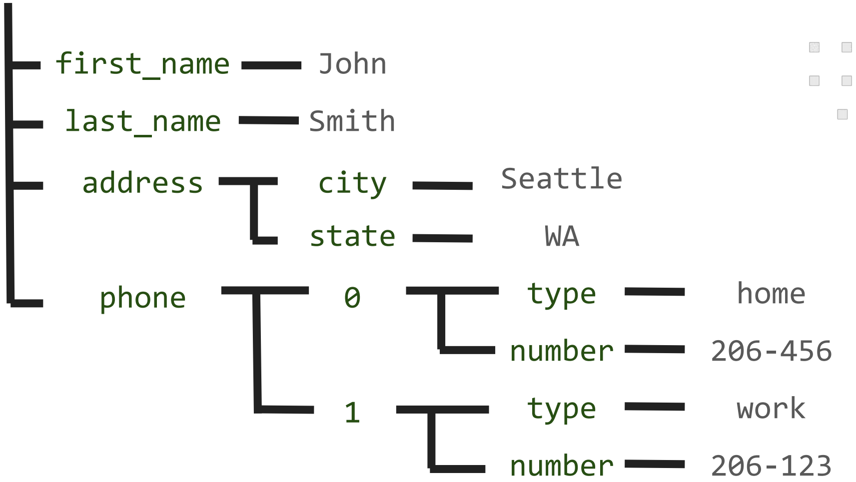
relationalAI

Relational Representation of JSON

We can represent JSON with first-order relations in graph normal form

After parsing, JSON is typically represented as a tree (right)

```
{
  "first_name": "John",
  "last_name": "Smith",
  "address": { "city": "Seattle",
               "state": "WA" },
  "phone": [
    { "type": "home",
      "number": "206-456" },
    { "type": "work",
      "number": "206-123" }
  ]
}
```

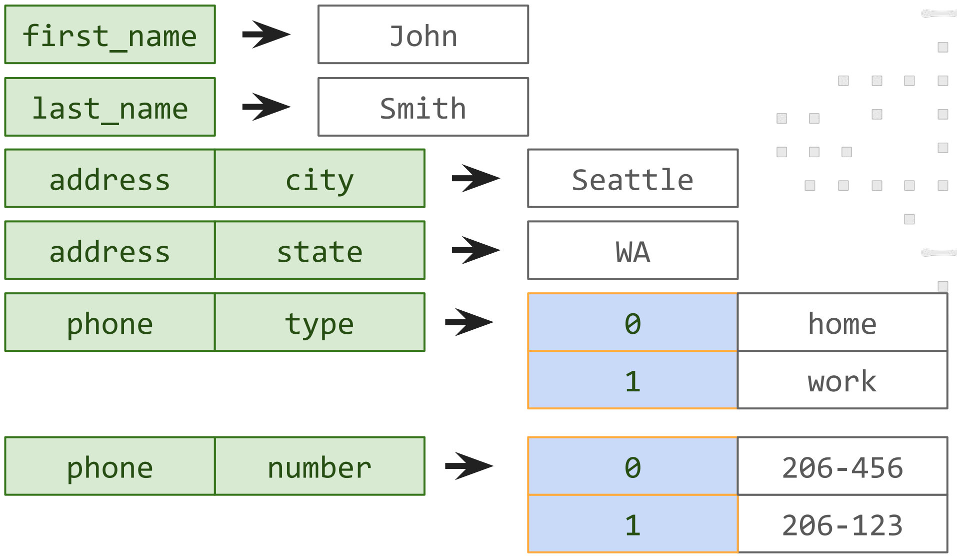


Relational Representation of JSON

Next, we organize the data by the path abstraction.

This is a relational representation of JSON

```
{
  "first_name": "John",
  "last_name": "Smith",
  "address": { "city": "Seattle",
               "state": "WA" },
  "phone": [
    { "type": "home",
      "number": "206-456" },
    { "type": "work",
      "number": "206-123" }
  ]
}
```



JSON on GNF benefits

Complete and Efficient Array Support

- GNF makes it possible to support arbitrary nested usage of arrays efficiently.

No Schema Inference and Inefficient Handling of `Erroneous` Data

- Relations can efficiently be overloaded by type (as opposed to a boxing type), so for JSON there is no need to infer a schema. All data is stored equally efficiently

Import+Query as well as Construct+Export

- Because a JSON document is a GNF relation, the same representation can also be constructed and exported as a JSON document. Import followed by export results in logically identical documents.

No special constructs in Query Language

- Because a JSON document is a relation, there is no need for constructs that mix relational and nested data. A document and subdocuments can be passed as arguments to abstractions.

GNF lets us support domain specific syntax

Rel - for relational and tensor dialects (see docs.relational.ai)

SQL - preliminary support using DuckDB

Legend - preliminary support via direct transpilation

GraphQL - TBD

SPARQL - TBD

GQL - TBD

…

GNF lets us support domain specific syntax

Time-series abstraction is easily expressed in GNF databases (special case of vector/tensor)

So is **functional programming** (pointwise and point-free)

So are **diagrammatic languages** (e.g. conceptual modeling in ORM - see appendix)

Mapping is left as “exercise to the reader”

GNF lets us support domain specific syntax. What else?

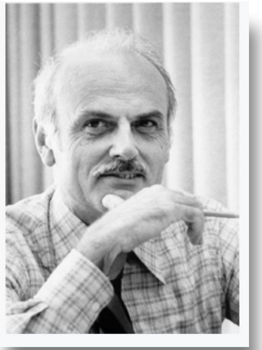
- **Eliminates the need for nulls** and multi-valued logics [Hoare's "billion dollar mistake"][Date][Libkin].
- **Supports DML**, i.e. insert, update, upsert, delete —> incrementally maintained materialized views
- **Improves semantic stability** by making the addition or removal of schema information easier as the application evolves (also schema on demand)
- **Improves analytic query performance** of queries that involve a smaller number of attributes than would normally exist in a wide table. The low information entropy of normalized tables allows compression schemes and efficiency approaching that of column stores
- **Supports temporal features** like transaction time and valid time for each piece of information in the database

That's a lot of abstraction goodness that we've been too scared to use because of fear of the performance hit of binary joins and incomplete query optimization

Appendix

RelationalAI

The Essence of the Relational Model



Information Retrieval P. BAXENDALE, Editor

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

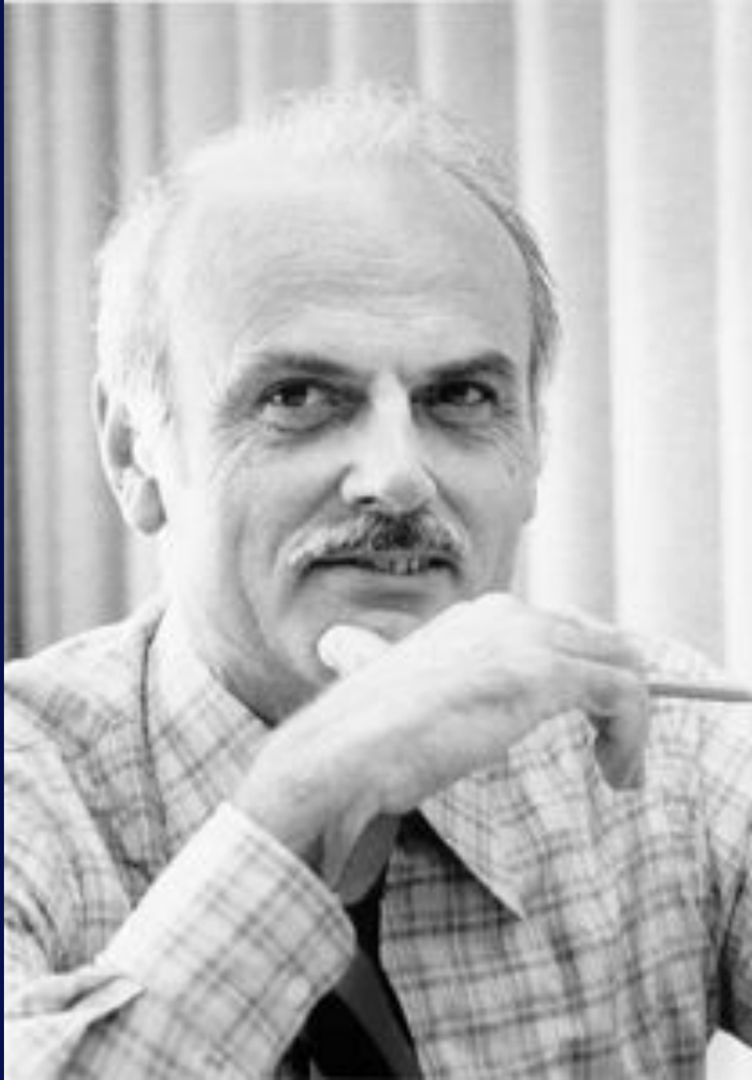
Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model.

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity
CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

Have relational database systems been sufficiently ambitious on this point?

Most people have never used a Relational Database




Relational
Databases
SMILE!
Vision
Reality

Betweenness Centrality

One of many of graph **centrality measures** which are useful for assessing the **importance of a node**.

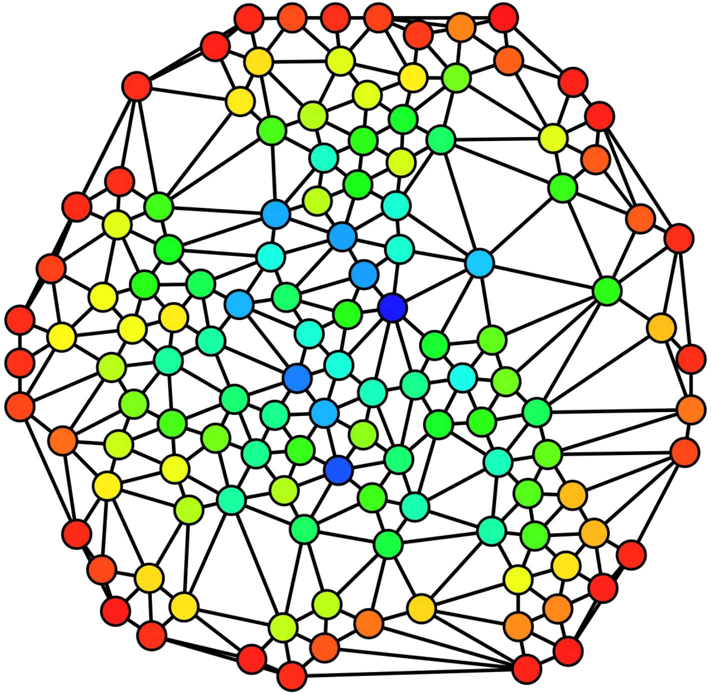
High Level Definition: Number of times a node appears on shortest paths within a network

Why it's Useful: Identify which nodes control information flow between different areas of the graph; also called "Bridge Nodes"

Business Use-Cases:

Communication Analysis: Identify important people which communicate across different groups

Retail Purchase Analysis: Which products introduce customers to new categories



RelationalAI

Betweenness Centrality

Brandes Algorithm is applied as follows:

1. For each pair of nodes, compute all shortest paths and capture nodes (less endpoints) on said path(s)
2. For each pair of nodes, assign each node along path a value of one if there is only one shortest path, or the fractional contribution (1/n) if n shortest paths
3. Sum the value from step 2 for each node; this is the Betweenness Centrality

Algorithm 1: Betweenness centrality in unweighted graphs

```
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
```

Betweenness Centrality

```
// Shortest path between s and t when they are the same is 0.
def shortest_path[s, t] = Min[
    v, w:
    (shortest_path(s, t, w) and v = 1) or
    (w = shortest_path[s,v] +1 and E(v, t))
]

// When s and t are the same, there is only one shortest path between
// them, namely the one with length 0.

def nb_shortest(s, t, n) = V(s) and V(t) and s = t and n = 1


// When s and t are *not* the same, it is the sum of the number of
// shortest paths between s and v for all the v's adjacent to t and
// on the shortest path between s and t.
def nb_shortest(s, t, n) =
   s != t and
   n = sum[v, m:
      shortest_path[s, v] + 1 = shortest_path[s, t] and E(v, t) and
      nb_shortest(s, v, m)
   ]
```

```
// sum over all t's such that there is an edge between v and t,
// and v is on the shortest path between s and t
def C[s, v] = sum[t, r:
   E(v, t) and shortest_path[s, t] = shortest_path[s, v] + 1 and
   (
      a = C[s, t] or
      not C(s, t, _) and a = 0.0
   ) and
   r = (nb_shortest[s, v] / nb_shortest[s, t]) * (1 + a)
] from a
```

```
// Note that below we divide by 2 because we are double
counting every edge.

def betweenness_centrality_brandes[v] =
   sum[s, p : s != v and C[s, v] = p]/2
```

Normal Forms

RelationalAI

Unnormalized

ISBN#	Title	Author	Author Nationality	Format	Price	Subject	Pages	Thickness	Publisher	Publisher Country	Publication Type	Genre ID	Genre Name
1590593324	Beginning MySQL Database Design and Optimization	Chad Russell	American	Hardcover	49.99	MySQL Database Design	520	Thick	Apress	USA	Book	1	Tutorial

relationalAI

First level of normalization - 1NF

Book

ISBN#	Title	Author	Author Nationality	Format	Price	Pages	Thickness	Publisher	Publisher Country	Publication Type	Genre ID	Genre Name
1590593324	Beginning MySQL Database Design and Optimization	Chad Russell	American	Hardcover	49.99	520	Thick	Apress	USA	Book	1	Tutorial
1590593324	Beginning MySQL Database Design and Optimization	Chad Russell	American	E-book	22.34	520	Thick	Apress	USA	Book	1	Tutorial
1234567890	The Relational Model for Database Management: Version 2	E. F. Codd	British	E-book	13.88	538	Thick	Addison-Wesley	USA	Book	2	Popular Science
1234567890	The Relational Model for Database Management: Version 2	E. F. Codd	British	Paperback	39.99	538	Thick	Addison-Wesley	USA	Book	2	Popular Science

Subject

ISBN#	Subject
1590593324	MySQL
1590593324	Database
1590593324	Design

relationalAI

Next level of normalization - 2NF

Book

ISBN#	Title	Author	Author Nationality	Pages	Thickness	Publisher	Publisher Country	Publication Type	Genre ID	Genre Name
1590593324	Beginning MySQL Database Design and Optimization	Chad Russell	American	520	Thick	Apress	USA	Book	1	Tutorial
1234567890	The Relational Model for Database Management: Version 2	E. F. Codd	British	538	Thick	Addison-Wesley	USA	Book	2	Popular Science

Subject

ISBN#	Subject
1590593324	MySQL
1590593324	Database
1590593324	Design

Format- Price

ISBN#	Format	Price
1590593324	Hardcover	49.99
1590593324	E-book	22.34
1234567890	E-book	13.88
1234567890	Paperback	39.99

Next level of normalization - 3NF

Book

ISBN#	Title	Author	Pages	Thickness	Publisher	Publication Type	Genre ID
1590593324	Beginning MySQL Database Design and Optimization	Chad Russell	520	Thick	Apress	Book	1
1234567890	The Relational Model for Database Management: Version 2	E. F. Codd	538	Thick	Addison-Wesley	Book	2

Subject

ISBN#	Subject
1590593324	MySQL
1590593324	Database
1590593324	Design

Format- Price

ISBN#	Format	Price
1590593324	Hardcover	49.99
1590593324	E-book	22.34
1234567890	E-book	13.88
1234567890	Paperback	39.99

Author

Author	Author Nationality
Chad Russell	American
E. F. Codd	British

Genre

Genre ID	Genre Name
1	Tutorial
2	Popular Science

Publisher

Publisher	Publisher Country
Apress	USA
Addison-Wesley	USA

Other normal forms

EKNF: Elementary key normal form
BCNF: Boyce–Codd normal form
4NF: Fourth normal form
ETNF: Essential tuple normal form
5NF: Fifth normal form
DKNF: Domain-key normal form
6NF: Sixth normal form

Each of the above eliminates some form of redundancy and decomposes the model into its elementary (atomic) building blocks.

relationalAI

Ultimate level of normalization - GNF

hasISBN#

Book	ISBN#
1	1590593324
2	1234567890

hasTitle

Book	Title
1	Beginning MySQL Database Design and Optimization
2	The Relational Model for Database Management: Version 2

hasAuthor

Book	Author
1	Chad Russell
2	E. F. Codd

hasNumPages

Book	Pages
1	520
2	538

...

hasPublisher

Book	Publisher
1	Apress
2	Addison-W esley

...

hasGenre

Book	Genre
1	1
2	2

hasSubject

Book	Subject
1	MySQL
1	Database
1	Design

FormatHasPrice

Book	Format	Price
1	1	49.99
1	2	22.34
2	2	13.88
2	3	39.99

hasName

Author	Name
1	Chad Russell
2	E. F. Codd

hasNationality

Author	Nationality
1	American
2	British

hasName

Genre	Name
1	Tutorial
2	Popular Science

hasName

Publisher	Name
1	Apress
2	Addison-Wesley

hasCountry

Publisher	Country
1	USA
2	USA

Ultimate level of normalization - GNF

hasISBN#

Book	ISBN#
1	1590593324
2	1234567890

hasTitle

Book	Title
1	Beginning MySQL Database Design and Optimization
2	The Relational Model for Database Management: Version 2

hasAuthor

Book	Author
1	Chad Russell
2	E. F. Codd

hasNumPages

Book	Pages
1	520
2	538

...

hasPublisher

Book	Publisher
1	Apress
2	Addison-Wesley

...

hasGenre

Book	Genre
1	1
2	2

Similar concept?

hasSubject

Book	Subject
1	MySQL
1	Database
1	Design

FormatHasPrice

Book	Format	Price
1	1	49.99
1	2	22.34
2	2	13.88
2	3	39.99

hasName

Author	Name
1	Chad Russell
2	E. F. Codd

hasNationality

Author	Nationality
1	American
2	British

hasName

Genre	Name
1	Tutorial
2	Popular Science

hasName

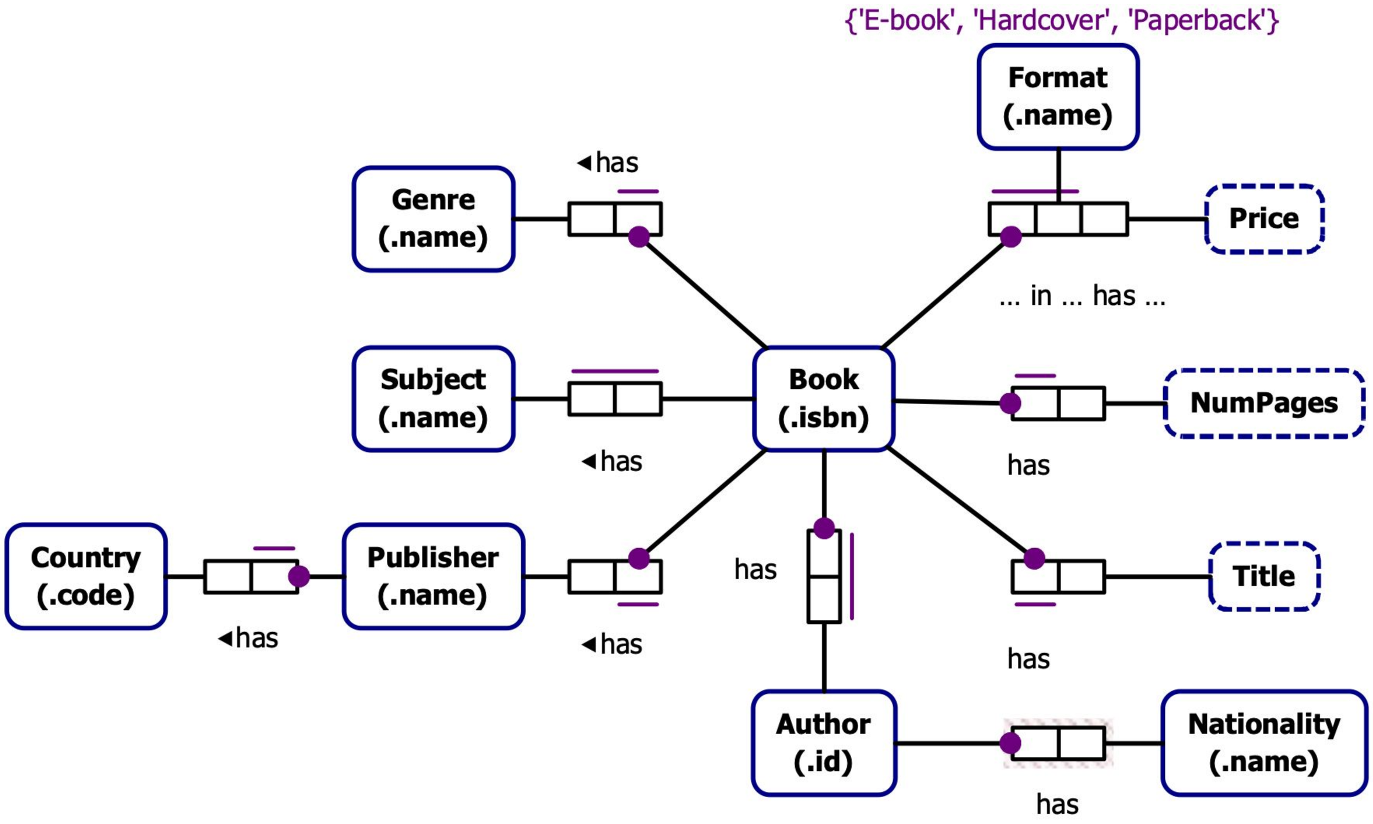
Publisher	Name
1	Apress
2	Addison-Wesley

hasCountry

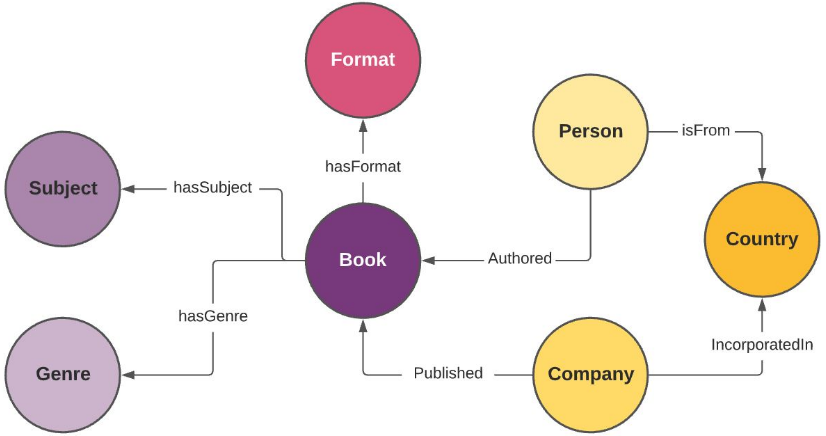
Publisher	Country
1	USA
2	USA

relationalAI

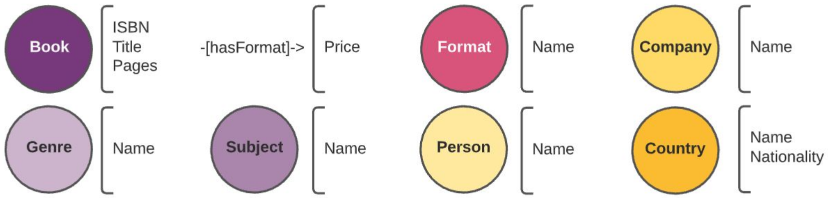
Ta-da -- A Relational Knowledge Graph!



Labeled Property Graph Schema



Node & Relationship Attributes

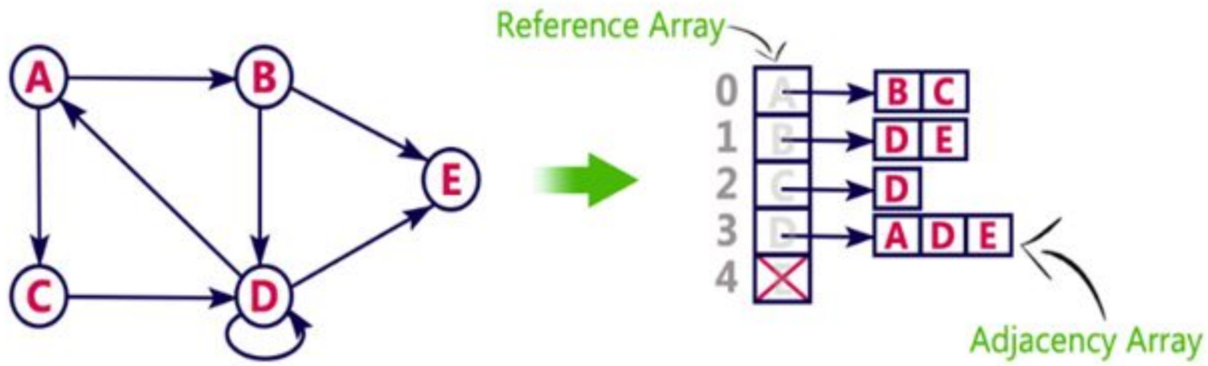


How should we represent graphs?

RelationalAI

How do you represent relationships in a graph?

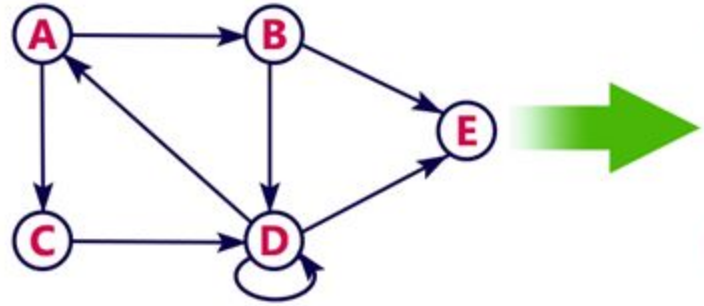
With pointers in an adjacency list



https://www.javatpoint.com/graph-theory-graph-representations

How do you represent relationships in a graph?

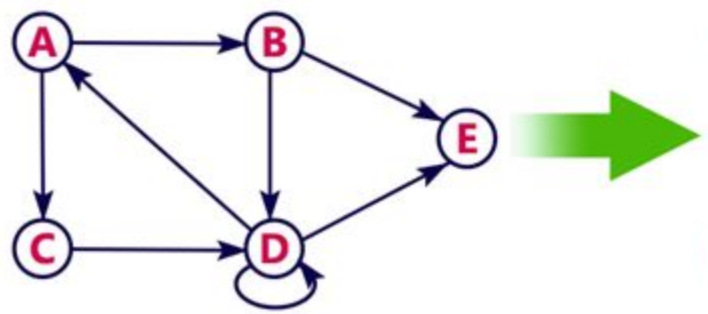
With an adjacency matrix



	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

How do you represent relationships in a graph?

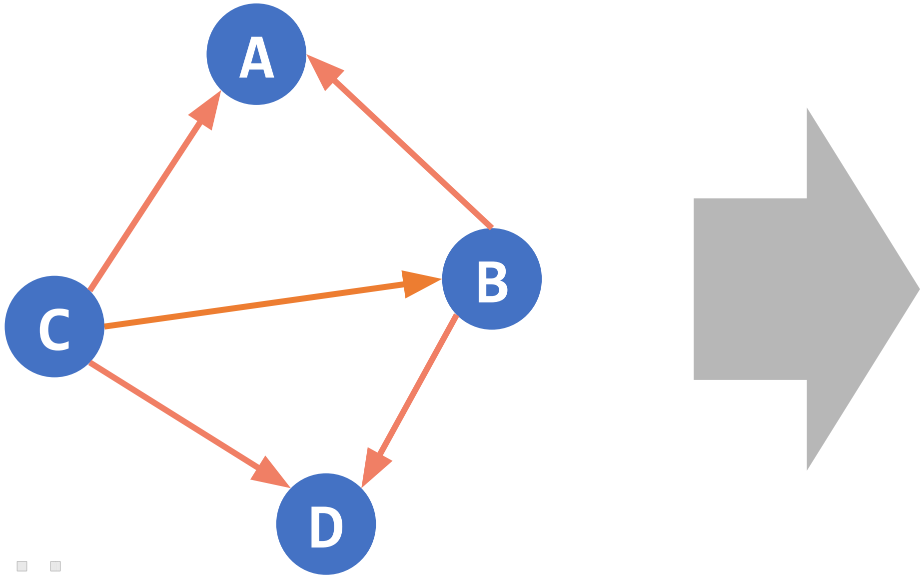
With an edge relation



SRC	DEST
A	B
A	C
B	D
B	E
C	D
D	A
D	D
D	E

relationalAI

Directed Graphs as a Relation




```
edge(B, A)
edge(B, D)

edge(C, A)
edge(C, B)
edge(C, D)
```

Relations are a universal abstraction!

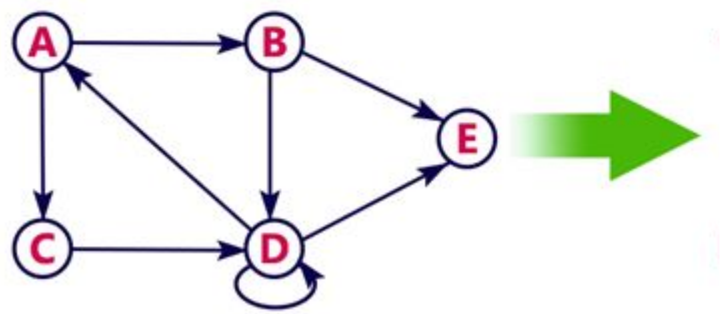
Graph	→ Binary relation
Hypergraph	→ n-ary relation with n > 2
Function	→ Relation with functional dependency constraint
Tensor	→ Function mapping tuple of integer indexes to a numeric value
Set	→ Unary relation
Bag	→ Function from set element to count

…

You can seperate the abstraction from the implementation…

Separation of the what from the how - data structures

Edge relation

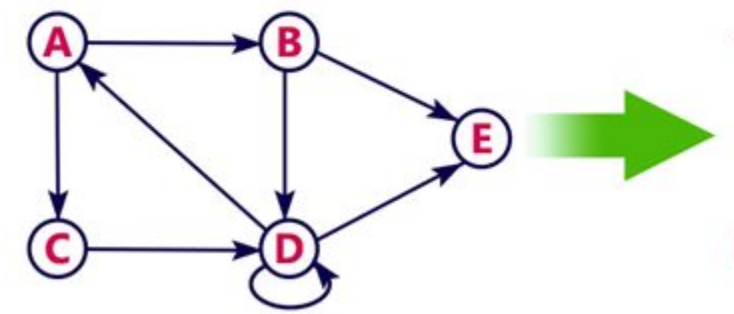


SRC	DEST
A	B
A	C
B	D
B	E
C	D
D	A
D	D
D	E

relationalAI

Separation of the what from the how - data structures

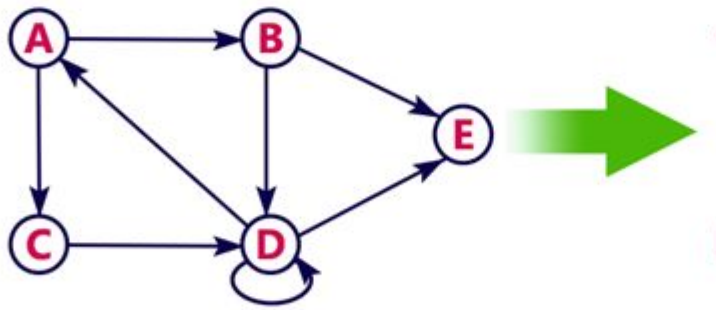
Edge relation - src to dest index



SRC	DEST
A	B
	C
B	D
	E
C	D
D	A
	D
	E

Separation of the what from the how - data structures

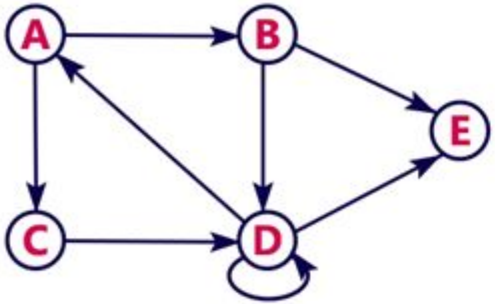
Edge relation



DEST	SRC
A	D
B	A
C	A
D	B
D	C
D	D
E	B
E	D

Separation of the what from the how - data structures

Edge relation - dest to src index



DEST	SRC
A	D
B	A
C	A
D	B
	C
	D
E	B
	D