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.
Use Case: Business Intelligence
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.
Q1-b

Total extended price

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

def result = sum[charge]

def charge =
  extendedprice * (1 - discount) * (1 + tax)
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
**Q5-u**

All customers in Asia (Join)

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

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

Navigational Notation
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

<table>
<thead>
<tr>
<th>orderkey</th>
<th>customer</th>
<th>date</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>500</td>
<td>2022-03-27</td>
<td>75</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>2022-03-27</td>
<td>43</td>
</tr>
</tbody>
</table>

- `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.
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

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Spark Dataframes</th>
<th>Spark GraphFrames</th>
<th>Neo4J Cypher</th>
<th>Tensor Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SQL</strong></td>
<td><code>SELECT source AS id, COUNT(*)</code> FROM edge GROUP BY id</td>
<td><code>result = edges.groupBy(&quot;src&quot;).agg(count(&quot;*&quot;)))</code></td>
<td><code>g = GraphFrame(nodes, edges)</code> <code>result = g.outDegrees</code></td>
<td><code>MATCH (n:node)-[r]-&gt;()</code> <code>RETURN n.id, COUNT(DISTINCT r) as degree</code></td>
<td><code>def degree[x] = count[edge[x]]</code></td>
</tr>
</tbody>
</table>

### 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.
## Path Count per Node (3 hops)

<table>
<thead>
<tr>
<th>Neo4j Cypher</th>
<th>SQL</th>
<th>Tensor Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATCH( (a:node)-[:POINTSTO]-&gt;(b:node) )&lt;br&gt;MATCH( (b:node)-[:POINTSTO]-&gt;(c:node) )&lt;br&gt;MATCH( (c:node)-[:POINTSTO]-&gt;(d:node) )&lt;br&gt;RETURN a.id, COUNT(*)</td>
<td>SELECT e1.source, COUNT(*)&lt;br&gt;FROM edge e1, edge e2, edge e3&lt;br&gt;WHERE&lt;br&gt;  e1.dest = e2.source AND&lt;br&gt;  e2.dest = e3.source AND&lt;br&gt;GROUP BY e1.source</td>
<td>def path3(a, b, c, d) =&lt;br&gt;  edge(a, b) and&lt;br&gt;  edge(b, c) and&lt;br&gt;  edge(c, d)&lt;br&gt;def result[a] = count[path3[a]]</td>
</tr>
</tbody>
</table>
Results: Path Count per Node (3 hops)
Graph Analytics

No explicit syntax for graphs

```python
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
```
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
```
Labelled Property Graphs as Relational Graphs

Director
name: Villeneuve
id: 2
director(2), 3

Director
name: Villeneuve
id: 2
director(2), 3

Actor
name: Chalamet
id: 1
acted(1, 3)
role(1, 3, "Paul Atreides")

Movie
title: Dune
year: 2021
id: 3
acted(1, 3)

Actor
name: Chalamet
id: 1
acted
role: Paul Atreides

Director
name: Villeneuve
id: 2
directed
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)
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
Use Case: Linear Algebra
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**

<table>
<thead>
<tr>
<th>L_ORDERKEY</th>
<th>L_LINENUMBER</th>
<th>L_PARTKEY</th>
<th>L_SUPPKEY</th>
<th>L_QUANTITY</th>
<th>L_EXTENDEDPRICE</th>
<th>L_DISCOUNT</th>
<th>L_TAX</th>
<th>L_RETURNFLAG</th>
<th>L_LINESTATUS</th>
<th>L_SHIPDATE</th>
<th>L_COMMITDATE</th>
<th>L_RECEIPTDATE</th>
<th>L_SHIPINSTRUCT</th>
<th>L_SHIPMODE</th>
<th>L_COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1552</td>
<td>93</td>
<td>1 17</td>
<td></td>
<td></td>
<td>24710.35</td>
<td>0.04</td>
<td>0.02</td>
<td>N</td>
<td>O</td>
<td>1996-03-13</td>
<td>1996-02-12</td>
<td>1996-03-22</td>
<td>DELIVER IN PERSON</td>
<td>TRUCK</td>
<td></td>
</tr>
<tr>
<td>1 674</td>
<td>75</td>
<td>2 36</td>
<td></td>
<td></td>
<td>56688.12</td>
<td>0.09</td>
<td>0.06</td>
<td>N</td>
<td>O</td>
<td>1996-04-12</td>
<td>1996-02-28</td>
<td>1996-04-20</td>
<td>TAKE BACK RETURN</td>
<td>MAIL</td>
<td></td>
</tr>
<tr>
<td>1 637</td>
<td>38</td>
<td>3 8</td>
<td></td>
<td></td>
<td>12301.04</td>
<td>0.10</td>
<td>0.02</td>
<td>N</td>
<td>O</td>
<td>1996-01-29</td>
<td>1996-03-05</td>
<td>1996-01-31</td>
<td>TAKE BACK RETURN</td>
<td>REG AIR</td>
<td></td>
</tr>
</tbody>
</table>

**Tensor Notation**

\[ l_{\text{extendedprice}}[o, \text{num}] \]

\[ l_{\text{shipdate}}[o, \text{num}] \]

\[ l_{\text{shipmode}}[o, \text{num}] \]
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

**Math**

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

**Rel** Our new relational language

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

**SQL**

```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
```
Use Case: JSON & semi-structured data

```json
{
  "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

We can represent JSON with first-order relations in graph normal form.
After parsing, JSON is typically represented as a tree (right).

```json
{
  "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
Have relational database systems been sufficiently ambitious on this point?

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 yields 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.
Most people have never used a Relational Database
Relational Databases

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
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

\[ C_B[v] \leftarrow 0, v \in V; \]
for \( s \in V \) do
\[ S \leftarrow \text{empty stack}; \]
\[ P[w] \leftarrow \text{empty list}, w \in V; \]
\[ \sigma[t] \leftarrow 0, t \in V; \quad \sigma[s] \leftarrow 1; \]
\[ d[t] \leftarrow -1, t \in V; \quad d[s] \leftarrow 0; \]
\[ Q \leftarrow \text{empty queue}; \]
\[ \text{enqueue } s \rightarrow Q; \]
while \( Q \text{ not empty} \) do
\[ \text{dequeue } v \leftarrow Q; \]
\[ \text{push } v \rightarrow S; \]
\[ \text{foreach neighbor } w \text{ of } v \text{ do} \]
\[ \quad \text{// } w \text{ found for the first time?} \]
\[ \quad \text{if } d[w] < 0 \text{ then} \]
\[ \quad \quad \text{enqueue } w \rightarrow Q; \]
\[ \quad \quad d[w] \leftarrow d[v] + 1; \]
\[ \quad \text{end} \]
\[ \quad \text{// shortest path to } w \text{ via } v? \]
\[ \quad \text{if } d[w] = d[v] + 1 \text{ then} \]
\[ \quad \quad \sigma[w] \leftarrow \sigma[w] + \sigma[v]; \]
\[ \quad \text{append } v \rightarrow P[w]; \]
\[ \quad \text{end} \]
\[ \text{end} \]
\[ \delta[v] \leftarrow 0, v \in V; \]
\[ \text{// } S \text{ returns vertices in order of non-increasing distance from } s \]
while \( S \text{ not empty} \) do
\[ \text{pop } w \leftarrow S; \]
\[ \text{for } v \in P[w] \text{ do} \]
\[ \quad \delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w]); \]
\[ \quad \text{if } w \neq s \text{ then} \quad C_B[w] \leftarrow C_B[w] + \delta[w]; \]
\[ \text{end} \]
end
```

RelationalAI
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
<table>
<thead>
<tr>
<th>ISBN#</th>
<th>Title</th>
<th>Author</th>
<th>Author Nationality</th>
<th>Format</th>
<th>Price</th>
<th>Subject</th>
<th>Pages</th>
<th>Thickness</th>
<th>Publisher</th>
<th>Publisher Country</th>
<th>Publication Type</th>
<th>Genre ID</th>
<th>Genre Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>Beginning MySQL Database Design and Optimization</td>
<td>Chad Russell</td>
<td>American</td>
<td>Hardcover</td>
<td>49.99</td>
<td>MySQL, Database, Design</td>
<td>520</td>
<td>Thick</td>
<td>Apress</td>
<td>USA</td>
<td>Book</td>
<td>1</td>
<td>Tutorial</td>
</tr>
</tbody>
</table>
## First level of normalization - 1NF

### Book

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

### Subject

<table>
<thead>
<tr>
<th>ISBN#</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>MySQL</td>
</tr>
<tr>
<td>1590593324</td>
<td>Database</td>
</tr>
<tr>
<td>1590593324</td>
<td>Design</td>
</tr>
</tbody>
</table>
Next level of normalization - 2NF

### Book

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

### Subject

<table>
<thead>
<tr>
<th>ISBN#</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>MySQL</td>
</tr>
<tr>
<td>1590593324</td>
<td>Database</td>
</tr>
<tr>
<td>1590593324</td>
<td>Design</td>
</tr>
</tbody>
</table>

### Format- Price

<table>
<thead>
<tr>
<th>ISBN#</th>
<th>Format</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>Hardcover</td>
<td>49.99</td>
</tr>
<tr>
<td>1590593324</td>
<td>E-book</td>
<td>22.34</td>
</tr>
<tr>
<td>1234567890</td>
<td>E-book</td>
<td>13.88</td>
</tr>
<tr>
<td>1234567890</td>
<td>Paperback</td>
<td>39.99</td>
</tr>
</tbody>
</table>
## Next level of normalization - 3NF

### Book

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

### Subject

<table>
<thead>
<tr>
<th>ISBN#</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>MySQL</td>
</tr>
<tr>
<td>1590593324</td>
<td>Database</td>
</tr>
<tr>
<td>1590593324</td>
<td>Design</td>
</tr>
</tbody>
</table>

### Format - Price

<table>
<thead>
<tr>
<th>ISBN#</th>
<th>Format</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>Hardcover</td>
<td>49.99</td>
</tr>
<tr>
<td>1590593324</td>
<td>E-book</td>
<td>22.34</td>
</tr>
<tr>
<td>1234567890</td>
<td>E-book</td>
<td>13.88</td>
</tr>
<tr>
<td>1234567890</td>
<td>Paperback</td>
<td>39.99</td>
</tr>
</tbody>
</table>

### Author

<table>
<thead>
<tr>
<th>Author</th>
<th>Author Nationality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chad Russell</td>
<td>American</td>
</tr>
<tr>
<td>E. F. Codd</td>
<td>British</td>
</tr>
</tbody>
</table>

### Genre

<table>
<thead>
<tr>
<th>Genre ID</th>
<th>Genre Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tutorial</td>
</tr>
<tr>
<td>2</td>
<td>Popular Science</td>
</tr>
</tbody>
</table>

### Publisher

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Publisher Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apress</td>
<td>USA</td>
</tr>
<tr>
<td>Addison-Wesley</td>
<td>USA</td>
</tr>
</tbody>
</table>
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.
# Ultimate level of normalization - GNF

<table>
<thead>
<tr>
<th>hasISBN#</th>
<th>hasTitle</th>
<th>hasAuthor</th>
<th>hasNumPages</th>
<th>hasPublisher</th>
<th>hasGenre</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book</td>
<td>ISBN#</td>
<td>Title</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1590593324</td>
<td>Beginning MySQL Database Design and Optimization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1234567890</td>
<td>The Relational Model for Database Management: Version 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hasSubject</th>
<th>FormatHasPrice</th>
<th>hasName</th>
<th>hasNationality</th>
<th>hasName</th>
<th>hasName</th>
<th>hasName</th>
<th>hasCountry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book</td>
<td>Subject</td>
<td>Author</td>
<td>Nationality</td>
<td>Name</td>
<td>Name</td>
<td>Name</td>
<td>Publisher</td>
</tr>
<tr>
<td>1</td>
<td>MySQL</td>
<td>Chad</td>
<td>American</td>
<td>Russell</td>
<td></td>
<td></td>
<td>Apress</td>
</tr>
<tr>
<td>2</td>
<td>Database</td>
<td>E. F.</td>
<td>British</td>
<td>Codd</td>
<td></td>
<td></td>
<td>Addison-Wesley</td>
</tr>
<tr>
<td>1</td>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>USA</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>USA</td>
</tr>
</tbody>
</table>
## Ultimate level of normalization - GNF

<table>
<thead>
<tr>
<th>hasISBN#</th>
<th>hasTitle</th>
<th>hasAuthor</th>
<th>hasNumPages</th>
<th>hasPublisher</th>
<th>hasGenre</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book</td>
<td>ISBN#</td>
<td>Title</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1590593324</td>
<td>Beginning MySQL Database Design and Optimization</td>
<td>1</td>
<td>Chad Russell</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1234567890</td>
<td>The Relational Model for Database Management: Version 2</td>
<td>2</td>
<td>E. F. Codd</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hasSubject</th>
<th>FormatHasPrice</th>
<th>hasName</th>
<th>hasNationality</th>
<th>hasPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book</td>
<td>Subject</td>
<td>Author</td>
<td>Author Nationality</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>MySQL</td>
<td>Chad</td>
<td>American</td>
<td>49.99</td>
</tr>
<tr>
<td>1</td>
<td>Database</td>
<td>Russell</td>
<td></td>
<td>22.34</td>
</tr>
<tr>
<td>1</td>
<td>Design</td>
<td></td>
<td>British</td>
<td>13.88</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>Popular Science</td>
<td>39.99</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hasSimilarConcept?</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>hasName</th>
<th>hasCountry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Publisher</td>
</tr>
<tr>
<td></td>
<td>Country</td>
</tr>
<tr>
<td>1</td>
<td>Apress</td>
</tr>
<tr>
<td>2</td>
<td>Addison-Wesley</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hasGenre</th>
<th>hasPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Price</td>
</tr>
<tr>
<td>1 Tutorial</td>
<td>99.99</td>
</tr>
<tr>
<td>2 Popular Science</td>
<td>13.88</td>
</tr>
</tbody>
</table>

**Similar concept?**
Ta-da -- A Relational Knowledge Graph!
How should we represent graphs?
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
How do you represent relationships in a graph?

With an edge relation

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>
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 separate the abstraction from the implementation...
Separation of the what from the how - data structures

Edge relation

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>
Separation of the what from the how - data structures

Edge relation - src to dest index

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>E</td>
</tr>
</tbody>
</table>
Separation of the what from the how - data structures

Edge relation

<table>
<thead>
<tr>
<th>DEST</th>
<th>SRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
</tr>
</tbody>
</table>
Separation of the what from the how - data structures

Edge relation - dest to src index

<table>
<thead>
<tr>
<th>DEST</th>
<th>SRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>