Advanced Data Management (CSCI 490/680)

Graph Databases

Dr. David Koop
Recent History in Databases

- Early 2000s: Commercial DBs dominated, Open-source DBs missing features
- Mid 2000s: MySQL adopted by web companies
- Late 2000s: NoSQL dos scale horizontally out of the box
- Early 2010s: New DBMSs that can scale across multiple machines natively and provide ACID guarantees
NewSQL

• 451 Group’s Definition:
  - A DBMS that delivers the scalability and flexibility promised by NoSQL while retaining the support for SQL queries and/or ACID, or to improve performance for appropriate workloads.

• Stonebraker's Definition:
  - SQL as the primary interface
  - ACID support for transactions
  - Non-locking concurrency control
  - High per-node performance
  - Parallel, shared-nothing architecture
OLTP Workload

- **Operation Complexity**
  - Simple
  - Complex

- **Workload Focus**
  - OLTP
  - Social Networks
  - Data Warehouses

The diagram represents the workload focus for OLTP, social networks, and data warehouses on a graph with operation complexity on the y-axis and workload types on the x-axis.

*Michael Stonebraker*
Ideal OLTP System

- Main Memory Only
- No Multi-processor Overhead
- High Scalability
- High Availability
- Autonomic Configuration
Spanner Overview

- Focus on scaling databases focused on OLTP (not OLAP)
- Since OLTP, focus is on sharding rows
- Tries to satisfy CAP (which is impossible per CAP Theorem) by not worrying about 100% availability
- External consistency using multi-version concurrency control through timestamps
- ACID is important
- Structured: universe with zones with zone masters and then spans with span masters
- SQL-like (updates allow SQL to be used with Spanner)
Spanner and the CAP Theorem

- Which type of system is Spanner?
  - C: consistency, which implies a single value for shared data
  - A: 100% availability, for both reads and updates
  - P: tolerance to network partitions

- Which two?
  - CA: close, but not totally available
  - So actually CP
External Consistency

- Traditional DB solution: **two-phase locking**—no writes while client reads
- "The system behaves as if all transactions were executed sequentially, even though Spanner actually runs them across multiple servers (and possibly in multiple datacenters) for higher performance and availability" [Google]
- Semantically indistinguishable from a single-machine database
- Uses multi-version concurrency control (MVCC) using **timestamps**
- Spanner uses **TrueTime** to generate monotonically increasing timestamps across all nodes of the system
## Google Cloud Spanner: NewSQL

<table>
<thead>
<tr>
<th>Feature</th>
<th>Cloud Spanner</th>
<th>Traditional Relational</th>
<th>Traditional Non-Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQL</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Consistency</td>
<td>Strong</td>
<td>Strong</td>
<td>Eventual</td>
</tr>
<tr>
<td>Availability</td>
<td>High</td>
<td>Failover</td>
<td>High</td>
</tr>
<tr>
<td>Scalability</td>
<td>Horizontal</td>
<td>Vertical</td>
<td>Horizontal</td>
</tr>
<tr>
<td>Replication</td>
<td>Automatic</td>
<td>Configurable</td>
<td>Configurable</td>
</tr>
</tbody>
</table>

[https://cloud.google.com/spanner/]
Spanner as "Effectively CA"

- Criteria for being "effectively CA"
  1. At a minimum it must have very high availability in practice (so that users can ignore exceptions), and
  2. As this is about partitions it should also have a low fraction of those outages due to partitions.
- Spanner meets both of these criteria
- Spanner relies on Google's network (private links between data centers)
- TrueTime helps create consistent snapshots, sometimes have a commit wait

[E. Brewer, 2017]
Throughput: Spanner vs. MySQL

Median Latency as Throughput Increases

Throughput (queries per second)

Latency (ms)

MySQL (median)
spanner 9 nodes (median)
spanner 15 nodes (median)
spanner 30 nodes (median)

[P. Bakkum and D. Cepeda, 2017]
Assignment 4

- World Education Data
- Collected/collated by UNESCO, World Bank, and OECD
- Transform World Bank Data
- Impute missing year data
- Integrate teacher and student numbers
- Fuse three datasets
- Think about how to integrate based on country
- Last part: country name can be any (consider 'first' aggregation option)
- Due Monday
Specific Types of Data
Graphs: Social Networks

[P. Butler, 2010]
What is a Graph?

- An abstract representation of a set of objects where some pairs are connected by links.

Object (Vertex, Node)

Link (Edge, Arc, Relationship)
What is a Graph?

- In computing, a **graph** is an abstract **data structure** that represents set objects and their relationships as **vertices** and **edges/links**, and supports a number of graph-related **operations**
- Objects (nodes): \{A, B, C, D\}
- Relationships (edges):
  \{ (D, B), (D, A), (B, C), (B, A), (C, A) \}
- Operation: shortest path from D to A

[K. Salama, 2016]
Different Kinds of Graphs

- Undirected Graph
- Directed Graph
- Pseudo Graph
- Multi Graph
- Hyper Graph
Graphs with Properties

- Each vertex or edge may have properties associated with it
- May include identifiers or classes

```plaintext
Person
name = 'Tom Hanks'
born = 1956

Person
name = 'Robert Zemeckis'
born = 1951

Movie
title = 'Forrest Gump'
released = 1994
```

ACTED_IN
roles = ['Forrest']

DIRECTED
Types of Graph Operations

- Connectivity Operations:
  - number of vertices/edges, in- and out-degrees of vertices
  - histogram of degrees can be useful in comparing graphs
- Path Operations: cycles, reachability, shortest path, minimum spanning tree
- Community Operations: clusters (cohesion and separation)
- Centrality Operations: degree, vulnerability, PageRank
- Pattern Matching: subgraph isomorphism
  - can use properties
  - useful in fraud/threat detection, social network suggestions
What is a Graph Database?

• A database with an explicit graph structure
• Each node knows its adjacent nodes
• As the number of nodes increases, the cost of a local step (or hop) remains the same
• Plus an Index for lookups

[M. De Marzi, 2012]
How do Graph Databases Compare?

90% of Use Cases

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational Databases</td>
<td>Key-Value Store</td>
</tr>
<tr>
<td>Document Databases</td>
<td>BigTable Clones</td>
</tr>
<tr>
<td>Graph Databases</td>
<td></td>
</tr>
</tbody>
</table>

[M. De Marzi, 2012]
Graph Databases Compared to Relational Databases

Compared to Relational Databases:

- Optimized for aggregation
- Optimized for connections

[Image: Diagram showing comparison between graph databases and relational databases, with nodes A1, A2, A3 on the left and nodes B1, B2, B3, B4, B5, B6, B7, C1, C2, C3 on the right, connected by lines to illustrate the optimization aspects.]
Graph Databases Compared to Key-Value Stores

Compared to Key Value Stores

Optimized for simple look-ups

Optimized for traversing connected data

[M. De Marzi, 2012]
Graph Databases Compared to Document Stores

Optimized for “trees” of data

Optimized for seeing the forest and the trees, and the branches, and the trunks

[M. De Marzi, 2012]
Graph Databases

D. Lembo and R. Rosati
Why Graph Database Models?

• Graphs has been long ago recognized as one of the most simple, natural and intuitive knowledge representation systems
• Graph data structures allow for a natural modeling when data has graph structure
• Queries can address direct and explicitly this graph structure
• Implementation-wise, graph databases may provide special graph storage structures, and take advantage of efficient graph algorithms available for implementing specific graph operations over the data

[R. Angles and C. Gutierrez, 2017]
Figure 1: Example of a genealogy expressed in the relational model (i.e. as tables on the left) and a diagram of its scheme on the right.

3.1 The basics: Labeled graphs

The most basic data structure for graph database models is a directed graph with nodes and edges labeled by some vocabulary. A good example is Gram [37], a graph data model motivated by hypertext querying.

A schema in Gram is a directed labeled multigraph, where each node is labeled with a symbol called a type, which has associated a domain of values. In the same way, each edge has assigned a label representing a relation between types (see example in Figure 2). A feature of Gram is the use of regular expressions for explicit definition of paths called walks. An alternating sequence of nodes and edges represent a walk, which combined with other walks conforms other special objects called hyperwalks.

For querying the model (particularly path-like queries), an algebraic language based on regular expressions is proposed. For this purpose a hyper-
Basic Labeled Model (Gram)

- Directed graph with nodes and edges labeled by some vocabulary
- Gram is a directed labeled multigraph
  - Each node is labeled with a symbol called a **type**
  - Each edge has assigned a label representing a **relation** between types

![Diagram of Basic Labeled Model (Gram)]

[R. Angles and C. Gutierrez, 2017]
Hypergraph Model (Groovy)

- Notion of edge is extended to **hyperedge**, which relates an arbitrary set of nodes
- Hypergraphs allow the definition of complex objects (undirected), functional dependencies (directed), object-ID and (multiple) structural inheritance

![Hypergraph Diagram]

[R. Angles and C. Gutierrez, 2017]
Hypernode Model

- Hypernode is a directed graph whose nodes can themselves be graphs (or hypernodes), allowing **nesting** of graphs
- **Encapsulates** information

![Schema and Instance Diagram]

[Figure 4: Hypernode Model. The schema (left) defines a person as a complex object with the properties name and lastname of type string, and parent of type person (recursively defined). The instance (on the right) shows the relations in the genealogy among different instances of person.]

Summarizing, the main features of the Hypernode model are: a nested graph structure which is simple and formal; the ability to model arbitrary complex objects in a straightforward manner; underlying data structure of an object-oriented data model; enhancement of the usability of a complex objects database system via a graph-based user interface.

3.4 Trees: The Semistructured model (JSON, OEM, XML)

The semistructured model was designed to describe data together with its schema in one place, also called “self-describing” data. Technically they are trees, the most simple version of a graph, but could describe, via references, general graphs.

The semistructured model was designed to overcome the limitation of both, structured data (fixed schema and format, precise rules) and unstructured data (loose schema, no format, little predictability). The early motivations were the modeling of documents (whose structure can be viewed as trees), data on the Web and data integration at Web scale [50, 33].

Among its advantages are the simple way to integrate new data, to model incomplete data, and the flexibility to query it without prior knowledge of schema. The drawbacks are mainly in the area of optimization, which becomes much harder as the structure of the data is not necessarily known in advance.

An early proposal in this direction was the data model OEM [74, 120] which proposed an extremely simple and elegant model of objects with identifiers and “links” to other objects , with a simple syntax (see Figure 5) which

[R. Angles and C. Gutierrez, 2017]
Semistructured (Tree) Model: (OEM Graph)

- "Self-describing" data like JSON and XML
- OEM uses pointers to data in the tree

**OEM Syntax**

```
{ person : &p1 { name : "George" ,
                 lastname : "Jones" }

person : &p2 { name : "Ana" ,
                 lastname : "Stone" }

person : &p3 { name : "Julia" ,
                 lastname : "Jones" ,
                 parent : &p1 ,
                 parent : &p2 }

person : &p4 { name : "James" ,
                 lastname : "Deville" }

person : &p5 { name = "David",
                 lastname : "Deville",
                 parent : &p3 ,
                 parent : &p4 }

person : &p6 { name = "Mary",
                 lastname : "Deville",
                 parent : &p4 ,
                 parent : &p3 }}
```

**OEM Graph**

![OEM Graph](image)

[R. Angles and C. Gutierrez, 2017]
RDF (Triple) Model

- Interconnect resources in an extensible way using graph-like structure for data
- Schema and instance are mixed together
- SPARQL to query
- Semantic web

[Image of RDF data model with instances and schema interconnected]

[Sources: R. Angles and C. Gutierrez, 2017]
Property Graph Model (Cypher in neo4j)

- Directed, labelled, attributed multigraph
- Properties are **key/value pairs** that represent metadata for nodes and edges

![Property Graph Model Diagram]

[R. Angles and C. Gutierrez, 2017]
Types of Graph Queries

- Adjacency queries (neighbors or neighborhoods)
- Pattern matching queries (related to graph mining)
  - Graph patterns with structural extension or restrictions
  - Complex graph patterns
  - Semantic matching
  - Inexact matching
  - Approximate matching
- Reachability queries (connectivity)

[R. Angles and C. Gutierrez, 2017]
Types of Graph Queries (continued)

• Analytical queries
  - Summarization queries
  - Complex analytical queries (PageRank, characteristic path length, connected components, community detection, clustering coefficient)

[R. Angles and C. Gutierrez, 2017]
Figure 8: Evolution of graph query languages: G [63], G+ [64], Graphlog [61], HPQL [104], THQL [141], GRE [37], Hyperlog [123], HNQL [103], PORL [72], SLQL [52], HQL [137], PRPQ [107], GraphQL [85], SPARQL [124], RLV [132], Cypher [14], ECRPQ [43], PDQL [41], GX-Path [106], SPARQL 1.1 [71] and RQ [127].

For the sake of space we will not present a complete review of graph query languages. Instead we describe some of the languages we consider relevant and useful to show the developments in the area. Moreover, we restrict our review to “pure” GQLs, that is those languages specifically designed to work with graph data models. Figure 8 presents this subset of languages in chronological order.

As we mentioned before, Cruz et al. [63] proposed the query language G. This language introduced the notion of graphical query as a set of query graphs. A query graph (pattern) is a labeled directed multigraph in which the node labels may be either variables or constants, and the edge labels can be regular expressions combining variables and constants. The result of a graphical query $Q$ with respect to a graph database $G$ is the union of all query graphs of $Q$ which match subgraphs of $G$. For instance, Figure 9 presents an example of graphical query containing two query graphs, $Q_1$ and $Q_2$. This query finds the first and last cities visited in all round trips from Toronto (“Tor”), in which the first and last flights are with Air Canada (“AC”) and all other flights (if any) are with the same airline. Note that the last condition is expressed by the edge labeled with regular expression $w^+$. Thanks to the inclusion of regular expressions, G is able to express recursive queries more general than transitive closure. However, the evaluation of queries in G is of high computational complexity due to its semantics based on simple paths.

G evolved into a more powerful language called G+ [64]. The notion of graphical query proposed by G is extended in G+ to define a summary

[Graph Query Languages

[R. Angles and C. Gutierrez, 2017]
Cypher

- Implemented by neo4j system
- Expresses reachability queries via path expressions
  - \( p = (a)-[\text{:knows}*-]->(b): \) nodes from \( a \) to \( b \) following \text{knows} edges
- \text{START} \ x=\text{node:person(name="John")}
  \text{MATCH} \ (x)-[\text{:friend}]->(y)
  \text{RETURN} \ y.\text{name}

[R. Angles and C. Gutierrez, 2017]
SPARQL (RDF)

• Uses SELECT-FROM-WHERE pattern like SQL

• SELECT ?N
  FROM <http://example.org/data.rdf>
  WHERE { ?X rdf:type voc:Person . ?X voc:name ?N }
Comparing Graph Database Systems: Features

### Data Storage

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>Main memory</th>
<th>External memory</th>
<th>Backend Storage</th>
<th>Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllegroGraph</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>DEX</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Filament</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>G-Store</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>HyperGraphDB</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>InfiniteGraph</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Neo4j</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Sones</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>vertexDB</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

### Operations/Manipulation

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>Data Definition Language</th>
<th>Data Manipulat. Language</th>
<th>Query Language</th>
<th>API</th>
<th>GUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllegroGraph</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>DEX</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Filament</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>G-Store</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>HyperGraphDB</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>InfiniteGraph</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Neo4j</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Sones</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>vertexDB</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

*Note: ● indicates availability.*

[R. Angles, 2012]
Comparing Graph Database Systems: Representation

Graph Data Structures

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>Graphs</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple graphs</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Nested graphs</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Attributed graphs</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Node labeled</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Node attribution</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Directed edges</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Edge labeled</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Edge attribution</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>AllegroGraph</th>
<th>DEX</th>
<th>Filament</th>
<th>G-Store</th>
<th>HyperGraphDB</th>
<th>InfiniteGraph</th>
<th>Neo4j</th>
<th>Sones</th>
<th>vertexDB</th>
</tr>
</thead>
</table>

Entites & Relations

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>Schema</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node types</td>
<td>Property types</td>
<td>Relation types</td>
</tr>
<tr>
<td>Object nodes</td>
<td>Value nodes</td>
<td>Complex nodes</td>
</tr>
<tr>
<td>Complex relations</td>
<td>Simple relations</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"Comparing Graph Database Systems: Representation

Graph Data Structures

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>Graphs</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple graphs</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Nested graphs</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Attributed graphs</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Node labeled</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Node attribution</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Directed edges</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Edge labeled</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Edge attribution</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>AllegroGraph</th>
<th>DEX</th>
<th>Filament</th>
<th>G-Store</th>
<th>HyperGraphDB</th>
<th>InfiniteGraph</th>
<th>Neo4j</th>
<th>Sones</th>
<th>vertexDB</th>
</tr>
</thead>
</table>

Entites & Relations

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>Schema</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node types</td>
<td>Property types</td>
<td>Relation types</td>
</tr>
<tr>
<td>Object nodes</td>
<td>Value nodes</td>
<td>Complex nodes</td>
</tr>
<tr>
<td>Complex relations</td>
<td>Simple relations</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[R. Angles, 2012]
Comparing Graph Database Systems: Queries

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>Type</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllegroGraph</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>DEX</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Filament</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>G-Store</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>HyperGraphDB</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>InfiniteGraph</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Neo4j</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Sones</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>vertexDB</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>Adjacency</th>
<th>Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Node/edge adjacency</td>
<td>k-neighborhood</td>
</tr>
<tr>
<td>Allegro</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>DEX</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Filament</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>G-Store</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>HyperGraph</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>InfiniteGraph</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Neo4j</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Sones</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>vertexDB</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

[R. Angles, 2012]