HyperGraphDB: Model and Applications

- Borislav Iordanov, Kobrix Software, Inc.

- Agenda:
  - About the project
  - Model Overview
  - Implementation Overview
  - Java OODB
  - Natural Language Processing
  - JSON database as a graph
  - OWLAPI and Distributed Ontology Versioning
About the Project

➢ Background: AI & Opencog
  ➢ http://www.opencog.org

➢ Dr. Harold Boley and Directed Recursive Labelnode Hypergraphs, circa 1977

➢ Codebase – a Java library
  ➢ http://www.hyperographdb.org
  ➢ http://code.google.org/p/hypergraphdb

➢ LGPL (minus BerkeleyDB – GPL)

➢ Help?
Data Model Basics

- Directed, typed, higher-order hypergraph:
  - Hyper: edges are sets instead of pairs
  - Directed: edges are ordered sets
  - Higher-order: edges can point to/contain other edges
  - Typed: each entity has a type and a value
- → essentially tuples that refer to each other
- → no fundamental difference between nodes and edges are unified into something called atoms.
Data Model - Terminology

- Atom – the basic entity in storage, the typed tuple.
- TargetSet(x) – the elements of the tuple
- Arity(x) – the size of TargetSet(x)
- Node – tuple with arity 0
- Link (or edge) – tuple with arity > 0
- IncidenceSet(x) – set of atoms pointing to x
- Type – an atom conforming to a special interface
- Value – data managed in storage by a type atom
- Handle – auto generated identifiers used throughout
Overview of APIs

- HyperNode Interface
  - Basic data operations & Querying
  - Implementations: HyperGraph main class, HGSubgraph, PeerHyperNode, Apps

- Querying & Traversals
- Indexing – by properties, by targets, custom
- Typing – customizable type ”tower”
- Transactions – MVCC, nested
- Pluggable Storage
Based on a key-value layer with duplicates support.

Two-layered architecture:

1. Primitive layer: handles (i.e. identifiers) and byte[]
2. Model layer: atoms (runtime database entities)

Handles/Identifiers are custom generated – UUIDs or longs or ints, etc.

A HyperGraphDB database consists of several key-value stores - some core and some user defined indices
Basic Layout

Tuple DB
- AtomID
- TypeID
- Target1ID
- ValueID

<TypeId, ValueId, Target1ID, ..., TargetnID>

TypeID
<TypeConstructorId, DescriptorId, ...>

Target1ID
<Type2Id, Value2Id, TargetnID, ...>

ValueID
<NestedValueID, ...>

Data DB

Value ID
Byte [] primitive data

Incidence Index
- ID
Sorted set of IDs of links pointing to ID

Predefined – Java Class

e.g., incidence set of Targetn:

TargetnID
<AtomId, Target1ID, ...>

Type Index
- Type ID
Sorted set of Atom IDs with that type
Implementation - Caching

⇒ Constant lookup of handle by atom and vice-versa
⇒ Constant traversals within RAM working set
Any Java object can be represented as a HGDB atom, out of the box

No need to implement a special interface for plain nodes

No need for ”primary key” (handles are implicit)

Every Java class/interface mapped to a HGDB type

Inheritance represented with binary links (HGSubsumes)

Customize storage via custom types
Hypergraphs come up frequently:

- Lexical/conceptual networks – WordNet
- Syntact structures in information extractions
- Semantic relationships (e.g. first-order logic representations)
- Knowledge bases
- Code at http://code.google.com/p/disko/ (used for semantic search & chatbot applications)
Apps: JSON Trees

➢ Built on mJson – http://sharegov.org/mjson
  ➢ Single universal type Json for all elements
➢ HyperNode view of the DB
➢ Primitives → immutable nodes with typed values
➢ Arrays → links/tuples of Json elements
➢ Objects → links/tuples of JsonProperty
➢ JsonProperty → link of name (a String atom) and value (a Json element)
{isbn:"978-0768329806",
entity:"book",
title:"Irregular Drinking"
authors:[{name:"Joe Cool",
id:504,
entity:"person"},
{name:"Pete Smart",
id:984,
entity:"person"}]
}
Consequences of "Knowledge Based" View

➢ Pros:
  ➢ Implicit graph indexing of everything!
  ➢ Given any value, or any property, traverse the graph to find (sub)structures that contain it
  ➢ Pattern matching: given a blueprint, fill in the blanks

➢ Cons:
  ➢ Waste of storage if not many duplicates
  ➢ Slower writes to ensure uniqueness
  ➢ Slow reads on deep trees, with few duplicates and frequent cache misses
Apps: Semantic Web

- RDF via Sail (by Ian Holsman)
  → Named graph support – trivial in HGDB
- Topic Maps
  → Natural representation of n-ary associations
- OWL 2.0 (by Thomas Hilpold)
  → Native persistance with small semantic gap:
    - Ontologies : subgraphs
    - Axioms : links
    - Entities : nodes
  → Performance for large ontologies and complex queries
2 Ontologies as subgraphs in the same repository

Axioms/Links (39 types) refer to entities, often indirectly

Description logic expressions as links

Entities atoms are shared b/w subgraphs/ontologies

Garbage collector decides based on roots and incidence

Native Persistence also for: Change Objects, Changesets, Versioned Ontologies
HGOWL Collaboration

- Distributed Revision Control for Ontologies
  - Local: commit, rollback, revert, history
  - P2P: push/pull of changesets
  - Branching and merging

- Available as:
  - Development kit, standalone server
  - Protege Plugin (OSGi)
  - URL: http://sharegov.org/protegehgdb/
HGOWL Pros (no Cons :) 

➢ HGOWL-API Users:
  ➢ Full OWL-API interface with seamless transactional persistence designed with performance in mind
  ➢ Ontology Versioning, P2P capabilities, Ontology Server, …
  ➢ Flexible toolbox as a Protege Plugin

➢ HGOWL-API Developers:
  ➢ Full control over Indexing
  ➢ Query extensibility and type customization
  ➢ OSGi Plugin Development

➢ Home page: http://sharegov.org/protegehgdb
Thanks

Q & A if there's time...and Q's
Email: borislav.iordanov@gmail.com