HyperGraphDB

Data Management for Complex Systems

(by borislav.iordanov@gmail.com)

About the Project

- Background: AI & OpenCog
  - http://www.opencog.org (Dr.Ben Goertzel et al.)
  - Dr.Harold Boley and Directed Recursive Labelnode Hypergraphs, circa 1977
- From prototype to prototype
- Codebase and licensing
  - http://code.google.com/p/hypergraphdb
  - http://www.hypergraphdb.org
  - LGPL (minus Berkeley DB)
- Help?

On Software Complexity

If I talk more than two minutes on this, please stop me…

Check out:

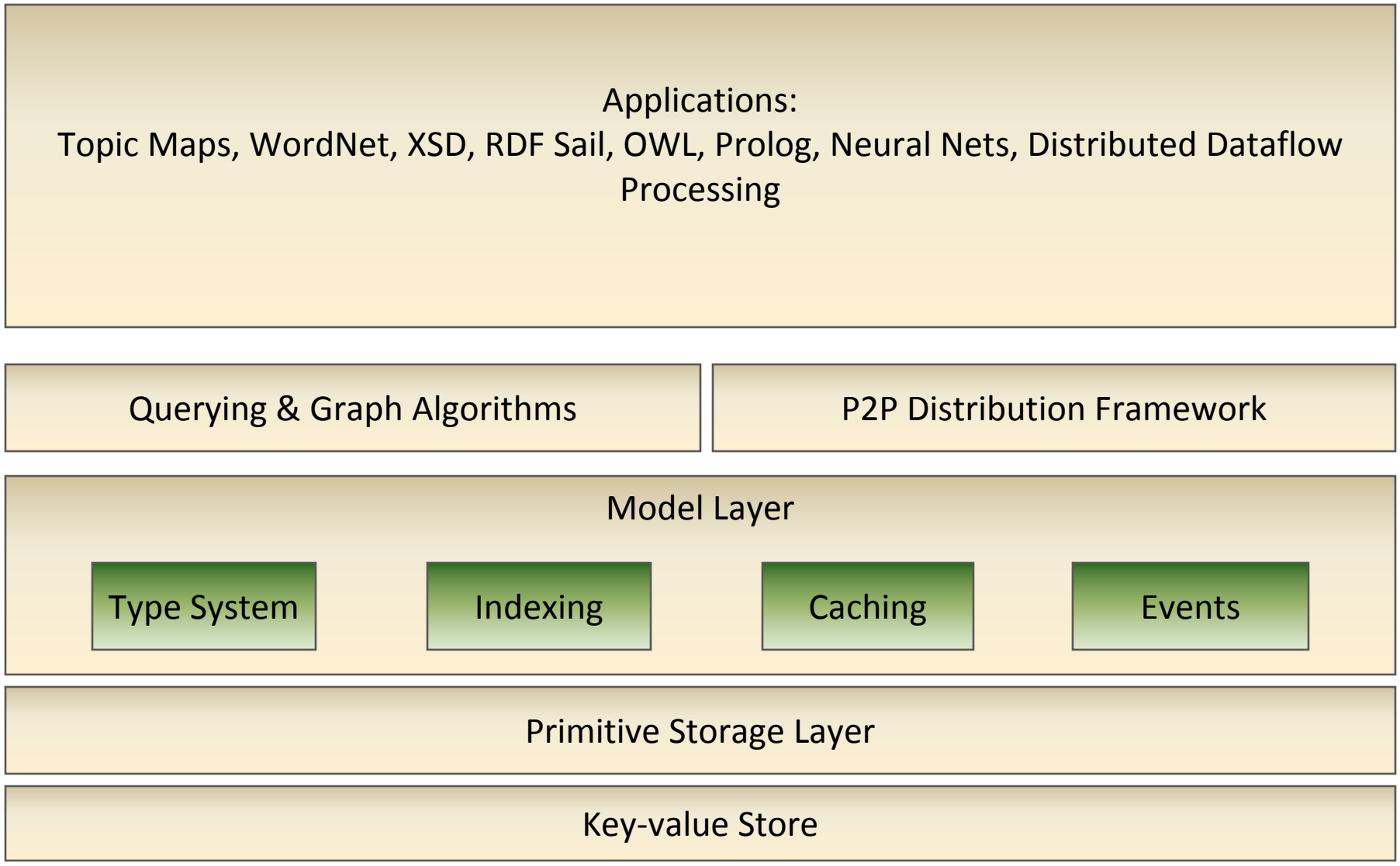
http://kobrix.com/documents/rse.pdf

http://kobrix.com/seco.jsp

Architectural Goals

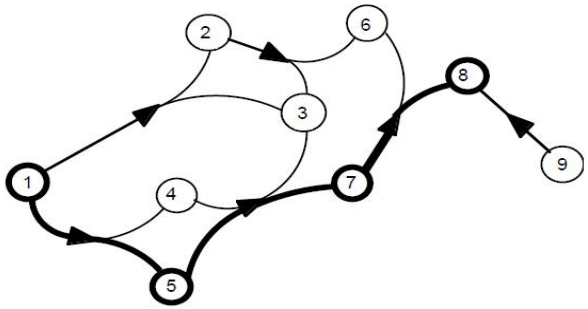
- Minimal API intrusiveness
- Database as a transparent extension of RAM
- Universal identification (independent of atom location)
- Programming language agnostic
- …yet naturally embedded/embeddable into the runtime environment
- “Frameworky” & Open – customize at various levels
- Reflective & Dynamic – as few predefined aspects as possible

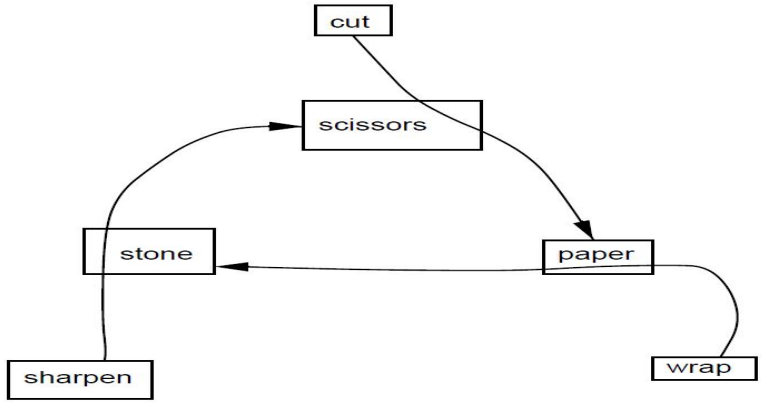
Architecture



What Is a Hypergraph?

- In a graph $G=(V, E)$, V is a set and E a set of pairs $e=\{v_1, v_2\}$ from V.
- Take e to be any subset $\{v_1,\ldots, v_n\}$ of V and you get a **undirected** hypergraph.
- Directed hypergraphs:





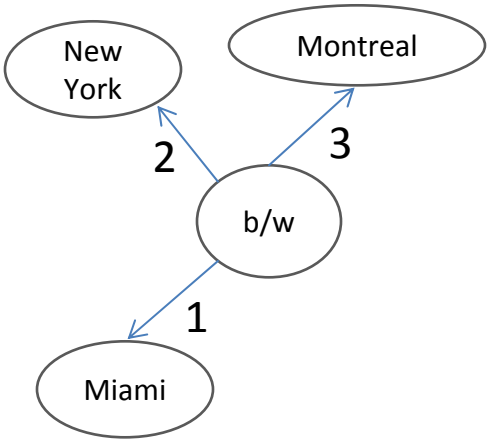
HyperGraphDB Model & Terminology

- V (nodes) + E (edges) = A (atoms)
- Atom = **target set** + "payload", that is
  a tuple of 0 (for **nodes**) or more (for **links**)
  atoms + a typed value
- **Incidence set(x) =** { y | x is in y's target set}, that is all links that "point" to x.
- **Arity(x)** = cardinality of its target set
- **Type(x)** = An atom conforming to a special interface
- **Value(x)** = Arbitrary data managed by Type(x)

Why a n-ary relationships

- Because sometimes more than 2 things stand in a relationship together!
- Examples:
  - between(Miami, New York, Montreal)
  - purchase(Mr. Hip, Apple, iPad, $$$)

Property Graph Model



VS

HyperGraphDB Model

Fewer database entities
More natural to work with API-wise.

Why higher-order relation?

- Because a relation is an abstract entity in a model, i.e. an **entity**…
- Examples:
  - Causal links between events (purchase and lottery win)
  - Contextualization (betweeness depends on transportation means)
  - Instead of sparse foreign keys
  - Standard tree-like structures
  - Rule representation: link premises and conclusion

Atoms and References

- Node atoms: any Java object
- Link atoms: implements HGLink
  - possible to decouple this interface at a small cost
- Universal reference: HGHandle
  - HGLiveHandle points to the Java ref and the…
  - …HGPersistentHandle which is the DB id
  - Valid in a distributed environment
  - Several varieties interacting with caching etc.

$\Rightarrow$ Can't use disk offsets as atom identifiers, key lookup required.
$\Rightarrow$ In RAM, HGHandle dereferencing is member dereferencing at best or hash lookup at worst.

Incidence Sets

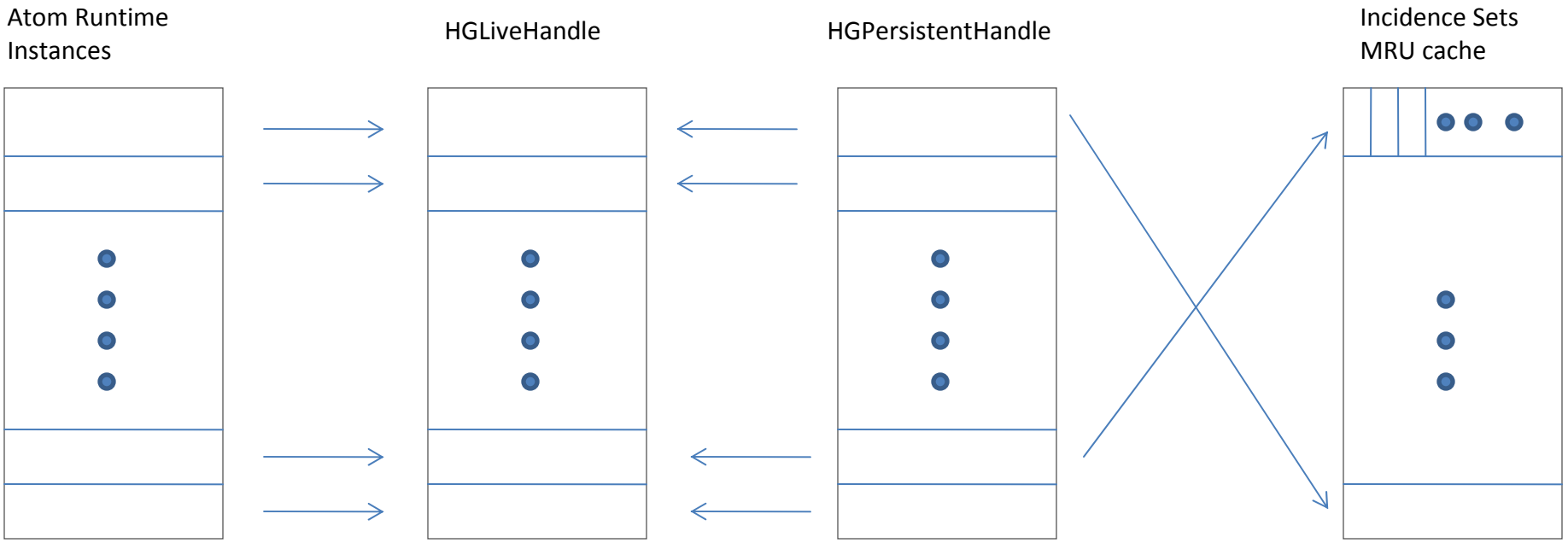
- Given atoms:

  L → *[A, B, C]* and *A* → *[C, D, E]*

The incidence set of *C* is *IS(C)={L, A}*

- To find adjacent atoms, examine all targets of all links in an atom's incidence set
- …so *A* is both an adjacent atom and an incident link to *C*

Caching



⇒ Constant lookup of handle by atom and vice-versa
⇒ Constant traversals within RAM working set

Storage Architecture

- Two layers – primitive and model layer
- Primitive Layer – a low-level graph of identities and raw data
- Model layer – a layout for representing typed hypergraph atoms

Primitive Layer

- A graph of identities and raw, byte[] data
- LinkStore

  ID -> [ID, …., ID]
- DataStore

  ID -> byte[]

  Current IDs are type 4 UUID

Model Layer

- Formalizes layout of primitives:

  AtomID -> [TypeID, ValueID, TargetID, ..., TargetID]

  TypeID := AtomID

  TargetID := AtomID

  ValueID -> [ID, …, ID] | byte[]

- A set of predefined indices:

  IncidenceIndex: AtomID -> SortedSet<AtomID>

  TypeIndex: TypeID -> SortedSet<AtomID>

  ValueIndex: ValueID -> SortedSet<AtomID>

Typing

- Data interpretation
  - Integrity and consistency
  - Customized storage model
  - Dynamic database schema
- Types as atoms
  - Reflectivity: domain model part of the data!
  - Type constructors (types of types) cover any type system
  - Meta data and reasoning: the type *Purchase* is the inverse of the type *Sale*.
- Bootstrapped by a set of predefined types
  - Frozen in cache, configurable, of course
  - Cover the Java object model and more

The HGAtomType Interface

// Create and return the runtime of an atom or some nested value

Object make(HGPersistentHandle handle, HGHandle[] targetSet, Set<HGHandle> incidenceSet)

The storage ValueID

Empty for nodes or nested values

The runtime instance of an atom can depend on its graph connections!

// Call storage layer to serialize instance and return a new or existing ValueID

HGPersistentHandle store(Object instance)

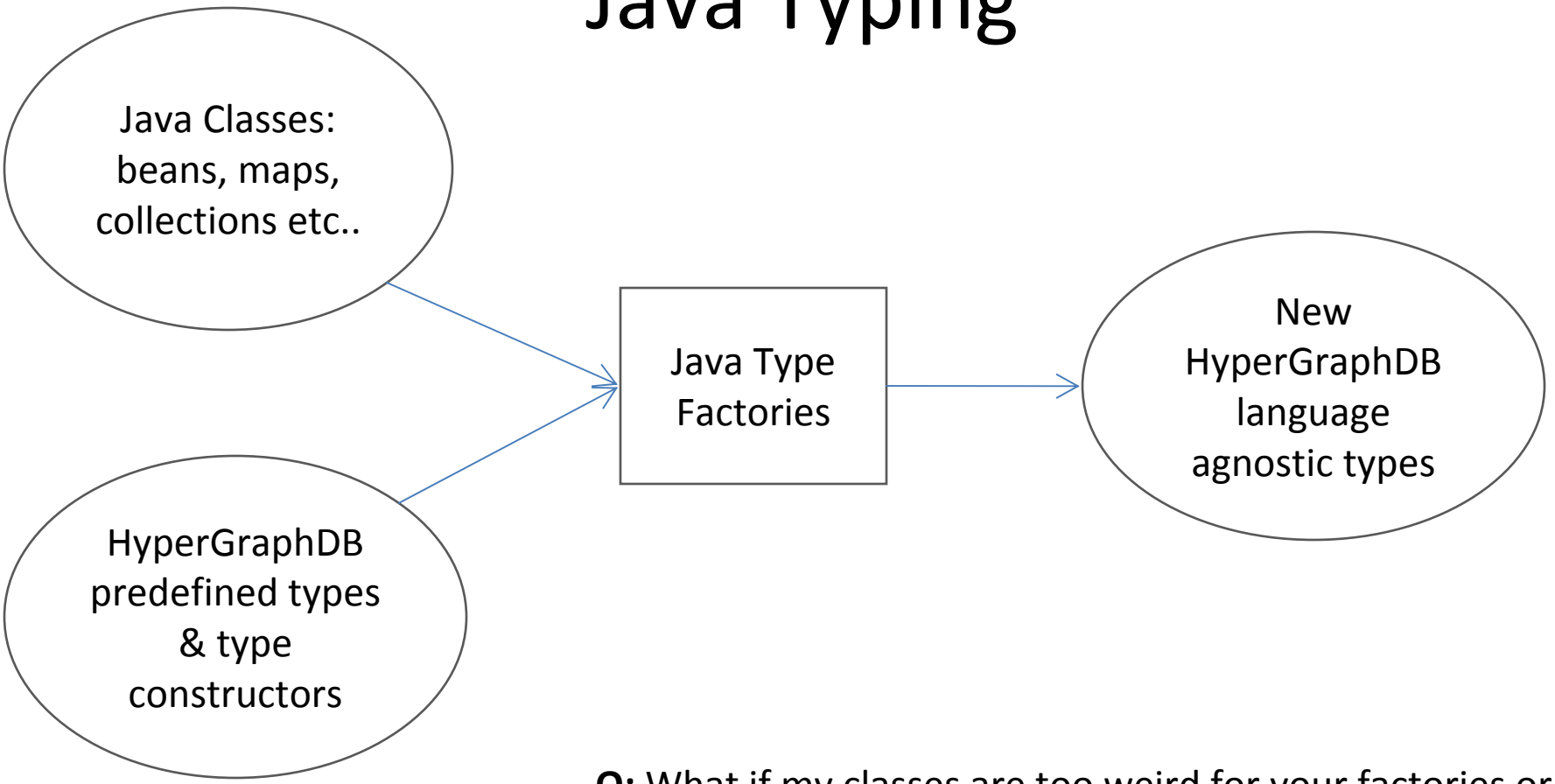
// Remove a value from storage layer given it's ValueID

void release(HGPersistentHandle handle)

// Used mainly as a sub-typing predicate in type constructors

boolean subsumes(Object general, Object specific)

Java Typing



Q: What if my classes are too weird for your factories or if my model is not represented in OO Java?
A: Just write your own HyperGraphDB types.

Q:What about type hierarchies?
A:Represented with a predefined *HGSubsumes* link – remember, types are just atoms.

Indexing

- Associate indexes with atom types – then indexing is automatic
- *HGIndexer*: given an atom, produce a key.
- Out-of-box implementations:
  - object property -> atom
  - target -> atom
  - target -> another target
  - target tuple -> atom
  - multi-key: compose any of the above

Querying

- Traversals – API for standard graph traversals. Hyper-traversals by jumping levels.
- Constrained atom sets (SQL style), API based, lazy evaluation
- (Vaporware) Graph patterns - a new API + comprehensive query language, coming up, looking for help to do it!

Transactions

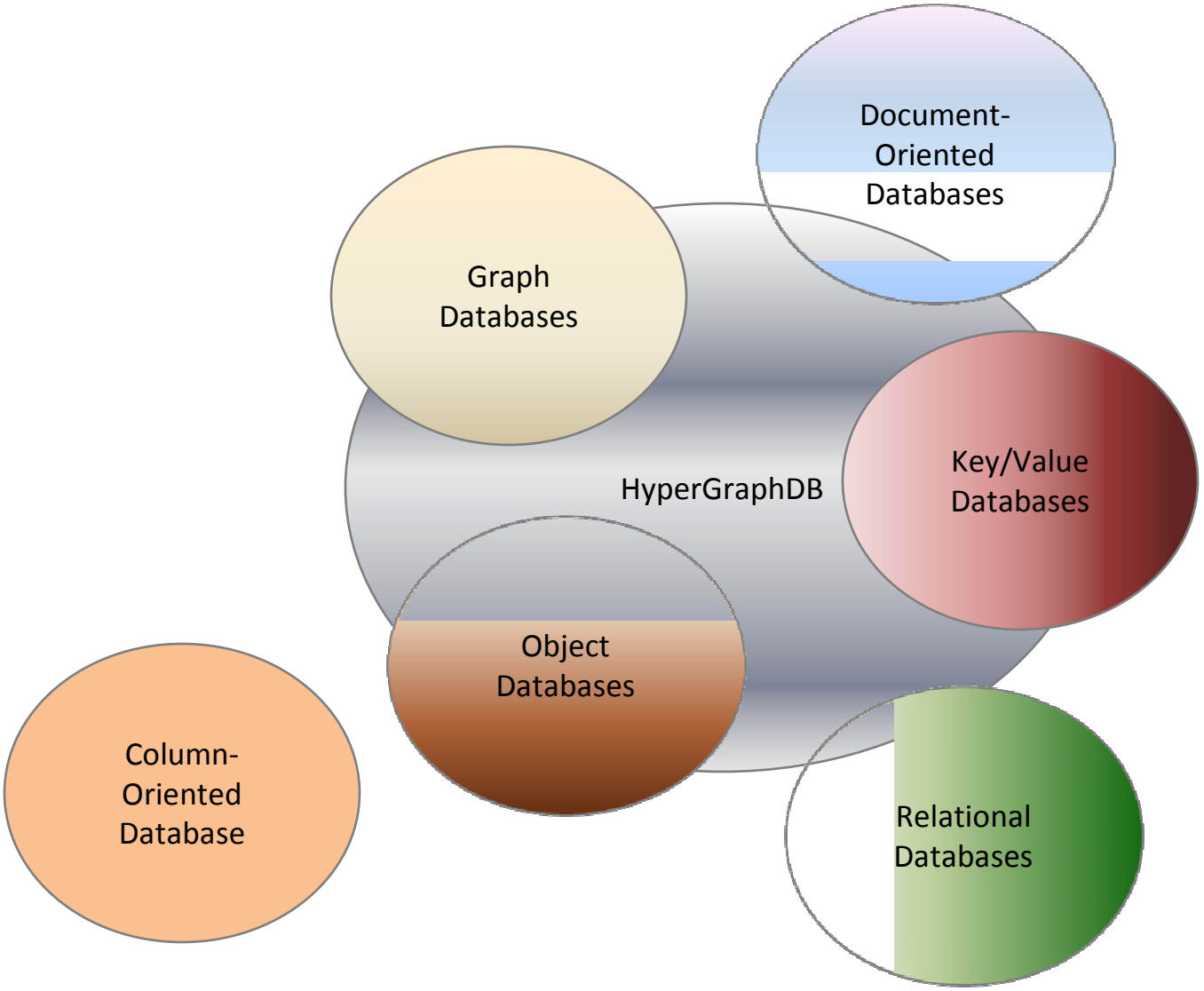
- MVCC in memory and on disk
- ACI(D), nested, thread-bound
- Read-only transactions
  - never conflict
  - very long queries or traversals remain isolated at no additional memory cost!
- Conflicts resolved through retries

  $\Rightarrow$ Be careful of side-effects within your own code!

Distribution

- Build on ACL (Agent Communication Language) foundation
- Pluggable presence&communication layer – XMPP (default), JXTA (available) or your own
- Nested workflows framework for agent (i.e. DB instance) conversations
- Primitive conversations such as subgraph transfer available
- Eventually consistent replication at model layer level.

NOSQL


Document-
Oriented
Databases
Graph
Databases
HyperGraphDB
Key/Value
Databases
Object
Databases
Column-
Oriented
Database
Relational
Databases

Roadmap

- Pattern matching query API & language
- Hypernodes (a.k.a. nested graphs)
- Auto sharding
- Transparent Distributed Queries
- Other Storage Engines
- Alternate Java->HGDB mapping where everything is an atom
- Auto Delete (a.k.a. managed atoms)
- More Runtime Control (e.g. class loading and transactional instrumentation)
- More Apps (e.g. OWL 2)
- Reasoning…maybe

In Summary

- HyperGraphDB -“just” a universal memory model with pointers, types, values and linked structures. Or a generic NOSQL framework.
- Software complexity is in representations – a richer meta model won’t reduce the complexity, but will allow it to breathe rather than suffocate the system.
- AI **is** mainstream…or should be…and will be. (witness the Semantic Web)