HypergraphDB - A Graph Database

First Steps in Using HyperGraphDB

Introduction

HyperGraphDB is an embedded database and as such it is meant to be used within the same process as your application. Therefore, learning to use HyperGraphDB (or HGDB for short) amounts to learning its API and the concepts behind it. The implementation is based on the very robust and ubiquitous BerkeleyDB database. There are very few configuration options that you need to be aware of. And there are virtually no setup/installation steps to get up to speed and use the database. You only need to incorporate the library in your application. In case of problems, such as if your application experiences a serious crash and the default recovery mechanism is not sufficient, you may need to rely on the BerkeleyDB tool suite for troubleshooting. Other than this, you will not need to know about any BerkeleyDB particularities.

While HyperGraphDB aims to be a very general and flexible storage framework in which it is in principle possible to represents various formalisms for organizing data, it can be used "out of the box" as an object-oriented Java database. To ease the introduction into its APIs, the present tutorial focuses mostly on that particular aspect.

The following short sections go through the basics of installing HyperGraphDB, creating a new database instance and playing with fundamental operations such as storing and retrieving information.

The pages are written mostly in tutorial style. And while there are no accompanying sources with all the examples, you are encouraged to copy and paste code from the tutorial and run it and play with it.

7. Graph Traversals
8. Indexing
9. Transaction Essentials

Start Here >>

Installation and Deployment

This page describes the installation of HyperGraphDB from an official release package. If you would like to build and install the system directly from source, please refer to the Installing from Source page. Note that every official release comes with source code, so it is always possible to examine the code, experiment, and apply patches if need be.

HyperGraphDB provides an embedded database library that you can use to create and manage databases in your application. With HyperGraphDB, there is no client/server communication layer, however, it is possible to create distributed databases with additional peer-to-peer (P2P) libraries. HyperGraphDB manages databases through its underlying storage mechanism, specifically Oracle's Berkeley DB (standard edition). HyperGraphDB supports both Berkeley DB's native C interface as well as the Java Edition (JE). The default storage implementation is JE, which performs even better than the native for small to moderate datasets. If you would like to use the native version, then Berkeley DB's native (JNI) library must be included in the system path of your application.

Getting HyperGraphDB with Maven

As of version 1.2 all HyperGraphDB and dependencies are managed in a Maven repository located at http://www.hypergraphdb.org/maven. To links HyperGraphDB to your project through Maven:

Add the HyperGraphDB repository to your pom:

```
 <repositories>
    <repository>
      <id>hypergraphdb</id>
      <url>http://hypergraphdb.org/maven</url>
    </repository>
  </repositories>
```

Add the relevant dependencies (minimal set of components with BerkeleyDB JE storage):

```
    <dependency>
        <groupId>org.hypergraphdb</groupId>
        <artifactId>hgdb</artifactId>
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>org.hypergraphdb</groupId>
        <artifactId>hgbdbje</artifactId>
        <version>1.2</version>
    </dependency>
```

Downloading and Installing HyperGraphDB

All official release packages are located at the Google Code downloads area. The latest build is linked to at HyperGraphDB download page.

HyperGraphDB releases are distributed in compressed archive files (*.tar.gz* for Unix platforms and *.zip* for Windows). To install a distribution, just unzip the archive into a directory of your own choosing.

Distribution Archive Content

The unzipped archive contains the following (where ${version} stands for the current version of this distribution):

readme.html	General information about the HyperGraphDB disribution.
LicensingInformation	The HyperGraphDB license conditions.
lib/hgdb-${version}.jar	Core HyperGraphDB library. You need a storage implementation in your classpath in addition to this jar.
hgdbp2p-${version}.jar	Additional peer-to-peer functionality.
hgbdbje-${version}.jar	Storage implementation based on BerkeleyDB Java Edition. You need the je-${jeversion}.jar in your classpath as well.
hgbdbnative-${version}.jar	Storage implementation based on BerkeleyDB native C library. You need the db-${dbversion}.jar in your classpath as well. At runtime, you need to add the relevant native libraries, depending on your platform, to the java system library path.
db-${dbversion}.jar	The Java bindings for BerkeleyDB native.
je-${jeversion}.jar	The BerkeleyDB Java Edition.
smack-3.1.0.jar	P2P XMPP library needed for the HyperGraphDB peer-to-peer framework.
smackx-3.1.0.jar	P2P XMPP library extensions needed for the HyperGraphDB peer-to-peer framework.
lib/native/	Native libraries needed to run this version of HyperGraphDB. While most Unix environments already have Berkeley DB installed in a standard location, you are strongly advised to use the version provided in the distribution.
apidocs/	API documentation in HTML format - each component has its own folder, hgdb for the core library (you'd mostly need to consult this), p2p for the peer-to-peer framework etc.
src/	The source code used to create the release package. Again each component is in its own subfolder.
ThirdPartyLicensing/	Licensing information for HyperGraphDB dependencies, notably Oracle's Berkeley DB Standard Edition.

Deploying Applications

In total, the HyperGraphDB library consists of its own JAR files, Berkeley DB's *db-${dbversion}.jar* file or *je-${jeversion}*, optionally the third-party JAR files for peer-to-peer communication, and the Berkeley DB JNI library if you're using the native version. The specific JARS required by your application depend on the deployment scenario.

You tell the Java Virtual Machine where to find native libraries in one of two ways:

- Specify the location using the *java* command line option *-Djava.library.path*. For example:

  ```
  java -Djava.library.path=$HGDB_ROOT/lib/native/$PLATFORM
  ```

Here, $HGDB_ROOT refers to your HyperGraphDB installation directory and $PLATFORM refers to your operating system (linux, macos or windows, or their 64bit versions).

- Add the location to the *PATH* environment variable (on Windows) or the *LD_LIBRARY_PATH* environment variable (on Linux/Unix).

NOTE for Windows users:

The Windows native libraries may require the installatioin of the Visual C++ runtime libraries. If you get a runtime error pertaining to DLL loading, try installing the MSVC++ redistribution package. As the location changes often, we're not providing a link, but you can easily search for it.

Configuration

As an embedded database, HyperGraphDB does not require any configuration outside of a few parameters that you can set at runtime. In fact, only a directory location where data will be stored is required. However, when deploying a node within a distributed environment, you will need to extra configuration depending on your particular deployment scenario. For more information, consult the DistributedHyperGraph Creating Distributed Databases] topic.

You configure HyperGraphDB by setting parameters to a HGConfiguration instance before opening a database. One important parameter there is the storage implementation (see *setStoreImplementation* method). By default, BerkeleyDB JE is used. You can also specify a different implementation class as a command line parameter:

-
Dorg.hypergraphdb.storage.HGStoreImplementation=org.hypergraphdb.storage.bdb.BDBStorageImplementation

The above will force HyperGraphDB to use the BerkeleyDB native implementation.

Next - Creating a Database >>

Creating a Database.

A HyperGraphDB database is an instance of the HyperGraph class

HyperGraphDB manages storage as a set of files in a directory. To create a new database, you need to designate a directory that will hold the data and write some Java code that creates and initializes a database instance in that directory. Here's an example:

```
import org.hypergraphdb.*; // top-level API classes are in this
package

public class HGDBCreateSample
{
    public static void main(String [] args)
    {
        String databaseLocation = args[0];
        HyperGraph graph;
        // ...
        try
        {
            graph = new HyperGraph(databaseLocation);
        }
        catch (Throwable t)
        {
            t.printStackTrace();
        }
        finally
        {
            graph.close();
        }
    }
}
```

As you can see, creating a database amounts to creating a new *HyperGraph* instance. If the database does not exist, it will be created. If it does exist, it will be opened. So the same code is used to create or open a database.

As an opened database holds operating systems resources open, it is wise to make sure it is closed in a finally block. It is also very important to properly close a database in order to avoid any data loss or corruption of the underlying low-level storage. HyperGraphDB may throw exceptions, but very few of the API methods throw checked exceptions. Usually the exception thrown will be a *HGException*, possibly wrapping some underlying cause.

Prefer the HGEnvironment for Managing Database Instances

The HyperGraph class in the code above is the main entry point into the database API. It represents a single database. While creating/opening a database by calling the *HyperGraph* constructor is a valid approach, it may be preferable to rely on the HGEnvironment class for such high-level operations. Opening a database is instead done thus:

```
....
HyperGraph graph = HGEnvironment.get(databaseLocation);
....
```

The main different between a call to *HGEnvironment.get* and a call to *new HyperGraph* is that the former will return an already opened database instance at that location. The *HGEnvironment* class maintains a static map of all databases currently open. If you want an "open if exists, otherwise throw an exception" behavior, *HGEnvironment* provides it through the *getExistingOnly* method.

Closing Databases

Note on closing databases - it is generally recommended that you close a HyperGraph instance as soon as you're done with it. However, in many cases databases remain open for the lifetime of an application. To ensure proper closing in such cases, you only need to make sure that the application exits gracefully. The

HGEnvironment registers a shutdown hook with the Java Virtual Machine to properly close all databases that are still open. A slight disadvantage is that sometimes exiting your application may take longer than usual because the system may be flushing caches and writing some remaining transactions to disk.

Configuration Options

It is possible to configure several of the runtime properties of a database instance when you open it. Some of those properties have to do with how the database will behave during the entire session, others affect only the startup process.

Configuration options are specified by create an instance of the HGConfiguration.class class and passing it as an extra parameter to the *HGEnvironment.get* method:

```
HGConfiguration config = new HGConfiguration();
config.setTransactional(false);
config.setSkipOpenedEvent(true);
HyperGraph graph = HGEnvironment.get(location, config);
```

The above opens (or creates if none exists at that location) a database without triggering the predefined HGOpenedEvent. In addition, the database will ignore any transactional directives - code that uses the transactions API, like much of the HyperGraphDB code itself, will work by using NOP stubs in place of true transactions.

Naturally, configuration options are ignored in calls to *HGEnvironment.get* for databases that have already been opened.

Note: there are no creation time configuration options at this point, only runtime options which you can vary every time you open your database instance.

<< Prev - Installing HyperGraphDB Next - Storing Data >>

Storing Data in a HyperGraphDB.

Store by Adding Any Java Object to a DB Instance

The short story is that you can just put in whatever object you have in a HyperGraphDB database:

```
HyperGraph graph = HGEnvironment("c:/temp/test_hgdb");
String x = "Hello World";
Book mybook = new Book("Critique of Pure Reason", "E. Kant");

graph.add(x);
HGHandle bookHandle = graph.add(mybook);
graph.add(new double [] {0.9, 0.1, 4.3434});
```

There is nothing "graphish" in this example - we are just adding some data in our database. The system will generally do the right thing and store your data in a way that would make it possible to reconstruct an equivalent run-time object from storage later. Storage of concrete values is handled by the HyperGraphDB type system. The type system is a completely customizable layer, but it was built to do the sensible thing by default. And in Java, the sensible thing can be summarized as follows:

- Translate primitive types and primitive arrays into byte buffers following common industry format.
- Interpret Java beans as record-like structures, according to the Java Beans conventions.
- Store built-in arrays and collections as sequence of objects.
- Store maps as sequences of pairs of objects.
- Record type inheritance information.
- Some sensible details that you will learn in the course of usage ;)

Note that when *mybook* is added, we assign the result to a variable of type *HGHandle*. This is essentially an identifier within the HyperGraphDB system, explained in more detail in the Atom Handles topic. A newly added object will have its handle/identifier assigned by the system. If a Java object is already known to the system, you can update it in storage without passing the handle. But you need the handle when you want to delete

some data:

```
...
// Add a new object to the database
HGHandle bookHandle = graph.add(mybook);

// At a later point, update an existing object's value:
mybook.setYearPublished(1988);
graph.update(mybook);

// At a later point, delete the object:
graph.remove(bookHandle);
```

Note that the *HGHandle* is needed to remove the object. You can also replace the object identified by that handle with a completely new object:

```
graph.replace(bookHandle, new Book(....));
```

The Storage Model

Every storage medium has a meta-model, a predefined structure of what things can be stored in it. A file is a sequence of bytes. A relational database is a set of relations (or "tables"). An XML file is a tree of markup elements. A HyperGraphDB database is a generalized graph of entities. The generalization is two-fold:

1. Links/edges "point to" an arbitrary number of elements instead of just two as in regular graphs
2. Links can be pointed to by other links as well.

Everything that gets stored in a HyperGraphDB is either a node or a link. A node is simply some object value that does not represent a relationship, it doesn't point to anything else. A **link** is HyperGraphDB's terminology for a graph edge or arrow. A link holds pointers to other entities in the graph and in general represents some sort of relationship. In all cases, we refer to the things in HyperGraphDB as **atoms**. When an atoms is a link, we call the number of atoms it points to its **arity**. Thus, nodes are atoms that have arity 0.

As a final piece of terminology pertaining to graph linkage, we call the set of all links

pointing to a given atom its **incidence set** and we call the set of atoms a given link points to its **target set**. If L is a link pointing to the atom A, we say that L is **incident** to A and that A is a **target** of L.

From a Java perspective, an atom is simply some object. From a conceptual perspective, an atom is an entity that can be related to other entities or that can represent such a relation. The primary role of HyperGraphDB as a database is to store entities networked together by relationships. There are no restrictions on what kind of data is bound to an entity or to a relationship. That is, the data associated with a HyperGraphDB atom can be anything. And because links can point to other links, higher order relationships (i.e. relationships between relationships) are possible. In effect, relations in HyperGraphDB are automatically reified.

Storing Graphs

The example above showed storing an atom in the database amounts to calling the *HyperGraph.add* method. To store an actual graph structure in a HyperGraphDB database, you would need to create links between atoms. HyperGraphDB links are objects that implement the HGLink interface. So to store a link between two entities (i.e. *atoms*), you would simply add an atom that is an instance of the *HGLink* interface.

There are a couple of general default implementations of the *HGLink* interface:

- The HGPlainLink class is a link implementation that carries no additional information with. It contains no additional data and its type is general with no special meaning.
- The HGValueLink class is similar to *HGPlainLink*, but it allows you to embed an arbitrary Java object in it as "payload". Instances of such links will take on the type of the "payload" object.
- The HGRel class is a labeled link implementation with type constraints to its target set (i.e. its arguments). Use of this type of links is documented in this topic.

Here's an example of using the *HGValueLink*:

```
HyperGraph graph = HGEnvironment("c:/temp/test_hgdb");
Book mybook = new Book("Critique of Pure Reason", "E. Kant");
HGHandle bookHandle = graph.add(mybook);
HGHandle priceHandle = graph.add(9.95);
```

```
  HGValueLink link = new HGValueLink("book_price", bookHandle,
priceHandle);
```

While in general HyperGraphDB strives for a minimal API intrusiveness, implementing meaningful representations is best done by defining your own *HGLink* implementations. Other predefined links with specific semantics and some of which are used by HyperGraphDB itself can be found in the atom package.

Note the use of the HGHandle interface in the above example. Handles are the basic means to refer to atoms in HyperGraphDB as detailed in this topic.

<< Prev - Creating a Database Next - Atom Handles >>

HypergraphDB - A Graph Database

One refers to object on a memory heap with memory addresses. In a relational database, one uses primary keys. On the internet, it's URIs. With HyperGraphDB, one uses HyperGraphDB handles - instances of HGHandle. When you add an atom with the *HyperGraph.add* method, you get back a *HGHandle*. When you query for atoms, you get back a set of handles. HyperGraphDB links also point to atoms by using handles. In other words, handles are to HyperGraphDB what object references are to Java or pointers to C/C++.

If you have a *HGHandle*, you can get the actual atom by calling *HyperGraph.get*. For example:

```
HGHandle handle = // some means to obtain a handle to an atom
Book book = (Book)graph.get(handle);
```

In general, when working with objects that reside in HyperGraphDB, it is preferable to use handles and retrieve the actual object on a need-by-need basis. This recommendation is based on:

1. The way caching of objects works in HyperGraphDB. The system will cache all currently loaded atoms and maintain a map b/w *HGHandle*s and Java object references. When a Java object is garbage collected, the cache clears its maps from it after a while. But the handle remains valid and the object will be automatically re-loaded upon the next *HyperGraph.get(handle)* request.
2. The fact that links in the graph are based on handles, not plain Java references. The *HyperGraph.get* method is pretty fast (most of the time it doesn't even involve a hash lookup), so it's ok to use *HGHandle* as your way of referring to objects stored in HyperGraphDB instead of plain Java references.

You will notice that HGHandle is actually just a marker interface. There are several implementation of it that may change in the future, so you shouldn't be relying on any concrete implementation. The only time you may be interested in a variety of a HyperGraphDB handle is if you want to persist the handle somewhere else. In those cases, you can get a persistent version of it by calling *HyperGraph.getPersistentHandle(handle)*. Persistent handles are represented by the

HGPersistentHandle interface. They can be converted to byte buffers and then read back with HGHandleFactory.makeHandle.

From Objects to Handles

If you have a Java object reference that is also a HyperGraphDB atom, you can obtain its handle through the *HyperGraph.getHandle(object)* method. Thus despite the advice given above, you're not required to refer to hypergraph atoms only through their handles. It would be cumbersome to have to obtain the object with calls to *HyperGraph.get* every time you need it.

However, you must be careful that the object whose handle you're trying to obtain is actually in the HyperGraphDB cache. It will be there either if it was loaded from HyperGraphDB through a previous *get* or if it was just recently added by a call to *add*.

In general, you would use the Java object reference whenever you have to work with the actual data value and/or you want to isolate an application layer from the HyperGraphDB API. And you would use the *HGHandle* interface when you are mindful of memory consumption and/or you are working mainly with the graph structure of your data, rather than individual atoms.

<< Prev - Storing Data Next - Atom Types >>

Atom types

HyperGraphDB is a Typed World

Understanding the details of the typing system is not essential for many uses of HGDB, but it is always helpful to have a general intuition of how things work in a piece of software so we present a brief introduction here.

Every value stored in HyperGraphDB is typed. All programming languages and database systems have types in one form or another, but what sets HyperGraphDB apart is that types themselves are atoms in the graph just like regular data. This makes HyperGraphDB into a reflexive database. So the type of every atom is also an atom and, as all atoms, is identified with a *HGHandle*, it is indexed, cached, can participate in graph relationships etc. The HGDB type is a different entity than the Java class of an atom. In fact, the Java class of an atom and the HGDB type of the same atom can be completely unrelated. However, in practice there is a correspondence between the two in that to most Java classes with instances stored as HGDB atoms, by default there are HGDB types that manage the low-level storage. And in most cases in practice you don't even need to know the *HGHandle* of an atom's type since the API in general accepts a Java *Class* and looks up the corresponding HGDB type automatically.

When you add an atom with code like this:

```
A a = new A(....);
graph.add(a);
```

the first thing the system does is try to find out whether there's a HGDB type corresponding to the Java class *A*. And if there is already a HGDB type associated with the class *A* it will simply use it. If not, it will create such a type and store it as a new *type atom* before continuing with the storage of *a*. This new atom type will be henceforth associated with the Java class *A.class* and reused when more instances of this class are added to the graph. To obtain the handle of the HGDB type corresponding to a Java class, you can make the following call:

```
HGHandle handleA =
graph.getTypeSystem().getTypeHandle(A.class);
```

You can then link to that handle or use wherever a type atom is expected. If you already have a type handle and you'd like to store a particular atom under that type, call:

```
A a = new A(...);
HGHandle typeHandle = ...//some means to obtain the atom type
graph.add(a, typeHandle);
```

This will work even if there's already some different atom type associated with the class *A*. Well, it will work provided the atom type identified by the *typeHandle* variable knows how to deal with Java instances of *A*. In fact, the main responsibility of an atom type is serializing runtime objects into a low-level database representation and converting them back from this low-level representation into runtime objects. Types are implementations of the HGAtomType interface. Take a look at the methods of that interface - they resemble a conventional CRUD API.

Since we've mentioned that HGDB types are "just" atoms, you may wonder what the types of those atom types are. They are atoms too, of course. The system is bootstrapped with a set of predefined types whose type is the special type *top* from HGTypeSystem class. But what about the type of a type created on the fly? For example, upon adding an instance of *A* above for the first time, a HGDB type corresponding to *A.class* is created and stored in the database, as an atom. The type of this newly created atom type is not *top*, but rather something called a *type constructor* : an atom type whose value range consists of other atom types. If you are familiar with functional programming or C++ templates, the concept shouldn't sound foreign.

For practical purposes, there are predefined implementations to handle Java POJOs, arrays, collections, maps and objects implementing the *java.io.Serializable* interface. If you want to store a Java objects that is neither serializable, nor does it follow the Java Beans naming conventions to expose its state, then you will need to develop a custom type for that Java class. Developing a custom type amounts to implementing the *HGAtomType* interface and is not hard at all - see the writing a custom type topic.

Once you've developed a custom type, you can plug it into the system with a call to *HGTypeSystem.addPredefinedType*.

As a last point about type management in HyperGraphDB, we note that inheritance is represented by special purpose link atoms of type HGSubsumes. The system uses such links when asked to, for example, to find all atoms of a given type or of any of its sub-types. *HGSubsumes* links are created between a class and its superclass and between a class and all of the interfaces it implements.

<< Prev - Atom Handles Next - Querying >>

Querying in HyperGraphDB.

There is no special purpose query language for HyperGraphDB, yet. Therefore, querying for atoms is performed with a special purpose API. Like in many other database systems, the API is based on conditional expressions that you create, submit to the query system and get back a set of atoms as the result.

Query API Overview

The conditional expressions for querying are build up out of classes in the org.hypergraphdb.query package. Some of the conditions apply to atom values (the Java object) while others apply to the graph structure (how atoms are link between each other). You can create a query condition by instantiating those classes. Here is an example:

```
    HGQueryCondition condition = new And(
              new AtomTypeCondition(Book.class),
              new AtomPartCondition(new String[]{"author"},
"George Bush", ComparisonOperator.EQ));
    HGSearchResult<HGHandle> rs = graph.find(condition);
    try
    {
        while (rs.hasNext())
        {
            HGHandle current = rs.next();
            Book book = graph.get(current);
            System.out.println(book.geTitle());
        }
    }
    finally
    {
        rs.close();
    }
```

The condition should be self-explanatory: it asks for atoms of type Book and whose author is George Bush. The AtomPartCondition lets you constrain the value of an atom by one of its object properties.

As you can see, the result is returned in the form of HGSearchResult instance. Search results are basically bi-directional iterators (you can go back to a previous item) that you need to close once you are done with them, similarly to a JDBC *ResultSet* object. If a search result is not closed, it will maintain locks on certain parts of the database which in turn is likely to cause a deadlock in your application.

That's simple enough, albeit a bit verbose. But there is a much more convenient syntax to work with HyperGraphDB queries. The HGQuery.hg class defines an extensive list of so called "factory methods" (methods that instantiate objects in lieu of constructing them directly) for creating and composing query conditions. It also defines several other general purpose utility methods which you are encouraged to examine.

Here is the above condition rewritten in this style:

```
// Use Java 5 static import facility to import the HGQuery.hg
// namespace.
import org.hypergraphdb.HGQuery.hg;

//
// Given some HyperGraph instance that has some Books added to
it.
//
HyperGraph graph = new HyperGraph(tutorialHyperGraphLocation);
List<Book> books = hg.getAll(hg.and(hg.type(Book.class),
hg.eq("author", "George Bush")));
for (Book b : books)
    System.out.println(b.getTitle());
```

For example, the *hg.and* takes an arbitrary number of arguments, each a query condition and returns a composite condition representing the conjuction of all sub-conditions. There are several methods for comparing values: *hg.eq*, *hg.lt*, *hg.gte* etc.

and they all will construct a corresponding condition for comparing against an atom's value or an atom's property value.

Of note in the above example is the *hg.getAll* method. It will load from storage and put in a standard Java list all atoms of the search result set. This is convenient when you don't expect a large result set or when you need the whole of it in RAM anyway. The *hg.findAll* method does the same, but it returns a list of *HGHandle*s instead of pre-loading the atom from disk.

You can create query objects and execute them repeatedly:

```
HGQuery query = HGQuery.make(graph, condition);
rs = query.execute();
//etc…
```

The advantage of creating a HGQuery object is that the condition is analyzed and a query plan is created and stored as part of that object, thus increasing performance each time the query is executed.

The following sections give a list of common query factory methods thematically grouped.

Logical Operators

The three standard logical operators *and*, *or* and *not* are available:

hg Factory Method	Description
and(c1, c2, ...)	performs a logical conjunction of conditions c1, c2...etc
or(c1, c2, ...)	performs a logical disjunction of conditions c1, c2...etc
not(c)	performs a logical negation of the conditions c

Querying Atom Values

The usual comparison operators are available when constraining the values of atoms in the result set. They are "enumed" in the ComparisonOperator class. For every operator, the *HGQuery.hg* supplies a factory method as a shortcut. In addition, value comparing factory methods come in two flavors: comparing atom values or comparing atom property values (a.k.a. bean properties in the Java world). Here's a

summary:

hg Factory Method	Description
value(x, op)	compare the atom's value against x using the ComparisonOperator op
part(path, x, op)	compare the atom's property referred by *path* against x using the ComparisonOperator op
eq(x), eq(path, x)	the atom (or its property identified by *path*) must be equal to x
lt(x), lt(path, x)	the atom (or its property identified by *path*) must be less than x
gt(x), gt(path, x)	the atom (or its property identified by *path*) must be greater than x
lte(x), lte(path, x)	the atom (or its property identified by *path*) must be less than or equal to x
gte(x), gte(path, x)	the atom (or its property identified by *path*) must be greater than equal to x

Naturally, the operators lt, lte, gt and gte can only be applied to values that are comparable.

Querying Graph Structure

A few conditions deal with the linkage between atoms. You can constrain an atom to be a link having a certain form or to be the target of a link. Here is table summarizing those conditions:

hg Factory Method	Description
target(x)	the atom must be a target of the link x.
incident(x)	the atom must be a link pointing to x.
link(h1, h2, ...)	the atom must be a link whose target set includes atoms h1, h2...etc.
orderedLink(h1, h2, ...)	the atom must be a link whose target set is ordered and where h1, h2...etc appear in exactly that order.
arity(n)	the atom has arity n.
disconnected()	the atom has no links pointing to it.

Note that:

```
    hg.link(h1, h2, h3) == hg.and(hg.incident(h1),
hg.incident(h2), hg.incident(h3))
```

In fact, this is how link lookups are implemented: by intersecting incidence sets of the desired link target set.

Typing Constraints

Whether you are querying atoms by their value or relationships in the graph, it is frequently necessary to include a typing constraint. For example, given an *AtomValueCondition*, the query system will easily infer the desired atom type from the Java class of the value. But this is impossible with a single *AtomPartCondition* for example.

Asking for atoms with a specific type is akin to a *from* clause in an SQL query because all atoms are automatically indexed by their type. This is done with *hg.type* factory method. If you don't care about specific type, but only about a super-type (i.e. base class or an interface in Java), you can use the *hg.typePlus* condition.

hg Factory Method	Description
type(class), type(handle)	the atom must have the specified type, either as a Java class or as a HGDB handle
typePlus(class), typePlus(handle)	the atom's must be either of the specified type or a of a type inheriting the specified type
subsumes(x)	the atom must subsume (i.e. be more general) the atom specified in the condition
subsumed(x)	the atom must **be** subsumed (i.e. be more specific) by the atom specified in the condition

<< Prev - Atom Types Next - Graph Traversals >>

Graph Traversals

One of the main advantages of HyperGraphDB as a database is its ability to store and manage very large graphs of relationships. It is mixing this and regular relational and object-oriented style databases that makes HyperGraphDB a powerful tool for information management and knowledge representation. So let's examine the basic APIs for walking around a HyperGraph and the algorithms provided out of the box. All interfaces and classes talked about in this section are in the org.hypergraphdb.algorithms package.

At the foundation of all graph related algorithms lies the basic operation of walking from node to node, or in HyperGraphDB lingo form atom to atom. This is represented by the HGTraversal interface. Traversing the graph is always done in some particular order, depending on the concrete traversing algorithm. A traversing algorithm simply produces atoms in a sequence and the *HGTraversal* interface is a specialization of the standard *java.util.Iterator* interface. The two standard graph traversal algorithms are implemented by the *HGBreadthFirstTraversal* and *HGDepthFirstTraversal* respectively.

Now, in HyperGraphDB all links are typed and have an arbitrary object attached to them. More importantly, each atom can conceptually participate in several independent structures in a graph. Therefore it is not immediately obvious which links should a traversal follow when walking from atom to atom. An essential component of the traversing algorithms in HyperGraphDB is that very decision: which of the adjacent atoms should be next visited. That decision is delegated to the user of a traversal algorithm - you :)

The adjacent atoms that must be visited are calculated by implementations of the HGALGenerator interface. Both the breadth-first and depth-first traversal implementations take an *HGALGenerator* instance as a constructor argument. The simplest generator is the SimpleALGenerator implementation. This implementation will list all adjacent atoms, disregarding types and values of atoms:

```
HGHandle myBook = ...// get the handle of a book of interest.
```

```
HGDepthFirstTraversal traversal =
    new HGDepthFirstTraversal(myBook, new
SimpleALGenerator(graph));

while (traversal.hasNext())
{
    Pair<HGHandle, HGHandle> current = traversal.next();
    HGLink l = (HGLink)graph.get(current.getFirst());
    Object atom = graph.get(current.getSecond());
    System.out.println("Visiting atom " + atom +
                       " pointed to by " + l);
}
```

Notice that the next method of the *HGTraversal* interface returns a pair of objects. The first element of this pair is the link that led to the current atom while the second is the atom itself.

A more interesting *HGALGenerator* is the DefaultALGenerator interface. The *DefaultALGenerator* implementation allows control over most aspects of graph link-atom selection that one would encounter in practice. The full constructor signature looks like this:

```
DefaultALGenerator(HyperGraph graph,
                   HGAtomPredicate linkPredicate,
                   HGAtomPredicate siblingPredicate,
                   boolean returnPreceeding,
                   boolean returnSucceeding,
                   boolean reverseOrder)
```

As you can see, the *DefaultALGenerator* can be configured with a predicate that constraints the links to be selected during traversal and a predicate that filters the atoms to be traversed. In addition, one can also specify the order in which atoms linked by a given link are visited. In classical graphs where all links are of arity 2, the order amounts to specifying directionality of the links of interest. In hypergraph, where links can potentially tie together tens of atoms, the order may be important. It

all depends on the representation of your domain. Suppose that the current atom during a traversal is *x* and that x is the target of some link that looks like this:

```
L = [a, b, c, x, d, e]
```

The AL generator will first check whether the link *L* satisfies the *linkPredicate*. Then if *returnPreceeding* is true it will examine each of the atoms a, b and c as potential siblings and it will returns all those that satisfy the *siblingPredicate*. Similarly, if *returnSucceeding* is true, it will examine and maybe return the atoms d and e. When *reverseOrder* is true, the AL generator considers the link L as if it had the following form:

```
L = [e, d, x, c, b, a]
```

This reversed order affects which siblings are first returned. As an example, imagine a graph where you have all sorts of publications (books, articles, blogs) as nodes and you have links of type CitedBy(X, Y) which means that publication X contains a quote from publication Y. To traverse all citations of articles published in the *Science* magazine, you could perform the following:

```
   DefaultALGenerator algen = new DefaultALGenerator(graph,

hg.type(CitedBy.class),

hg.and(hg.type(ScientificArticle.class),

hg.eq("publication", "Science")),
                                                        true,
                                                        false,
                                                        false);
   HGTraversal traversal = new
HGBreadthFirstTraversal(startingArticle, algen);
   ScientificArticle currentArticle = startingArticle;
   while (traversal.hasNext())
   {
       Pair<HGHandle, HGHandle> next = traversal.next();
```

```
        ScientificArticle nextArticle =
graph.get(next.getSecond());
        System.out.println("Article " + current + " quotes " +
nextArticle);
        currentArticle = nextArticle;
    }
```

<< Prev - Querying Next - Indexing >>

How to use indices

HyperGraphDB allows you to index atoms by their attributes. Internally, various indices are maintained around the basic organizational layout of the hypergraph data. For example, because every atoms X has an associated incidence set holding all links pointing to it, the set of those links is readily available and can be efficiently intersected with other incidence sets. But to quickly retrieve a set of atoms based on their values, one needs to explicitly create an index.

At the lowest level, indices are just key-value tables that the storage layer manages. There are also bi-directional indices where a set of keys matching a given value can be retrieved. Some type implementations work directly with the storage layer to maintain internal indices normally hidden from the user. Such internal indices are of no concern to us here. Suffice it to mention that given a unique name, you can create an index using the HGStore API and then put whatever you want in it as long as you can translated your data to/from byte buffers.

Indexing at the level of atoms is supported by an HGIndexManager that is associated with every *HyperGraph* instance. Every time an atom is added, removed or replaced, the *HyperGraph* will trigger an event with its *HGIndexManager* to update all relevant indices.

Indices themselves are created by registering *indexers*, which are implementations of the HGIndexer class, with the index manager. An *HGIndexer* is essentially responsible for creating a key given an atom. It is always associated with a specific atom type. So indices are always type-based. Moreover, sub-types are automatically indexed when an index is registered for a super-type.

In practice, the two most frequently used *HGIndexer* implementations are ByPartIndexer and ByTargetIndexer. The *ByPartIndexer* lets you create an index based on some atom property. For example if you have a *SiteUser* Java bean, with a bean property called *email*, you can index all users by their email like this:

```
HGHandle siteUserType =
graph.getTypeSystem().getTypeHandle(SiteUser.class);
graph.getIndexManager().register(new ByPartIndexer(siteUserType,
"email");
```

Now, when you query for site users by email (e.g. *hg.and(hg.type(SiteUser.class), hg.eq("email", "bill@microsoft.com"))*), the index will be used.

The *ByTargetIndexer* lets you index links by targets at specific positions. Take the predefined *HGSubsumes* link as an example which links something *general* (target at position 0) to something *specific* (target at position 1). You can index all *HGSubsumes* atoms by their second target like this:

```
graph.getIndexManager().register(new
ByTargetIndexer(graph.getTypeSystem().getTypeHandle(HGSubsumes.class),
 1));
```

Note that indexing by link targets is only useful when doing queries on ordered links. Otherwise, for unordered links the implicit indexing by incidence sets suffices. To take advantage of the index above, you would write a query like this:

```
List<HGHandle> L = hg.findAll(hg.and(hg.type(HGSubsumes.class),
hg.orderedLink(hg.anyHandle(), someHandle)));
```

Note the use of *hg.anyHandle()* at position 0 of the ordered link condition. It is important to be explicit about the exact form of an ordered link in your query. Otherwise, the query system will not be able to associate the provided value (*someHandle* in the example above) with an available index.

When such indexers are registered with the system, an automatic indexing process is triggered the next time the database is opened. If you want to force the indexing to happen right now, call the following API:

```
graph.runMaintenance();
```

If you have existing atoms of the type specified in the indexer, they will all be added to the index and this can take some time. Indexer can also be removed by calling *HGIndexManager.unregister*. Remove an indexer doesn't take much time.

Note that *HGIndexer* instances are stored as HGDB atoms. For instance, one can list all by-value-part indices with the following query:

```
List<ByPartIndexer> byPartIndexers = hg.getAll(graph,
hg.type(ByPartIndexer.class));
```

That said, removing an *HGIndexer* atom without going through the *HGIndexerManager.unregister* method would be a bad idea because the underlying storage won't be cleaned up.

[<< Prev - Graph Traversals](#) [Next - Transaction Essentials >>](#)

Transaction Handling.

HyperGraphDB is transactional. For performance reasons, HyperGraphDB's transactions are by default ACI, but not D. That is, they are atomic, consistent and isolated, but not durable. This means that upon failure, some of the recently committed transactions may be lost. This is generally acceptable, especially in a Java environment where JVM crashes are relatively rare. If you need durability, that's possible as well.

In addition, transactions can be nested so that sub-operations of more complex operations can fail, while the top-level transaction succeeds. Every update operation that you perform is automatically encapsulated in a transaction so most of the time you don't have to worry about it. However, when you want to perform several operations as a transactional unit, you can wrap them in a top-level transaction as shown below.

In order to alleviate the API, none of the public HyperGraphDB methods take a transaction object as a parameter. Instead, a current transaction is associated with every thread. It is theoretically possible to have multiple threads share the same current transaction, by working directly with the transactions API, but this is error prone and it should be avoided.

To wrap a long, complex operation in a transaction, the usual coding pattern works:

```
HyperGraph graph = …;

graph.getTransactionManager().beginTransaction();
try
{
    // do your work here…
    // …
    // end work unit

    graph.getTransactionManager().commit();
```

```
}
catch (Throwable t)
{
   // false means the transaction failed.
   graph.getTransactionManager().abort();
}
```

However, there is a slight caveat. HyperGraphDB handles locking conflicts leading to deadlocks by randomly aborting one of the transactions involved. Thus if there is a deadlock situation between two transactions, it will be automatically detected and one of the transactions will be aborted with a *DeadlockException* which indicates that it must be retried. Because there is no predetermined order in which deadlocked transactions are aborted, they are all guaranteed to eventually succeed if retried enough times. This approach works quite well in practice, but the above coding pattern must be modified to accommodate the retries. Such a retry loop is implemented in the HGTransactionManager.transact method which you can call like this:

```
graph.getTransactionManager().transact(new Callable<Object>()
{
    public Object call()
    {
        // transaction unit of work
    }
}
```

A common situation is when you want a piece of code to be encapsulated in a transaction, but you'd like to "reuse" the current transaction if there is one in effect (remember transactions are bound to the current thread) or create a new transactions if there is no current one in effect. This use case is implemented in the following API:

```
graph.getTransactionManager().ensureTransaction(new
Callable<Object>()
{
    public Object call()
```

```
    {
        // transaction unit of work
    }
}
```

As mentioned above, deadlock detection is an important aspect of ensuring seamless concurrent access to the database. However, it is not a panacea against deadlocks in your application. If you are using locks on Java runtime data structures, deadlock may result due to the interplay between them and underlying database locking. Suppose you have a regular Java lock L and a database lock D. Suppose thread A acquires L and is then waiting for D while thread B acquires D and is then waiting for L. This sort of deadlock is not going to be automatically detected because L is not managed by the database. In general, you should be mindful of this sort of situation and simply avoid it. You can also use an implementation of the *ReadWriteLock* Java interface that is based on database locks: the *org.hypergraphdb.transaction.BDBTxLock* class. You need to provide a unique *byte[]* that identifies your lock. This implementation participates in deadlock detection and simplifies somewhat mixed concurrent access to RAM structures and disk data. However, caution must be used again. First, the implementation needs the same current transaction to be in effect when locking and unlocking. Second, you must make sure that any code that is executed within a transaction and that modifies RAM data structures is reentrant and can be safely aborted and retried an indefinite number of times.

The full transaction API is documented in the *org.hypergraphdb.transaction* package.

<< Prev - Indexing