MIT-LL Graph QuBE
USER MANUAL

Charlie Dagli	Michael Yee	Gordon Vidaver	Wade Shen
dagli@ll.mit.edu	myee@ll.mit.edu	gordon.vidaver@ll.mit.edu	swade@ll.mit.edu

The following will outline the MIT-LL Graph QuBE system. Specifically, this document will outline the build process, launching and use and will direct the user to distributed example code that ingests and extract features for a sample data set.

1. Building

GraphQuBE is written in Java with Apache Maven for build automation and dependency management. Accordingly, users building the GraphQuBE project will need to setup Maven locally (either stand-alone or as an IDE plugin). Obtaining Maven and setting it up for a particular IDE are beyond the scope of this document, however, once it is installed, the directions below can be used to build GraphQuBE using Maven from the command line.

The GraphQuBE Maven project is configured using the Project Object Model stored in the **pom.xml** file in the project's root directory. This file should be setup properly and not require any modifications by the user. If operating behind a firewall, the user might have to add the proxy server to the Maven **settings.xml** file. In Windows, this file is typically located in the following location:

C:\Users\user\.m2\settings.xml

The proxy can be set in the appropriate block in the xml:

```
<settings>...
<proxies>
<proxy>
<id>myproxy</id>
<active>true</active>
<protocol>http</protocol>
<host>...</host>
<port>...</port>
</proxy>
</proxies>...
</settings>
```

To build GraphQuBE from the command line in the project root:

```
>>mvn compile
>>mvn package
```

The first call builds the source and runs code generation to create .java files from Avro .avdl files. The second call creates a jar file for GraphQuBE and copies all dependencies to **./target/lib**.

2. Data Ingest

GraphQuBE is distributed with a small Bitcoin dataset and corresponding example code to illustrate to developers how one would go about performing the ETL process on raw transactional data.

For ease of use, this example data is located at:

./src/main/resources/bitcoin_small_feats_tsv/

This directory will contain the following input files:

1. **bitcoin-20130410-smaller.tsv**: Raw Bitcoin Transaction Data
2. **bitcoin_ids_xdata.txt**: Mapping from Bitcoin IDs to self-reported metadata
3. **btcToDollarConversion.txt**: A time-based exchange-rate table.

Once the project has been built, data ingest using this example data can be performed via:

>>java -cp target/classes/*;target/lib/* mitll.xdata.dataset.bitcoin.ingest.BitcoinIngest <1> <2> <3>

where:

<1> (input data) = src/main/resources/bitcoin_small_feats_tsv/bitcoin-20130410-smaller.tsv
<2> (database location and name) = src/main/resources/bitcoin_small_feats_tsv/bitcoin
<3> (output dir) = src/main/resources/bitcoin_small_feats_tsv/

This class ingests the raw data, extracts features for both the entities and transactions and saves them out in the corresponding resource directory as feature files and an h2 database, respectively.

The output of this process is the following collection of files:

1. **bitcoin.h2.db**: An h2 database containing all the transactions in the raw input file along with derived features.
2. **bitcoin_features_standardized.tsv**: A feature-file containing standardized entity-level features for all the entities in the raw input file.
3. **bitcoin_raw_features.tsv**: A feature-file containing the raw entity-level features for all the entities in the raw input file.
4. **pair.txt**: An index-file containing all unique pairs of transactions between entities.

These outputs will be used as resource files for GraphQuBE's query-by-example services. New dataset bindings need to be written for each new dataset; the above is provided as an example of how one might structure such a binding.

3. Running GraphQuBE

Graph QuBE entity and pattern search can be performed by launching the **GraphQuBEServer** class in the package **mitll.xdata**. The GraphQuBE service can be launched by running the class in an IDE or command line. Alternately, the GraphQuBE server can be launched from the packaged jar file via:

>>java -cp graph-qube-0.0.1-SNAPSHOT.jar;lib/* mitll.xdata.GraphQuBEServer <port> <h2_db_path>

When run, this class launches a local REST service (via the Spark micro-framework) which exposes entity and pattern search functionality via Avro queries. The **<port>** parameter is the port on which the service will be exposed. The **<h2_db_path>** is the directory where the h2 database file generated by the ingest process is located. Please note: if launching the service from the **jar** file, copy the h2 database file to the root directory (where the **jar** file lives) to ensure the service can find it.

Once the GraphQuBE server has been started, entity and pattern searches can be submitted through REST queries of the type shown below. For both entity and pattern search, the form of REST queries will continue to be the same even as underlying technologies improve. Unless otherwise stated, the goal is to allow for forward-compatibility with newer releases of the tool.

4.1 Entity Search

Once the GraphQuBE service is running, entity search queries are performed using the exposed entity search method. An example of such a search is given below for the entity labeled **5104** (for the service running on port 8087):

http://localhost:8087/entity/search/example?id=5104

The expected results for these types of queries is an html table showing a rank ordered list of entities and their corresponding similarity results to the query entity. By default, the first return should have a matching score of 1.0 and should be the query entity itself.

Expected results for this query are located (relative to this document) in: **./results/entity_query.html**

4.2 Pattern Search

Similarly, pattern search queries are performed using the exposed pattern search method. The pattern search method is a REST query using Oculus's Influent API[1] and Avro serialization is used to both post and return results in JSON format. An example of such a search is given below for interactions between two example entities (for the service running on port 8087):

http://localhost:8087/pattern/search/example?hmm&max=20&example={"uid":"PD0","name":"Pattern Descriptor 0","description":null,"entities":[{"uid":"E0","role":{"string":"Entity Role 0"},"entities":null,"tags":null,"properties":null,"examplars":{"array":["5104"]},"constraint":null},{"uid":"E1","role":{"string":"Entity Role 1"},"entities":null,"tags":null,"properties":null,"examplars":{"array":["725672"]},"constraint":null}],"links":[]}

[1] The Influent API is beyond the scope of this manual. Please see corresponding documentation for more details.

The argument, **max,** is the maximum number of returns desired. The argument, **example,** passes in the entities (and the inferred pattern of behavior between them) as an object compatible with Oculus's Influent API. The user should make sure the boolean flag **hmm** is set in order to make sure temporal behavior matching is turned on.

Results are serialized in order for downstream visualization tools (such as Oculus's Influent) to modularly plug in GraphQuBE into their back-end. Alternately, adding the **svg** boolean to the query turns on basic visualization of results.

Expected results for this query are located (relative to this document) in:

./results/pattern_query.json
./results/pattern_query.html