MOBBED (Mobile Brain-Body-Environment Decision-making) Part II: User Guide

By

Jeremy Cockfield, Kyung min Su, and Kay Robbins

Department of Computer Science

University of Texas at San Antonio

April 2013

UTSA CS Technical Report TR-2013-006

Updated from UTSA CS Technical Report TR-2012-006.

A companion technical report entitled: MOBBED (Mobile Brain-Body-Environment Decision-making) Part I: Database Design is available as TR-2013-005.

This report was revised 7/4/2013 to reflect the addition of data cursors and a revised commit strategy. This report was revised 8/12/2013 to reflect the addition of double precision fetches and some minor revisions of table names. This report was revised 8/25/2013 to remove extractdb as its function has been superseded by generalization of getdb.

Table of Contents

1. Getting Started with MOBBED

1.1 Overview

MOBBED is a database and MATLAB front end designed to facilitate the large scale analysis of multi-modal event-rich data collections. MOBBED provides facilities for searching, querying, annotating, and caching of intermediate calculations. MOBBED can also be used with the MATLAB Parallel Processing Toolbox to take advantage of multicore desktops as well as clusters. These installation instructions assume that you will not be working with the Java source, but are using the JAR files that come with the distribution. The Java source, in the form of an Eclipse project, is available separately.

1.2 Requirements

The MOBBED database has been tested with MATLAB 2012a and PostgreSQL 9.2 on systems running either 64-bit Windows 7 or Ubuntu. No other toolboxes are required.

1.3 Installation

MOBBED assumes that PostgreSQL has been installed and is available either on the local machine or on a machine that is accessible through a network.

1. Install PostgreSQL by downloading and executing the appropriate installer for your platform (e.g., Windows 64 bit, Linux, etc.). You can follow the default installation with the following exceptions:

a) The administrator account for the server is *postgres*. You will be asked for a password for the database administrator. The examples in this guide use the password *admin*, but you should choose (and remember) a more secure password. Make note of the PostgreSQL installation and data directories, as well as the port number for the server.

b) **Do not press the Finish button, until you have unchecked the option to install items using the stackbuilder**. Some items in the stack builder cause the database to perform very slowly.

c) The pgAdmin3 GUI for administering the PostgreSQL server comes with the distribution. You will find it in the `bin` subdirectory of your PostgreSQL installation. This tool is a very handy companion for monitoring the databases that you create. You probably should create a shortcut on your desktop for this tool so that you can monitor the various databases and database servers that you are working with.

2. Install the MOBBED package by unzipping it where you want it to be.

3. Setup up the paths. Make the `mobbedDB` directory your current working directory in MATLAB and execute the `setup.m` script. This adds the appropriate directories to the MATLAB path and the needed Java jar files to the dynamic Java classpath. If you are using EEG structures, you should also add EEGLAB and its subdirectories to your path. You will need to execute the `setup.m` script each time you start MATLAB. Alternatively, you can add commands to your permanent MATLAB configuration.

3. Test the MOBBED package. To test your installation, can run the `examples_paper.m` script located in the `examples` subdirectory after editing the script to reflect your database authentication information. Working code for the examples in this user manual can be found in the script `examples/examples_manual.m`. MOBBED also comes with unit tests that you can execute by typing `runtests mobbed_tests`.

1.4 The Mobbed class

MOBBED consists of three layers: a PostgreSQL database, a Java layer that handles interaction with the database, and a MATLAB class, `Mobbed`, whose public methods are summarized in Table 1. The `Mobbed` class provides structure-oriented input, output, and searching capabilities to allow users to access to the database without having to learn SQL. The database operations reflected by each of these methods either complete successfully in their entirety (i.e., treated as a transaction), or the state of the database is restored to the point when the operations started (rolled back).

Table 1: A summary of the public methods of the `Mobbed` class.

Method	Description
`mat2db`	Create and store a dataset in the database.
`db2mat`	Retrieve a dataset from the database.
`data2db`	Create a data definition and store corresponding data in database.
`db2data`	Retrieve a data definition and associated data from the database.
`getdb`	Retrieve rows from a single table.
`putdb`	Create or update rows from a single table.
`close`	Disconnect from the database. (Further calls cause an exception.)

If you are unfamiliar with classes and objects in MATLAB, you just have to remember a few things. Object-oriented programming allows you to specify how to group data or state and to create functions or methods meant specifically for that data. The definition or blueprint for such a definition is called a *class* or *class definition*. The blueprint for working with this database is in the `Mobbed` class, contained in `Mobbed.m`.

Once you have such a class definition, you can create objects containing data and methods that follow the blueprint by calling the class constructor (designated by the class name followed by any required initialization information). For example:

```
DB = Mobbed('shooter', 'localhost', 'postgres', 'admin');
```

creates a connection to the shooter database on the local machine. The call returns a handle or reference to the object (e.g., a `Mobbed` object). Creating a `Mobbed` object creates "an open connection" to the database as described in Section 4. Once you have a handle to the database (say `DB`), the public methods listed above are called using a syntax similar to that of ordinary functions with the handle as the first argument. For example:

```
mat2db(DB, ...)
```

MATLAB also permits objects to be called using the traditional object-oriented notation:

```
DB.mat2db(...)
```

The object-oriented approach is a natural for databases. Not only does an object-oriented representation allow you to organize related methods, but the object itself encapsulates state, so you don't have to pass information such as passwords and database URLs on every call.

`MOBBED` also has static methods for creating and deleting databases as summarized in Table 2. These methods are called like ordinary functions, but must be qualified by the class name:

```
Mobbed.createdb(...)
```

Static methods don't have internally stored object state information and rely on information passed as arguments. The `createdb` and `deletedb` cause actions that happen completely outside of MATLAB.

Table 2: A summary of the static methods of the `Mobbed` class.

Method	Description
`closeAll`	Closes all open MOBBED database connections for workspace cleanup.
`createdb`	Create a PostgreSQL database.
`createdbc`	Create a PostgreSQL database using database credentials from a file.
`deletedb`	Delete a PostgreSQL database.
`deletedbc`	Delete a PostgreSQL database using database credentials from a file.
`getFromCredentials`	Create a new MOBBED connection using credentials from a file.

1.5 MOBBED organization

You do not have to worry much about the details of the database organization in order to make effective use of MOBBED. MOBBED identifies each database item using a UUID (universally unique identifier). These 128-bit values are represented as hexadecimal strings in MATLAB. For example: `'4acb34d2-19eb-42cf-b996-d7cad6b8fb7d'` could identify a dataset or a tag or an event. The phrase "Universally unique" means that the identifiers will not conflict with identifiers that appear anywhere else. This approach allows programs to fetch data from multiple sources or to merge data from different databases without having to reassign identifiers.

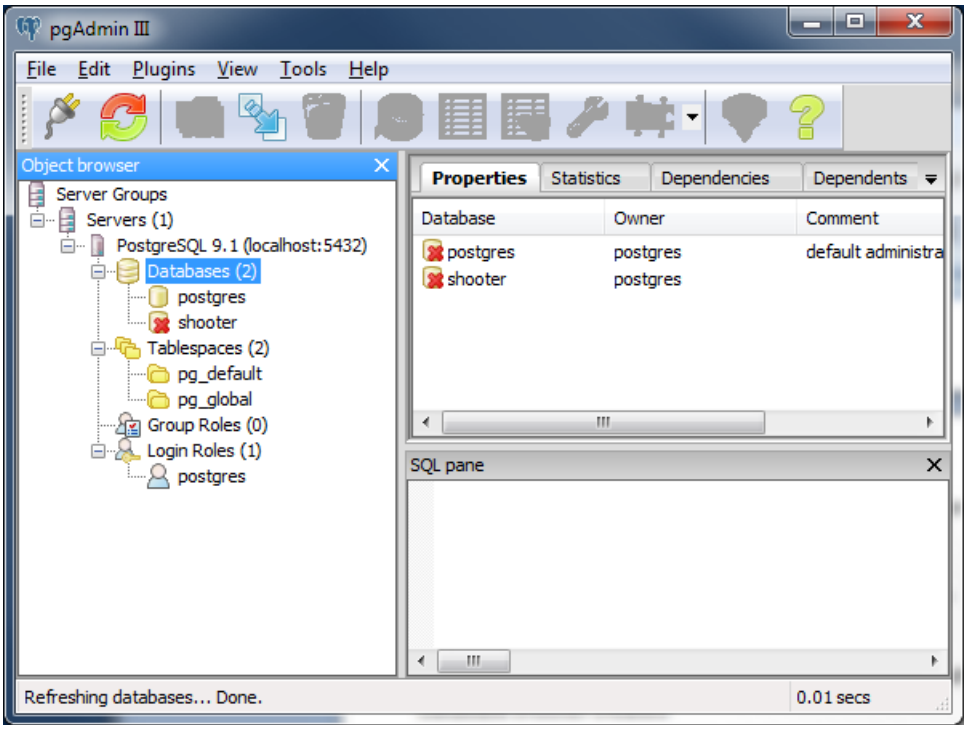
MOBBED has a relatively flat and simple organization designed for straightforward queries as illustrated in subsequent sections of this document. Table 3 summarizes the MOBBED database tables. The details of MOBBED organization are further described in a technical reports (Hossain et al., 2012). MOBBED follows a naming convention similar to that used for Rails (Hartl, 2012). Database tables are generally named in the plural (e.g., `DATASETS`). The column names for a given table have individual names separated by underbars. They begin with the singular version of the table name. For example, the `DATASETS` table has column names such as `dataset_uuid`, `dataset_namespace`, and `dataset_name`. You can read rows from a MOBBED database table using `getdb` and create or update rows using `putdb`. These methods allow users to formulate database actions in terms of MATLAB structures rather than SQL queries, making them easier for MATLAB users to incorporate databases into their scripts. The structure fields representing columns should be lowercase.

Table 3: A summary the tables in MOBBED.

Table name	Purpose
ATTRIBUTES	Structured metadata (either string or numeric).
COLLECTIONS	Membership of items in a named group.
COMMENTS	String comments along with attribution information.
CONTACTS	Contact information for individuals.
DATADEFS	Definitions of data items.
DATAMAPS	Maps data items to entities.
DATASETS	Index information for datasets.
DEVICES	Identity information about particular experimental apparatus.
ELEMENTS	Identity and metadata of sensors that produce data streams (e.g., EEG electrodes).
EVENT_TYPE_MAPS	Maps events to their event types.
EVENT_TYPES	Defines and describes event types.
EVENTS	Events.
MODALITIES	Dataset modalities (currently *simple*, *eeg*, and *generic*).
NUMERIC_STREAMS	Data for data definitions corresponding to collections time-stamped vectors.
NUMERIC_VALUES	Data for data definitions corresponding to single arrays.
SUBJECTS	Demographic information for persons who are experimental subjects.
TAGS	Maps tag strings to entities.
TRANSFORMS	Maps transform strings to datasets for caching purposes.
XML_STREAMS	Data for data definition corresponding to collections of time-stamped xml blobs.
XML_VALUES	Data for data definitions corresponding to single xml blobs.

2. Creating a database

To create a MOBBED database, you must first have installed PostgreSQL and created a user on a host machine (not necessarily your local machine). These steps are performed outside of MATLAB, usually using the installer that comes with the PostgreSQL distribution and the pgAdmin tools. This installation procedure does not actually create a database, but rather it installs a database server that can manage many databases and simultaneous user connections. The installation also includes the pgAdmin tools, which allows you to examine and manage various aspects of their databases. The following figure shows a pgAdmin III view after creating the shooter database. If you click on the shooter database, you can view the actual tables and values in the database:



The default user is the system administrator account `'postgres'`, and the examples assume that the password for this account is `'admin'`. You should modify the default password for security reasons. It is also a good idea to create a separate user account for your MATLAB work. The static `createdb` method of the MATLAB `Mobbed` class enables you to create a database from within MATLAB once you have installed PostgreSQL on the machine that will be your database server. This machine does not have to be your local machine, although it could be.

Example 2.1: Create a MOBBED database named *shooter* on the local machine owned by the *postgres* user who has password *admin*.

```
Mobbed.createdb('shooter', 'localhost', 'postgres', 'admin', 'mobbed.sql')
```

The first argument is the name of the database, and the second argument is the name of the machine hosting the database (in this case the local machine). To create a database on a different machine, replace `'localhost'` with the IP address or URL of the host machine. The default port for the database server is 5432, but if you have multiple database servers on the same machine (for example if

you are running multiple versions of PostgreSQL), you will need additional ports. If you wanted to access a database server listening at port 5433, you would replace `'localhost'` with `'localhost:5433'` in the above example. The `mobbed.sql` script, which comes with the MOBBED distribution, contains the SQL code needed to create the database. The `createdb` function throws an exception if an error occurs.

Example 2.2: Create a MOBBED database named *shooter* on the machine *visual.cs.utsa.edu* owned by the *postgres* user who has password *admin*.

```
try
  Mobbed.createdb('shooter', 'http://visual.cs.utsa.edu', ...
                  'postgres', 'admin', 'mobbed.sql')
catch ME   % If database already exists, creation fails and warns
    warning('mobbed:creationFailed', ME.message);
end
```

This example assumes that the PostgreSQL database has been installed on `visual.cs.utsa.edu`. You should substitute the URL for your own server. This example places the call to create the database in a try-catch block to avoid having a script exit with an error. If the database already exists, this code simply outputs a warning and continues.

MATLAB Syntax

```
Mobbed.createdb(dbname, hostname, username, password, script)
Mobbed.createdb(dbname, hostname, username, password, script, verbose)
```

Table 4: Summary of arguments for the `Mobbed createdb` static method

Name	Type	Description
`dbname`	Required	A string containing name of the database. A database of this name cannot already exist on the specified server. Note: The name must begin with a letter or an underscore. Subsequent characters can be letters, underscores, digits, or dollar signs.
`hostname`	Required	A string containing the host name or IP address of the machine that is running the PostgreSQL server on the default port (5432). To access a server listening on a different port, append a colon followed by the port number to the host name.
`username`	Required	A string containing the user name of the owner of the newly created database. This name must correspond to an existing database user who is allowed to create a database. The default installation of PostgreSQL automatically creates a *postgres* user who can administer the database.
`password`	Required	The password for user.
`script`	Required	The name of the script containing the commands to create a MOBBED database. The `mobbed.sql` file comes with the MOBBED distribution. This function could be used to create different databases from valid SQL.
`verbose`	Optional	If `true` (the default), then MOBBED outputs informative messages during the creation process, otherwise it suppresses messages.

3. Deleting a database

Once created, a database has permanent existence until it is deleted, independently of whether MATLAB is running. Users can use the pgAdmin tools or web-based tools to examine the data outside of MATLAB. The `deletedb` method of `Mobbed` deletes a particular database from the specified server. If the database doesn't exist, MOBBED outputs a warning, but does not throw an exception.

Example 3.1: Delete a locally stored MOBBED database named *shooter* owned by the *postgres* user who has password *admin*.

```
Mobbed.deletedb('shooter', 'localhost', 'postgres', 'admin');
```

Deletion from MATLAB is meant mainly for clean-up of temporary databases. For managing more archival databases and particularly databases that are shared, you should consider using the administrative tools that come with PostgreSQL. You should be sure that you have closed the connections to this database or exit MATLAB before trying to delete the database before using these tools.

MATLAB Syntax

```
Mobbed.deletedb(dbname, hostname, username, password)
Mobbed.deletedb(dbname, hostname, username, password, verbose)
```

Table 5: Summary of arguments for the `Mobbed deletedb` static method.

Name	Type	Description
`dbname`	Required	A string containing name of the database to be deleted.
`hostname`	Required	A string containing the host name or IP address of the machine that is running the PostgreSQL server on the default port (5432). To access a server listening on a different port, append a colon followed by the port number to the host name.
`username`	Required	A string containing the user name of the owner of the newly created database. This name must correspond to an existing user in the database. The default installation of PostgreSQL automatically creates a `'postgres'` user.
`password`	Required	The password for user.
`verbose`	Optional	If `true` (the default), then MOBBED outputs informative messages during the deletion, otherwise it suppresses messages.

4. Connecting to a database in MATLAB

Once you have created a PostgreSQL database, you can connect to it within MATLAB by creating a `Mobbed` object using the constructor of the `Mobbed` class.

Example 4.1: Open a database connection to a locally stored MOBBED database named *shooter* owned by the *postgres* user who has password *admin*.

```
DB = Mobbed('shooter', 'localhost', 'postgres', 'admin');
```

Use the reference to `DB` as the first argument of `Mobbed` methods to access this database. You can create multiple connections to the same database or to different databases by creating multiple `Mobbed` objects. If you have the MATLAB parallel processing toolbox, you can create a separate connection in each worker and use them to simultaneously write to the database.

MATLAB Syntax

```
DB = Mobbed(dbname, hostname, username, password)
DB = Mobbed(dbname, hostname, username, password, verbose)
```

Table 6: Summary of arguments for the `Mobbed` constructor.

Name	Type	Description
`dbname`	Required	A string containing name of the database, which must exist on the specified host.
`hostname`	Required	A string containing the name or IP address of the machine that is running the PostgreSQL server on the default port (5432). To access a server listening on a different port, append a colon followed by the port number to the host name.
`username`	Required	A string containing the user name of a user (role) that has permission to access the database. This name must correspond to an existing user in the database.
`password`	Required	The password for user.
`verbose`	Optional	If `true` (the default), then `Mobbed` outputs informative messages during the connection process, otherwise it suppresses messages.
`DB`	Output	A handle to the database connection to be used in subsequent accesses to the database. This connection should be closed before reusing the variable.

5. Disconnecting from a database in MATLAB

Open connections to the database should be closed explicitly when you finish using them unless you exit MATLAB. Suppose `DB` is a MATLAB variable holding a `Mobbed` object. To close the database connection represented by `DB`, call the `close` method.

Example 5.1: Close the open database connection represented by `DB`.

```
close(DB)
```

It doesn't hurt to call `close` multiple times on an already closed connection. However, once you close a connection, you cannot use it for other operations without creating a completely new MATLAB `Mobbed` object.

MATLAB Syntax

```
close(DB)
```

Table 7: Summary of arguments for the `Mobbed close` public method.

Name	Type	Description
`DB`	Required	A handle to an open MOBBED database connection.

In a typical work session, you may create many different MOBBED connections and if you don't close these connections explicitly, the resources will not be released unless you exit MATLAB completely. This issue becomes problematic over extended work sessions, particularly when your script throws exceptions, thereby avoiding your carefully placed `close(DB)` commands. Executing a MATLAB `close all` command does not release these resources. MOBBED keeps track of all open connections and provides a static `closeAll` method for clean-up.

Example 5.2: Close all of the open MOBBED database connections as part of cleaning up the MATLAB workspace.

```
Mobbed.closeAll()
```

MATLAB Syntax

```
Mobbed.closeAll()
```

6. Storing a dataset in the database

The dataset is the fundamental organizational unit in MOBBED database. Use the `mat2db` function to store datasets in the database. To store a dataset, first retrieve an empty structure from the database by calling `db2mat` with no arguments except the database connection, fill the structure with the information, and then call `mat2db` with the filled in structure as an argument.

Example 6.1: Upload an EEGLAB `EEG` structure to the database through open connection `DB`.

```
load eeglab_data_ch.mat;              % load a previously saved EEG structure
s = db2mat(DB);                       % get empty structure to fill in
s.dataset_name = 'eeglab_data';       % dataset name is required
s.data = EEG;                         % set data to be stored
sUUID = mat2db(DB, s);                % store in database DB
```

The `db2mat` in Example 6.1 retrieves an empty structure, `s`. Most of the fields of this structure directly correspond to the columns of the `DATASETS` table. A complete list of fields with their descriptions appears in Table 9. The only required arguments are the dataset name and the actual data. The example sets these fields in the `s` structure and then calls `mat2db` to store the dataset. The call returns a cell array containing the UUID string (universally unique identifier) of the new dataset. You can store multiple datasets in a single call to `mat2db` by replacing `s` with an array of structures.

Example 6.2: Upload a second, tagged version of an EEGLAB `EEG` structure into the database through open connection `DB`.

```
s = db2mat(DB);                       % get empty structure to fill in
s.dataset_name = 'eeglab_tagged';     % dataset name is required
s.data = EEG;                         % set data to be stored
sUUID = mat2db(DB, s, 'Tags', {'EyeTrack', 'Oddball', 'AudioLeft'});
```

Example 6.2 stores another copy of the EEG structure in the database under the name *eeglab_tagged*. The value of `'IsUnique'` is *true*, specifying that the combination of the dataset's namespace and name must be unique and an attempt to overwrite fails. If the `'IsUnique'` argument is *false*, the dataset version number is incremented before storing a dataset with the same name and namespace as a previously stored dataset. The dataset has three tags associated with it: *EyeTrack*, *Oddball*, and *AudioLeft*. The tags allow unstructured searching across multiple items in the database.

MOBBED events have a user-defined type that allows you to retrieve similar events more easily. In Example 6.2, no event types were specified, so MOBBED created a new type for each unique value of `event.type`. (Since this is EEG modality, MOBBED assumes the incoming data is an EEGLAB EEG structure). However, if you have a collection of EEG datasets, you will want to reuse the types so that you can identify the events of the same type across the collection.

The `mat2db` method has an optional `'EventTypes'` argument that allows you to pass existing event types for MOBBED to reuse. The `mat2db` method also has a second return argument that is a cell array containing the union of the event types passed in and the events that MOBBED had to create because it encountered a new type of event. The event types are represented by their UUID strings. The process is shown in Example 6.3.

Example 6.3: Reuse event types created when storing multiple, related `EEG` structures.

```
s = db2mat(DB);                          % get empty structure to fill in
s.data = EEG;                            % store EEG with new set of event types
s.dataset_name = 'original EEG';
[~, uniqueEvents] = mat2db(DB, s);
s.data = EEG1;                           % store EEG1 reusing event types
s.dataset_name = 'EEG1';
[~, uniqueEvents] = mat2db(DB, s, 'EventTypes', uniqueEvents);
```

MATLAB Syntax

```
[UUIDs, uniqueEvents] = mat2db(DB, datasets)
[UUIDs, uniqueEvents] = mat2db(DB, datasets, ...)
```

Table 8: Summary of arguments for the `Mobbed mat2db` public method.

Name	Type	Description
`DB`	Required	A handle to an open MOBBED database connection.
`datasets`	Required	A structure array reflecting the columns of the `DATASETS` table in the MOBBED database. The fields are shown in Table 9.
`'EventTypes'`	Name-Value	A cell array of UUIDs of the event types associated with this dataset. Section 10 describes how common event types facilitate search.
`'IsUnique'`	Name-Value	If true (the default), the `mat2db` method throws an exception if the (namespace, name) combination already appears in the database. If false, MOBBED creates a new version of the dataset and increments the version number if the (namespace, name) combination already appears in the database.
`'Tags'`	Name-Value	A string or a cell array of strings specifying the tags to be associated with the datasets stored in this operation. Tags are a form of unstructured annotation described in Section 11.
`UUIDs`	Output	A cell array of UUID strings corresponding to the datasets stored in the database by this operation.
`uniqueEvents`	Output	A cell array of UUID strings corresponding to the unique event types used in this operation, including those passed in by the `'EventTypes'` argument.

Table 9 below provides a specification of the `datasets` structure used in `mat2db`. The only field that is required to have a value is the `dataset_name`. Usually the `data` field is also set with the actual dataset. If the `data` field is not empty, MOBBED stores its value as a large binary object so that `db2mat` can return an exact copy of what was stored. The remaining fields are either optional or are assigned by `mat2db`. When the `data` field is empty, the dataset is used as an organizing structure, and the actual data is stored in multiple data definitions as described in Section 14.

Table 9: Summary of fields of the `datasets` argument of the `Mobbed mat2db` public method.

Field name	Field type	Description
`dataset_uuid`	uuid string (do not set unless created from other items)	The UUID identifying the dataset in the database. This field should not be set unless the dataset was generated elsewhere.
`dataset_session_uuid`	uuid string of the session	The UUID of the session in which this dataset was recorded. The session usually refers to datasets that were collected in a single sitting, although a session may contain many runs.
`dataset_namespace`	string (optional)	The namespace of this dataset. If not set, MOBBED uses the default namespace `'mobbed'`. Often a laboratory, institution, or investigator URL is used as the namespace to avoid conflicts with similarly-named datasets. If you organize your datasets into different namespaces, you can more easily isolate datasets that belong together when searching.
`dataset_name`	string (required)	The name of the dataset written to the database. The combination of (namespace, name, and version) is unique in MOBBED. **You are required to set this field.**
`dataset_version`	integer (do not set)	An integer indicating a version number for this dataset. You should not set the value of this field.
`dataset_contact_uuid`	uuid string (optional)	The UUID of the owner of this dataset. If this is null, MOBBED uses the UUID for the default contact which is `'system'`.
`dataset_creation_date`	timestamp (do not set)	A timestamp indicating when the dataset was stored in the database.
`dataset_description`	string (optional)	A description of the dataset.
`dataset_parent_uuid`	uuid string (optional)	The UUID of the parent dataset. You can use a parent to, for example, indicate that this dataset is derived from another dataset and inherits its events. However, MOBBED does not use the parent implicitly in its search.
`dataset_modality_uuid`	uuid string or string name (optional)	The UUID or name of the modality used for this dataset. The modality determines the format of the data, particularly the writing of the events and attributes. MOBBED currently supports *simple*, *eeg*, and *generic* modalities. EEG is the default.
`dataset_oid`	internal object ID (do not set)	The object ID of the actual data of this dataset used to identify the large binary object containing the data in the database.
`data`	contains data that will be stored as a large binary object.	This field contains the actual dataset data in its entirety. It can be in any format that can be saved by MATLAB as a single variable. The value of this field is retrieved as the data by `db2mat`.

7. Retrieving a dataset based on dataset UUID

You must know a dataset's UUID in order to retrieve it from a MOBBED database. You may know the UUID because you saved it from a previous operation or because you performed a search for UUIDs meeting specified search criteria.

Example 7.1: Retrieve a group of datasets corresponding to the UUIDs contained in `UUIDs`.

```
datasets = db2mat(DB, UUIDs);
```

The fields of the `datasets` structure array are specified in Table 9. All of the fields have values in the returned structure, with the `.data` field containing the actual data.

MATLAB Syntax

```
datasets = db2mat(DB)
datasets = db2mat(DB, UUIDs)
```

Table 10: Summary of the arguments of the `Mobbed db2mat` public method.

Name	Type	Description
`DB`	Required	A handle to an open MOBBED database connection.
`UUIDs`	Optional	A UUID string or a cell of UUID strings corresponding to dataset(s) that have been stored in the database. If not included, `db2mat` returns an empty structure to be filled in by the user for calling `mat2db`.
`datasets`	Output	A structure array that is similar in form to the structures used for storing using `mat2db`.

8. Searching

MOBBED uses the `getdb` function to retrieve rows of a specified table using qualified searches. Rowsets from any of the tables listed in Table 3 can be retrieved as a MATLAB structure array, with the field names corresponding to column names.

Example 8.1: Retrieve a list of all of the datasets.

```
s = getdb(DB, 'datasets', inf);           % retrieve all rows from datasets
```

In this example, the structure array, `s`, has fields that mirror the columns of the DATASETS table. (See Table 9.) Unlike `mat2db`, `getdb` only contains table rowsets and not the actual dataset data.

Example 8.2: Retrieve a list containing a maximum of 10 datasets.

```
s = getdb(DB, 'datasets', 10);  % retrieve a maximum of 10 rows from datasets
```

Currently `getdb` only supports retrieval of all rows or a specified number of rows.

Example 8.3: Retrieve up to 10 datasets whose names are `'eeg*'`.

```
s = getdb(DB, 'datasets', 0);         % get empty datasets structure
s.dataset_name = 'eeg*';              % dataset name must be 'eeg*'
sNew = getdb(DB, 'datasets', 10, s)   % retrieve these datasets
```

This example uses qualified search. We first retrieve an empty structure and fill in the field qualifications for searching. In this case we are asking specifically for datasets whose name is `'eeg*'`. The search qualification requires an exact match of the name `'eeg*'`. (Multiple copies might appear either with different namespaces or with different versions.)

A more typical search qualification might ask for all datasets whose name starts with `'eeg'`, that is the search qualification is a regular expression as shown in the next example.

Example 8.4: Retrieve up to 10 datasets whose names start with `'eeg'`.

```
s = getdb(DB, 'datasets', 0);         % get empty datasets structure
s.dataset_name = 'eeg*';              % dataset name starts with 'eeg*'
sNew = getdb(DB, 'datasets', 10, s, 'RegExp', 'on'); % retrieve these datasets
```

In Example 8.3, `'eeg*'` was treated as a specific name that had to be matched exactly. In Example 8.4, regular expression parsing is turned on, so `'eeg*'` is treated as a regular expression.

Users familiar with regular expressions can use regular expressions to qualify the search of any string field. Specifically, any search of a string type column can be qualified by setting the corresponding field value to a specific string or a cell array of specific strings. If `'RegExp'` is `'on'`, then these strings are treated as regular expressions for the search. UUID column search can be qualified by listing a specific UUID as a string or a group of UUIDs as a cell array of strings. For example, you might want to search for all datasets that have a certain parent dataset.

Double columns such as the `event_start_time` can also be searched. The following example searches for events that happen 1 second, 2 seconds, or 3 seconds after the start of the data file.

Example 8.5: Extract all events in the database that happen 1 second, 2 seconds or 3 seconds after the start of the dataset.

```
s = getdb(DB, 'events', 0);     % get empty events structure
s.event_start_time = [1, 2, 3]; % get events within epsilon of these times
sNew = getdb(DB, 'events', inf, s);
```

The `sNew` variable contains all such events of all types from all datasets, since the search did not qualify either the `event_dataset_uuid` or the `event_type_uuid`. The search of Example 8.5 has an implied default precision, *epsilon*, which you can set by calling the `setPrecision` method of `Mobbed`. The search returns rows of the EVENTS table whose event start times are in [*a* − *epsilon*, *a* + *epsilon*]. Here *a* can be any of the specified start times 1, 2, or 3. The *epsilon* allows users to define the meaning of "simultaneous" to account for the timing precision of the experiments as well as to look for concurrences.

A second form of qualification for numeric searches allows you more control in specifying the precision. This qualification can be used for epoching and other types of retrieval as shown by the next example.

Example 8.6: Extract all events in the database that happen within [−0.5, 1] of the times 1 second, 2 seconds or 3 seconds after the start of the dataset.

```
s = getdb(DB, 'events', 0);             % get empty events structure
s.event_start_time.values = [1, 2, 3]; % get events within range of these times
s.event_start_time.range = [-0.5, 1];  % range specification times
sNew = getdb(DB, 'events', inf, s);
```

This search returns rows of the EVENTS table whose event start times are in [*a* – 0.5, *a* + 1]. Here *a* can be any of the specified start times 1, 2, or 3. Qualifications for date columns are planned for a future release.

The `getdb` method also supports qualified searches on tags and attributes. Tags, which are unstructured labels used to annotate data, are described in more detail in Section 11. Attributes, which are annotations that are associated with a certain field in a data structure, are discussed in Section 12.

Example 8.7: Retrieve up to 10 datasets whose names start with `'eeg'`. Each dataset must have the `'EyeTrack'` tag and either the `'VisualTarget'` tag or a tag that starts with the phrase `'Audio'`.

```
s = getdb(DB, 'datasets', 0);     % get empty datasets structure
s.dataset_name = 'eeg*';          % dataset name starts with 'eeg'
sNew = getdb(DB, 'datasets', 10, s, 'RegExp', 'on'...
          'Tags', {{'EyeTrack'}, {'VisualTarget', 'Audio*'}})
```

Tag search qualifications are listed in a cell array. The qualification in each cell must be satisfied. However, if a cell array entry is another cell array, then at least one of the inner conditions must be

satisfied. Only two levels of nesting are allowed. Attribute search qualifications follow the same rule as tags.

Example 8.8: Retrieve events of the first dataset retrieved in Example 8.5 in blocks of 100 events. whose names start with `'eeg'`. Each dataset must have the `'EyeTrack'` tag and either the `'VisualTarget'` tag or a tag that starts with the phrase `'Audio'`.

```
s = getdb(DB, 'datasets', 0);     % get empty datasets structure
s.dataset_name = 'eeg*';          % dataset name starts with 'eeg'
sNew = getdb(DB, 'datasets', 10, s, 'RegExp', 'on'...
          'Tags', {{'EyeTrack'}, {'VisualTarget', 'Audio*'}})
```

The `getdb` method also supports data cursors for iteratively fetching the results of a query. In the following example, we plan to process 100 events at a time correspond to the dataset whose UUIDs correspond to the datasets fetched in Example 8.8.

Example 8.9: Retrieve events for the datasets of Example 8.8 in batches of 100 events using a data cursor. Find the unique event-types present

```
uniqueTypes = { };              % start with an empty set of types
UUIDs = {sNew.dataset_uuid};    % fetch the UUIDs of the query datasets
s = getdb(DB, 'events', 0);     % get template for retrieving events
s.event_dataset_uuid = UUIDs;    % set search criteria
s = getdb(DB, 'events', 100, s, 'DataCursor', 'mycursor');
while ~isempty(s)
   uniqueTypes = union(uniqueTypes, unique({s.event_type_uuid}));    % process
   s = getdb(DB, 'events', 100, 'DataCursor', 'mycursor'); % get next
end
```

The first `getdb` initializes a data cursor called `'mycursor'` as well as retrieving up to 100 events corresponding to the dataset whose UUID is `UUIDs{1}`. The `getdb` in the loop fetches the next 100 events after processing the previous set. MOBBED supports data cursors only for the `getdb` method. Single datasets are always retrieved in their entirety from the externally stored binary object.

MATLAB Syntax

```
outS = getdb(DB, table, limit)
outS = getdb(DB, table, limit, inS)
outS = getdb(DB, table, limit, inS, ...)
```

Table 11: Summary of the arguments of the `Mobbed getdb` public method.

Name	Type	Description
`DB`	Required	A handle to an open MOBBED database connection.
`table`	Required	A string specifying the name of the table to retrieve rows from
`limit`	Required	The maximum number of rows to retrieve. If zero, an empty row structure is returned. If `inf`, all rows are returned.
`inS`	Optional	A structure whose rows mirror the rows of the table. This structure is used to specify search qualifications
`'Attributes'`	Name-value	Nested cell array of attributes used as search qualifications

`'DataCursor'`	Name-value	String specifying the name of the currently active data cursor
`'RegExp'`	Name-value	If the value is `'on'`, then string column qualifiers are treated as regular expressions to be matched. If the value is `'off'`, (the default) then string column qualifications are treated literally.
`'Tags'`	Name-value	Nested cell array of tags used as search qualifications
`outS`	Output	A structure whose rows mirror the rows of the table. This structure is the result of the specified search qualifications.

9. Writing individual rowsets to the database

MOBBED has a general `putdb` method that allows users to write individual rowsets to the database. This method should be used cautiously --- usually for tagging and assigning attributes and other metadata or for mapping data items to a dataset or collection. Changes initiated by `putdb` are not made permanent until you commit the transaction. Generally, you should call `commit` before starting a series of `putdb` operations to clear previous pending transactions.

Example 9.1: Update a dataset description to an existing dataset.

```
s = getdb(DB, 'datasets', 0);     % get an empty structure to fill in
s.dataset_uuid = '4acb34d2-19eb-42cf-b996-d7cad6b8fb7d';
s = getdb(DB, 'datasets', 1, s);     % retrieve the dataset
s.dataset_description = 'dataset that comes with EEGLAB';
putdb(DB, 'datasets', s);
```

Modification of existing rowsets generally follow this pattern. The writing of the individual rowsets generally follows this pattern. The second `getdb` retrieves the existing record to be modified.

MATLAB Syntax

```
putdb(DB, table, inS)
```

Table 11: Summary of the arguments of the `Mobbed putdb` public method.

Name	Type	Description
`DB`	Required	A handle to an open MOBBED database connection.
`table`	Required	A string specifying the name of the table to storerows.
`inS`	Required	A structure whose rows mirror the rows of the table. This structure is used to specify the rows to be stored.

10. Events

MOBBED is designed to facilitate analysis of event-rich data. An event is an incident or item that is associated with a particular point in time. Events have a type and a start time. They may also have an end time and any number of other attributes. In traditional experimental setups, events occur as part of the experiment --- say when an experimental stimulus is delivered or when the system detects a user response to that event. However, in more natural settings, an event can be any labeled and time stamped occurrence including incidental changes in environmental conditions or even the output of classification algorithms computed during processing and analysis.

Events are typically used in two distinct ways --- for selection and for annotation. In selection, an analyst will isolate short intervals of data around the time of particular events in order to differentiate between response when the event occurs from when it does not. In annotation, the analyst might perform classification based on some algorithm and then try to find out what the portions in the same class have in common. Having the same event occur during the intervals in a particular class provides an explanation of the meaning of that class of data.

Individual events are stored in the EVENTS table, whose columns are described in Table 12. Events are associated with an entity and have a type as well as a start time. Events can also occur over an interval of time and hence have an end time. Events also have a certainty, which is a value between 0 and 1 indicating how certain the event occurrence is. A certainty value of 1 indicates an event that is completely certain. Actual experimental events have a certainty of 1. However, many computational labeling algorithms, including most classification algorithms, provide a probability measure of accuracy with a value between 0 and 1.

Table 12: Summary of the columns of the EVENTS table of MOBBED.

Field name	Field type	Description
`event_uuid`	uuid string (do not set)	The UUID of the event in the database. This field should not be set by users.
`event_dataset_uuid`	uuid string	The UUID of the dataset associated with this event.
`event_type_uuid`	uuid string	The UUID of the type of event. If not supplied, MOBBED will create a new type.
`event_parent_uuid`	uuid string	The UUID of the parent to this event.
`event_start_time`	double	A double specifying the time in seconds of the start of this event, relative to the start of the dataset or other organizing entity.
`event_end_time`	double	An optional double value specifying the time in seconds of the end of this event relative to the start of the dataset or organizing entity. If the end time is not given, MOBBED uses the event start time.
`event_position`	integer	A sequence number identifying this event. This may be empty if not provided when the event is stored.
`event_certainty`	double	A double value between 0 and 1 indicating the certainty of this event.

A significant difficulty with event analysis is the lack of a common nomenclature to describe similar events. MOBBED supports detailed metadata for events in three different ways: event types, tagging, and attributes. Each event has a type, which is specified by an entry in the `EVENT_TYPES` table. When storing a group of similar experiments, you want to reuse the event types whenever possible so that you can search for similar events by event type across similar datasets.

Example 10.1: Store ten copies of the `EEG` structure using the same event types for all copies.

```
s = db2mat(DB);                  % get empty structure to fill in
s.data = EEG;                    % set data to be stored
sNewF = cell(10, 1);             % save room to get created UUIDs
uniqueEvents = {};               % start with no event types and accumulate
for k = 1:10
    s.dataset_name = ['data' num2str(k) '.mat']; % set the dataset name
    [sNewF(k), uniqueEvents] = mat2db(DB, s, 'EventTypes', uniqueEvents);
end
```

The `mat2db` function allows you to pass a cell array of event type UUIDs and uses these types if the event type string matches that of the event type. In the above example, the first call to `mat2db` passes an empty set of event types. The `mat2db` function returns a cell array that is the union of the unique event types passed in and the new ones that were created during the call. Thus, in Example 10.1, the event types created for the first dataset were reused for the remaining datasets, although each dataset has its own set of actual events. This use of common event types facilitates looking for patterns across multiple datasets and experiments.

Events often have additional metadata that can be used to map common events. Event types can have tags to facilitate mapping across experiments. Events themselves can have attributes and tags. Tags and attributes are described in more detail in Sections 11 and 12, respectively.

Example 10.2: Retrieve all events associated with the dataset identified by UUID and having event type identified by eUUID.

```
s = getdb(DB, 'events', 0);      % get empty structure to fill in
s.event_dataset_uuid = UUID;      % search for events from a particular dataset
s.event_type_uuid = eUUID;       % search for events only of a particular type
events = getdb(DB, 'events' inf, s); % search for events from a dataset
```

The return value is a structure array containing all of the events of type `eUUID` from the dataset identified by `UUID`. The fields of the `events` structure array of Example 10.2 are summarized in Table 12.

11. Tags

Tags provide an unstructured way of annotating data to make searching easier. Tags are strings that can be associated with any item to provide additional information. Tags may be inserted for any item in the database that has its own UUID.

Example 11.1: Create a *"/Label/Event/Type"* tag for the item with UUID *"34bab916-3675-4099-bb92-792fb89de020"*, which corresponds to an event type item.

```
s = getdb(DB, 'tags' 0);      % get empty structure to fill in
s.tag_name = '/Label/Event/Type';
s.tag_entity_uuid = '34bab916-3675-4099-bb92-792fb89de020';
s.tag_entity_class = 'event_types'; % name of table where entity is defined
putdb(DB, 'tags', s);         % store the tag
```

Example 11.2: Create a *"Subject 1"* tag for all datasets whose name starts with *"S001"*.

```
s = getdb(DB, 'datasets', 0); % get empty structure to fill in
s.dataset_name = 'S001*';    % set search criteria
datasets = getdb(DB, 'datasets', inf, s, 'RegExp', 'on');
s = getdb(DB, 'tags' 0);      % get empty structure to fill in
s.tag_name = 'Subject 1';     % specify the tag
s.tag_entity_class = 'datasets';
for k = 1:length(datasets)
   s.tag_entity_uuid = datasets(k).dataset_uuid;
   putdb(DB, 'tags', s);         % store a tag-entity association
end
```

Example 11.2 has two steps: find the UUIDs of the items to be tagged and then apply the tag. The columns of the TAGS table are given in Table 13.

Table 13: Summary of the columns of the TAGS table of MOBBED.

Field name	Field type	Description
tag_name	string	A string representing the actual tag, which could be a pathname.
tag_entity_uuid	uuid string	The UUID of an entity associated with this tag.
tag_entity_class	string	The name of the database table that this entity comes from.

12. Attributes

Attributes provide a flexible method of storing string or numeric metadata about a dataset or other entity. Attributes have a specific position within the organizational structure of their dataset. Table 14 summarizes the columns of the ATTRIBUTES table. The key feature of attributes is that they can represent not only the values of numeric and string metadata, but also their position within the original dataset structure.

Example 12.1: EEGLAB stores dataset events in a substructure array called `EEG.event`. The event types for the dataset are in `{EEG.event.type}`. Researchers will often create additional fields in the event substructure to incorporate additional information about the event. For example events of type target, might have an `EEG.event.dist` field containing the distance of the target from a baseline position.

The fields that are created in this manner are completely up to user discretion. The `mat2db` function automatically stores the extra fields of `EEG.event` and `EEG.chanlocs` as attributes. The field names (e.g., `data.event.dist`) are stored as path strings (e.g., `'./event/dist'`) in `attribute_path`.

Table 14: Summary of the columns of the ATTRIBUTES table of MOBBED.

Field name	Field type	Description
`attribute_uuid`	uuid string (do not set)	The UUID of the event after written to the database. This field should not be set by the user.
`attribute_entity_uuid`	uuid string	The UUID of the entity that this attribute qualifies.
`attribute_entity_class`	string	The name of the database table that this entity comes from.
`attribute_organization_uuid`	uuid string	The UUID of the organizing entity – usually a dataset or a collection. This value can match the entity UUID.
`attribute_path`	string	Pathname corresponding to location in the data structure for this attribute.
`attribute_numeric_value`	double	A double containing the value of this attribute, if it is numeric.
`attribute_value`	string	A string representation of this attribute.

13. Transforms and caching

MOBBED provides a very simple provenance mechanism that users can incorporate into their everyday workflow to facilitate caching, reuse, and standardization of workflows. The steps are as follows:

1. Store the original or starting data in MOBBED and obtain its UUID.
2. Apply the processing pipeline to the data to obtain a new dataset.
3. Store the resulting data in MOBBED and obtain its UUID.
4. Choose a transform string that unambiguously identifies the transformed data.
5. Add the UUID and transform string to the TRANSFORMS table.

Ideally these operations would be incorporated in wrapper functions as part of a standardized pipeline.

Example 13.1: Load a dataset, filter it, and store the results in the database.

```
% Store original dataset in the database
load eeglab_data_ch.mat;              % load a previously saved EEG structure
s = db2mat(DB);                       % get empty structure
s.dataset_name = 'original_eeglab_data';      % dataset name is required
s.data = EEG;                         % set data to be stored
sUUID = mat2db(DB, s);                % store original dataset

% Filter the data and store the filtered dataset
EEG = pop_eegfilt(EEG, 1.0, 0, [], 0);  % filter EEG dataset
s.dataset_name = 'eeglab_data_filtered.set';   % set up for storage
s.data = EEG;                         % put data in structure for storing
sUUIDNew = mat2db(DB, s);             % store the filtered dataset

% Cache the transform for future quick retrieval
t = getdb(DB, 'transforms', 0); % retrieve an empty transform structure
t.transform_uuid = sUUIDNew{1};       % set the fields
t.transform_string = ['pop_eegfilt(' sUUID{1} '),1.0,0,[],0)' ];
t.transform_description = 'Used EEGLAB FIR filter [1.0, 0]';
putdb(DB, 'transforms', t);           % set the fields
```

Example 13.2: Use the transforms to retrieve the filtered data rather than recomputing the values.

```
t = getdb(DB, 'transforms', 0);       % retrieve an empty structure
t.transform_string = ['pop_eegfilt((' sUUID{1} '),1.0,0,[],0)' ];
cached = getdb(DB, 'transforms', inf, t); % get UUID of result
filtEEG = db2mat(DB, cached(1).transform_uuid); % get dataset
```

Programs such as BCILAB (Delorme et al., 2011) use fully parenthesized expressions and cache the results locally. A small adaption to this pipeline can use a backend database as a third-level cache.

14. Storing additional data

The MOBBED dataset is a fundamental data unit. Supported dataset formats or modalities are *simple*, *eeg*, and *generic*. These data modalities are discussed in more detail in Section 16. However, often, besides the basic structure, researchers will want to store additional data. Storing data in this way enables users to search and perform other database operations. MOBBED provides the `data2db` and `db2data` methods for storing and retrieving data items.

Currently MOBBED supports five (5) different data formats: `'EXTERNAL'`, `'NUMERIC_VALUE'`, `'NUMERIC_STREAM'`, `'XML_VALUE'`, and `'XML_STREAM'`. The `'EXTERNAL'` format specifies that the data should be stored as a large binary object. This representation permits efficient retrieval, but only the metadata associated with the data definition may be searched. The `'NUMERIC_VALUE'` format specifies a data value consisting of a real-valued vector of arbitrary length. The item can be operated on using certain PostgreSQL operations. Similarly, the `'NUMERIC_STREAM'` consists of a group of time-stamped vectors that are individually searchable. The `'XML_VALUE'` and `'XML_STREAM'` behave similarly except that the data uses self-identifying XML blobs rather than numeric vectors.

Example 14.1: Explode the data from an EEGLAB EEG structure as individual frames that can be searched.

```
sdef = db2data(DB);                          % get an empty template
sdef.datadef_format = 'NUMERIC_STREAM';      % set the format (required)
sdef.datadef_sampling_rate = EEG.srate;      % specify equally spaced samples
sdef.description = [EEG.setname ' individual frames'];
sdef.data = EEG.data;                        % set the data
sdefUUID = data2db(DB, sdef);       % store the individual frames in database
```

The first step in Example 14.1 retrieves an empty structure, `sdef`. Since the data format is specified as a numeric stream, each column of `sdef.data` is stored as a time-stamped value. Time stamps are double values representing the time in seconds since the start of the data. The time stamps may be given explicitly as a vector of double values corresponding to times of the columns of `sdef.data`. However, in this case a sampling rate is given, so the samples are assumed to be equally-spaced. The only required fields are the dataset format and the actual data. A complete list of fields with their descriptions appears in Table 15. Table 16 describes the columns of the `DATADEFS` table. The final call to `data2db` returns a cell array containing the UUID of the data definition that was stored. You can store multiple data definitionsin a single call to `data2db` by replacing `sdef` with an array of structures.

Example 14.2: Associate the data defined in Example 14.1 with the datasets whose UUIDs are contained in the array `UUIDs`. When retrieved, the data will be mapped to the structure `EEG.dataEx`.

```
smap = getdb(DB, 'datamaps', 0);          % get the template
smap.datamap_def_uuid = sdefUUID{1};      % UUID of data from Example 14.1
smap.datamap_path = '/EEG/dataEx';        % load destination
for k = 1:length(UUIDs)
   smap.datamap_entity_uuid = UUIDs{k};
   smap.datamap_entity_class = 'datasets';
   putdb(DB, 'data_maps', smap);
end
```

Example 14.2 creates entries in the DATAMAPS table that associate each of the datasets identified by UUIDs with the data created in Example 14.1.

MATLAB Syntax

```
UUIDs = data2db(DB, datadefs)
```

Table 15: Summary of the arguments of the Mobbed data2db public method.

Name	Type	Description
DB	Required	A handle to an open MOBBED database connection.
datadefs	Required	A structure array containing the information needed to store the data.
UUIDs	Output	A cell array containing the UUIDs of the data definitions created in this call to data2db.

Table 16: Summary of fields of the datadefs structure argument of data2db.

Field name	Field type	Description
datadef_uuid	uuid string (do not set)	The UUID of the data definitionafter written to the database. This field should not be set by the user.
datadef_format	string	One of the following strings specifying the data format: 'NUMERIC_VALUE', 'NUMERIC_STREAM', 'XML_VALUE', 'XML_STREAM', or 'EXTERNAL'.
datadef_sampling_rate	double	Sampling rate in Hz. This is used for stream data when the time between samples is fixed.
datadef_timestamps	double vector	Time in seconds from the beginning of each data point for the individual vectors in a stream. This field is not filled in for non-stream formats or when the sampling rate of a stream is fixed.
datadef_oid	oid string (do not set)	Identifier of the external data blob containing this data if the data format was 'EXTERNAL'.
datadef_description	string	String describing this data definition,
data	data to set	Actual data for this data definition.

15. Retrieving auxiliary data

You must know a data item's UUID in order to retrieve it from the MOBBED database. You may know the UUID because you saved it from a previous operation or because you performed a search for UUIDs meeting specified search criteria. You can also use the data mappings to retrieve all auxiliary data associated with a particular item.

Example 15.1: Retrieve the data identified by the data definition UUID in the variable `dUUID`.

```
datadefs = db2data(DB, dUUID);
```

The `datadefs` structure will have the fields identified in Table 16. The `datadefs.data` field will contain the actual data. If `dUUID` is a cell array containing multiple UUIDs, the `datadefs` structure will be a structure array.

Example 15.2: Retrieve all auxiliary data associated with the primary dataset identified by `pUUID`.

```
smap = getdb(DB, 'datamaps', 0);          % get an empty data map template
smap.datamap_entity_uuid = pUUID;         % find data items mapped to pUUID
dmaps = getdb(DB, 'datamaps', inf, smap); % find data map entries
datadef = db2data(DB, dmaps);             % retrieve all of those
```

The DATAMAPS table stores a structure path so that the data can be placed appropriately when extracted from the database. Suppose in Example 15.2 that the first entry of the `dmaps` structure array has a datamap path specified by `dmaps(1).datamap_path = '/EEG/dataEx'`. Then the corresponding data will be in `datadefs(1).data.EEG.dataEx`. Table 17 summarizes the arguments of `db2data`.

MATLAB Syntax

```
datadefs = db2data(DB)
datadefs = db2data(DB, UUIDs)
datadefs = db2data(DB, dmaps)
```

Table 17: Summary of the arguments of the `Mobbed db2data` public method.

Name	Type	Description
`DB`	Required	A handle to an open MOBBED database connection.
`UUIDs`	Optional	A UUID string or a cell of UUID strings corresponding to previously stored data definition(s). If not included, `db2data` returns an empty structure to be filled in by the user for calling `data2db`.
`dmaps`	Optional	An alternative form of the second argument specifying a structure array of DATAMAPS table entries to fetch.
`datadefs`	Output	A structure array containing the retrieved results. The fields of the output structure are those listed in Table 16. If the second argument was the UUIDs of the data items, the output data items will be in `datadefs.data`. However, if a data map was provided as the second argument, the data will be stored in the appropriate structure under `datadefs.data`.

16. Defining dataset modalities

The dataset is the fundamental organizational unit in MOBBED, and although MOBBED is flexible enough to support complex dataset organizations by combining low-level `getdb` and `putdb` operations, most users will prefer high-level `mat2db` and `db2mat` operations for storing and retrieving datasets. These high-level operations require that the incoming data be in specific formats, specified by the dataset modality.

MOBBED currently supports three modalities: *simple*, *eeg* (the default), and *generic*. The *simple* modality simply stores the dataset as a large binary object and does not explode events. Users use this modality for simple archiving of datasets and are free to assign additional tags and attributes for special purpose searching. The *eeg* modality assumes the data is in an EEGLAB EEG structure. It stores the entire structure as a large binary object, but fully explodes the events and their attributes as well as channel information and other attributes for searching and manipulation. The *generic* modality allows the flexible creation of datasets that have exploded events and metadata.

Users who wish to implement a new modality called `XXX` simply identify the modality by adding an entry to the modality table using `putdb` and implement a class called `XXX_Modality` that has `retrieve` and `save` methods. The implementations can use `putdb` and `getdb` to unfold the desired structures into the database.

16.1 Simple modality

The simple modality stores the dataset as an external binary object and does not explode events or other metadata into the database. The *simple* modality is useful for data archival.

Example 16.1: Store the array `xray` as a simple dataset.

```
s = db2mat(DB);
s.dataset_name = 'my simple dataset'; % dataset name is required
s.data = xray;                        % set data to be stored
s.dataset_modality_uuid = 'simple';   % dataset modality is not the default
sUUID = mat2db(DB, s);                % store in database DB
```

In the above example, the dataset modality is specified by its name rather than UUID. As a convenience, the `mat2db` method accepts either the modality name or its UUID.

Example 16.2: Store the array `xray` as a simple dataset, qualifying with additional tags.

```
s = db2mat(DB);
s.dataset_name = 'my simple dataset 1'; % dataset name is required
s.data = xray;                          % set data to be stored
s.dataset_modality_uuid = 'simple';     % dataset modality
sUUID = mat2db(DB, s, 'Tags', {'Image', 'Left Femur'});
```

Example 16.3: Retrieve datasets that have an `'Image'` tag.

```
dataspecs = getdb(DB, 'datasets', inf, 'Tags', {'Image'});
```

Example 16.3 does not retrieve the actual dataset data, but only the structures corresponding to entries in the DATASETS table. A further call to `db2mat` retrieves the data as shown in the following example.

Example 16.4: Retrieve the actual data for the datasets retrieved in Example 16.3.

```
datasets = db2mat(DB, {dataspec.dataset_uuid});
```

The `datasets` structure array has the fields specified in Table 9. The `datasets.data` fields contain the original dataset.

16.2 Eeg modality

MOBBED uses *eeg* as the default modality and assumes that the dataset follows the EEGLAB `EEG` structure. The strategy explodes event and channel information into the database so that it can be searched. However, MOBBED also stores the entire `EEG` structure as a large binary object so that it can be retrieved intact. The steps are as follows:

1) Create a row in the DATASETS table corresponding to the dataset.
2) Store the `chanlocs` field as a top-level element (entire cap) in the ELEMENTS table.
3) Store the individual channels in ELEMENTS table with top-level cap as the parent.
4) Store all of the specific channel information as attributes.
5) Store the events individually in the EVENTS table, creating attributes for additional fields in the `EEG.event` and `EEG.urevent` structures.
6) Store the entire `EEG` structure as an external large binary data object.
7) Commit the entire transaction.

EEGLAB stores two types of events in the `EEG` structure. The `EEG.urevent` holds the original events, which are stored with an event parent UUID of the Null parent (`'591df7dd-ce3e-47f8-bea5-6a632c6fcccb'`). Properly represented `EEG` structures have events corresponding to urevents, and each event should have an event parent UUID corresponding to some urevent.

16.3 Generic modality

The *generic* modality allows some flexibility in representing the data. We assume the dataset is in a MATLAB structure, which can have an arbitrary number of fields. MOBBED processes only the `element`, `event`, `feature`, and `metadata` fields of the structure. Each of these, if present, is assumed to be an array of structures within the overall dataset structure `x`:

```
x.
   element.
      label       {string}
      position    {integer}
      description {string – could be empty or missing}
      other       {arbitrary number of fields of string or double}
   event.
      type        {string}
      position    {number of event in the event stream}
      stime       {double value indicating start time in seconds}
      etime       {double value indicating end time in seconds}
      certainty   {double value in [0, 1] indicating probability}
      other       {arbitrary number of fields of string or double}
   feature.
      type        {numeric_value, numeric_stream, xml_value, xml_stream}
      value       {depends on type}
      description {string – could be empty or missing}
      other       {arbitrary number of fields of string or double}
   metadata.
      other       {arbitrary number of fields of string or double}
```

The `element` structure array, if present, represents the detectors (e.g., channels in EEG recordings). Each element within the array should have a label (a string) and a position (an integer). The *`other`* field is italicized to indicate that it is a place holder for other fields. These other fields can have any names. MOBBED automatically converts their values to strings and stores them as structured attributes so that they can be restored on retrieval.

The `event` structure array, if present, stores the events associated with this dataset. If the `etime` is missing, the event end time is assumed to be the same as the `stime`. If the `certainty` field is missing, the event is assumed to have a certainty value of 1.

The `feature` structure array, if present, allows users to store computed or other types of features in the database for future interrogation. The `metadata` structure array, if present, allows users to map additional metadata into the database.

MOBBED ignores any additional fields of `x`, but stores the entire data structure as external file that can be retrieved through the database.

16.4 Adding a modality

As mentioned above, MOBBED currently supports three modalities: *simple*, *eeg* (the default), and *generic*. *Generic* offers flexible data representation, but a user may want to use a modality that is more tailored to the structure of their data. This can be accomplished by creating a new modality. There are two steps to creating a new modality: create a new modality type and create a class that encapsulates how to store the data.

We assume that you have already created a MOBBED database and created a `Mobbed` object called `DB`. We use `putdb` to create a new modality type called *xray:*

```
m = getdb(DB, 'modalities', 0); % get a template structure for modalities
m.modality_name = 'xray';
m.modality_platform = 'matlab';
m.modality_description = 'xray modality';
mUUID = putdb(DB, 'modalities', m);
```

The `putdb` returns the UUID of the newly created modality, say `'191df7dd-ce3e-47f8-bea5-6a632c6fcccb'.` In order to maintain consistency with other databases that you will be creating, you should also insert the following lines at the end of your `mobbed.sql` script, since modalities are permanent.

```
-- execute
INSERT INTO MODALITIES (MODALITY_UUID, MODALITY_NAME, MODALITY_PLATFORM,
MODALITY_DESCRIPTION) VALUES ('191df7dd-ce3e-47f8-bea5-6a632c6fcccb',
    'xray', 'MATLAB', 'xray modality');
```

The final step in creating a new modality is to provide a class that encapsulates how to store dataset data for this modality. The writing of the entry in the `DATSETS` table is common to all modalities. The class name for the new class must be `XRAY_Modality`. The definition of the class should be in the `helpers` subdirectory. (See `helpers/EEG_Modality.m` for an example.) The class must have a `store` method that has the signature:

```
function uniqueEvents = store(DB, datasetUuid, data, eventUuids)
```

The `store` method determines how fields of the dataset are exploded into various database tables (e.g., `events`, `attributes`, `datadefs`) so that the data can be searched. The data itself is always stored as a blob for retrieval. It is possible to also provide a `load` method to augment the storage of the data blob on loading.

The *simple* modality presents the simplest example as it only stores the data blob and doesn't explode other information into the database. The `store` method calls the `storeFile` utility function to save the data as a large binary object, but does not explode any other information in the database.

```
classdef SIMPLE_Modality
    % Represents datasets that are only stored as blobs and not exploded

    methods(Static)
        function uniqueEvents = store(DB, datasetUuid, data, eventUuids)
            tStart = tic;
            uniqueEvents = {};
            DbHandler.storeFile(DB, datasetUuid, data, true);
            if DB.Verbose
                fprintf('Data saved to DB: %f seconds \n', toc(tStart));
            end
        end % store
    end % static methods
end % SIMPLE_Modality
```

17. Parallel processing

MOBBED also supports parallel processing and multi-threading if the user has the Matlab Parallel Computing Toolbox as illustrated by the following two examples. The `threads` variable contains the number of workers used for parallel computing. For local processing, this should be less than the number of cores the desktop has. The `fUUIDs` variable is a cell array containing a list of UUIDs of datasets to be processed by each thread or worker.

Example 17.1: Use multiple threads to process multiple groups of datasets.

```
matlabpool(threads)
parfor k = 1:length(fUUIDs)
   doDbPar(dbName, host, user, password, fUUIDS{k});
end
matlabpool close
```

In the above example, each entry element of fUUIDs is a list of dataset UUIDs corresponding to the datasets to be processed by a particular thread, under the assumption that processing each group of datasets is independent. PostgreSQL support many simultaneous connections. Notice that, we don't pass an open connection to the database, which is not serializable. Instead, the database credentials are passed as strings.

The key to using MOBBED in parallel processing is to open the connection to the database within the worker itself rather than passing an open connection as an argument, since all arguments to the worker functions must be serializable. An example of the `doDbPar` function is shown below.

Example 17.2: Example of a serializable function that can be executed by multiple threads.

```
function doDbPar(dbName, host, user, password, dataUUIDs)
   if ~isempty(dataUUIDs)
      DB = Mobbed(dbName, hostName, userName, password, false);
      for k = 1:length(dataUUIDs)
          dataset = db2mat(DB, dataUUIDs{k});
          % Do stuff to this dataset
      end
      close(DB);
  end
end
```

The distribution provides a number of example functions that illustrate various uses of parallel processing.

18. Using stored credentials

The scripts in this user manual explicitly contain user password information, which causes several difficulties from a software distribution and security point of view. To alleviate these difficulties, MOBBED provides users with the option of storing their credential information as a property file on their presumably in a password-protected local machine. The property file is just a text file with name-value pairs specifying the username, password, and other information.

Example 18.1: A sample credential file for creating or access a database.

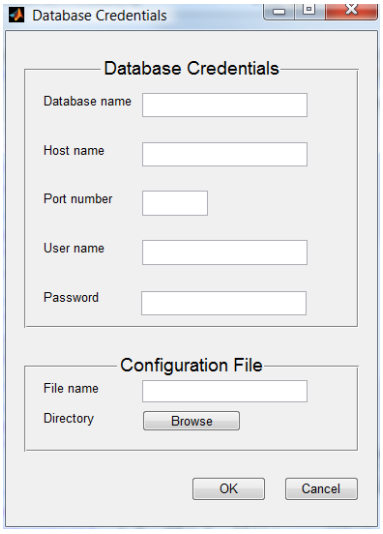
```
#Thu Apr 11 16:21:52 CDT 2013
hostname=localhost\:5432
password=admin
dbname=mobbed
username=postgres
```

The `createCredentials` static method of `Mobbed` displays a GUI that allows the user to specify values for the various credential properties and presumably in a password-protected local machine. The property file is just a text file with name-value pairs specifying the username, password, and other information.

Example 18.2: The following call brings up a GUI to create a credential file.

```
configPath = Mobbed.createCredentials
```

When you close theGUI, the `configPath` variable contains the full pathname of the configuration file for future use. The GUI is shown in the figure below.



Once you have created a credential file, MOBBED provides three static methods for creating and accessing a database using these credentials:

```
Mobbed.createdbc(filename, script)
Mobbed.deletedbc(filename)
DB = Mobbed.getFromCredentials(filename)
```

The `createdbc` method creates a database assuming all of the credential information is in the property file specified by filename. The `getFromCredentials` method returns a handle to a `Mobbed` object corresponding to a database with credential information in the property file. These methods allow users to distribute working scripts without exposing their username/password information.

19. Acknowledgments

The authors acknowledge helpful conversations with Christian Kothe, Nima Bigdely Shamlo, Alejandro Ojeda, Arno Delorme, and Scott Makeig, all of University of California San Diego as well as Scott Kerick, and Jeanne Vettel of the Army Research Laboratories, and Tony Johnson of DCS Corporation. This research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-10-2-0022. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

20. References

pgAdmin PostgreSQL administration and management tool homepage [Online]. Available: http://www.pgadmin.org/development/ [Accessed].

phpPgAdmin homepage [Online]. Available: http://phppgadmin.sourceforge.net/doku.php?id=start [Accessed].

Delorme, A., Mullen, T., Kothe, C., Akalin Acar, Z., Bigdely-Shamlo, N., Vankov, A., and Makeig, S. (2011). EEGLAB, SIFT, NFT, BCILAB, and ERICA: New Tools for Advanced EEG Processing. *Computational Intelligence and Neuroscience* 2011.

Hartl, M. (2012). *Ruby on Rails Tutorial: Learn Web Development with Rails (2nd Edition).* Addison-Wesley Professional.

Hossain, A., Cockfield, J., and Robbins, K.A. (2012). "MOBBED (Mobile Brain Body Environment Decision Making) Data Infrastructure". University of Texas at San Antonio).