DPDQ: Differentially Private Data Queries

Staal A. Vinterbo

Time-stamp: <2015-03-09 18:08:10 staal>

Contents

Synopsis . . . 2
- Main scripts . . . 2
- Supporting scripts . . . 5

Executive summary . . . 6
- Rationale . . . 6
- Capabilities . . . 6
- System Overview . . . 6

Features . . . 9

Using *DPDQ* . . . 10
- Pre-deployment . . . 10
- Using the system . . . 12
- Scaling . . . 19

Details . . . 25
- Differential Privacy . . . 25
- The Gnu Privacy Guard . . . 25
- Protocols . . . 26
- Adding new query types . . . 30
- Risk accounting policies . . . 31

Miscellaneous . . . 31
- Automated deployment example . . . 31
- Logistic Regression . . . 32

Acknowledgements . . . 32

Author Staal A. Vinterbo (sav@ucsd.edu)

Download Download installable Python package here.

Printer Friendly PDF version of this document: here.

Synopsis

DPDQ is a client/server approach to secure and interactive querying of data with differential privacy risk expenditure accounting.

Main scripts

dpdq_qserver.py

```
usage: dpdq_qserver.py [-h] [-k KEY] [-g GPGHOME] [-p QUERY_SERVER_PORT]
                       [-S RISK_SERVER_KEY] [-A RISK_SERVER_ADDRESS]
                       [-P RISK_SERVER_PORT] [-l LOGFILE] [-q QUERYMODULE]
                       [-v] [--allow_alias] [--allow_echo]
                      database_url

Query processing server (version: 0.25). This program allows clients to
request information about datasets in the database it is connected to. As
these requests potentially carry privacy risk, a risk accountant is queried
for adherence to the current risk policy before serving the information to the
client.

positional arguments:
  database_url          The RFC 1738 style url pointing to the database. The
                        format is
                        dialect+driver://username:password@host:port/database.

optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     the key identity of the query server user (default:
                        "QueryServer").
  -g GPGHOME, --gpghome GPGHOME
                        the folder in which to find key ring files (default:
                        ".").
  -p QUERY_SERVER_PORT, --query_server_port QUERY_SERVER_PORT
                        the ip address port the query processing server will
                        listen to client requests on (default: 8123).
  -S RISK_SERVER_KEY, --risk_server_key RISK_SERVER_KEY
                        the key identity for the risk accountant server user
                        (default: "RiskAccountant").
  -A RISK_SERVER_ADDRESS, --risk_server_address RISK_SERVER_ADDRESS
                        the risk accountant server address (default:
                        "localhost").
  -P RISK_SERVER_PORT, --risk_server_port RISK_SERVER_PORT
                        the port the risk accountant server listens on
                        (default: 8124).
  -l LOGFILE, --logfile LOGFILE
                        the file that information about the state of the query
                        processing server and communications with both clients
                        and the risk accountant is written to (default:
                        "query.log").
  -q QUERYMODULE, --querymodule QUERYMODULE
                        a python module that contains additional query type
                        processors (default: "None").
```

```
  -v, --version         display version number and exit.
  --allow_alias         Allow connecting clients to query on behalf of another
                        user. This should only be allowed if the client is
                        trusted, e.g., implements external identification and
                        authentication.
  --allow_echo          Allow echo requests. Useful for debugging clients.
```

dpdq_rserver.py

```
usage: dpdq_rserver.py [-h] [-k KEY] [-P RISK_SERVER_PORT] [-e {threshold}]
                       [-g GPGHOME] [-l LOGFILE] [-m MODULE] [-v]
                       database_url

Risk Accounting Server (version: 0.25). This program answers requests about
users' privacy risk history and whether a user is allowed to incur further
risk according to the current risk policy.

positional arguments:
  database_url          The RFC 1738 style url pointing to the risk database.
                        The format is
                        dialect+driver://username:password@host:port/database.

optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     the key identity for the risk accountant user
                        (default: "RiskAccountant").
  -P RISK_SERVER_PORT, --risk_server_port RISK_SERVER_PORT
                        the ip address port the risk accountant listens for
                        requests on (default: 8124).
  -e {threshold}, --enforce_policy {threshold}
                        risk management policy to enforce (default:
                        "threshold").
  -g GPGHOME, --gpghome GPGHOME
                        the folder in which to find key ring files (default:
                        ".").
  -l LOGFILE, --logfile LOGFILE
                        the file information about transactions and the state
                        of the accountant is written to (default: "risk.log").
  -m MODULE, --module MODULE
                        a python module that contains additional policies
                        (default: "None").
  -v, --version         display version number and exit.
```

dpdq_web.py

```
usage: dpdq_web.py [-h] [-k KEY] [-s QUERY_SERVER_KEY] [-p PORT] [-g GPGHOME]
                   [-f HOSTSFILE] [-u USER] [--use_alias_fingerprint]
                   [-i INFOPAGE] [-v] [-d]

DPDQ web server (version: 0.25). This program starts a Twisted web server that
allows connecting clients to request information from and about datasets from
a query processing server.
```

```
optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     the key identifier for the web-server
  -s QUERY_SERVER_KEY, --query_server_key QUERY_SERVER_KEY
                        the key identifier for the query server user (default:
                        "QueryServer").
  -p PORT, --port PORT  the webserver port (default: 8082).
  -g GPGHOME, --gpghome GPGHOME
                        the directory in which to find key ring files
                        (default: ".").
  -f HOSTSFILE, --hostsfile HOSTSFILE
                        URL pointing to known hosts python dict.
  -u USER, --user USER  The user to query on behalf on, if not given by
                        REMOTE_USER (default: Demo).
  --use_alias_fingerprint
                        use the alias key fingerprint as identifier when
                        sending requests.
  -i INFOPAGE, --infopage INFOPAGE
                        URL pointing to the DPDQ homepage. Used in the help
                        dialog to point to more information. If not supplied
                        http://ptg.ucsd.edu/~staal/dpdq is used.
  -v, --version         display version number and exit.
  -d, --debug           display debug info.
```

dpdq_cli.py

```
usage: dpdq_cli.py [-h] [-k KEY] [-s QUERY_SERVER_KEY]
                   [-a QUERY_SERVER_ADDRESS] [-p QUERY_SERVER_PORT]
                   [-g GPGHOME] [-u USER] [-f] [-v] [-n] [-d]

Text based query client (version: 0.25). This program allows requesting
information from and about datasets from a query processing server.

optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     the key identifier for the user this client acts on
                        behalf of (default: "Alice").
  -s QUERY_SERVER_KEY, --query_server_key QUERY_SERVER_KEY
                        the key identifier for the query server user (default:
                        "QueryServer").
  -a QUERY_SERVER_ADDRESS, --query_server_address QUERY_SERVER_ADDRESS
                        the query server ip address (default: "localhost").
  -p QUERY_SERVER_PORT, --query_server_port QUERY_SERVER_PORT
                        the query server ip address port (default: 8123).
  -g GPGHOME, --gpghome GPGHOME
                        the directory in which to find key ring files
                        (default: ".").
  -u USER, --user USER  the user alias to act on behalf on (default: "None").
  -f, --filter          act like a filter: read commands from stdin and print
                        output to stdout. This is useful for batch execution
                        of commands.
  -v, --version         display version number and exit.
  -n, --nowrite         disallow writing results to file.
  -d, --debug           display debug info.
```

Supporting scripts

dpdq_riskuser.py

```
usage: dpdq_riskuser.py [-h] [-g GPGHOME] [-t TOTALTHRESHOLD]
                        [-q QUERYTHRESHOLD] [-u] [-k] [-i] [-v] [-n]
                        database_url user

Add user to the risk accounting data base. Creates database if it does not
exist.

positional arguments:
  database_url          The RFC 1738 style url pointing to the database. The
                        format is
                        dialect+driver://username:password@host:port/database.
  user                  the user identifier.

optional arguments:
  -h, --help            show this help message and exit
  -g GPGHOME, --gpghome GPGHOME
                        the directory in which to find key (and possibly
                        database) files (default: ".").
  -t TOTALTHRESHOLD, --totalthreshold TOTALTHRESHOLD
                        the total risk threshold (default: 10).
  -q QUERYTHRESHOLD, --querythreshold QUERYTHRESHOLD
                        the per query risk threshold (default: 3).
  -u, --update          Update existing user instead of add
  -k, --kill            Delete existing user
  -i, --info            Show user information and history
  -v, --version         display version number and exit.
  -n, --nokey           Use user identifier directly instead of looking up key
                        fingerprint.
```

dpdq_csv2db.py

```
usage: dpdq_csv2db.py [-h] [-v] [-n] database_url csvfile

Import a data set in csv file into the database. Creates meta data. Note that
the meta data created is computed from the data and is *not* differentially
private. The dataset is given the basename of the csvfile. If the database
does not exist, it is created.

positional arguments:
  database_url         The RFC 1738 style url pointing to the database. The
                       format is
                       dialect+driver://username:password@host:port/database.
  csvfile              the csv file the data is in.

optional arguments:
  -h, --help           show this help message and exit
  -v, --version        display version number and exit.
  -n, --no_categorize  Don't interpret integer columns with a full complement
                       of values in range(x) where x < 5 as categorical
```

Executive summary

Rationale

How well privacy is protected when making information available about a group of individuals is still a question that generally remains unanswered. Not knowing how good privacy safeguards are can result in inadequate or overly conservative protection, as well as reluctance in individuals to consent to the use of their data. Quantifying privacy risk associated with privacy safeguards is necessary for balancing risk and information shared, allowing selection of appropriate safeguards, as well as providing guarantees regarding privacy to individuals.

Differential privacy is an emerging standard for quantifying privacy risk to an individual incurred by the disclosure of information computed from a dataset. Risk measured by differential privacy can be tracked over time, different datasets, and information computed by different methods, allowing for implementation of risk accounting policies.

DPDQ aims at being a lightweight, scalable, and easily deployable networked system for making information in datasets available in a secure and differentially private manner, as well as serve as a platform for a growing number of differentially private methods. A key feature is the ability to incorporate new datasets, methods for query processing, and implementations of risk accounting policies with minimal effort and disruption of service.

Capabilities

The system comes with three different query types preinstalled. Among these is the histogram query type. A histogram can be thought of as a "compression" of data, by construction of equivalence classes in the data, and only keeping a representative for each class together with a count of the number of class members. Creating equivalence classes can be done by projecting the data onto a subset of attributes/dimensions, as well as partitioning attribute values by, for example, discretizing.

Given a histogram, one can "reconstitute" data by

- expanding equivalence class representatives into a number of rows proportional with the size of the class
- reversing any attribute value partitioning by randomly substituting by randomly substituting a value in the partition class (e.g., a discretization range).

This reconstituted data can then be visualized and analyzed using standard tools freely without any further privacy risk.

It is easy to add new query types, as well as risk accounting policies, as run-time loadable modules. For example, the complete code needed for a loadable risk policy that implements the current default policy is given as an example below. A more sophisticated example, can be seen in the complete module that implements a new query type that implements differentially private logistic regression.

System Overview

DPDQ consists of three main parts:

1. a *client* that sends a query to a query processing server,
2. a *query processing server*, and
3. a *risk accountant server*.

The client:

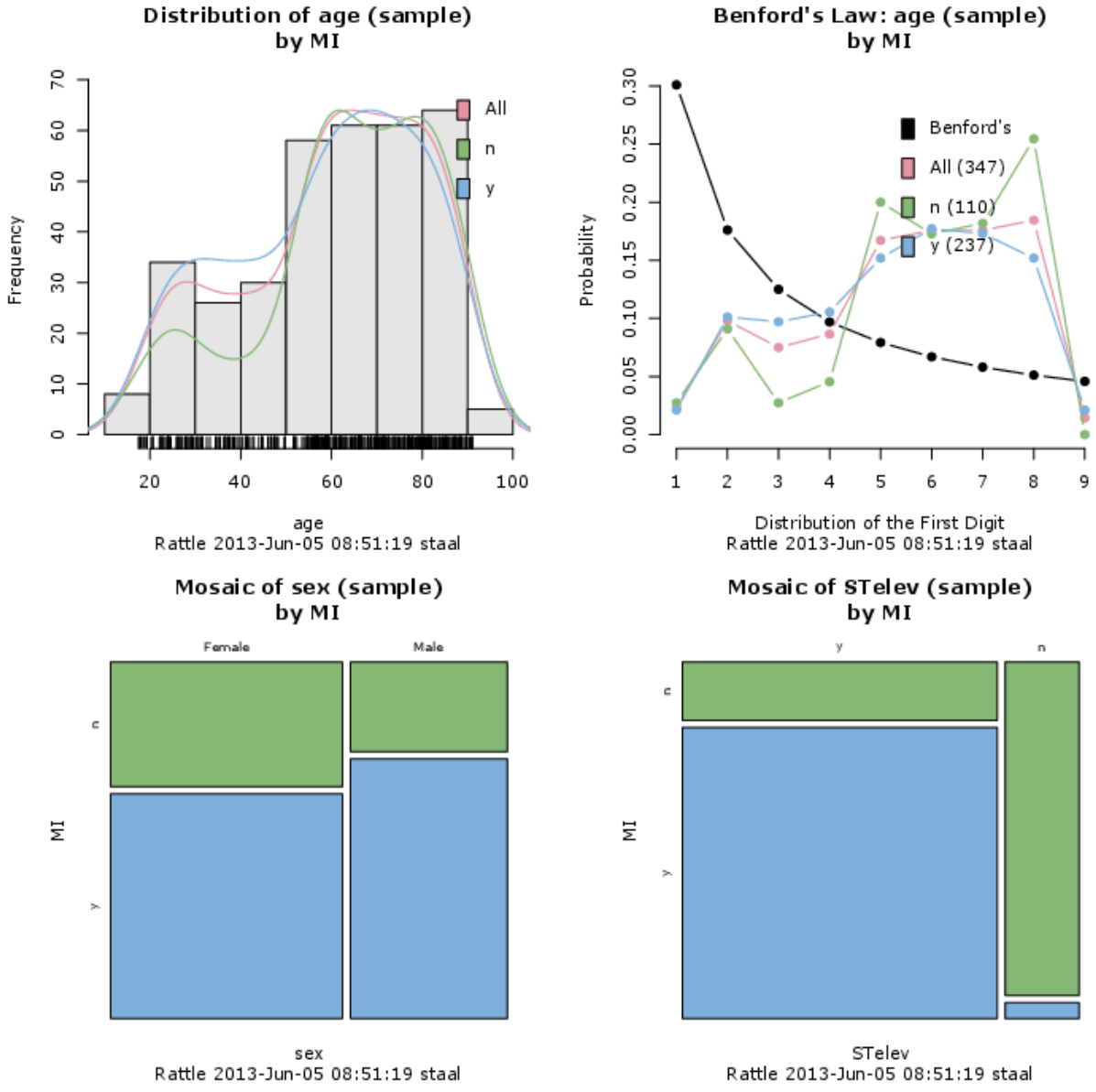

Figure 1: Post histogram query analysis of myocardial infarction (MI) data

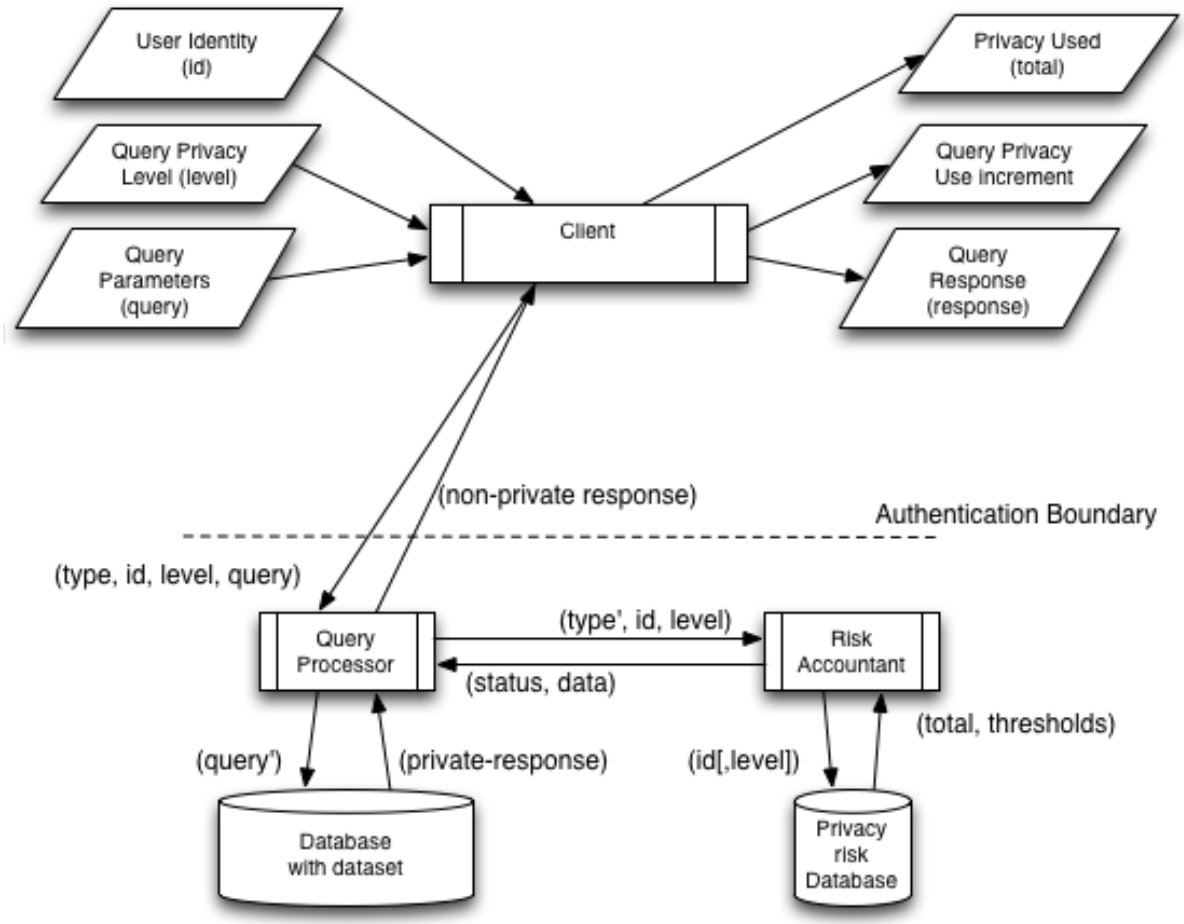

Figure 2: The interactions in the DPDQ system

a. allows the user to formulate a query, typically a statistic or a histogram for a subset of the data,
b. allows the user to send the query to the server
c. displays the server response

The query processing server:

a. receives a query from the client
b. asks the risk accountant server whether the user is registered there, the requested query privacy risk level is below that users predetermined per query threshold, and whether the user is still within his/her allotted total privacy risk budget.
c. if the risk accountant approves, the processing server computes a differentially private response and sends it back to the client, otherwise notifies the client about the response from the risk accountant.

The risk accountant server:

a. determines if the requested risk is allowed under the current risk accounting policy. The default policy is that the risk is allowed if it is below the user's per query risk amount, and the sum of the current total and the requested risk is below the user's total allowed cumulative risk amount.
b. notifies the processing server whether the client can be given the query response subject to the current accounting policy, and if the risk is allowed the user is credited with it.

All communication is encrypted. Identification and authentification is done using public key fingerprints and cryptographic signatures, avoiding the use of passwords that can be forgotten or inadvertently shared.

Features

- **Secure**
  - Public key encryption based identification and authentication: no passwords transmitted.
  - All communication is encrypted.
  - Allows choice of most suitable database management system.
- **Privacy Preserving**
  - Differentially private with accounting of risk expenditure over time that supports scriptable risk accounting policies.
  - Allows guarantees of privacy that are separate from guarantees of access control.
  - Allows guarantees of de-identification.
- **Easily deployable**
  - Simple setup that requires only python and GPG.
  - Supports many database backends "out of the box", including SQLite, MySQL, Postgresql, Oracle, SQL Server, and Firebird.
  - Designed to support self-configuring clients: clients configure the user interface dynamically to reflect the currently relevant dataset and available query types.
  - No redeployment or software update required when adding
    * risk accounting policies,
    * query types, and
    * datasets.
  - Runs on any platform python and GPG can run on (includes Windows, Linux, MacOS X, CentOS, BSD Unix, System V Unix).

  - Oriented towards data sets and is only dependent on dataset metadata being available, but otherwise agnostic to underlying database schema. This allows control and flexible definitions of data sets.

- **Flexible**
  - Supports hot-pluggable query types and data sets.
  - Supports hot-pluggable risk accounting policies.
  - Minimal deployment does not require any additional database software as SQLite is distributed with python.
- **Scalable**
  - Supports distribution of workload:
    * datasets can be distributed among different databases,
    * multiple servers can use the same database, and
    * multiple query processing servers can share the same risk accounting server.
  - Centralized management is possible through the use of distributed file systems and networked databases.
  - Adaptation to any access control system is possible by using the *DPDQ* web-server behind a reverse proxy.
- **Open source** Allows community involvement as well as site-specific customization

Using *DPDQ*

Pre-deployment

Pre-deployment consists of installation, server key-generation, exchange and signing, and setting up a risk accounting database.

Dependencies *DPDQ* is implemented in Python, and depends on the availability of

- python version 2.7, version 2.7.3 or newer
- the gnu privacy guard
- python packages:
  - twisted version 12.0.0 or higher
  - python-gnupg version 0.3.3 or higher
  - sqlalchemy version 0.8.0 or higher
  - texttable version 0.8.1 or higher
  - numpy version 1.6.1 or higher
  - jinja2 version 2.7 or higher

The SQLite SQL database engine is distributed with python, and can be used for the databases containing both data sets and risk accounting information. However, MySql, PostgreSQL, or other database engines supported by sqlalchemy (see here for more information) can be used if available.

Installation The *DPDQ* distribution is available from the download page, and can be installed by any of the standard ways of installing python packages. An example is:

```
$ pip install dpdq-VERSION.tar.gz
```

where `VERSION` is the version of the package file downloaded. The above command will also download and install the python packages on which *DPDQ* depends.

Information security

Identification, authentication, and communications security In a computer system, *identification* is a user giving the system an identity "token" (typically a username), i.e., saying "I am Alice". *Authentication* is the system satisfying iself (or rather its owners) that the identification is true. Communications security is restricting communication to intended recipients.

In *DPDQ*, all communication between clients, query processing servers, and risk accounting servers, is by passing messages which contain a request (e.g., a query for information in a dataset) or a response. A client can act on behalf of a user, and we will use client and user interchangeably. Each instance of QP and RA are also associated with their respective users. In this context, identification is the determination of the user associated with the sender of the message, and authentication is verification of the sender. Further we associate rights with users, and an unknown user has no rights.

For identification, authorization, and communications security, *DPDQ* depends on public key encryption, and uses The Gnu Privacy Guard for this. In public key encryption schemes, each user has a (public key, private key) pair with the important property that a message encrypted (locked) with one of the keys can be decrypted (unlocked) only with the other. This is used as follows. If Alice has Bob's public key, then she can encrypt a message to Bob with this key. As long as Bob keeps his private key private, he is the only one that can decrypt the message. If Bob has Alice's public key, then she can "sign" a message by encrypting it (or a unique representation of it) with her own private key. Bob can then verify Alice as the sender by checking that he can decrypt the message with Alice's public key.

A *DPDQ* user is identified by his/her public key and signatures for authenticating the sender. This allows the separation of computer system users from *DPDQ* users allowing QP and RA to run in the same computer system users process space. Communications security is achieved by encrypting messages with the recipients public key. An advantage of the public key encryption use in *DPDQ* is that once the keys have been distributed, further communication and system use does not rely on passwords that can be lost, forgotten, and must be entered every time a user wants to log in. The disadvantage is that a user's keys must be available, which can restrict mobility.

Rights management is described next in rights.

Rights A client (users and QP wrt. to RA) can have two rights:

1. communication rights
2. query/risk expenditure rights

Users' communication rights give access to metadata about datasets, letting the user query QP for what datasets and query types that are available. Communication rights for QP also imply query rights.

The communication rights are primarily (primarily because a client also needs the server's (QP or RA) public key, which could be published to the general population facilitating key distribution), given by importing and signing (with the servers private key) the user's public key into the server's keyring. Note that the client also needs to sign the server's key with his/her private key.

A user's query rights equate rights to expend risk (or query about their stored risk allowances) are given by adding the user (identified by the user's public key fingerprint) to the RA's risk database, and at the same time allocating a total risk allowance, as well as a per query risk allowance.

Per dataset query type restrictions Information about datasets is stored in metadata. This information includes what query types are considered appropriate for each dataset, allowing fine-grained control of information type release.

Database security *DPDQ* can be used with any database that sqlalchemy supports. These include SQLite, MySQL, Postgresql, Oracle, SQL Server, and Firebird. Communication with these is done through [Python DBAPI v2.0(http://www.python.org/topics/database/DatabaseAPI-2.0.html) compliant drivers providing access to their respective client APIs. This allows the choice of database (among the available) that implements the most appropriate security features. For Oracle 11g, release 1, information on how to secure client connections is given here. These instructions (as do similar instructions for postgresql v 9.1) include securing database client to database server by encrypting the connection. If a database does not support this natively, this can be done by "tunneling". An example is ssh port forwarding, where the application that is being tunneled can stay oblivious of this fact.

The risk accounting database The risk accounting database (RAD) is where RA keeps information about user's

- overall risk threshold. A user's cumulative risk expenditure is not allowed to exceed this.
- per query risk threshold. An individual query is not allowed to cause risk beyond this threshold.
- risk expenditure history.

Users are identified by their GPG public key fingerprint. The risk database is relational and has two tables:

```
users(id text, info text, tt real, qt real)
history(id text, eps real, time text)
```

where `id` is the user's public key fingerprint, `info` is information about who the user is, `tt` is overal total threshold, `qt` is the per query threshold, `eps` is a query expenditure, and `time` is the timestamp RA put on this expenditure. For sqlite the following SQL statements create these tables:

```
CREATE TABLE history (
    id VARCHAR(60),
    eps FLOAT NOT NULL,
    time DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL,
    FOREIGN KEY(id) REFERENCES users (id)
);
CREATE TABLE users (
    id VARCHAR(60) NOT NULL,
    info VARCHAR(200) NOT NULL,
    tt FLOAT NOT NULL,
    qt FLOAT NOT NULL,
    PRIMARY KEY (id)
);
```

The script `dpdq_riskuser.py` (see Synopsis) can be used for risk database management, and will create the database when adding a user if a database does not exist. Similarly to `dpdq_cvs2db.py`, the database dialect used is determined by the URL for the database. For example, to create an SQLite database in the current directory and insert Alice into it, one can use: `dpdq_riskuser.py sqlite:///risk.db Alice`.

Using the system

After the pre-deployment, datasets and users have to be added, and use has to be monitored.

Adding data sets As presented in the section on features, *DPDQ* operates on datasets as opposed to databases, even though the datasets happen to be stored in databases. This allows the system to be agnostic to the underlying database schema, with the exception to dataset metadata tables explained shortly.

In practice, datasets are either tables or views. Each database can hold many datasets, and in order to expose these to the QP, and the client through the QP, the QP must have a standard way to find the datasets. This is accomplished through metadata tables. These tables hold information about

- which datasets are available
- for each data set:
  - name
  - textual description
  - for each attribute in the data set:
    - type
    - textual description
    - for each categorical value
      - name
      - textual description
    - value ranges for numeric data
  - which differentially private methods are known to be appropriate for this dataset

The above is implemented by the following tables

```
datasets(name, size, description)
attributes(name, set, type, description)
bounds(attribute, set, lower, upper)
values(attribute, set, name, description)
processors(set, type)
```

where `datasets.name` is the name of the table or view representing the dataset, `size` is an upper bound on the size of the data set, `datasets.description` is a description of the data set, `attribute.name` is a column name in the dataset identified by `attribute.set`. The type of the attribute is held in `attribute.type` and can take on the following values:

```
0.  categorical,
1.  integer,
2.  float,
3.  string,
4.  date (tbd),
```

A description of the attribute is stored in `attribute.description`. If the column `bounds.attribute` in dataset `bounds.set` is of a numerical type (1 and 2), upper and lower bounds (mainly used by the differentially private methods) are stored in `bounds.lower` and `bounds.upper`, respectively. If an attribute is categorical, then each of its value and a description is stored in `values`. Finally, for each dataset named in `processors.set` has the names of its known appropriate query types (methods) stored in `processors.type`.

SQLite statements for creating these tables are:

```
CREATE TABLE datasets (
    name VARCHAR(50) NOT NULL,
    size INTEGER NOT NULL,
```

```
    description VARCHAR(200) NOT NULL,
    PRIMARY KEY (name)
);
CREATE TABLE attributes (
    name VARCHAR(50) NOT NULL,
    "set" VARCHAR(50),
    type INTEGER NOT NULL,
    description VARCHAR(200) NOT NULL,
    PRIMARY KEY (name),
    FOREIGN KEY("set") REFERENCES datasets (name)
);
CREATE TABLE bounds (
    attribute VARCHAR(50),
    "set" VARCHAR(50),
    lower FLOAT NOT NULL,
    upper FLOAT NOT NULL,
    FOREIGN KEY(attribute) REFERENCES attributes (name),
    FOREIGN KEY("set") REFERENCES datasets (name)
);
CREATE TABLE "values" (
    attribute VARCHAR(50),
    "set" VARCHAR(50),
    name VARCHAR(50) NOT NULL,
    description VARCHAR(200) NOT NULL,
    FOREIGN KEY(attribute) REFERENCES attributes (name),
    FOREIGN KEY("set") REFERENCES datasets (name)
);
CREATE TABLE processors (
    "set" VARCHAR(50),
    type INTEGER NOT NULL,
    FOREIGN KEY("set") REFERENCES datasets (name)
);
```

When adding a dataset these tables must be updated as meta data is needed for three things:

1. Passing to the client to help with query formulation (e.g., creating a user interface to support this),
2. translation of client query into executable SQL implementing the selection, and
3. as attribute co-domain (e.g., maximum range) descriptions for the query processors.

A script to help with the import of a dataset in CSV format is provided as `dpdq_csv2db.py` (see Synopsis). Note that this tool extracts bounds and values from the data and consequently these are not differentially private. In order to be able to guarantee differential privacy, the meta data must be independent of the particular dataset. After the use of the tool, the ranges should be edited to achieve this. As an example, the bounds for an attribute "age" should be chosen without looking at the data (e.g., 0 to 100).

User management A client (on behalf of a user) can have two types of rights, as mentioned in the section on Information security. To get communication rights, the user must exchange public keys with the server in question (see Information security). Furthermore, both the user and the server must sign each others keys. To get query rights, the user must be registered in the risk accounting database used.

For adding, removing, as well as changing the risk profile (see The risk accounting database), the script `dpdq_riskuser.py` is provided (see Synopsis).

Running the servers In order for a query processing server and a risk accounting server to be able to communicate, the need communication rights and must exchange public keys with the server in question (see Information security).

The servers can be started as shown in the Synopsis. In the example deployment script (see Automated deployment example), they are started similarly to:

```
$ dpdq_rserver.py -g /tmp/dpdq/r sqlite:////tmp/dpdq/r/risk.db
$ dpdq_qserver.py -g /tmp/dpdq/q sqlite:////tmp/dpdq/q/warehouse.db
```

Query processing server The query processing server can load new metadata from its database, updating its knowledge on available datasets, as well as load additional query types from a python file or package given by the `--querymodule` option. Both of these can be done at startup, or is done when the server is sent the SIGUSR1 signal. For details on adding new query types this way see the section on new query types. For other startup options, see Synopsis.

Risk accounting server The risk accounting server can load additional risk accounting policies by importing a package implementing policies as described in Risk accounting policies. This package, given by using the `--risk_policies` option, can be loaded on startup, or by sending the SIGUSR1 signal to the process. At startup, any loaded policy can be activated by the `-e` or `--enforce_policy` option. The default policy called `threshold` denies information queries if the requested risk plus the user's current total expenditure exceed the total threshold set for the user, or if the requested risk exceeds the per query threshold set for the user. For other startup options, see Synopsis.

Monitoring The servers and users can be monitored by keeping track of the log files (see Synopsis for how to specify the location of the log files), and the risk histories in the risk databases.

Using the text based client The text based client is designed to help compose a query for information on a dataset. It offers a command prompt where a user can enter commands.

A query consists of:

- the dataset of interest
- the columns (variables) in that dataset of interest (at least one is needed)
- the predicate determining wanted dataset rows
- the type of query (wanted type of information)
- the value of any parameters for that query type
- the epsilon quantifying differential privacy risk you allow

Each of the above have an associated command, and once all are determined, the query can be run.

If your system allows it, the client will use TAB completion. This means that pressing TAB will complete what you are currently typing if there is a unique way to do so. If there is no unique completion, pressing TAB again will list all possible completions of what you are currently typing.

The available commands are:

`help [<command>]` get help. If a command is given, help is given on that command. `dataset DATASET`
select the dataset DATASET to query. `columns COLUMN [COLUMN]*`
set columns (data attributes) you are interested in. If left unset all columns are used. `predicate PREDICATE`

input selection predicate. A predicate is a disjunction of a possibly negated conjunction of terms. Example: `gender == male and age > 30 or not gender == female and age > 30`. Note that categorical values should not be quoted. An empty predicate selects all. `type TYPE`
sets query type. A query type is the information item to be computed from the rows in the dataset that match the predicate. `value PARAMETER VALUE`
set parameter for query type. `run [OUTFILE]`
send query to query server. If OUTFILE is given, the computed result is copied to the file. `list datasets | types`
list available datasets or query types. `show settings | dataset DATASET | type TYPE | risk`
show settings or info about dataset, query type, or user risk levels. `quit`
quit the client.

The client can also read a list of commands from standard input if given the `-f` (or alternatively `--filter`) option. For example if a file `commands.txt` contains the following:

```
dataset iris
columns Species
type simple_count
predicate Species == versicolor and Petal_Length < 4
run
```

and we give the following command (linux/unix):

```
$ cat command.txt | dpdq_cli.py -g ~/.gnupg -f
```

the output looks similar to this:

```
count: 13
```

Using the web server/graphical web user interface *DPDQ* also comes with a web-server `dpdq_web.py`. The web-server acts as a client to a query processor, and any connecting client "inherits" the web-server's communication rights (see rights).

Important: as the web server is designed to run behind a reverse proxy (see Implementing external user management) that takes care of securing external communication, as well as authentication, it does **not** perform any authentication of connecting clients.

It is however possible to use the web-server on a single user machine (e.g., a laptop or desktop) without the reverse proxy by using a firewall to restrict connections to the web server to originate from the local machine. On a linux machine using iptables this can be done for port 8082 by:

```
$ iptables -A INPUT -i lo -p tcp --dport 8082 -j ACCEPT
$ iptables -A INPUT -p tcp --dport 8082 -j DROP
```

We can then replace

```
$ dpdq_cli.py -k Alice -u Demo
```

with

```
$ dpdq_web.py -k Alice -u Demo -p 8082
```

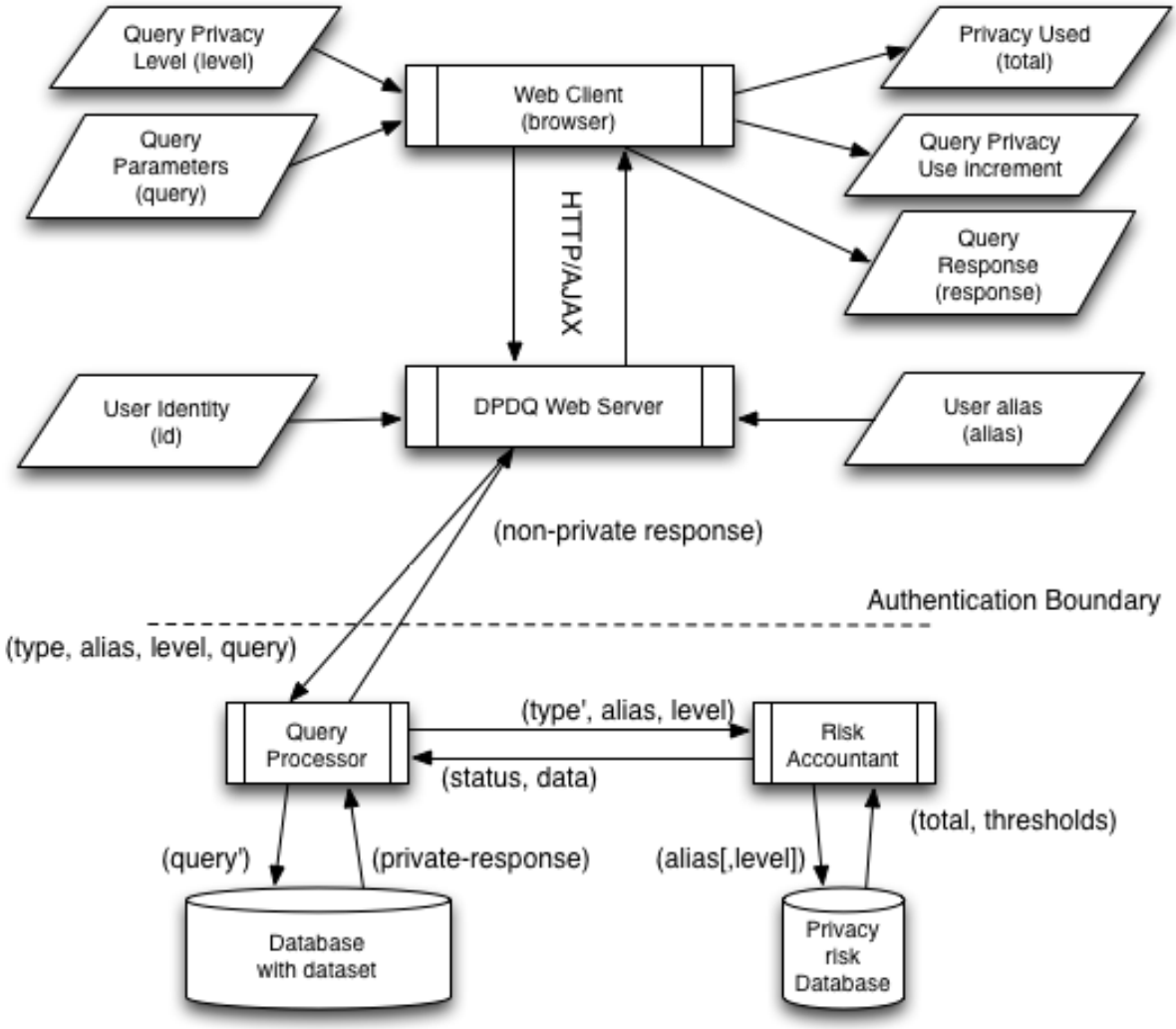

Figure 3: The web-server QP client

or

```
$ dpdq_cli.py -k Alice
```

with

```
$ dpdq_web.py -k Alice -u Alice --use-alias-fingerprint -p 8082
```

and point the browser to `localhost:8082`.

Note that the query processor server must be run with the `--allow-alias` option for the `-u` option to have any effect. If the query processor does not allow aliases to be given, the user associated with the web server's key is used (which might be what is wanted in the single user scenario) .

The graphical user interface the web server presents provides the same functionality as the text client, including downloading data created from histogram queries.

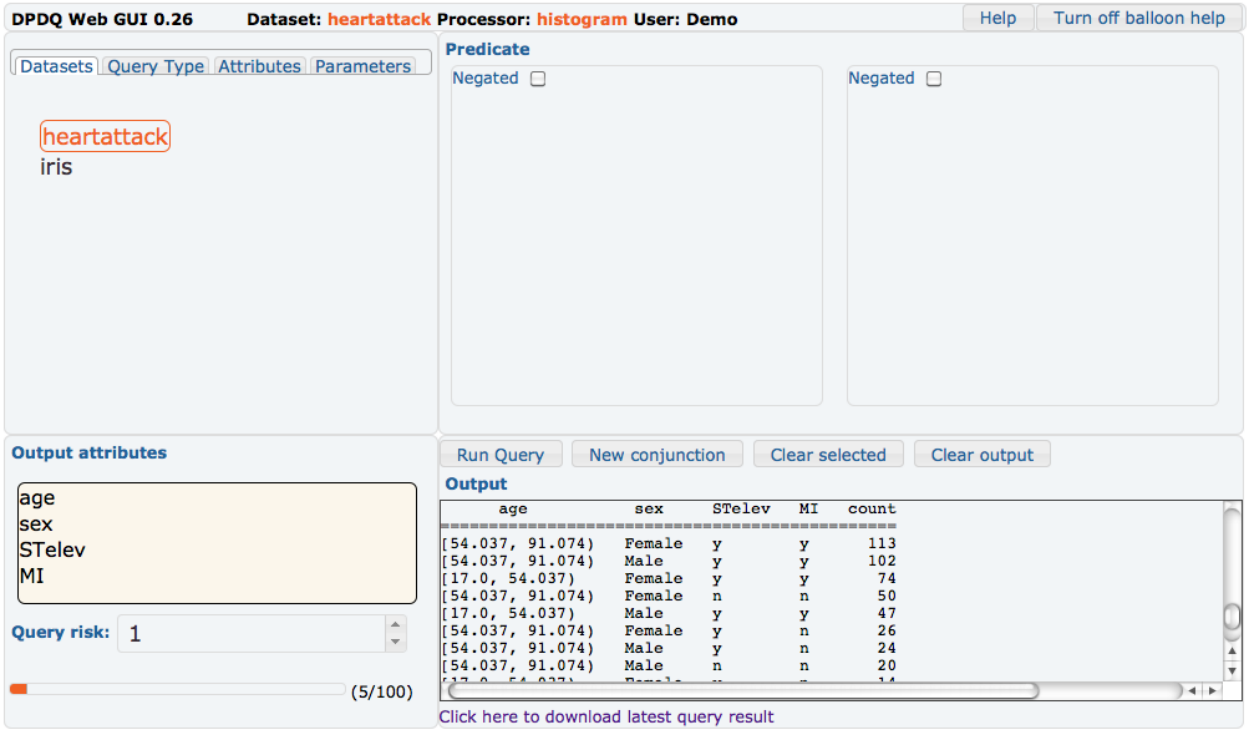

Figure 4: The DPDQ graphical user interface.

On connecting, the user is presented with a page allowing a selection among available query processors. The configuration of these is done by supplying the `--hostsfile` option (see Synopsis).

The graphical user interface (GUI) has the following main components:

- Status header. Here the currently selected data set and query type are displayed.
- A tabbed selection access box in the upper left corner. In this box, the user has access to dataset, query type, attributes, and query type parameters, with a tab for each.

- Selection "receiving" areas for attribute selections. Each is activated selecting it (mouse click or finger touch on touch devices), then populated by selecting attribute names from the attribute selection tab. Selecting an attribute name in an area will remove the entry from that area.

  There are two types of receiving areas, the first is the "Output Attributes" area below the tabbed selection box. Query types that use specific attributes will receive these as input. The second type is the "predicate conjunction" box. A predicate is a disjunction (logical or) of possibly negated conjunctions (logical and) of propositions. Each conjunction has its own receiving area. When active, selecting an attribute from the attributes tab inserts an "attribute operator value" proposition into the conjunction box. Selecting the operator part opens a drop-down menu of choices. If the attribute is categorical, selecting the value part will do the same. Numerical and string attributes can be edited in place. Selecting the attribute name part will remove the proposition from the conjunction.

  Initially, two conjunction areas are shown, but any number can be added by selecting the "New conjunction" button. Added conjunction can be deleted again by selecting the "Delete" button inside it. Checking the "Negated" checkbox will negate the meaning of the conjunction. If all conjunction boxes are empty, the predicate is taken to be true for all, meaning that all the data is given to the method implementing the query type.

  The active selection area can be cleared by selecting the "Clear selection".

- Risk selection and monitoring area. Here the user can select the differential privacy risk requested for the next query. The box also shows how much the user has expended, together with a progress bar that indicates the accumulated expenditure in relation to a set threshold.

- Output box. Responses from the query processor are shown here.

A query is sent to the query processor by selecting the "Run query" button.

Tooltip (balloons with text that appear when hovering over particular areas) help is activated by default. This can be turned off, and on again, by selecting the button with the corresponding text in the upper right corner.

Scaling

The basic system has five compontents:

- The client,
- the query processing server (QP),
- the risk accounting server (RA),
- the database containing the data sets (Datasets DBMS), and
- the risk accounting database (Risk DBMS).

The client need only connect to the QP, the QP connects to both the Datasets DBMS and the RA, while the RA connects to the Risk DBMS and accepts communications from usually only the QP. Both the QP and RA are implemented as single-threaded programs. While both are event-driven, for the QP this means that it can receive another query while it is waiting for a response from the RA, queries are generally processed in a first come first served manner. This means that if a computation, e.g., computing a query response for the QP or applying a policy for the RA, takes a long time, new clients will notice a delay in service and even time out.

There is, however, nothing wrong with deploying multiple QP and RA instances at the same time. This scenario can be seen here:

Particularly useful in this context is "deamonizing" a server that needs to perform significant computations. Here, a daemon process receives an incoming request and starts a new server for that connection. This new server has as its only task to serve that connection. Modern operating systems such as unix/linux variants have tools for this.

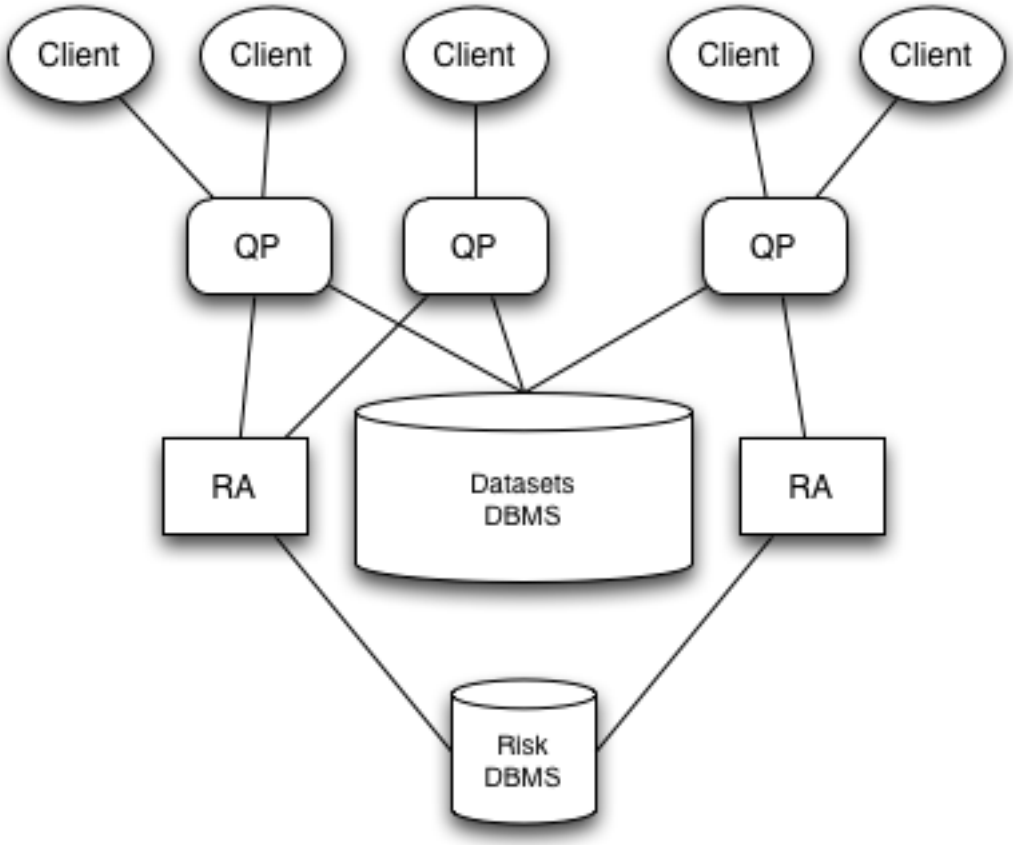

Figure 5: A larger deployment scenario

An even larger scale deployment can consist of multiple instances of the "larger deployment scenario" depicted above.

Centralizing management Both QP and RA can connect to their respective databases over a network connection. This allows centralizing these databases, even within a single centralized database management system or warehouse.

Currently, public and private keys are stored in a file system directory. These can be centralized as well by using (possibly encrypted) networked file systems (e.g., NFS).

Implementing external user management Larger organizations often have their own organization-wide identification and authentication systems. An example is Active Directory. In order to incorporate *DPDQ* into such a system, access to the client can be regulated using the external system. For example a web-server can act as a client, and access to the web-server (client services) can be controlled by the external system. In this scenario, the web-server is the sole client with communication rights for possibly a multitude of QP instances. Information query rights are however still maintained on a per user-base in the Risk accountants risk database, and the `alias` field is used to communicate with the QP. Any mechanism can be used to propagate user query rights to *DPDQ* by inserting user credentials into the risk databases used.

To support external user management, *DPDQ* provides a web-server `dpdq_web.py` (see Synopsis and use description. This web server does not do any user authentication, nor does it support encrypted communication with its clients. This is because it is designed to be used behind a *reverse proxy* that takes care of these things. The user identity to use is read from the `REMOTE_USER` header variable.

A simple example setup using `dpdq_web.py` A simple setup using apache2 `mod_proxy` and `mod_rewrite` as well as basic authentication is described in the following. All processes are running on the same machine.

Assume that a user `dpdq` on a linux/unix type system with world readable home directory `~dpdq` being `/home/dpdq` has directories

```
- `~dpdq/r` -- containing risk accountant server key files and an
  sqlite risk database `risk.db`
- `~dpdq/q` -- containing query processor server key files and an
  sqlite database with datasets `warehouse.db`
- `~dpdq/c` -- containing client key files
- `~dpdq/store` -- for storing information not accessible in the
  web tree.
```

All of these directories should have restricted permissions.

Furthermore, we assume that *DPDQ* is installed, and that we are running an apache2 web server on the machine.

In order to require basic authentication (or adapt for LDAP if active directory is wanted), we make sure that there exists a file `~dpdq/www/.htaccess` containing

```
AuthType Basic
AuthName "Password Required"
AuthUserFile "/home/dpdq/store/.htpasswd"
Require valid-user
```

The file `/home/dpdq/store/.htpasswd` contains usernames and password pairs for users authorized, and we assume that the web server can read the file. If this is not the case,

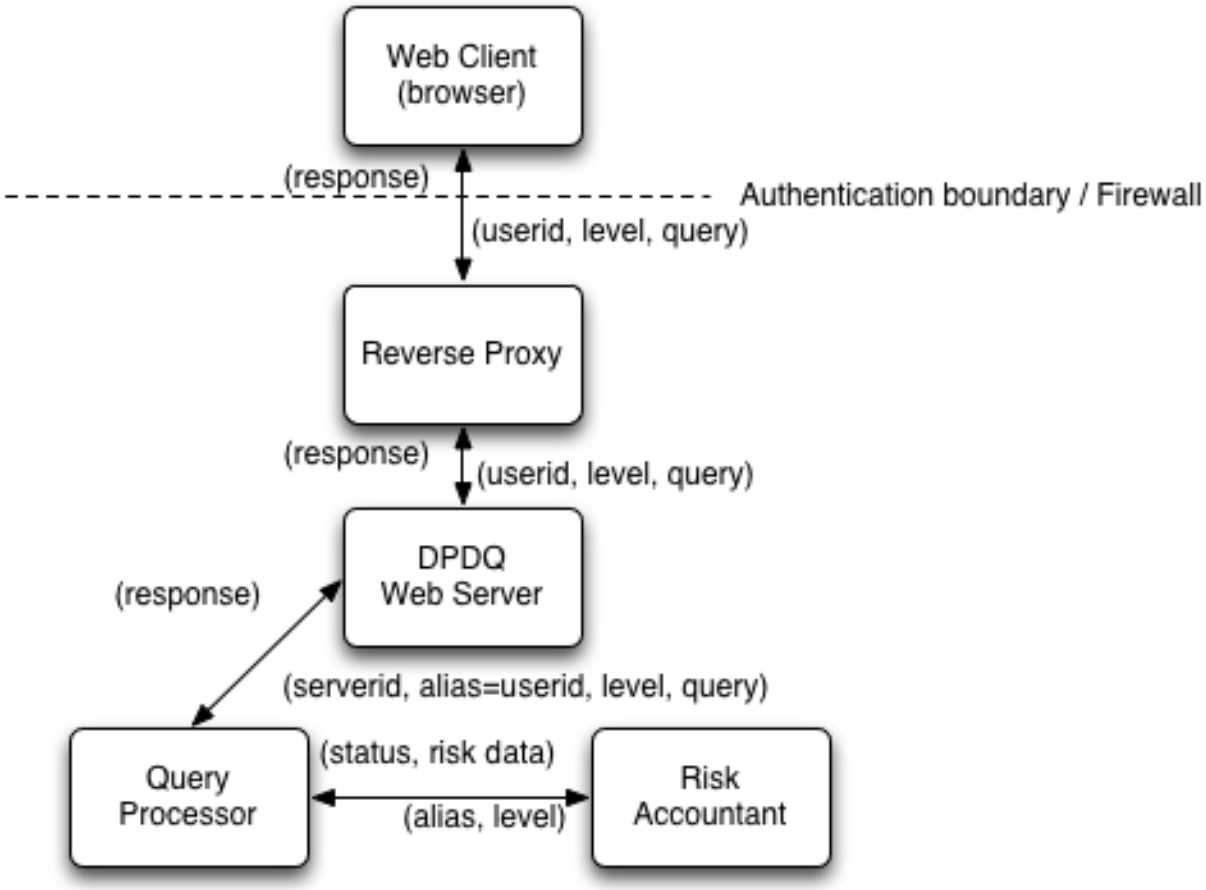

Figure 6: A simple reverse proxy setup

```
$ sudo chgrp www-data ~dpdq/store
$ chmod g+xr ~dpdq/store
$ chmod o-rwx ~dpdq/store
```

could be used.

We now restrict connections to the web server (assuming this server is listening to port 8082) to originate from the local machine by:

```
$ iptables -A INPUT -i lo -p tcp --dport 8082 -j ACCEPT
$ iptables -A INPUT -p tcp --dport 8082 -j DROP
```

Then if our machine has address `ptg.ucsd.edu` (and assuming apache2 has been configured to allow rewrite from .htaccess files) we add the two following lines to `~dpdq/www/.htaccess`:

```
RewriteEngine On
RewriteBase /

RewriteRule ^(.*)$ http://ptg.ucsd.edu:8082/$1 [P,L]
RewriteRule ^$ http://ptg.ucsd.edu:8082/ [P,L]
```

The web-server can give the user a choice of query processors to connect to. These choices are given as a URL that points to what amounts to the text definition of a python dictionary keyed on `address:port` strings with display names as values. For our example, this file `qpservers.txt` contains:

```
{
   'localhost:8123' : 'Demo'
}
```

What remains is to add a user (in this case 'Demo'), and start the servers. This can be done by:

```
# generate a user 'Demo'
$ dpdq_riskuser.py -n sqlite:////home/dpdq/r/risk.db Demo
$ htpasswd ~dpdq/store/.htpasswd Demo

# start servers
$ (cd q; dpdq_qserver.py --allow_alias sqlite:///warehouse.db &> out.log &)
$ (cd r; dpdq_rserver.py sqlite:///risk.db &> out.log &)
$ (cd c; dpdq_web.py -f file:qpservers.txt -p 8082 &> out.log &)
```

The above can be adapted to be more secure by securing communication with the apache 2 web server (e.g., by using ssl), and using a different type of authentication. Furthermore, the setup allows running each component on a separate machine, possibly with a firewall between each element in the communication chain.

A simple setup using `shellinabox` and `dpdq_cli.py` A simple setup that allows web access without any application programming is combining shellinabox with apache cgi. In this setup, `shellinaboxd`, an AJAX based terminal emulator, is used to provide cgi access to `dpdq_cli.py`, and the web-server takes care of identification and authentication. To explain the setup in more detail, assume that a user `dpdq` on a linux/unix type system with world readable home directory `~dpdq` being `/home/dpdq`

1. is allowed to run serve documents and cgi scripts ending in .cgi from `~dpdq/www`, and

2. has directories
    - `~dpdq/r` – containing risk accountant server key files and an sqlite risk database `risk.db`
    - `~dpdq/q` – containing query processor server key files and an sqlite database with datasets `warehouse.db`
    - `~dpdq/c` – containing client key files
    - `~dpdq/store` – for storing information not accessible in the web tree.

    All of these directories should have restricted permissions.

Furthermore, we assume that cgi scripts are run as the `www-data` system user, and that *DPDQ* is installed.

In order to require basic authentication (or adapt for LDAP if active directory is wanted), we make sure that there exists a file `~dpdq/www/.htaccess` containing

```
AuthType Basic
AuthName "Password Required"
AuthUserFile "/home/dpdq/store/.htpasswd"
Require valid-user
```

The file `/home/dpdq/store/.htpasswd` contains usernames and password pairs for users authorized, and we assume that the web server can read the file. If this is not the case,

```
$ sudo chgrp www-data ~dpdq/store
$ chmod g+xr ~dpdq/store
$ chmod o-rwx ~dpdq/store
```

could be used.

Now we need a `cgi` file to call `shellinabox`. Let `~dpdq/www/dpdq.cgi` contain

```
#!/bin/bash

if [ ! -z "${REMOTE_USER}" ]; then
    shellinaboxd --localhost-only --cgi -t \
    -s/:`id -n -u`:`id -n -g`:/tmp:"dpdq_cli.py -g /home/dpdq/c -n -u ${REMOTE_USER}"
else
    cat <<EOF
Status: 401 Unauthorized to access the document
WWW-authenticate: Basic realm="Password Required"
Content-type: text/plain

Login needed.

EOF
fi
```

What remains is to give `www-data` ownership of `~dpdq/c` because gpg will not read key files that do not belong to the executing user which is `www-data`, add a user (in this case 'Demo'), start the servers, and make `dpdq.cgi` executable. This can be done by:

```
# change ownership
$ sudo chown -R www-data:www-data ~dpdq/c
```

```
# generate a user 'Demo'
$ dpdq_riskuser.py -n sqlite:////home/dpdq/r/risk.db Demo
$ htpasswd ~dpdq/store/.htpasswd Demo

# start servers
$ (cd q; dpdq_qserver.py sqlite:///warehouse.db &> out.log &)
$ (cd r; dpdq_rserver.py sqlite:///risk.db &> out.log &)

# make cgi script executable
$ chmod a+x ~dpdq/www/dpdq.cgi
```

The above can be adapted to be more secure by securing communication with the web server (e.g., by using ssl), using a different type of authentication, and by running the web-server and the cgi programs `shellinabox` and `dpdq_cli.py` on a different machine than the query processor and risk accountant, which can be separated from their respective databases as well. If the web-server is compromised, neither the risk database nor the data warehouse then are accessible except by the encrypted protocol.

Details

Differential Privacy

A famous proof by Gregory Chaitin shows that it is impossible to prove that data is random without knowledge about how the data was produced. Using knowledge about how information is produced in order to be able to provide proofs is central to differential privacy, which measures privacy risk resulting from disclosing information computed by a particular randomized method. Formally, we define differential privacy as:

> A randomized algorithm A is ϵ-differentially private if for any measurable set of outputs S, $P\left(A(D) \in S\right) \leq e^{\epsilon} P\left(A(D') \in S\right)$, where D, D' are any two databases of n records that share $n-1$ records in common. The probability is taken over the randomness in A.

In particular, if the information s computed by a method A applied to data D, i.e., $s = A(D)$, differential privacy measures the probability of learning anything about any record r in D, when given access to s and all records of D except r, independent of outside information. Note that learning anything about r also includes the identity of the person r belongs to, and consequently differential privacy also quantifies de-identification. What this means is that one can give a person a guarantee that data participation and the subsequent disclosure of analysis results will not increase the risk of anyone learning anything about her/him by more than a factor e^{ϵ}. The value ϵ can be chosen to comply with privacy policy.

Protecting privacy inherently means a degradation of the quality of information that can be disclosed. For differential privacy, this degradation comes from randomness introduced in the method, with a reciprocal relationship between accuracy of the result and privacy risk incurred. Differential privacy research focuses on optimally balancing this relationship, and successful differentially private methods (i) guarantee very small privacy risk, and (ii) provide accurate results.

The Gnu Privacy Guard

The system depends on public key encryption for identification, authentication, and communications security as described in the section on information security. DPDQ uses the Gnu Privacy Guard(GPG) for this. By default, GPG keeps its files in .gnugp in the current user's home directory. If QP and RA are run by the same user, it might be useful to specify a separate directory for each of QP and RA. The location of the GPG files is given using the `-g` or `--gpghome` option to the supplied scripts. Detailed information on GPG, key generation, key signing, and key management can be found here. A shell script that can serve as an example of how key distribution and signing is described in the automated deployment example.

Protocols

All messages (requests and responses) are

1. converted to their string representations
2. signed, encrypted, and ascii armored by GPG,
3. wrapped in the netstring format, i.e., string "hello world!" is encoded as "12:hello world!,"

before transmission via the Transmission Control Protocol(TCP).

In the following format specifications

`tuple(a,b,...,c)` means a python tuple containing `a,b,...,c` `tuple(t)`
means a python tuple with elements of type `t` `dict(a,b)`
means the python dictionary with key type `a` and value type `b`. `{ ... }`
means an explicitly given python dictionary that must have exactly the keys listed. `|`
separates alternatives `list(a)`
means a non-empty python list of `a` elements `empty_list`
means an empty python list `string`
is a python string `float`
is a python float `number`
is either a `float` or an integer

Client – Query processing server The interaction between client and server consists of a number of rounds where the client sends a string representation of a *request* to the server, and the server sends a string representation of an *answer* back. The client can either request metadata, the user's current risk usage, or an answer to a dataset query.

Format specification:

```
request       = tuple(type, alias, epsilon, query)
answer        = tuple(status, type, response)
type          = get_meta | answer_query | get_user_risk
get_meta      = 0
answer_query  = 1
get_user_risk = 2
response      = dict | string
epsilon       = float
alias         = None | string
```

The field `query` is not used if `type` is `get_meta`, or `get_user_risk` and can be set to `None`. The string `alias` allows a client to act on behalf of another user identified in `alias`. A value for alias other than `None` is taken to be a request of alias use if this is enabled in the query processing server.

The first element in the answer, `status`, is a status code:

```
QP_OK            = 0 # everything is fine
QP_ERROR_BUDGET  = 1 # budget exceeded
QP_ERROR_RA      = 2 # user not found in risk database
QP_ERROR_QUERY   = 3 # malformed query
QP_ERROR_INTERNAL = 4 # internal error (this is bad)
```

The second element in the answer is the request `type` that this is an answer to. If the status code is `QP_OK`, the third element `response` is a dictionary, otherwise it is a string explaining what went wrong. For a risk usage request, the returned `response` dictionary format is:

```
{ 'total' : float,
  'tt'    : float,
  'qt'    : float }
```

where `total` is the key of the current usage, `tt` is the key of maximum cumulative allowed risk, and `qt` is the per query allowed risk.

Dictionaries returned by information requests are query type dependent.

Metadata Metadata is needed for three things:

1. Passing to the client to help with query formulation (e.g., creating a user interface to support this),
2. translation of client query into executable SQL implementing the selection,
3. and as attribute co-domain (e.g., maximum range) descriptions for the query processors.

Metadata format specification:

```
metadata         = { 'datasets'   : dataset,
                     'operators'  : operators,
                     'processors' : processors }

datasets         = dict(name, dataset)
dataset          = { 'name' : name,
                     'attributes ' : attributes,
                     'description' : description,
                     'size'  : integer,
                     'processors' : tuple(name) })
attributes       = dict(name, attribute)
attribute        = num_attribute | cat_attribute | string_attribute
num_attribute    = { 'bounds' : { 'lower' : number
                                  'upper' : number },
                     'description' : description,
                     'type' : 1 | 2 }
cat_attribute    = { 'description' : description,
                     'type' : 0,
                     'values' : dict(name, description) }
string_attribute = { 'description' : description,
                     'type' : 3 | 4 }

operators        = dict(type, opdict)
opdict           = dict(name, { 'description' : description,
                                'literal' : literal })

processors       = dict(name, processor)
processor        = { 'name' : name,
                     'description' : description,
                     'parameters' : parameters })
parameters       = dict(name, parameter)
```

```
parameter        = num_parameter | cat_parameter | string_parameter
num_parameter    = { 'bounds'      : { 'lower' : number
                                       'upper' : number },
                     'description' : description,
                     'default'     : number,
                     'type' : 1 | 2 }
cat_parameter    = { 'description' : description,
                     'type'        : 0,
                     'default'     : name,
                     'values'      : dict(name, description) }
string_parameter = { 'description' : description,
                     'default'     : string
                     'type'        : 3 | 4 }


name             = string
description      = string
literal          = string
number       = float | integer
type             = 0 | 1 | 2 | 3 | 4
```

The value type encoding is:

```
categorical = 0
integer     = 1
float       = 2
string      = 3
date        = 4  (which is a type of string format)
```

Query A query consists of:

- the name of the dataset of interest
- the column names (variables) in that dataset of interest (at least one is needed)
- the predicate determining wanted dataset rows
- the type of query (wanted type of information)
- the value of any parameters for that query type
- the epsilon quantifying differential privacy risk you allow

The predicates allowed are disjunctions of possibly negated conjunctions. A conjunction is encoded as a list of descriptors, each of which consists of a three-tuple (attribute, operator, value) encoding terms of the kind attribute operator value. For example the term `age == 3` is encoded as `('age', '==', 3)`.

Format specification:

```
 query       = tuple(tuple(dataset_name, predicate, attributes),
               tuple(processor_name, parameters))
 predicate   = list(tuple(negbit, conjunction))
 conjunction = list(descriptor)
 descriptor  = tuple(attribute_name, operator_name, value)
 attributes  = list(attribute_name)
 parameters  = empty_list | list(tuple(parameter_name, value))
```

```
value       = string | number
negbit      = 0 | 1
dataset_name   = name
processor_name = name
attribute_name = name
operator_name  = name
parameter_name = name
name           = string
```

All `name` elements must correspond to the `name` elements in the metdata.

Query Processor – Risk accounting server The risk accounting server accepts two types of queries, one to check if a query for a given user `user` with a certain risk amount (`epsilon`) can be answered, and one to query a users risk information.

The `check` query can be formulated as

status equals 1 if
`epsilon` is not more than the user's per query threshold
and
the user's current total risk expended + epsilon is not more total risk allowed
otherwise
status equals 0

The user information requested by the `info` query is

- the user's current total risk expenditure
- the user's per query risk threshold
- the user's total risk allowed

The specification of the query format is:

```
query   = check | info
check   = tuple(0, user, epsilon)
info    = tuple(1, user)
epsilon = float
user    = string
```

The specification of the response format is:

```
response        = check_response | info_response | error_response
check_response  = tuple(0, status)
info_response   = tuple(0, cumulative_risk, total_threshold, query_threshold)
error_response  = tuple(error_code, error_description)
status          = 0 | 1
cumulative_risk = float
total_threshold = float
query_threshold = float
error_code      = user_not_found | malformed_query | internal_error
user_not_found  = 1
malformed_query = 2
internal_error  = 3
error_description = string
```

Both queries and responses are sent as strings.

Adding new query types

The query processor comes with three query types preinstalled. The first, `simple_count`, produces a noisy count of rows matching the query predicate rows by adding a Laplace($2/\epsilon$, 0) distributed random deviate and rounding the result to the nearest integer (see the section on differential privacy for an explanation of the parameter ϵ). The second, `user_pref_count`, also returns a perturbation of the true number of rows that match the quer predicate. The perturbation and the parameters can be explored here and are described in detail in the following paper:

Protecting count queries in study design.
Vinterbo SA, Sarwate AD, Boxwala AA.
J Am Med Inform Assoc. 2012 Sep 1;19(5):750-7. Epub 2012 Apr 17.

The third, `histogram`, first discretizes the numerical columns, produces a histogram, and adds Laplace($2/\epsilon$, 0) to each count (including the 0-valued entries). Finally entries with a value less than a value $A\log(n)/\epsilon$ are removed. The approach is based on results in the following paper:

Jing Lei.
Differentially private m-estimators.
NIPS 2011, pages 361-369, 2011.

The query processing server can load new metadata from its database, updating its knowledge on available datasets, as well as load additional query types from a python module given by the `--querymodule` option. Both of these can be done at startup, or is done when the server is sent the SIGUSR1 signal. The given python module must contain a python dictionary `processors` with the new query type names as keys, and dict values as follows:

```
{ 'name' : name,
  'f'    : f,
  'query_edit' : edit_f, # optional field
  'meta' : f_meta }
```

where

`name` is the same as the key for this dict in `processors`. **`f`**
is a function `f(eps, parms, meta, result)` where

`eps` is the differential privacy risk ϵ allowed **`parms`**
is a list of `(parameter_name, value)` tuples as given in the query. If the query type has a field `query_edit`, the `parms` list gets a
`('orig_query', {'predicate' : predicate, 'attributes' : attributes})`
tuple appended, where `predicate` and `attributes` are the query parts received (see Query)

`meta` is the metadata for the queried dataset as described in the metadata specification **`result`**
is an iterator for data row tuples extracted from the dataset. The row entries correspond to the `attributes` in the query and all satisfy the query predicate

`edit_f` is a function `edit_f(predicate, attributes)` where

`predicate` is the `predicate` part of the query sent (see Query) **`attributes`**
is the `attribute` part of the query sent (see Query)

the `edit_f` function, if it exists, is called to allow the query type to edit the `predicate` and `attribute` parts before they are used. The return value of `edit_f` is a`(predicate,attribute)` tuple with the (potentially) rewritten `predicate` and `attribute` parts.

`f_meta` is python dictionary containing the metadata for processor as specified in the metadata specification.

Note that dataset metadata should be updated to reflect the addition of a new query type, as it otherwise will not be available for application.

An example of how a loadable query type can be implemented can be found in the section on Logistic Regression.

Risk accounting policies

The default risk accounting policy described aboveis implemented as:

```
def threshold_policy(eps, tt, qt, total_sum, history):
    '''implement the threshold policy

    parameters:
       eps : currently requested risk
       tt  : total allowed risk
       qt  : per query allowed risk
       total_sum : the current total risk expenditure
       history: an iterator for the rows of the history table for this user.

    implements: eps + total_sum <= tt and eps <= qt'''
    return eps + total_sum <= tt and eps <= qt
```

With interface dict:

```
threshold_policy = {
    'name' : 'threshold',
    'implementation' : threshold_policy,
    'description' : threshold_policy.__doc__
    }
```

and dict of available policy interfaces:

```
policies = { 'threshold' : threshold_policy }
```

The package that implements new policies to be loaded must implement the same interface dict and also contain a `policies` dict as shown. If the module is loaded in response to a SIGUSR1 signal, the module must also contain an element `policy` containing the name of the policy to be named . For example, to load a package with the above contents and start using `threshold` in response to a signal, the following line must be added:

```
policy = 'threshold'
```

Miscellaneous

Automated deployment example

A shell script that deploys a small demonstration system can be found in this gzipped tar archive. The archive also contains a directory with CSV files that are used as datasets, as well as a small python program to help with key generation.

The test deployment involves the two servers and a user Alice. The script

1. does the following key management steps
    1. generates separate directories for each of the servers and Alice
    2. generates keys for the servers and Alice
    3. distributes the public keys among the parties and signs the received public keys for each party
2. generates needed databases
    - a data "warehouse" containing datasets to connect the query processing server to
    - a risk accounting database to connect the risk accounting server to
3. adds user Alice to the risk accounting database
4. launches the servers
5. launches the text-based client for the user Alice that connects to the query processing server

Logistic Regression

Here is an implementation of a differentially private logistic regression query type: query_dplr.py. It can be loaded into the query processing server using the `-q` or `--querymodule` option. For example, the automated deployment example attempts doing this by adding `-q aux/query_dplr.py` to the option list. This assumes that the directory in which the server was started contains a sub-directory `aux` which contains the file.

The query type depends on R, rpy2, and the `R` library PrivateLR.

Acknowledgements

This implementation was supported by NIH Roadmap for Medical Research grant U54 HL108460 and NIH grant R01 LM07273.