dbinfer Documentation

Release 0.1.0b

Howard Chivers

Jun 27, 2016

Sensitive data are usually protected by access controls which allow subjects (users, programs) specific types of access (e.g. read, write, modify) to constrain the viewing or extraction of information (confidentiality), or the ways in which it may be changed (Integrity), or both.

Internet Users are familiar with how browsing behaviour is tracked and consolidated to provide a profile of 'interests' which are then used to provide targeted advertising. This 'linkage' of information may be helpful (if you enjoy advertising!) or regarded as a violation of privacy.

The problem of inferring sensitive information from data fragments or statistics which are otherwise not sensitive is a common concern of database designers. For example, queries against public census information or customer databases that provide aggregated statistics may be contrived to intersect in such a way that they reveal information about single identities in the system.

dbinfer is a progam which supports inference experiments by providing access to a small database of 'Students and Exam Grades'. Program features include command-line and API interfaces, statistical queries, and pre and post query inference management policies.

Contents:

CHAPTER

ONE

FUNCTIONS AND EXAMPLES

This program uses a predefined database of student grades on an imaginary cyber security course. There are a total of 100 students in the database; the grades vary from 1-5 for each module and 4 modules are defined: threat, network, crypto, and forensic.

1.1 Command Line Functions

The command line functions provided by *dbinfer* are described in detail in the *Command Line Reference* section; a brief discussion of the algorithms used in this program is given in *Algorithms* together with output examples with verbose printing.

dbinfer can be invoked on the command line directly as in the examples below, or as a module, using:

```
python -m dbinfer
```

In linux you will probably need to specify python3:

```
python3 -m dbinfer
```

The module also provides an API to allow itto be accessed from a Python program, brief examples follow this command line section.

Normally it will be already be configured to access the database (see *Installation*), but if required command line options may be included to override the database connection parameters, for example:

```
dbinfer -h 172.0.0.1
dbinfer>
```

See *Command Line Reference* for other connection parameters.

When the program is running from the command line it provides a *dbinfer*> prompt for user commands.

The query behaviour is determined by the user type, policies which place inference restrictions on the results, and an optional verbose output mode. The default is the *guest* user, no inference control and no verbose output. For example, to change the user to *staff*:

```
dbinfer> config -u staff
dbinfer>
```

The query command allows the user to provide an SQL *WHERE* clause; in other words to set a logical expression which is used to select database rows. No other SQL is available to the user and it is not necessary to know SQL to use this program. For example, to query a row with id (primary key) of 57:

```
dbinfer> query id='57'
Results database for staff user, query:  id='57'
Total results returned = 1

      --------  Student  -------    ------------- Grade ------------
 id   first     last      gender    threat  network crypto  forensic
 57   Lily      Moore          F         5       2       5       3

dbinfer>
```

Working as *staff* user the query prints the whole database row. The field names (given above the table) can be used in an arbitrary logical expression using comparisons = > >= < <= <> logical operators AND OR NOT XOR and parenthesis. For example:

```
dbinfer> query gender='F' AND (forensic>'3' or crypto>'4') and threat='5'
Results database for staff user, query:  gender='F' AND (forensic>'3' OR␣
↪crypto>'4') AND threat='5'
Total results returned = 4

      --------  Student  -------    ------------- Grade ------------
 id   first     last      gender    threat  network crypto  forensic
 38   Millie    Hernandez      F         5       3       4       4
 57   Lily      Moore          F         5       2       5       3
 75   Sienna    Robinson       F         5       3       4       4
 97   Jessica   Wilson         F         5       3       3       4

dbinfer>
```

Note that query values used for comparison must be quoted, e.g. `'57'`, while field names, e.g. `gender`, do not use quotations. This is standard in SQL.

The guest user does not have access to the database, but to only the average marks achieved in each module. For example, using the same query:

```
dbinfer> config -u guest
dbinfer>  query gender='F' AND (forensic>'3' or crypto>'4') and threat='5'
Results database statistics for guest user, query:  gender='F' AND␣
↪(forensic>'3' or crypto>'4') and threat='5'
Total results returned = 4
mean results:
threat     5.00
network    2.75
crypto     4.00
forensic   3.75

dbinfer>
```

Statistical results of this sort are often filtered to prevent inference of individual data items. *dbinfer* demonstates two common types types of filter policy, the first is *n results limited to k%*. This means the no n rows in the result set used to calculate the statistic may account for more than k% of the mean. In this program n=1 and k=30%, meaning that queries in which any row accounts for more that 30% of the mean value are not allowed. For example, using the same query:

```
dbinfer> config -pk
dbinfer> query gender='F' AND (forensic>'3' or crypto>'4') and threat='5'
Error: Query not allowed by result restriction policy
```

```
dbinfer>
```

The sum of the crypto grades in these 4 rows is 16, the policy threshold is therefore 0.3 * 16 = 4.8, so if any row in the result set has a crypto score in excess of 4.8 the query will not be permitted. Here Lily has a crypto grade of 5 so the policy prevents the user from seeing the results. In this result it is only the crypto field that restricts the output, although the calculation is done for every field.

This policy is applied after the results have been calculated and it is vulnerable to attacks which calculate individual rows from the intersections of larger queries. The simplest such attack is to obtain two result sets that are large but differ in only one row; that row can then be calculated by subtraction. To mitigate this it is possible to detect combinations of fields which could be used in this way. This is the *lattice* policy which rejects any query using a set of fields that could be used in some combinatino to isolate a single row. In this case it has the same result:

```
dbinfer> config -pl
dbinfer> query gender='F' AND (forensic>'3' or crypto>'4') and threat='5'
Error: Query not allowed by lattice policy
dbinfer>
```

These policies are discussed in more detail in *Algorithms*.

Finally, there is an exit command:

```
dbinfer> exit
Database Closed
```

Clean exits are a good idea since they release the database connection.

1.2 API Functions

The API provides the same functionality as the command line. A `DbInfer` object is first obtained:

```
>>> from dbinfer import DbInfer
>>> dbapi = DbInfer()
```

This builds a connection to the database; the connection parameters are provided by defaults or *dbinfer.ini*. Connection parameters may be also be overridden when the object is built; for example:

```
>>> dbapi = DbInfer(host='127.0.0.1')
```

See *Module Functions* for other connection parameters and defaults, and for detailed options for the methods described below.

Instead of a *config* command as at the command line there are three separate methods used to set the policies: `setUser(user)`, `setPolicy(policy)` and `setVerbose(bool)`. A user may be *staff* or *guest*, and a policy may be *k*, *l* or *n* with the result described in *Module Functions*. For example, to set the user to *staff*:

```
>>> dbapi.setUser('staff')
```

A query takes a logical WHERE clause, as described for command lines above, and returns a row count and a list of tuples:

```
>>> dbapi.query('id="57"')
(1, [(57, 'Lily', 'Moore', 'F', 5, 2, 5, 3)])

>>> dbapi.query("gender='F' AND (forensic>'3' or crypto>'4') and threat='5'
 ↪")
(4, [(38, 'Millie', 'Hernandez', 'F', 5, 3, 4, 4), (57, 'Lily', 'Moore', 'F
 ↪', 5, 2, 5, 3),
(75, 'Sienna', 'Robinson', 'F', 5, 3, 4,4), (97, 'Jessica', 'Wilson', 'F',␣
 ↪5, 3, 3, 4)])
```

Note that either form of quotation is acceptable but the values must be quoted and the fields unquoted within the query string.

Guest queries return a single row with the mean of each grade for the rows selected by the query.:

```
>>> dbapi.setUser('guest')
>>> dbapi.query("gender='F' AND (forensic>'3' or crypto>'4') and threat='5'
 ↪")
(4, [5.0, 2.75, 4.0, 3.75])
```

and a policy violation raises a ValueError:

```
>>> dbapi.setPolicy('k')
>>> dbapi.query("gender='F' AND (forensic>'3' or crypto>'4') and threat='5'
 ↪")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\eclipse\cp_information\dbinfer\dbinfer.py", line 305, in query
  raise ValueError('Query not allowed by result restriction policy')
ValueError: Query not allowed by result restriction policy
```

Finally, the database connection should be closed:

```
>>> dbapi.close()
```

CHAPTER

TWO

COMMAND LINE REFERENCE

This program requires database connection parameters to connect to a MySql database. The default values are given below; these may be overidden by values in a dbinfer.ini file in the same directory as the executable, or by the following command line arguments.

The command line options are:

-u	–user	string	user name to access database (default 'cyber-practicals')
-p	–password	string	user password (default '1kjg4GIiu5')
-h	–host	string	host IP address , or 'localhost' (default 'localhost')
-d	–database	string	database name (default 'examresults')
? or help			print usage message

If the program is invoked from the command line it provides a command shell which allows a range of queries and optional inference restriction policies over a pre-built test table.

Commands supported by the shell are:

config	Configure how queries are processed, the command takes the following options:			
	-u	–user	string	Specify the user as 'guest' or 'staff' (guests have statistical access, staff have full access).
	-p	–policy	string	A single character which specifies the inference restriction policy. `k`: restrict queries where 1 row is greater than 30% of any field. `l`: lattice restriction. `n`: no policy.
	-v	–verbose	string	To enable or disable verbose printing. 't' or 'true', 'f' or 'false respectively.
exit	Close the program.			
query	The text following the command is a complete WHERE clause, see *Functions and Examples*. In staff mode rows are printed, in guest mode a statistical summary is printed giving the number of rows selected and the mean of the result fields in those rows.			
rebuild	Rebuid the test database.			

The field names in the database table are: `id`: unique row key, `first` and `last` names, `gender`, then exam grades in the range 1-5 in fields corresponding to study modules: `threat`, `network`, `crypto` and `forensic`.

CHAPTER

THREE

PROGRAM API REFERENCE

The API interface provides the same functions as the command line. `rebuild()` is a module function, other actions are provided as methods wihin a `DbInfer` class. Note that there is no *exit* method, instead there is a `close()` method, which closes the dabase connection which is otherwise held open by the `DbInfer` object; configuration is provided by separate methods: `setUser`, `setPolicy` and `setVerbose`.

3.1 Module Functions

`dbinfer.`**`rebuild`**`(...)`

Rebuilds the database table. Arguments to the function are optional database connection specifications, if they are not provided the default values are used unless alternative values have been provided in a `dbinfer.ini` file.

Optional arguments are:

user (default *cyber-practicals*)

password (default *1kjg4GIiu5*)

host (default *localhost*)

database (default *examresults*)

3.2 Module Classes

class `dbinfer.`**`DbInfer`**`(...)`

The *`DbInfer`* class provides query functions on the test database. See *Algorithms* for more detail, including output provided by verbose mode.

The class init takes the same optional attributes as the `rebuild()` function and they are used in the same way. A *`DbInfer`* object opens a connection to the database and this remains open until `close()` is called.

The methods of this class raise a `ValueError` exception if an input parameter is invalid.

The methods of this class are:

`setUser``(`*user*`)`

The user must be `'guest'` or `'staff'`. (Default 'guest') Queries when a *staff* user is set may include user name fields (`first`, or `last`) and will return whole rows.

Queries after the *guest* user is set return a single row with mean values for the exam grades calculated from all rows that match the query.

setPolicy(*policy*)

The policy may be `'k'`:restrict queries where 1 row is greater than 30% of any field, `'l'`: lattice restriction, or `'n'`: no policy. See *Algorithms* for more detail on how these policies are applied.

setVerbose(*bool*)

The parameter must be `True` or `False`, and sets the verbose mode accordingly. Examples of verbose output are give in *Algorithms*, they include how policies are applied to rows and reporting text in a query which is detected as an illegal field or keyword.

query(*where*)

This queries the database; the parameter is the logical condition of an SQL WHERE clause. See *Functions and Examples* for more detail on such clauses.

The field names in the database table are: `id`: unique row identifier, `first` and `last` names, `gender`, then exam grades in the range 1-5 in fields corresponding to study modules: `threat`, `network`, `crypto` and `forensic`.

The query returns (count, list) where the count is the number of rows selected and the list is a list of tuples. If *staff* user is selected each tuple is a complete row with fields in the order given above; *guest* queries a single tuple in the list contains the means of the module grades.

close()

Close the database connection gracefully. After this has been executed no further operations will be possible using this object.

CHAPTER

FOUR

ALGORITHMS

The problem of inference control has been the subject of considerable research, prompted by the difficulty of defining information sensitivity in databases and in providing a balance between confidentialty and functionality. The algorithms implemented here illstrate some of these problems and are documented by Denning in Inference Control in Statistical Databases.

4.1 Bad Design Practice in dbinfer

dbinfer provides some facilities that should **not** be provided to users of operational databases. The query selector is interpreted as a complete WHERE clause; here it allows a wide range of experiments without the need to be fully conversant with SQL, but allowing users to the ability to substitute strings into SQL also facilitates the injection of malicious SQL code. Good practice is to use constrained queries in which only the parameter values are provided by the user - either prepared queries or safe restricted substitutions.

dbinfer does filter the query provided by the user to remove the more obvious injection opportunities:

```
dbinfer> query True; DROP TABLE results
Error: Invalid keyword in where clause
```

Verbose mode will print the first illegal keyword or symbol:

```
dbinfer> config -vt
dbinfer> query True; DROP TABLE results
Invalid keyword in where clause: ;
Error: Invalid keyword in where clause
```

4.2 Sensitive Statistics (n of k%)

The measure that no field in n rows must exceed k% of the total statistic for that field is used both as a definition of sensitivity and in some implementations as the simplest form of inference control. In *dbinfer* the measure implemented is that no field in any 1 row must exceed 30% of the reported mean for that field; this is the *bdinfer* `k` policy.

The verbose option can be used to show how the policy decision is made. For example:

```
dbinfer> config -pk
dbinfer> query gender='F' AND (forensic>'3' or crypto>'4') and threat='5'

Result Row                                          Error Fields
```

```
(38, 'Millie', 'Hernandez', 'F', 5, 3, 4, 4)
(57, 'Lily', 'Moore', 'F', 5, 2, 5, 3)              [6]
(75, 'Sienna', 'Robinson', 'F', 5, 3, 4, 4)
(97, 'Jessica', 'Wilson', 'F', 5, 3, 3, 4)

Error: Query not allowed by result restriction policy
```

In addition to the standard error message, the rows returned from the query are listed and the Error Fields column identifies those fields that violate the policy. In this case field 6 (value of 5 which corresponds to the crypto field) in row id 57 violates the policy: the sum of the crypto grades in these 4 rows is 16, the policy threshold is therefore 0.3 * 16 = 4.8.

4.3 Lattice Constraints

The Denning paper referenced above describes a wide range of inference control options and approaches; the form of lattice constraint imlemented in *dbinfer* is one of the most conservative - ie it provides strong security at the cost of permitting fewer query possibilities.

Since the history of a user's queries may be unknown it may be difficult to tell if a user is making one of a series of queries which, taken together, can be used to identify information about a particular individual. To be safe the only option is to prevent queries where some combination of the fields in the query could possibly select a single row in the database. If such a query was allowed then it would often be possible to contrive a sequence of queries which could be added or subtracted to reveal that row.

The algorithm used by *dbinfer* is fairly simple and allows this process to be visualised (however, note that this policy is not easy to implement on large databases, hence many other ideas on the subject). The sequence is:

1. The query is parsed to extract all field names quoted by the query.
2. The identified fields are read from every row in the table.
3. The values in each field are concatenated into a characteristic string.
4. A count is maintained of how many times each string occurs in the table.
5. If there are any characteristic strings with single occurrences the query is rejected.

In other words, if for a given set of fields there exists a combination of field values that can select a single row, then that set of fields is prohibited in a query.

Using the example above:

```
dbinfer> config -pl
dbinfer> query gender='F' AND (forensic>'3' or crypto>'4') and threat='5'

Attribute Values     Count
             F114        1     (vulnerable)
             F133        2
             F134        1     (vulnerable)
             F143        1     (vulnerable)
             F223        1     (vulnerable)
             F233        1     (vulnerable)
             F234        1     (vulnerable)
             F243        3
             F244        1     (vulnerable)
```

```
            F253          1      (vulnerable)
            F313          1      (vulnerable)
            F314          1      (vulnerable)
            F333          3
            F334          1      (vulnerable)
            F343          2
            F344          4
            F353          3
            ... etc
```

Four fields are quoted in this query: *gender*, *threat*, *crypto* and *forensic* (in field order), so for example the characteristic string F1114 corresponds to the query:

```
dbinfer> query gender='F' AND threat='1'AND crypto='1' AND forensic='4'
Results database for staff user, query:  gender='F' AND threat='1'AND␣
↪crypto='1' AND forensic='4'
Total results returned = 1

      --------  Student  -------    ------------- Grade ------------
 id   first     last      gender    threat  network crypto  forensic
 55   Isabella  Miller         F         1        3       1        4
```

which returns a single row. Note that this policy is not concerned with the actual field values or logic in the query, just what can be achieved from some combination of these fields; it is concerned with the potetial of an ensemble of queries.

Of course, the more fields quoted in a query the more likely they are to be able to index a single row so this policy will tend to allow queries with few fields and reject those with many. A final example to stress the difference between policy evaluation and the query:

```
dbinfer> query gender='F' AND threat='5'

Attribute Values     Count
               F1        5
               F2        8
               F3       16
               F4       14
               F5        7
               M1        8
               M2       15
               M3       13
               M4       11
               M5        3

Results database statistics for guest user, query:  gender='F' AND threat=
↪'5'
Total results returned = 7
mean results:
threat     5.00
network    2.57
crypto     3.57
forensic   3.43
```

Only two fields are present, and no combination of their values is able to select a single row, so the policy allows the query to proceed. The query of `gender='F' AND threat='5'` is equavalent to the policy enumeration of *F5* which, as expected, is the number of rows returned by the query.

CHAPTER

FIVE

INSTALLATION

dbinfer is usually pre-installed as part of an experiment VM, this section should only be of interest to developers or lecturers who wish to configure their own systems.

dbinfer is compatible with Python 3, it is not available for Python 2. Python 3 must installed and *pip* enabled before you begin installation.

dbinfer is distributed as a packaged souce file, it can be installed using PIP as follows:

```
pip install --upgrade --no-index <filename>
```

If you are installing an alpha or beta copy it is also necessary to use the `--pre` option, many Linux systems will require you to expicitly use python 3, since they support both, e.g.:

```
python3 -m pip install --upgrade --no-index --pre <filename>
```

5.1 Configuration

dbinfer will have been installed to the site-packages library for python 3.

If the database connection requirements are not the same as the program defaults (see *Command Line Reference*) place a file named **dbinfer.ini** in the directory with the scripts; it should have a single section (params) and a lines that specify any new connection parameters, for example:

```
[params]
host = 127.0.0.1
```

It is possible to configure new default values for the other connection parameters in the same way.

After installation it is necessary to build the database table to be used for the experiment, open the dbinfer command line and use the *rebuild* command:

```
>dbinfer
dbinfer> rebuild
(1, 'Matilda', 'Adams', 'F', 4, 3, 1, 3)
(2, 'Oliver', 'Alexander', 'M', 4, 2, 4, 3)
(3, 'Eva', 'Allen', 'F', 4, 3, 3, 3)
...
dbinfer> exit
Database Closed
```

The data for this table is in the *results.csv* file in a *data* subdirectory of the installed module; it can be substituted with other data if required but the schema is fixed.

CHAPTER

SIX

LICENCE

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

d