Department of Informatics, University of Zurich

Facharbeit

Multiple linear regression in databases

Markus Neumann

Matrikelnummer: s08-706-442

Email: markus.neumann@math.uzh.ch

July 17, 2014

supervised by Prof. Dr. M. Böhlen and O. Dolmatova

University of Zurich[UZH]

Department of Informatics

Abstract

Matrix operations have many applications in various fields of research and industry, where big datasets and relational database management systems prevail. Performing matrix operations inside a database can be expensive and therefore efficient algorithms are required. The MAD project provides a recent set of such algorithms from which I implemented the 'ordinary least squares' algorithm that approximates the linear regression coefficient, as Facharbeit in the course of my minor at UZH. This work presents an introduction to linear regression, the issues with its application in a database system as well as the implementation details of the algorithm in PostgreSQL.

Zusammenfassung

Matrizen sind weit verbreitet in verschiedenen Gebieten der Forschung und Industrie, wo riesige Datenmengen in relationalen Datenbanksystemen alltäglich sind. Die Berechnung von Matrizenoperationen innerhalb einer Datenbank kann sehr rechenintesiv sein, weshalb effiziente Algorithmen sehr gefragt sind. Das MAD Projekt bietet eine Sammlung solcher Algorithmen, von welchen ich die 'Methode der kleinsten Quadrate', welche den linearen Regressionskoeffizienten approximiert, als Facharbeit im Nebenfach meines Studiums an der UZH implementiert habe. Diese Arbeit gibt eine Einführung in lineare Regression, ihre Einbindung in Datenbanksysteme, sowie die Details der Implementierung des Algorithmus in PostgreSQL.

Contents

List of Figures

List of Listings

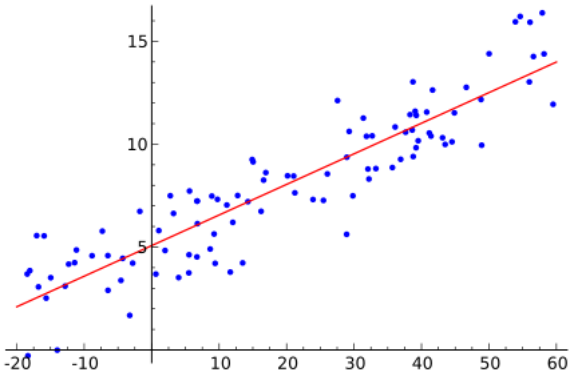

Figure 1: Graphical example of simple linear regression.
Source: Wikipedia [2014c]

1. Introduction

In the past few years the amount of data collected and stored in databases has increased significantly, as well as the available and affordable storage and computational power. At the same time, the requests to Database Management Systems (DBMS) have changed in their nature. Originally, databases were designed to store data and serve them back to the user on demand. The typical queries answer questions like "how many employees earn more than $\$Z$ per month?" or "what items of type X need a packing box of type Y?". Whenever there were questions about mathematical or statistical informations of the data, one exported the data to a file, imported it into an appropriate tool, like Matlab or R, computed the results within and served them back to the database. With increasing size of the datasets, the transport into and out of the database became very time-intensive. Thus, there have been more and more requests of performing statistical and advanced mathematical tasks inside DBMS itself. The MAD project focuses on exactly that. It provides a set of tools to perform these tasks in an efficient way directly inside the DBMS.

I used Hellerstein et al. [2012] as the motivation and basis for this work. The focus is placed on the linear regression method they present in section 4.1.

2. Linear regression

Linear regression is an approach in statistics for modelling a *linear* relation between n pairs of values $(x_i, y_i)_{i=1}^n$ such that $y_i = \beta_0 + \beta_1 x_i$. In of *simple* linear regression, where x_i are scalars, this can interpreted graphically by drawing values (x_i, y_i) as points in the euclidean plain and then fitting a line into that set of points in such a way that the line is as close as possible to all points (see Figure 1).

In case of *multiple* linear regression the values x_i are vectors of dimension d, denoted by $X_i = (x_{ij})_{j=1}^d$. In order to model the linear relation, the equation changes to

$$y_i = \beta_0 + \beta_1 x_{i1} + ... + \beta_d x_{id}$$

After subtracting β_0 from both sides, this can be rewritten as

$$X \cdot \beta = Y, \tag{1}$$

where X denotes the $n \times d$ matrix that has X_i as i-th row, $\beta = (\beta_i)_{i=1}^d$ a vector of length d and $Y = (Y_i)_{i=1}^n = (y_i - \beta_0)_{i=1}^n$ a vector of lenght n.

β can be approximated in a way, called the Ordinary Least Squares (OLS) approximation, that the *sum of squared residuals*

$$S(\beta) := \sum_{i=1}^{n} (Y_i - X_i \cdot \beta)$$

is minimized. This is achieved by multiplying X^T to both sides of Equation 1, resulting in

$$X^T \cdot X \cdot \hat{\beta} = X^T \cdot Y. \tag{2}$$

(Zwillinger [2000], Wikipedia [2014a,c])

3. Matrix operations in databases

If you google on how to do linear algebra in a database, the first answer you usually get is: "don't!" (stackoverflow [2014]).

This is because relational databases are designed to process data on an entry-by-entry basis in a very efficient manner. Even massive datasets can be handled quickly by moving records from disk to memory and back intelligently. Matrix operations pose a problem there: they typically need access to both rows and columns at the same time. E.g. in order to multiply two $n \times n$ matrices $A = (a_{ij})_{i,j=1}^n$ and $B = (b_{ij})_{i,j=1}^n$, one has to calculate n^2 entries. Every entry is computed as $\sum_{k=1}^n a_{ik} b_{kj}$ and hence needs to multiply and sum up $2n$ points. Typically, this is done by storing those $2n$ values (say row vector $A_i = (a_{ij})_{j=1}^n$ and column vector $\bar{B}_j = (b_{ij})_{i=1}^n$) in memory and then computing c_{ij}. After that, A_i is kept in memory and $\bar{B}_j$ is replaced by $\bar{B}_{j+1}$ to compute $c_{i,j+1}$. Now suppose we can only store $n - 1$ values in memory. Thus the DBMS would have to read all entries from disk at every step and hence computation would become very slow. There are ways to work around this by splitting matrices in smaller chunks in such a way that one can perform the arithmetic operations on those chunks and eventually combine the results, but so far none of the popular DBMS provide implementations for this kind of operations. Recently there is a lot research effort going into this topic and the MAD project is one working solution.

4. MADlib 'Ordinary Least Squares' algorithm

The task at hand is to solve Equation 2, resulting in

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T Y \tag{3}$$

Hellerstein et al. [2012] perform the steps as shown in algorithm 1. I will explain the details later, after discussing the following preliminaries:

Algorithm 1 Compute $\hat{\beta}$

1: $M \leftarrow X^T X$
2: $N \leftarrow X^T Y$
3: $M' \leftarrow (M)^{-1}$
4: $\hat{\beta} \leftarrow M'N$

4.1. How to multiply matrices

The ordinary way of multiplying two matrices $M \in \mathbb{K}^{n \times d}$ and $N \in \mathbb{K}^{d \times n}$ is by computing the *inner product*

$$(M \cdot N)_{ij} = \sum_{k=1}^{d} M_{ik} \cdot N_{kj}, \tag{4}$$

i.e. the entry i, j of the resulting matrix is computed as the vector-vector product between the i-th row vector of M and j-th column vector of N (Beezer [2006] and Wikipedia [2014b]). The situation in step 1 of algorithm 1 is a little different. Here, one has to multiply $X \in \mathbb{K}^{n \times d}$ with its transpose, which gives

$$(X^T X)_{ij} = \sum_{k=1}^{n} X_{ki} \cdot X_{kj} \tag{5}$$

So we are actually multiplying d^2 column vectors with each other to get the resulting matrix $(X^T X) \in \mathbb{K}^{d \times d}$. That's perfectly fine if you are multiplying matrices by hand or if you are using a system where you are able to load complete matrices or at least two column vectors at a time into memory.

There is a different approach to multiply matrices, that is easy to implement and efficient to execute in a DBMS.

Instead of multiplying column vectors one can multiply row vectors (*outer product*) and then sum up all the resulting matrices. Let's look at an example to illustrate this:

Example: Matrix multiplication Let $\mathbb{K} = \mathbb{R}$ and $M := \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. So $M^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$. I will call the resulting matrix P.

1. inner product

$$P_{11} = \begin{pmatrix} 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 3 \end{pmatrix} = 1 + 9 = 10$$

$$P_{12} = \begin{pmatrix} 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 4 \end{pmatrix} = 2 + 12 = 14$$

$$P_{21} = \begin{pmatrix} 2 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 3 \end{pmatrix} = 2 + 12 = 14$$

$$P_{22} = \begin{pmatrix} 2 & 4 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 4 \end{pmatrix} = 4 + 16 = 20$$

which gives

$$P = \begin{pmatrix} 10 & 14 \\ 14 & 20 \end{pmatrix}$$

2. outer product

$$P_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 & 1 \cdot 2 \\ 2 \cdot 1 & 2 \cdot 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$
$$P_2 = \begin{pmatrix} 3 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} 3 & 4 \end{pmatrix} = \begin{pmatrix} 3 \cdot 3 & 3 \cdot 4 \\ 4 \cdot 3 & 4 \cdot 4 \end{pmatrix} = \begin{pmatrix} 9 & 12 \\ 12 & 16 \end{pmatrix}$$
$$P = P_1 + P_2 = \begin{pmatrix} 10 & 14 \\ 14 & 20 \end{pmatrix}$$

4.2. Aggregate functions

To take full advantage of the functionality of the DBMS and the above mentioned method of matrix multiplication, it is natural to use an *aggregate function* to compute $X^T X$ and $X^T Y$, as the summands of the outer product can be computed independent of each other and the order of their summation does not matter.

Definition

> "An *aggregate function* computes a single result from multiple input rows." (PostgreSQL [2014a])

Usually a database system operates on tables in a row-by-row or entry-by-entry manner. Since there are situations where one wants to request information about some kind of collection of entries instead of single entries, most DBMS provide some built in aggregate functions such as `AVG`, `SUM` or `COUNT` and even support user defined aggregate functions (PostgreSQL [2014a] and PostgreSQL [2014c]).

The main benefits of aggregate functions are that they

- can (depending on the DBMS) be parallelized and tuned to high performance and
- are memory efficient since they don't need to keep track of all entries but only store the information that needs to be propagated further on.

Hellerstein et al. [2012] provide this definition

> "a user-defined aggregate consists of a well-known pattern of two or three user-defined functions:
>
> 1. A *transition function* that takes the current transition state and a new data point. It combines both into a new transition state.

2. An optional *merge* function that takes two transition states and computes a new combined transition state. This function is only needed for parallel execution.
3. A *final function* that takes a transition state and transforms it into the output value.

"

In order for the aggregate function to be parallelizable, its *transition function* has to be commutative and associative (Hellerstein et al. [2012]). Although matrix multiplication in general is *not* commutative, the process itself clearly is since the order of the summands does not matter for the result. In the case of matrix multiplication as in example 2, the corresponding transition function would look like algorithm 2.

Algorithm 2 "outer product" matrix multiplication

Input: M: current transition state, x: row vector
  $d \leftarrow \dim(x)$
  for $i = 1$ **to** d **do**
    for $j = 1$ **to** d **do**
      $M[i][j] \leftarrow M[i][j] + x[i] \cdot x[j]$
    end for
  end for
Output: M

As the transition function works only on one vector (one record of the database) at a time, it does not matter in which order that we process the rows. Let us look at example 2 again in order to enlighten this:

$$\begin{aligned} P_1 + P_2 &= \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix} + \begin{pmatrix} 9 & 12 \\ 12 & 16 \end{pmatrix} \\ &= \begin{pmatrix} 9 & 12 \\ 12 & 16 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix} \\ &= P_2 + P_1 \end{aligned}$$

Algorithm details We can combine steps 1 and 2 from algorithm 1 into a single transition function (algorithm 4) and do the inversion in the final function (algorithm 5). The aggregate function that wraps those two functions is shown in algorithm 3.

As of now, PostgreSQL does not support parallel execution of aggregates (Berkus [2014]). Hence I did not write a merge function but I want to show the outline of it in algorithm 6 for the sake of completeness.

5. Implementation

Database management system I worked with PostgreSQL as DBMS.

Algorithm 3 Aggregate function "matrix transposed times matrix"

Input: X matrix, recieved row-by-row, Y column vector, received entry-by-entry

initialize transition state $M = \texttt{ARRAY[][]}$
for all row in X **do**
    update M by transition function (algorithm 4)
end for
if parallel environment **then**
    combine all M_i by merge function (algorithm 6)
end if
calculate $\hat{\beta}$ by final function (algorithm 5)
return $\hat{\beta}$

Algorithm 4 Transition function

Input: M transition state, x row vector, y entry

$d \leftarrow \dim(x)$
for $i = 1$ **to** d **do**
    for $j = 1$ **to** d **do**
        $M[i][j] \leftarrow M[i][j] + x[i] \cdot x[j]$
    end for
    $M[i][d+1] \leftarrow M[i][d+1] + x[i] \cdot y$
end for
return M

Algorithm 5 Final function

Input: M transition state

$N \leftarrow M[:][d+1]$
invert $M[:][1:d]$ as M'
$\hat{\beta} \leftarrow M' \cdot N$
return $\hat{\beta}$

Algorithm 6 Merge function

Input: M_1 transition state, M_2 transition state

$M \leftarrow M_1 + M_2$
return M

Procedural language I had to choose one out of the four procedural languages that are supported by PostgreSQL.

- `PL/pgSQL` SQL procedural language
- `PL/Tcl` Tcl procedural language
- `PL/Perl` Perl procedural language
- `PL/Python` Python procedural language

  PostgreSQL [2014b]

I chose `PL/pgSQL` because of the following reasons:

- Personal interest.
  I learned some basics of the language during the database systems course in the spring semester 2014(Böhlen and Gamper [2014]), on which I wanted to build and get some practice in that language.
- Natural choice.
  Since PostgreSQL was fixed as DBMS, it was the natural choice to stick to its own procedural language.
- I know neither C nor Perl and learning one of those for this project would have taken to much time.

Table structure There a several different ways how one can store matrices in a database table, from which the following two are probably the most popular:

1. The values can be stored in *sparse* representation, where every entry consist of a `(column number, row number, value)` triple as shown in listing 1.

Listing 1 Table sparse matrix

```
col | row | value
----+-----+-------
  1 |   1 |     1
  2 |   1 |     2
  1 |   2 |     3
  2 |   2 |     4
```

2. The rows of a matrix can be stored as arrays as shown in listing 2.

Listing 2 Table row-by-row matrix

```
 values | row_id
--------+--------
  [1,2] |      1
  [3,4] |      2
```

The sparse representation is more generic and has a wider field of use cases than the array representation, but for the computation of $X^T \cdot X$ the array representation is very efficient. One can compute every entry of the result by reading at most two entries from the table into memory, as long as two arrays fit into memory and hence the amount of reads performed on the database is smaller than with sparse representation. I chose a variation of the array representation in order to have the same structure as Hellerstein et al. [2012]. The difference to listing 2 is, that the y values get stored in the same table as the X values. Therefore, I don't need to store the row_id because the computation of $\hat{\beta}$ does not depend on the order of X and y, as long as corresponding points are kept together.

5.1. Functions

Since the aggregate described in algorithm 3 needs to keep track of the processed values of X^TX and X^TY and since `PL/pgSQL` supports only a single variable as output of as step function, I had to define a custom datatype in order to store the complete information. It consists of a two dimensional array for X^TX and a one dimensional array for X^TY. I defined another custom data type, `LinRegOut`, used as output of the aggregate function. This comprises the one dimensional array `beta` and the time interval `inversiontime`, that keeps track of the amount of time used to invert X^TX that is stored for performance analysis purposes as is shown in listing 5.

5.1.1. Aggregate function

The implementation of algorithm 3 is straight forward in `PL/pgSQL` as it needs only of the type definition for the transition state `STYPE`, the transition function `SFUNC` and the final function `FINALFUNC` (listing 6).

5.1.2. Transition function

The transition function in listing 7 works basically as described in algorithm 4. The differences are language specific: since `PL/pgSQL` can't access single fields of custom composite data types inside a transition function directly, I had to define temporary variables for X^TX and X^TY respectively. In the first `IF`-block, the values of the transition state are read into those temporary variables, or if the transition state is not yet present (i.e. we are working on the first input values), the temporary variables are initialized to zero. Then the current vector-vector product $X_i^T \cdot X_i$ and the current vector-value product $X_i^T \cdot Y_i$ are added to the temporary variables. In the end those variables are cast back to `MyType` and returned.

5.1.3. Final function

The final function in listing 8 performs all the steps of algorithm 5. The inversion of X^TX is done by a separate function, in order to easily change in case of performance issues (listing 9). I decided to compute the LU-decomposition of X^TX first and invert the resulting lower-left and upper-right triangular matrices separately. In mathematical terms, this means that we can find such matrices L a lower-left matrix and U an upper-right matrix that $X^TX = L \cdot U$. Hence the inverse of X^TX can be computed as $(LU)^{-1} = U^{-1} \cdot L^{-1}$. This is because the algorithm for inverting a triangular matrix with the Gauss algorithm is much easier to implement, than inverting a full matrix. (Sauter [2014])

5.2. Discussion

I think that this algorithm is a really good way to solve the given task. Since it takes full advantage of the aggregate feature, it is only usable in DBMS that support aggregates. The best place to use it hence would be those DBMS that support parallelisation and aggregate at the same time. In the case of PostgreSQL it is still a very efficient way to compute linear regression. The downside is, that the algorithm is not applicable to other data structures or matrix multiplications in general as it depends on the array representation of the data.

6. Tests and results

6.1. Prerequisites

All tests were performed in PostgreSQL 9.3.4 running on a virtual Ubuntu 14.04 server. The hardware specifications are shown in listing 3.

Listing 3 Hardware specifications

CPU	2 Intel QEMU Virtual CPUs @ 2.4GHz
CPU cache	4MB
RAM	8GB

6.2. Data creation

I had no real-life data at hand to test the `LinReg` function, so I created random matrices of different sizes (listing 10).

I took $n \in \{10^i | 2 \leq i \leq 6\}$ and $d \in \{2^k | 3 \leq k \leq 6\}$. The entries were generated using `PL/pgSQL`'s built in `random()` function, that creates uniformly distributed values in the range $[0, 1.0)$. Since numbers are rarely that small in reality, I multiplied each random value by 1000. To improve reliability of the results and identify server lags, I created 5 different random tables for every n, d pair. This makes a total of 100 tables that can be used for tests.

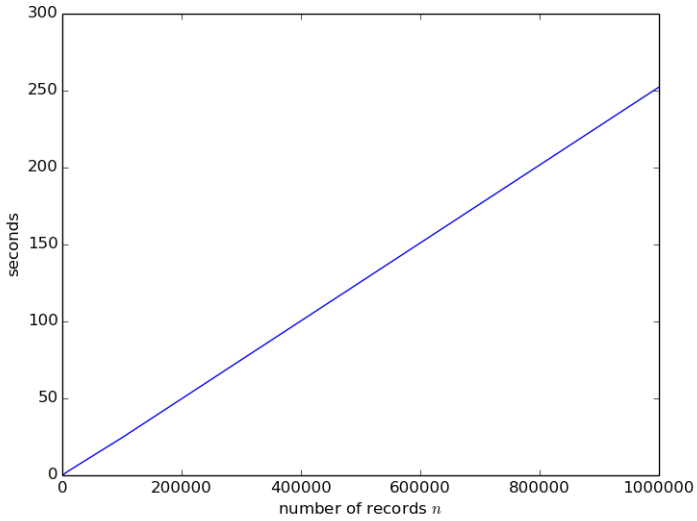

Figure 2: Total time for $d = 8$

6.3. Complexity

Before running the tests, let us look at the complexity of the code to get an idea of what to expect from the tests. Suppose we have a table with n records, where each x entry has length d. The transition function (subsubsection 5.1.2) gets executed n times. Inside the transition function we have two nested loops that iterate both from 1 to d. The final function (subsubsection 5.1.3) then performs the LU-decomposition that needs $\frac{d(2d-1)(d-1)}{6} \approx d^3$ iterations to compute the decomposition and additional d^3 operations to invert the two matrices. After that, the final function needs d^2 iterations to compute $\hat{\beta}$.

This makes a total of approximately $n \cdot d^2 + d^3 + d^3 + d^2$ iterations on the data. Now to classify this in $\mathcal{O}$ notation, we have to determine whether $n \cdot d^2$ or d^3 is the term of higher computational significance (Hromkovic [2008]). As soon as $n \geq d$, the first term is bigger, which is given for linear regression. Hence I have a code complexity of $\mathcal{O}(n \cdot d^2)$.

6.4. Tests

For every table that I created with the `buildTests()` function, I started a timer, called the `LinReg()` function on that table and stored the runtime and the inversion time in the results table. (See listing 11)

6.5. Results

A complete listing of the test times is shown in listing 14. To compare the results against the theory I built up in 6.3, I looked at the results of all tests first where $d = 8$ and $d = 32$. As you can see in both Figure 2 and Figure 3, the time increase is linear to the increase of n, which fits

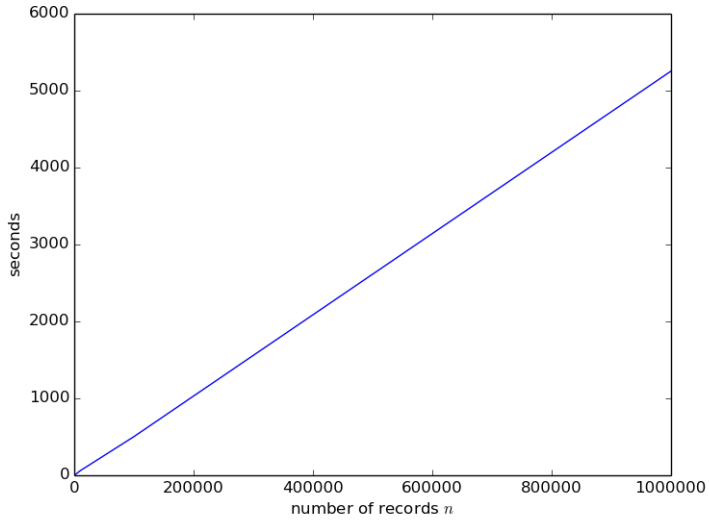

Figure 3: Total time for $d = 32$

exactly the expectations from subsection 6.3.[1] Now I fix n at 10000, 100000 and 1000000 and look at the change in time for the different values of d. The blue lines in Figure 4,Figure 5 and Figure 6 show the runtime, whereas the green lines show the slope of $y = \left(\frac{x}{2}\right)^2$ in Figure 4, $y = x^2$ in Figure 5 and $y = (3x)^2$ in Figure 6. The time increase for constant n is always in $\mathcal{O}(d^2)$, only the factor changes with the size of n, meaning my results are consistent.

6.6. Correctness

In order to check if my computation of $\hat{\beta}$ is correct, I performed another test. I created 20 tables with $n = 1000$ and $d = 16$ where the Equation 1 has $\beta = \mathbf{1}$ as exact solution. This is achieved by setting the entries of X again as random values in the interval $[0, 1000)$ and then computing Y as $Y[i] = \sum_{k=1}^{n} X_i[k]$ (listing 12). I computed $\hat{\beta}$ with the `LinReg` function and stored $\max(|\hat{\beta} - \beta|)$ in a table (listing 13). The results are shown in listing 4. All differences are smaller than 10^{-12}, which means that the error of the computation of $\hat{\beta}$ is in a range that can be ignored in most applications and is probably due to the arithmetic precision of the computer.

[1] All plots were produced by the python script listed in listing 15.

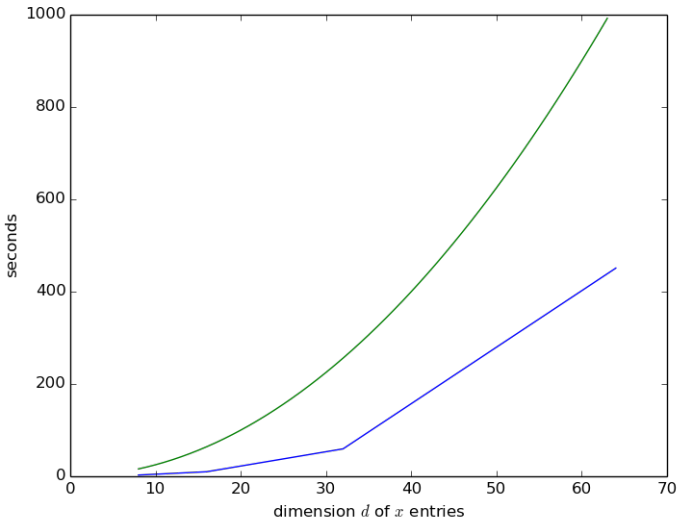

Figure 4: Total time for $n = 10000$

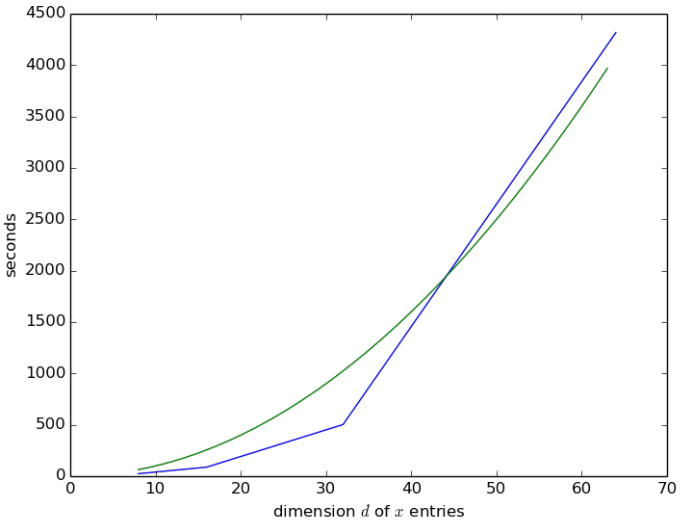

Figure 5: Total time for $n = 100000$

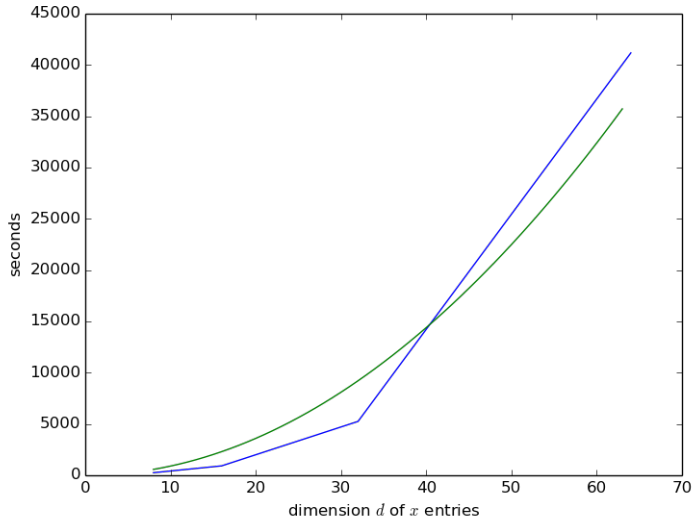

Figure 6: Total time for $n = 1000000$

References

R. A. Beezer. A First Course in Linear Algebra. 2006. URL http://linear.ups.edu.

J. Berkus. Postgresql new development priorities 4: Parallel query, 2014. URL http://www.databasesoup.com/2013/05/postgresql-new-development-priorities-4_20.html. [Online; accessed 11-July-2014].

M. Böhlen and J. Gamper. Slides for the "database systems" course at ifi@uzh, spring 2014, 2014. URL http://www.ifi.uzh.ch/dbtg/teaching/courses/DBS.html.

J. M. Hellerstein, F. Schoppmann, D. Z. Wang, E. Fratkin, and C. Welton. The MADlib Analytics Library or MAD Skills , the SQL. *Proceedings of the VLDB Endowment*, pages 1700–1711, 2012.

J. Hromkovic. *Hromkovic, Informatik:*. Vieweg+Teubner Verlag, 2008. ISBN 9783834806208. URL http://books.google.ch/books?id=AuuM5NC-I5MC.

PostgreSQL. Postgresql manual 9.3 aggregate functions, 2014a. URL http://www.postgresql.org/docs/9.3/static/tutorial-agg.html. [Online; accessed 9-July-2014].

PostgreSQL. Postgresql manual 9.3 server programming, 2014b. URL http://www.postgresql.org/docs/9.3/static/server-programming.html. [Online; accessed 11-July-2014].

Listing 4 Results of correctness test

```
 run_id |        error
--------+---------------------
      1 |  1.3988810110277e-13
      2 | 1.11910480882216e-13
      3 | 1.27897692436818e-13
      4 | 9.05941988094128e-14
      5 | 1.18349774425042e-13
      6 | 8.34887714518118e-14
      7 | 1.20792265079217e-13
      8 | 9.72555369571637e-14
      9 | 7.74935671188359e-14
     10 | 7.90478793533111e-14
     11 | 6.52811138479592e-14
     12 | 1.17239551400417e-13
     13 | 7.19424519957101e-14
     14 | 1.09690034832965e-13
     15 |  1.7608137170555e-13
     16 | 1.32338584535319e-13
     17 | 9.45910016980633e-14
     18 | 1.04805053524615e-13
     19 |  1.4122036873232e-13
     20 | 1.13686837721616e-13
(20 rows)
```


PostgreSQL. Postgresql manual 9.3 user-defined aggregates, 2014c. URL http://www.postgresql.org/docs/9.3/static/xaggr.html. [Online; accessed 9-July-2014].

S. Sauter. Numerische lineare algebra. Technical report, Universität Zürich, Institut für Mathematik, 2014. URL http://www.math.uzh.ch/index.php?ve_vo_det&key2=2158. [Online; accessed 11-July-2014].

stackoverflow. Mysql matrix multiplication, 2014. URL http://stackoverflow.com/questions/15502622/mysql-matrix-multiplication. [Online; accessed 10-July-2014].

Wikipedia. Linear regression — wikipedia, the free encyclopedia, 2014a. URL http://en.wikipedia.org/w/index.php?title=Linear_regression&oldid=615012889. [Online; accessed 8-July-2014].

Wikipedia. Matrix multiplication — wikipedia, the free encyclopedia, 2014b. URL http://en.wikipedia.org/w/index.php?title=Matrix_multiplication&oldid=616366532. [Online; accessed 11-July-2014].

Wikipedia. Ordinary least squares — wikipedia, the free encyclopedia, 2014c. URL http://en.wikipedia.org/w/index.php?title=Ordinary_least_squares&oldid=615091560. [Online; accessed 11-July-2014].

D. Zwillinger. *standard probability Statistics tables and formulae standard probability Statistics tables and formulae*. 2000. ISBN 1584880597.

A. SQL code

```
CREATE TYPE MyType AS (
  XtX DOUBLE PRECISION[][],
  Xty DOUBLE PRECISION[]
);

CREATE TYPE LinRegOut AS (
    beta DOUBLE PRECISION [],
    inversiontime INTERVAL
);
```

Listing 5: File MyType.sql

```
CREATE AGGREGATE LinReg(DOUBLE PRECISION[], DOUBLE PRECISION)(
  /*--------
   * Computes the beta-value of multi-linear regression.
   * (i.e. solves X*beta=y)
   * Usage: LinReg(X,y), where
   * - X DOUBLE PRECISION[]
   * - y DOUBLE PRECISION
   * Returns:
   * - beta DOUBLE PRECISION[]
   * - invtime DOUBLE PRECISION
   *--------
   */
  -- stores intermediate values of Xt*X and Xt*y
  STYPE = MyType,
  -- computes Xt*X and Xt*y
  SFUNC = LinRegStep,
  -- computes beta as (Xt*X)-1*Xt*y
  FINALFUNC = LinRegFinal
);
```

Listing 6: File LinReg.sql

```
CREATE OR REPLACE FUNCTION LinRegStep
       (aggr_state MyType, x DOUBLE PRECISION[], y DOUBLE PRECISION)
RETURNS MyType AS
$$
  /*--------
   * stepfunction of aggregate "LinReg"
   *
   * computes xt*x and xt*y from the
   * current row of matrix X and current
   * value of vector y.
   *--------
   */
  DECLARE
    -- length of vector x = dimension of Xt*X
    dim INTEGER;
    -- temporary arrays to compute current values
    XtX_tmp DOUBLE PRECISION[][];
    Xty_tmp DOUBLE PRECISION[];
  BEGIN
    -- read 'dim'
    dim := array_length(x,1);
    -- initialize output according to current state
    IF aggr_state IS NULL THEN
      -- on empty state, intialize output with 0-matrix and 0-vector
      XtX_tmp := array_fill(0,ARRAY[dim,dim]);
      Xty_tmp := array_fill(0,ARRAY[dim]);
    ELSE
      -- initialize output with current state
      XtX_tmp := (aggr_state).XtX;
      Xty_tmp := (aggr_state).Xty;
    END IF;
    -- compute the current XtX addition
    FOR i IN 1..dim LOOP
      FOR j in 1..dim LOOP
        XtX_tmp[i][j] := XtX_tmp[i][j]+x[i]*x[j];
      END LOOP;
      -- compute the current Xty addition
      Xty_tmp[i] := Xty_tmp[i]+x[i]*y;
    END LOOP;
    -- cast result to MyType, so aggregate can handle it
    RETURN (XtX_tmp,Xty_tmp)::MyType;
  END;
$$ LANGUAGE plpgsql;
```

Listing 7: File LinRegStep.sql

```
--DROP FUNCTION LinRegFinal(MyType) CASCADE;
CREATE OR REPLACE FUNCTION LinRegFinal(final_state MyType)
RETURNS LinRegOut AS
$$
  /*--------
   * final function of aggregate "LinReg"
   *
   * inverts resulting matrix Xt*X and
   * computes beta = (Xt*X)-1*Xt*y
   *
   * returns beta and timestamp of LUinversion
   *--------
   */
  DECLARE
    XtX_fin DOUBLE PRECISION[][];
    Xty_fin DOUBLE PRECISION[];

    time1 TIMESTAMP;
    time2 TIMESTAMP;

    dim INTEGER;

    beta DOUBLE PRECISION[];
    inversiontime INTERVAL;
  BEGIN
    -- fetch the values from final state
    XtX_fin := (final_state).XtX;
    Xty_fin := (final_state).Xty;
    -- read the dimension
    dim := array_length(Xty_fin,1);
    -- initialize resulting vector
    beta := array_fill(0,ARRAY[dim]);
    -- invert XtX (so far with LU and piecewise Gauss)
    /* we want to know the effect on total runtime,
     * so we take timestamps before and after.
     */
    time1 := clock_timestamp();
    XtX_fin := LUinvert(XtX_fin);
    time2 := clock_timestamp();
    -- store time difference
    inversiontime := time2-time1;
    -- compute (XtX)-1*Xty
```

```
    FOR i IN 1..dim LOOP
      FOR j IN 1..dim LOOP
        beta[i] := beta[i]+XtX_fin[i][j]*Xty_fin[j];
      END LOOP;
    END LOOP;
    -- all done
    RETURN (beta, inversiontime)::LinRegOut;
  END;
$$ LANGUAGE plpgsql;
```

Listing 8: File LinRegFinal.sql

```
CREATE OR REPLACE FUNCTION LUinvert(M DOUBLE PRECISION[][])
RETURNS DOUBLE PRECISION[][] AS
$$
  /*--------
   * computes the inverse of matrix M
   * by decomposing M into lowerleft and upperright
   * triangular matrices L and U, s.t. M=L*U,
   * inverting both L and U with Gauss algorithm
   * and computing Minv as Uinv * Linv.
   *
   * INPUT: M DOUBLE PRECISION[][]
   *
   * OUTPUT: Minv DOUBLE PRECISION[][]
   *--------
   */
  DECLARE
    -- matrix M has size 'dim x dim'
    dim INTEGER;
    -- inverse of M
    Minv DOUBLE PRECISION[][];
    -- lower-left matrix and inverse
    L DOUBLE PRECISION[][];
    Linv DOUBLE PRECISION[][];
    -- upper-right matrix and inverse
    U DOUBLE PRECISION[][];
    Uinv DOUBLE PRECISION[][];

    tmp_val DOUBLE PRECISION;
  BEGIN
    -- read the dimension
    -- /!\ M is supposed to be square, but not checked /!\
    dim := array_length(M,1);
```

```
--initialize all matrices
Minv := array_fill(0,ARRAY[dim,dim]);
L := array_fill(0,ARRAY[dim,dim]);
Linv := array_fill(0,ARRAY[dim,dim]);
U := array_fill(0,ARRAY[dim,dim]);
Uinv := array_fill(0,ARRAY[dim,dim]);
--compute the LU factorization of M
FOR k IN 1..dim LOOP
  L[k][k] := 1;
  Linv[k][k] := 1;
  U[k][k] := M[k][k];
  Uinv[k][k] := 1;
  FOR i IN k+1..dim LOOP
    L[i][k] := M[i][k]/U[k][k];
    U[k][i] := M[k][i];
  END LOOP;
  FOR i IN k+1..dim LOOP
    FOR j IN k+1..dim LOOP
      M[i][j] := M[i][j] - L[i][k]*U[k][j];
    END LOOP;
  END LOOP;
END LOOP;
--compute Linv and Uinv
FOR k in 1..dim LOOP
  FOR i in 1..k LOOP
    Linv[k][i] := Linv[k][i]/L[k][k];
    Uinv[i][k] := Uinv[i][k]/U[k][k];
  END LOOP;
  FOR i in k+1..dim LOOP
    FOR j in 1..k LOOP
      Linv[i][j] := Linv[i][j]-L[i][k]*Linv[k][j];
      Uinv[j][i] := Uinv[j][i]-U[k][i]*Uinv[j][k];
    END LOOP;
  END LOOP;
END LOOP;
--compute Minv as Uinv*Linv
FOR i IN 1..dim LOOP
  FOR j IN 1..dim LOOP
    tmp_val := 0;
    FOR k IN 1..dim LOOP
      tmp_val := tmp_val + Uinv[i][k]*Linv[k][j];
    END LOOP;
    Minv[i][j] := tmp_val;
  END LOOP;
```

```
    END LOOP;
    -- all done
    RETURN Minv;
  END;
$$ LANGUAGE plpgsql;
```

Listing 9: File LUinvert.sql

B. Tests

```
CREATE OR REPLACE FUNCTION buildTests()
RETURNS VOID AS
$$
  DECLARE
    ndim INTEGER;
    ddim INTEGER;
    x DOUBLE PRECISION[];
    y DOUBLE PRECISION;
  BEGIN
    FOR n in 2..6 LOOP
      FOR d in 3..6 LOOP
        ndim := 10^n;
        ddim := 2^d;
        FOR id in 1..5 LOOP
          EXECUTE 'CREATE TABLE test'
            || format('%s%s%s',ddim,ndim,id)
            || '(x DOUBLE PRECISION[],y DOUBLE PRECISION)';
          FOR r IN 1..ndim LOOP
            y := random()*1000;
            x := array_fill(1,ARRAY[ddim]);
            FOR i IN 1..ddim LOOP
              x[i] := random()*1000;
            END LOOP;
            EXECUTE 'INSERT INTO test'
              || format('%s%s%s',ddim,ndim,id)
              || ' VALUES ('
              || format('%L,%L',x,y)
              || ')';
          END LOOP;
        END LOOP;
      END LOOP;
    END LOOP;
  END;
$$ LANGUAGE plpgsql;
```

Listing 10: File buildTests.sql

```
CREATE OR REPLACE FUNCTION runTests()
RETURNS VOID AS
$$
  DECLARE
    ndim INTEGER;
```

```
    ddim INTEGER;

    starttime TIMESTAMP;
    endtime TIMESTAMP;
    totaltime INTERVAL;

    currentout INTERVAL;
  BEGIN
    CREATE TABLE IF NOT EXISTS
      results(ddim INTEGER, ndim INTEGER, inversiontime INTERVAL,
        totaltime INTERVAL);
    FOR n in 1..6 LOOP
      FOR d in 3..6 LOOP
        ndim := 10^n;
        ddim := 2^d;
        FOR id in 1..5 LOOP
          starttime := clock_timestamp();
          EXECUTE 'SELECT (LinReg(x,y)).inversiontime FROM test'
            || format('%sd%sn%si',ddim,ndim,id) INTO currentout;
          endtime := clock_timestamp();
          totaltime := endtime -starttime;
          INSERT INTO results
            VALUES (ddim,ndim,currentout,totaltime);
        END LOOP;
      END LOOP;
    END LOOP;
  END;
$$ LANGUAGE plpgsql;
```

Listing 11: File runTests.sql

```
CREATE OR REPLACE FUNCTION buildBetaTests()
RETURNS VOID AS
$$
  DECLARE
    ndim INTEGER;
    ddim INTEGER;
    x DOUBLE PRECISION[];
    y DOUBLE PRECISION;
    ytmp DOUBLE PRECISION;
  BEGIN
    FOR i in 1..20 LOOP
      ndim := 10^3;
      ddim := 16;
```

```
      EXECUTE 'CREATE TABLE IF NOT EXISTS beta_test'
        || format('%sd%sn%sid',ddim,ndim,i)
        || '(x DOUBLE PRECISION[],y DOUBLE PRECISION)';
      FOR r IN 1..ndim LOOP
        ytmp := 0;
        x := array_fill(1,ARRAY[ddim]);
        FOR j IN 1..ddim LOOP
          x[j] := random()*1000;
          ytmp := ytmp + x[j];
        END LOOP;
        y := ytmp;
        EXECUTE 'INSERT INTO beta_test'
          || format('%sd%sn%sid',ddim,ndim,i)
          || ' VALUES ('
          || format('%L,%L',x,y)
          || ')';
      END LOOP;
    END LOOP;
  END;
$$ LANGUAGE plpgsql;
```

Listing 12: File buildTests.sql

```
CREATE OR REPLACE FUNCTION runBetaTests()
RETURNS VOID AS
$$
  DECLARE
    ndim INTEGER;
    ddim INTEGER;

    starttime TIMESTAMP;
    endtime TIMESTAMP;
    totaltime INTERVAL;

    currentout INTERVAL;
    currenterror DOUBLE PRECISION[];
    error DOUBLE PRECISION;
    maxerror DOUBLE PRECISION;
  BEGIN
    CREATE TABLE IF NOT EXISTS
      beta_results(run_id INT, error DOUBLE PRECISION);
    FOR i in 1..20 LOOP
      ndim := 10^3;
      ddim := 16;
```

```
      EXECUTE 'SELECT (LinReg(x,y)).beta FROM beta_test'
        || format('%sd%sn%sid',ddim,ndim,i) INTO currenterror;
      maxerror := 0;
      FOR i in 1..ddim LOOP
        error := ABS(1-currenterror[i]);
        IF error > maxerror THEN
          maxerror := error;
        END IF;
      END LOOP;
      INSERT INTO beta_results VALUES (i,maxerror);
    END LOOP;
  END;
$$ LANGUAGE plpgsql;
```

Listing 13: File runBetaTests.sql

C. Results

ndim	ddim	totaltime	inversiontime
100	8	00:00:00.023746	00:00:00.002564
100	8	00:00:00.024939	00:00:00.003362
100	8	00:00:00.023171	00:00:00.002412
100	8	00:00:00.0254	00:00:00.002379
100	8	00:00:00.0232	00:00:00.002436
1000	8	00:00:00.26267	00:00:00.002564
1000	8	00:00:00.285489	00:00:00.0029
1000	8	00:00:00.280187	00:00:00.003273
1000	8	00:00:00.279535	00:00:00.004159
1000	8	00:00:00.269056	00:00:00.003036
10000	8	00:00:02.694995	00:00:00.002369
10000	8	00:00:02.543247	00:00:00.002332
10000	8	00:00:02.564963	00:00:00.004127
10000	8	00:00:02.693838	00:00:00.002653
10000	8	00:00:03.082352	00:00:00.004899
100000	8	00:00:23.726925	00:00:00.002501
100000	8	00:00:25.201353	00:00:00.002994
100000	8	00:00:24.343541	00:00:00.002566
100000	8	00:00:24.142769	00:00:00.002506
100000	8	00:00:24.603467	00:00:00.002504
1000000	8	00:04:05.098265	00:00:00.004447
1000000	8	00:04:06.844248	00:00:00.005147
1000000	8	00:04:20.993924	00:00:00.002401

1000000	8	00:04:21.864473	00:00:00.002414
1000000	8	00:04:06.783134	00:00:00.002382
100	16	00:00:00.112679	00:00:00.017259
100	16	00:00:00.091801	00:00:00.015146
100	16	00:00:00.112025	00:00:00.018014
100	16	00:00:00.112796	00:00:00.022704
100	16	00:00:00.135746	00:00:00.025885
1000	16	00:00:00.815234	00:00:00.015561
1000	16	00:00:00.901771	00:00:00.015476
1000	16	00:00:00.89798	00:00:00.024165
1000	16	00:00:00.933688	00:00:00.015925
1000	16	00:00:00.865702	00:00:00.016614
10000	16	00:00:09.162817	00:00:00.017146
10000	16	00:00:10.426016	00:00:00.015481
10000	16	00:00:10.376657	00:00:00.016865
10000	16	00:00:09.892446	00:00:00.028012
10000	16	00:00:09.445893	00:00:00.016338
100000	16	00:01:32.174325	00:00:00.015269
100000	16	00:01:23.07713	00:00:00.015008
100000	16	00:01:33.950572	00:00:00.014945
100000	16	00:01:27.25961	00:00:00.016372
100000	16	00:01:23.720913	00:00:00.014934
1000000	16	00:15:03.185172	00:00:00.015556
1000000	16	00:14:50.271601	00:00:00.01532
1000000	16	00:15:15.317476	00:00:00.020588
1000000	16	00:15:29.345151	00:00:00.020867
1000000	16	00:15:59.534883	00:00:00.015298
100	32	00:00:00.706183	00:00:00.138679
100	32	00:00:00.7699	00:00:00.17718
100	32	00:00:00.65598	00:00:00.141333
100	32	00:00:00.679164	00:00:00.158792
100	32	00:00:00.764569	00:00:00.212274
1000	32	00:00:04.89616	00:00:00.164341
1000	32	00:00:05.607052	00:00:00.188645
1000	32	00:00:05.15957	00:00:00.150948
1000	32	00:00:05.119471	00:00:00.146039
1000	32	00:00:05.313862	00:00:00.207736
10000	32	00:00:59.118931	00:00:00.206354
10000	32	00:01:02.757765	00:00:00.1889
10000	32	00:00:59.517943	00:00:00.16702
10000	32	00:00:56.774501	00:00:00.146103
10000	32	00:00:58.940245	00:00:00.161413
100000	32	00:08:06.410799	00:00:00.15987
100000	32	00:08:07.161503	00:00:00.164449

```
  100000 |    32 | 00:08:43.129977 | 00:00:00.174698
  100000 |    32 | 00:08:36.184827 | 00:00:00.164435
  100000 |    32 | 00:08:20.626028 | 00:00:00.14783
 1000000 |    32 | 01:25:43.143062 | 00:00:00.151011
 1000000 |    32 | 01:26:27.980122 | 00:00:00.166694
 1000000 |    32 | 01:26:45.59492  | 00:00:00.169309
 1000000 |    32 | 01:28:51.414537 | 00:00:00.147906
 1000000 |    32 | 01:29:59.636742 | 00:00:00.158089
     100 |    64 | 00:00:06.976099 | 00:00:02.632695
     100 |    64 | 00:00:06.354614 | 00:00:02.288769
     100 |    64 | 00:00:06.496339 | 00:00:02.232434
     100 |    64 | 00:00:06.61293  | 00:00:02.376079
     100 |    64 | 00:00:06.342603 | 00:00:02.25155
    1000 |    64 | 00:00:49.607221 | 00:00:02.533729
    1000 |    64 | 00:00:51.139592 | 00:00:02.867665
    1000 |    64 | 00:00:50.99478  | 00:00:02.492939
    1000 |    64 | 00:00:47.315666 | 00:00:02.448716
    1000 |    64 | 00:00:45.623201 | 00:00:02.121799
   10000 |    64 | 00:07:07.092826 | 00:00:02.316163
   10000 |    64 | 00:07:22.747058 | 00:00:02.263725
   10000 |    64 | 00:07:14.827523 | 00:00:02.428302
   10000 |    64 | 00:07:51.585268 | 00:00:02.244976
   10000 |    64 | 00:07:58.985783 | 00:00:03.273587
  100000 |    64 | 01:13:54.132076 | 00:00:02.44789
  100000 |    64 | 01:11:44.955574 | 00:00:02.544321
  100000 |    64 | 01:14:59.77437  | 00:00:02.500936
  100000 |    64 | 01:10:08.247361 | 00:00:02.33984
  100000 |    64 | 01:08:59.886234 | 00:00:02.207661
 1000000 |    64 | 11:50:16.324562 | 00:00:02.342342
 1000000 |    64 | 11:23:20.379733 | 00:00:02.241259
 1000000 |    64 | 11:01:12.279202 | 00:00:02.06376
 1000000 |    64 | 10:54:03.991939 | 00:00:01.983473
 1000000 |    64 | 12:02:32.056664 | 00:00:02.30654
(100 rows)
```

Listing 14: File allresults.txt

D. Appendix: other code

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D as ax3d
import numpy as np

def main():
    #read file
    with open('results.txt','r') as infile:
        rawdata = infile.readlines()
    #read data
    ndim, ddim, invtime, tottime = zip(
        *[read_data_line(line.split('|'))
          for line in rawdata[2:-2]])

    fig = plt.figure()

    dplt8 = fig.add_subplot(111)
    xs = [ndim[i] for i in range(len(ndim)) if ddim[i]==8]
    ys = [tottime[i] for i in range(len(ndim)) if ddim[i]==8]
    dplt8.plot(xs,ys)
    dplt8.set_xlabel(r'number of records $n$')
    dplt8.set_ylabel('seconds')
    plt.savefig('d=8_plot.png',transparent=True)

    fig = plt.figure()

    dplt32 = fig.add_subplot(111)
    xs = [ndim[i] for i in range(len(ndim)) if ddim[i]==32]
    ys = [tottime[i] for i in range(len(ndim)) if ddim[i]==32]
    dplt32.plot(xs,ys)
    dplt32.set_xlabel(r'number of records $n$')
    dplt32.set_ylabel('seconds')
    plt.savefig('d=32_plot.png',transparent=True)

    fig = plt.figure()

    nplt4 = fig.add_subplot(111)
    xs = [ddim[i] for i in range(len(ndim)) if ndim[i]==10000]
    ys = [tottime[i] for i in range(len(ndim)) if ndim[i]==10000]
    nplt4.plot(xs,ys)
    nplt4.plot([x for x in range(8,64)],
               [(0.5*y)**2 for y in range(8,64)])
    nplt4.set_xlabel(r'dimension $d$ of $x$ entries')
```

```
    nplt4.set_ylabel('seconds')
    plt.savefig('n=10000_plot.png',transparent=True)

    fig = plt.figure()

    nplt5 = fig.add_subplot(111)
    xs = [ddim[i] for i in range(len(ndim)) if ndim[i]==100000]
    ys = [tottime[i] for i in range(len(ndim)) if ndim[i]==100000]
    nplt5.plot(xs,ys)
    nplt5.plot([x for x in range(8,64)],
               [y**2 for y in range(8,64)])
    nplt5.set_xlabel(r'dimension $d$ of $x$ entries')
    nplt5.set_ylabel('seconds')
    plt.savefig('n=100000_plot.png',transparent=True)

    fig = plt.figure()

    nplt6 = fig.add_subplot(111)
    xs = [ddim[i] for i in range(len(ndim)) if ndim[i]==1000000]
    ys = [tottime[i] for i in range(len(ndim)) if ndim[i]==1000000]
    nplt6.plot(xs,ys)
    nplt6.plot([x for x in range(8,64)],
               [(3*y)**2 for y in range(8,64)])
    nplt6.set_xlabel(r'dimension $d$ of $x$ entries')
    nplt6.set_ylabel('seconds')
    plt.savefig('n=1000000_plot.png',transparent=True)

def read_data_line(line):
    return float(line[0]),float(line[1]), \
        translate_time(line[2]),translate_time(line[3])

def translate_time(string):
    '''takes a timestring of the form
    'hh:mm:ss.msmsmsms' and returns seconds as float'''
    #trim to remove additional whitespaces
    string = string.strip()
    doubles = [float(field) for field in string.split(':')]
    factors = [3600,60,1]
    return sum([t*f for t,f in zip(doubles,factors)])

if __name__=='__main__':
    main()
```

Listing 15: File producePlots.py