Dutch Cadastral Dataset on PostgreSQL and the PostGIS spatial extension

a performance test

M.J.Vermeij
Faculty of Civil Engineering and Geosciences
Department of Geodesy
Delft University of Technology

Preface

This is a report describing a research into the performance of the PostgreSQL open source database management system when handling spatial data. The data used, is data from the spatial part of the dataset of the Dutch cadastre, LKI. Support for spatial data types is still not fully evolved in many DBMS's and this is therefore an interesting field for research.

The research was performed during the course Geo-DBMS casestudy (ge4631), as part of my study in Geodesy at the department of Geodesy of the Delft University of Technology. Besides the results of the research, it was also important to gain some hands on experience on the field of spatial databases.

During the research I was helped by a number of people and I would hereby like to thank: drs C.W. Quak for his help with Unix, Perl and all kinds of practical problems I encountered during the research, drs T.P.M.Tijssen for his actions as system administer (like moving data, to prevent the loading of the database to fail, on a Sunday) and prof.dr.ir. P.J.M. van Oosterom for the overall guidance of the research. I also would like to thank the developers of PostGIS for quickly and usefully answering the questions I posted on the PostGIS mailing list.

Delft, February 2002,

Maarten Vermeij

Summary

This report describes the performance testing of the Open Source DBMS PostgreSQL when handling spatial cadastral data. The data used in testing was the spatial part of the cadastral database of 3 cadastre offices, Arnhem, Rotterdam and Zoetermeer. All tables in the database contain spatial attributes. During testing it was established that PostgreSQL alone didn't provide all the necessary support for spatial attributes. This lack in functionality was overcome by the use of the PostGIS extension to PostgreSQL. Using this extension all attributes and indices defined in the database definition could be created and used, and all required testing queries could be performed. The use of the extension caused the loading process to become more complicated and slower. The extension did allow queries to be performed using the indices, which wasn't the case when using only PostgreSQL.

The query performance of PostgreSQL/PostGIS isn't near as good as Oracle or Ingres on most parts, since the queries used had to use functions that couldn't use indices. Queries that only used index scans provided a performance similar to that of Oracle. This is the case when using only bounding boxes. This type of query can be also be handled by PostgreSQL only and that results in query times that are better then Oracle or Ingres. It must be noted that the Informix 8.11 DBMS outperforms all these DBMS when it comes to querying. Query times using this DBMS are fraction of those from the other DBMS's.
Since developments of PostgreSQL and especially PostGIS are planned, which should enhance the performance of the combination, future testing can be interesting. Also, since both PostgreSQL and PostGIS are open source software, the missing functionality could be created by writing the necessary routines yourself, if the required knowledge and time are available.

Table of Contents

1 Introduction

This report describes a research into the performance of the open source database PostgreSQL when used to handle a cadastral dataset. During the research it was determined that the geometric support offered by PostgreSQL didn't offer all the necessary functionality for indexing and querying of spatial data, since several required data types and operators weren't available and indexing on some types wasn't directly supported. A short investigation on the internet led to the PostGIS extension for PostgreSQL. This package, offers some additional geometric functionality for PostgreSQL. The PostGIS extension was used to handle all the geometric attributes of the database.
The objective of the research was to give a comparison of the performance of the PostgreSQL DBMS with regards to the previously tested systems Oracle 8i spatial and Ingres and also Informix, although less information on the last one was available. The main comparison will be in the field of the performance (speed) of the loading and querying of geometric cadastral data. The question to be answered therefore is:

> *How does the PostgreSQL DBMS perform in the loading and querying of geometrical spatial cadastral data in comparison with other DBMS's such as Oracle spatial, Ingres and Informix?*

The cadastral database is split into two parts, an administrative part, called AKR and a geometric part, called LKI. Due to limitations in time only tests with LKI where performed. This also limits the number of queries to be performed since some of the queries use a combination of both datasets. The queries used in this research are the same as the ones used in the 'Spatial DBMS testing with data from the Cadastre and TNO NITG [7] report. The comparisons to the other DBMS's are based upon the results presented in that report. Readers interested in the results presented here are therefore advised to also read the that report. The hardware and OS used for this research are the same as the once used for the research presented in that report.

This report has more ore less the same structure as the research that was performed. First there is a description of the data, which was used in the research, which in this case is data from the Cadastre of the Netherlands. Secondly the DBMS PostgreSQL used in the research will be discussed in chapter 3. Since PostgreSQL didn't provide al the functionality needed on spatial data types an extension to the system, PostGIS was used. Chapter 4 gives a description of the PostGIS extension. After all the software that is used is described, a description of the process of loading and indexing the data is given in chapter 5, together with the problems encountered in these actions. Now that the data is ready to be used the queries can be performed. A description of the querying of the data is given in chapter 6. The results of the test are presented in chapter 6. Finally in chapter 7 conclusions will be drawn, based on the findings of the research described in the previous chapters.

2 Dutch Cadastral Dataset

The database of the Dutch cadastre is split into two parts. This is the result of its historical development. The administrative data is stored in a database called AKR, whereas the geometric part has later been added into a database called LKI. Links between the two are made using a number of attributes in the two databases. Due to limitations in time only the geometrical database LKI was tested in this research.

The LKI database consists of 7 different tables. During the research the data of three cadastre offices was used, Arnhem, Rotterdam and Zoetermeer which results in the following total number of records per table in the final database.

Table	**No. of records**
xfio_boundary	10,044,511
xfio_gcpnt	60,986
xfio_line	3,502,313
xfio_parcel	3,820,699
xfio_parcelover	42,852
xfio_sympnt	2,054,463
xfio_text	1,640,122
Totals	**21,165,946**

Table 2.1: Number of records per table in the test set.

No explicit use of the content of the records has been made during this research, except to obtain counting results in the querying.

3 PostgreSQL

This chapter gives a brief description of the PostgreSQL DBMS that was used in this research. Since this research is primarily aimed at the spatial functionality and performance these will be described separately in paragraph 3.2

3.1 The DBMS

The DataBase Management System used in the research is called PostgreSQL. This piece of software originates from the University of California at Berkeley, which also produced the first versions of the Ingres DBMS [4]. PostgreSQL is Open Source software, which means that the source code of the software is (in this case freely) available to everyone interested. This potentially means that everyone interested can make adjustment, enhancements and extensions to the software. But there still is a central group of developers which maintains the 'official' version of the software, with help of many interested and able programmers who offer their views, ideas and skills. This is necessary in order to maintain a consistent product, which is the result of the work of so many individuals. This is especially important since a DBMS is a very complicated piece of software, which should be able to be trusted to handle important data. That is also a reason why many companies stick with commercial software, since then it is often possible to buy guaranteed support, whereas with free software you have to rely on the goodwill of the developers. This is why some open source developers offer paid support contracts. The software itself remains free but you pay for the support.

PostgreSQL has been designed to run on UNIX and UNIX-like platforms. During this research it was located on SUN Enterprise E3500 server with Sun Solaris 7 (for precise specifications see Appendix A.1). PostgreSQL is also distributed in some Linux distributions. It ran perfectly on my own computer using Suse Linux 7.2 (appendix A.3). Although this was only tested with a very limited amount of data, the results of that test provided some interesting results. With this limited set the loading and indexing of the data was performed at least twice as fast as on the SUN machine. However no tests were preformed with larger datasets or with querying. The cause of the difference has not been thoroughly investigated, but using the UNIX top command showed that PostgreSQL used only one of the two available processors of the SUN machine, which with 400 MHz is approximately half as fast as the 900 MHz Athlon processor. Furthermore the SUN machine is used by multiple users, although at the time of the loading little other activity than the PostgreSQL process was shown. Also differences in the PostgreSQL setup are possible since the package provided with Suse 7.2 was already pre-configured and this configuration was not altered or checked. Nevertheless, this is an indication that the loading process was limited by CPU power rather than disk access speed.

Since PostgreSQL is distributed as Open Source software it is normally made available as source code compressed in a single archive file. The first thing that therefore had to be done was to extract the source code files from the archive and compile them using a locally available compiler. The actual compilation is dependent on some configuration options, which are made using a script that is provided with the source code. During this configuration several options can be set, such as the directory where de DBMS itself is to be installed, what default directory will be used for the data and what clients are to be installed.

The PostgreSQL DBMS uses a client-server architecture. The actual data handling and most file access is performed by the server part. This has some implications on the necessary computer access rights. It is preferable that the database server is run under a user account, which has been specifically arranged for this purpose. This way the security of the database and the computer can be maximised and unwanted computer access by the DBMS or database users can be avoided. For example only the server has to be able to access the directories where the data in the database is stored. It is not necessary for database users to have this capability since all database handling is performed through the client applications which in turn access the server. In fact in most situations it is undesirable that users can access these files directly, since this can potentially lead to a corrupt database.

When starting the database server several options can be given to this program, such as a different location of the data files, or an option to allow network access to the server. This last option is important when accessing the database using other than the standard clients.

PostgreSQL maintains its own list of users with individual and group wise right management. This way the rights of users can be limited to avoid unwanted access to the data, being it either intentionally or unintentionally. The user management has much in common with that of operating systems such as UNIX. The user account with all rights is called the super-user. This account can be used to create new users. New users can have a range of authorities, ranging from only reading a database to being a super-user with all possible authorities. Again as with operating system users, it is advisable to limit the users' rights to what they need to be able to do, to avoid unwanted operations on databases.

As mentioned before, access to the server is performed by clients. Clients provide the user with an interface to be able to access their database and thus to perform operations like creating new databases, load data into the database and retrieve data by issuing queries. The standard PostgreSQL client is called psql. This program provides a textual interface, which can be run on any UNIX terminal or via a remote connection such a telnet. On start up of the client the user must provide a username and the name of the database to be accessed. If necessary a password has to be provided to access that database. If the user is authorised to use the specified database, a command prompt in psql will become available, which can be used to enter commands, including SQL queries.

3.2 Spatial functionality

This research focuses primarily on the spatial abilities of the DBMS. PostgreSQL offers some spatial abilities by providing a number of geometric data types. This set of data types provides all the types which are used in the LKI database, therefore the loading of the spatial attributes should be possible in PostgreSQL. Interesting data types are the ones with variable length such as path and polygon. This overcomes the problem of having to define a maximum number of points in for example a polygon, which could result in losing lots of disk space when just a few polygons contain many points and most have only a few points.

Geometric Type	Storage	Representation	Description
Point	16 bytes	(x,y)	Point in space
Line	32 bytes	((x1,y1),(x2,y2))	Infinite line
Lseg	32 bytes	((x1,y1),(x2,y2))	Finite line segment
Box	32 bytes	((x1,y1),(x2,y2))	Rectangular box
Path	4+32n bytes	((x1,y1),...)	Closed path (similar to polygon)
Path	4+32n bytes	[(x1,y1),...]	Open path
Polygon	4+32n bytes	((x1,y1),...)	Polygon (similar to closed path)
circle	24 bytes	<(x,y),r>	Circle (center and radius)

Table3.1: PostgreSQL spatial attributes [4]

The next part of the functionality that is important is the ability to create indices on the spatial attributes. For this purpose PostgreSQL provides the R-Tree index [8]. This type of index is especially useful for spatial data. The LKI database model defined indices on point and path attributes. Unfortunately PostgreSQL only allows indices on boxes and circles. The point and path attributes can be indexed by indexing their bounding boxes. This is possible since PostgreSQL allows the indexing of functions of the attributes. An index on a path attribute would be defined as follows:

```
CREATE INDEX xfio_boundary_0b ON xfio_boundary USING RTREE
(BOX(path_attribute));
```

Here the BOX function returns a box that is the bounding box of the path attribute. Using this 'trick' it is possible to create all the indices defined in the database model. The problem now lies in the using of these indices. This is unfortunately where PostgreSQL's native spatial abilities show a large shortcoming. PostgreSQL is only able to use the RTRee indices when querying with one of the following operators:

Operator	Description	Usage
&&	Overlaps?	box '((0,0),(1,1))' && box '((0,0),(2,2))'
&<	Overlaps to left?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Overlaps to right?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<	Left of?	circle '((0,0),1)' << circle '((5,0),1)'
>>	Is right of?	circle '((5,0),1)' >> circle '((0,0),1)'
@	Contained or on	point '(1,1)' @ circle '((0,0),2)'
~=	Same as	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Table 3.2: PostgreSQL spatial operators [4]

Of the attributes used in this research only the box attributes (or the bounding boxes of other spatial attributes) could be examined in overlap queries. The use of other operators or functions like the <-> distance operator or the # and ?# intersection operators did not allow more precise overlap queries that take the exact shape of the geometries into account. No way was therefore found to perform any spatial queries except box overlaps. Fortunately a solution to the problem was found in the PostGIS extension to PostgreSQL which is the subject of the next chapter.

4 PostGIS

The native spatial capabilities of PostgreSQL are fairly limited, but fortunately an extension is available which enhances PostgreSQL on this terrain, PostGIS [2]. PostGIS is being developed by a Canadian database consulting company, Refractions Research Inc. Although their product didn't have all the desired and planned functionality at the time of this research, it was used to handle all the geometric attributes in the database. PostGIS is planned to be fully Open GIS compliant. PostGIS offers at least the following spatial attributes [3], which also are defined in the OpenGIS simple features specification:

- GEOMETRY
- POINT
- LINESTRING
- POLYGON
- GEOMETRYCOLLECTION
- MULTIPOINT
- MULTILINESTRING
- MULTIPOLYGON

Of these the POINT, POLYGON and LINESTRING are used in the table definitions and MULTILINESTRING is used in the testing queries. The POLYGON data type is used to represent the box attributes in the database, since PostGIS doesn't have a special box attribute. The MULTILINESTRING type is used only in querying and is not present in the PostgreSQL geometric data types set. The GEOMETRY type can be used to store any type of single geometry types. This can be useful when the actual type of the spatial attribute isn't known at the design time of the database.

For loading data PostGIS can use the Open GIS Well Known Text Format (with 3d extentions), which is part of the OpenGIS "Simple Features for SQL" specification [1]. This format differs somewhat from the format that was used in the data files that were used during this research. A description of the conversion that had to be made to these files is given in chapter 5.

Besides the fact that PostGIS offers new functionality for spatial data handling within PostgreSQL, PostGIS also brings along a different indexing method. PostgreSQL provides an R-Tree based indexing method for spatial data-types. The indexing method used in PostGIS is called GiST, which stands for Generalized Search Tree [9]. Early versions of PostGIS allowed the user to decide whether to use the PostgreSQL R-Tree index or the PostGIS GiST index, version 0.6, which was used in this research, and higher only allow GiST indices. The developers offer two reasons for this limitation:

- Building an R-Tree index on a large table of geometries can take over twice as long as a GiST index on the same table.
- R-Tree indices in PostgreSQL cannot handle features, which are larger than 8K in size. GiST indices can, using the "lossy" trick of substituting the bounding box for the feature itself.

The first argument could be tested by indexing attributes which can be defined by both PostgreSQL native geometric attributes and PostGIS attributes, such as boxes. Actually boxes

are one of very few PostgreSQL's geometric attributes that can be indexed using PostgreSQL's native R-Tree index. The creation of an GiST index on the bbox attribute in the xfio_boundary table containing 10,044,511 records took 5h41m17s using the PostGIS GiST index and 4h51m34s using PostgreSQL's native R-Tree index. So the argument that the GiST indices can be created faster, is not confirmed by this test.

PostGIS is planned to be fully OpenGIS compliant, which could be very important in the acceptance and usability of the PostgreSQL/PostGIS combination, since it would increase the usability in combination with software and data from other providers. Once version 1.0 is available it will be submitted for OpenGIS compliance testing. The developers of PostGIS like to compare PostGIS (and the combination with PostgreSQL) with ESRI's SDE or Oracle's Spatial extension.

Like PostgreSQL, PostGIS is open source software, it is therefore again necessary that before being able to use PostGIS, it is compiled. When a specific database is to use the PostGIS extension, this extension then first has to be installed for that specific database. This is done through running a SQL script, which is provided with the source code of PostGIS. This script defines al the necessary functions, objects etc, used by PostGIS. PostGIS requires the PL/pgSQL procedural language extension in PostgreSQL, so if this isn't present it first has to be installed using:

```
CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
  '/export/home/postgres/pgsql/lib/plpgsql.so' LANGUAGE 'C';

CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
HANDLER plpgsql_call_handler
LANCOMPILER 'PL/pgSQL';
```

The OpenGIS "Simple Features Specification for SQL" defines standard GIS object types, the functions required to manipulate them, and a set of meta-data tables [1]. In order to ensure that meta-data remain consistent, operations such as creating and removing a spatial column are carried out through special procedures defined by OpenGIS. Two meta-data tables are created by PostGIS when it is installed in a database: geometry_columns and spatial_ref_sys. The geometry_columns table is used to store meta data on the columns containing PostGIS spatial attributes.

The GEOMETRY_COLUMNS table definition is as follows:

```
CREATE TABLE GEOMETRY_COLUMNS (
  F_TABLE_CATALOG VARCHAR(256) NOT NULL,
  F_TABLE_SCHEMA VARCHAR(256) NOT NULL,
  F_TABLE_NAME VARCHAR(256) NOT NULL,
  F_GEOMETRY_COLUMN VARCHAR(256) NOT NULL,
            COORD_DIMENSION INTEGER NOT NULL,
  SRID INTEGER NOT NULL,
  TYPE VARCHAR(30) NOT NULL
    )
```

To add a PostGIS attribute to a table definition, the following function, which is installed during the PostGIS installation, can be used:

ADDGEOMETRYCOLUMN(<DB_NAME>, <TABLE_NAME>, <COLUMN_NAME>, <SRID>, <TYPE>, <DIMENSION>).

By calling this function a geometry attribute is added to the indicated table and an appropriate record is inserted in the geometry_columns table. The attribute description given by PostgreSQL using the \d command describes all PostGIS attributes as GEOMETRY regardless of the type indicated in the ADDGEOMETRYCOLUMN function. None of the fields available in the GEOMETRY_COLUMNS table where used explicitly in the research.

Besides the usage in this research, PostGIS can also be used to provide Websites with geographical abilities. One application that can use the PostgreSQL/PostGIS combination as a source for its data is the Minnesota Mapserver, an internet web-mapping server [4].

PostGIS also comes with a program, which can convert ESRI Shape files into insert statements that can read by PostgreSQL/PostGIS to fill the database. This can be an important feature since ESRI ArcView is a widely spread GIS application.

It must be noted that PostGIS is a work in progress, so some of the desired (and often already planned) functionality wasn't available at the time of the research. In this research this most notably concerns the intersection of features where the exact boundaries are to be considered and not just their bounding boxes. Fortunately workarounds were possible but it would easier if it was already easily possible. Questions regarding PostGIS that weren't answered in the online documentation could be asked on a forum on the internet. This was done several times during the research and resulted in helpful reactions, often from the developers themselves, an example that support for Open Source software can be very useful and effective.

5 Building the database

5.1 Creating the tables

The first action after the creation of the database is the creation of the tables that are used to store the data. A table consists of a number of columns, which define the various attributes. Each row in a table represents a record. Tables are created using the SQL `CREATE TABLE` statement. Using this statement all the attributes desired can be defined and an empty table is created. Initially all the attributes defined were native PostgreSQL attributes, including the geometry types point, box and path (polyline). The path and polygon attributes are special since they can hold an arbitrary number of nodes. This isn't possible in standard relational databases, but this is possible in PostgreSQL since it's an object-relational DBMS. The table and index creation statements originally used in the research with Oracle and Ingres [6] describing the database model couldn't be used directly to construct the database model in PostgreSQL. Although all of these DBMS's support the SQL standard, there are often differences between the exact formulation of some commands at the level of database creation, especially regarding spatial attributes. Also at the level of the database creation there are differences in offered functionality between DBMS's which can lead to differences in the exact content of the scripts, for example by the use of work-arounds for missing functionality. To create the later-to-be-used PostGIS tables, the OpenGIS function ADDGEOMETRYCOLUMN is used [3]. This function creates the geometry column as well as adding an appropriate record to the geometry_columns table, which contains some meta information regarding the added attribute. This meta information, except the type of the geometry, wasn't explicitly used in this research and the standard create table statement could have sufficed, according to the PostGIS manual. What the exact procedure for adding a PostGIS attribute looks like, wasn't investigated.

5.2 Loading the data

Now that the 7 tables of the LKI database model are ready, they can be filled with data. The data for this research was available in the form of files which were created by an ASCII dump from an Ingres database. Each table in the database had its own file which contained one record per line. The attributes in each line were separated by tab characters. A study of the PostgreSQL online documentation lead to the COPY command as the most suitable for the loading of large amounts of data in a single action. Some practising with the command resulted in the surprisingly simple line:

COPY *tablename* FROM '*path/filename*'

for loading a single file into a table. Using this simple command on all 7 tables and their accompanying data files, resulted in the tables be filled correctly, including the geometric attributes. Apparently the Ingres export format used to create the text files, was fully compatible with PostgreSQL. This was only the case when using only PostgreSQL attributes and not using PostGIS.

Unfortunately it isn't possible to fill the PostGIS-fields using the PostgreSQL copy command. A direct conversion between the corresponding PostgreSQL and PostGIS attributes also

wasn't possible. The documented way to fill records containing PostGIS attributes was the use of one SQL INSERT statement per record:

```
INSERT INTO ROADS_GEOM (ID,GEOM,NAME ) VALUES (1,GeometryFromText('LINEST
RING(191232 243118,191108 243242)',-1),'Jeff Rd');
```

Example taken from [3]

Unfortunately this is a very slow process, and is not very feasible when loading large amounts of data, as was the case in this research. The solution to this problem used in this research was to first load the PostGIS attributes into text attributes using the previously described COPY command and then update the PostGIS attributes by their text to geometry functions. After the conversion it was intended to drop these text columns, which were of no use after the conversion. Although the online documentation of PostgreSQL does describe the syntax of the DROP COLUMN command, an execution of this command resulted in an error, which indicated that this feature isn't implemented in PostGreSQL (yet). To overcome this problem the loading process was extended somewhat. First the final table layout is created, which only contains the actual wanted attributes and not the text attributes necessary for the conversion of the geometric attributes. After that the loading is performed file by file. First a temporary table constructed which contains both the geometric attributes and their corresponding text attributes. The geometric attributes are located in the last columns of each table.

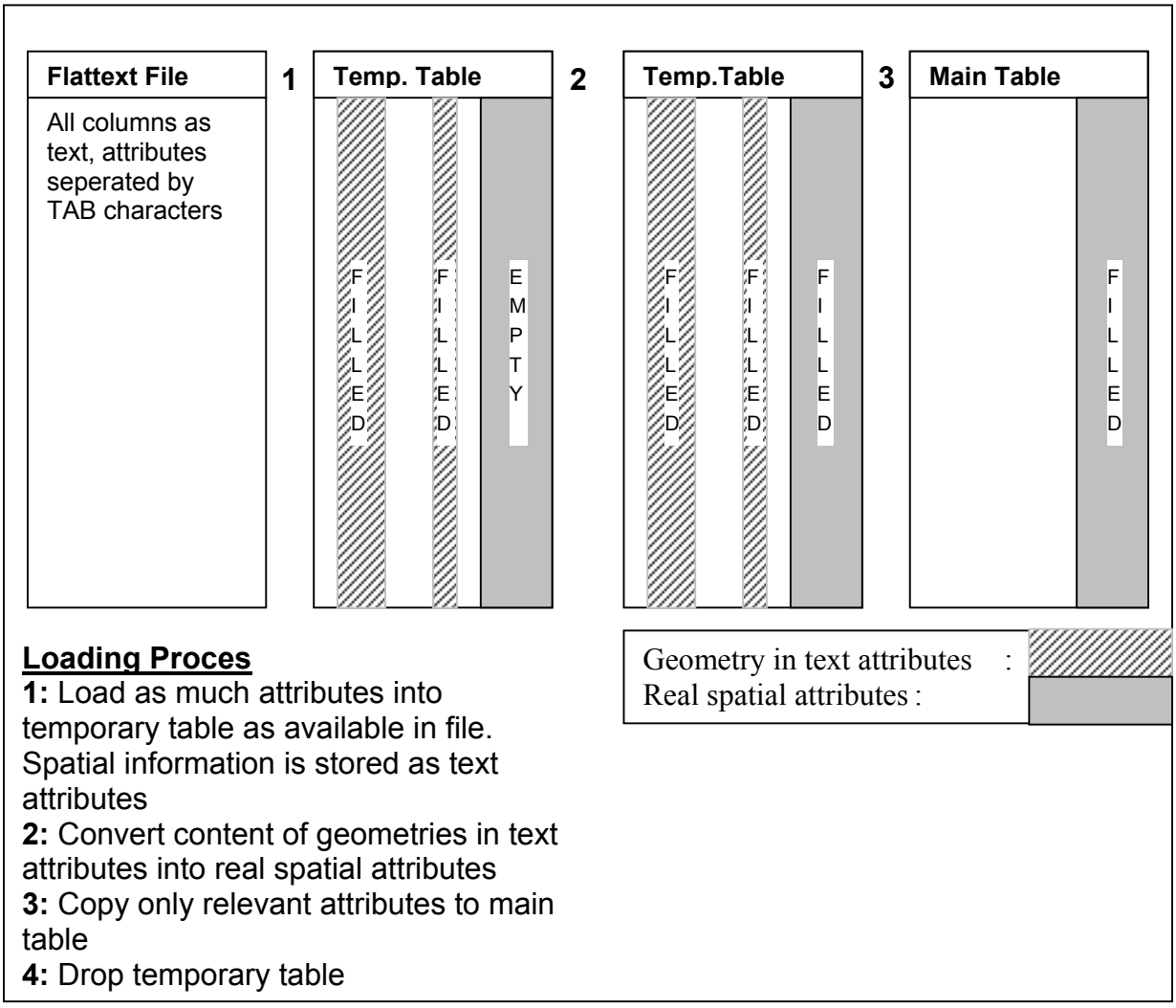


Figure 5.1: Schematic of PostgreSQL/PostGIS data loading process

When the PostgreSQL COPY statement is called it fills as much attributes as are present in the data-file. The result is that all attributes are being filled except the geometric types. When the loading of the file is completed, the geometric attributes are update using the content of their corresponding text attributes. We now have a table that contains all the required data plus the now surplus text attributes. All the required fields are now inserted into the main table, which is thus only filled with data that was originally intended to be there. Finally the temporary table is dropped. This process is schematically represented in figure 5.1.

One more problem had to be solved when converting the data from its textual representation to its PostGIS form. The required input format used by PostGIS is slightly different from the one used in the data files, which (accidentally) was compatible with PostgreSQL geometric format. The differences are indicate in table 5.1.

Type	PostGreSQL	PostGIS
Point	(1,1)	(1 1)
Box	((0,0),(1,1))	X
Polygon	((1,1),(1,0),(0,0))	(1 1,1 0,0 0)
Path (Polyline)	((0,0),(0,1),(1,1))	(0 0,0 1,1 1)

Table 5.1: Difference in textual representation of spatial attributes

Although it would be possible to create functions within the database to perform the changes necessary, it was decided to perform the actions outside the database, directly on the input files. The conversions were made using Perl scripts. Each table has a corresponding script which was used to process the input files for that table. The scripts have the following general layout:

Code	Comment

`#!/usr/local/bin/perl`
`#5  xfio_boundary.shape`
`#16  xfio_boundary.bbox`
	`Call Perl to process this script`
	`Polyline attribute`
	`Box attribute`

`while(<>)`	
`{`	`Start`
`@regel = split ('\t',$_);`	`Split line into array using TAB character as delimiter`
`@regel[5] =~ s/,/ /g;`	`Convert all commas to spaces`
`@regel[5] =~ s/\) \(/,/g;`	`Convert ) ( to ,`
`@regel[5] =~ s/\(\(/\(/g;` `@regel[5] =~ s/\)\)/\)/g;`	`Convert (( to (` `Convert )) to )`
`@regel[16] =~ s/,/ /g;`	`Convert all comma's to spaces`
`@regel[16] =~ s/\) \(/\),\(/g;`	`Convert ) ( to ),(`
`Print join ( "\t" , @regel );`	`Output using TAB as delimiter`
`}`	`End`

Table 5.2: Input file conversion script. Example for xfio_boundary.

Files were converted using the following command:

```
gunzip -c ${TMP}/xfio_boundary.copy.gz | \
perl conv_xfio_boundary > ${TMP}/xfio_boundaryA.dat
```

which is a combination of the decompressing of the available data files and converting them to the necessary input format for PostgreSQL/PostGIS. The text files were originally compressed to save disk space. The additional conversion of the input files and the loading process using the temporary table does cause some additional processing time but this process shouldn't have to be done very often. Most changes to the database will be the addition or changing of a limited number of records, not the loading of the entire database. Although reportedly the Dutch Cadastre frequently reloads the entire dataset, which would increase the importance of fast loading and indexing times.

Errors reported during the first loading trials of the entire data set revealed that some records in the dataset contained the \ character. This character is interpreted specially by PostgreSQL, mostly meaning that the following character is ignored. The consequence is that if a \ precedes a TAB character, which is the default delimiter, all columns within a line are shifted one to the left. This can lead for example to text fields that are attemptedt to be converted to numeric values or attributes that remain empty while being required to be filled. This problem was solved by adding a line to all converting scripts to duplicate all \ characters. This way PostgreSQL interprets them as a single \ character resulting in the data being correctly loaded.

The loading times of the various tables are shown in table 5.2. Displayed is the loading time without the conversion of the flat text files, but with the internal conversion of the spatial attributes.

Table	**No. of records**	**PostgreSQL**	**Oracle**	**Ingres**
xfio_boundary	10,044,511	09:45	04:27	18:16
xfio_gcpnt	60,986	00:04	00:01	00:06
xfio_line	3,502,313	02:11	01:13	05:04
xfio_parcel	3,820,699	04:06	01:11	06:26
xfio_parcelover	42,852	00:01	00:01	00:04
xfio_sympnt	2,054,463	00:57	00:15	02:59
xfio_text	1,640,122	00:49	00:13	02:26
Totals	**21,165,946**			

Table 5.3: Loading times per table (hh:mm)

As can be seen in table 5.3 the PostgreSQL loading times are slower than those of Oracle but faster than Ingres. In [7] it is stated that the slow times of Ingres are the result of the use of the COPYREL command instead of the faster COPY command. This was done since the standard COPY command doesn't support the loading of spatial data from ASCII text. PostgreSQL does support this, but using PostGIS a workaround has to be used, as described. Perhaps a similar procedure as used here could be used on Ingres to speed up that loading process.

The disk size occupied by the data is far greater than that used by Oracle or Ingres. The xfio_boundary table uses 8,330 Mb without indices and 13,7 Mb with indices. Oracle uses 4.8 Gb for this table and Ingres 2.5Gb, both without indices. The entire database including the indices has reached a size of about 27Gb and was even larger during indexing and VACUUM ANALYZE (see page 13).

5.3 Indexing

One very important feature of DBMS's is their ability to create, maintain and use indices. Indices allow a DBMS to find records more quickly then when sequentially searching an

entire table. If an attribute is frequently used in searches, it is likely that an index would increase the search speeds for such queries.
An often used method for database indexing is the B-Tree method but this type can not be used for spatial data. For spatial data R-Tree [8] indices are a better choice or the similar GiST index [9]. Although indices can improve the performance of queries they do produce some additional workload when changing the content of table (inserting, deleting or altering of records), because the indices should be kept up to date. It is therefore useful that when loading large amounts of data into a table, the indices on that table are dropped before the loading and recreated after the loading. This way the DBMS doesn't update the index after each record is inserted, which can be a very time consuming procedure.

As stated in chapter 3, PostgreSQL does provide the ability to use R-Tree indices on spatial data types, but this isn't implemented to work (directly) on all the used spatial data types. The GiST index offered by PostGIS is able to do this. Indices are created using SQL commands like:

```
CREATE INDEX XFIO_BOUNDARY_0B ON XFIO_BOUNDARY
USING GIST (SHAPE GIST_GEOMETRY_OPS);
```

The creation of indices can be quite time consuming on large tables but it can reduce search time dramatically. Since the tables in this research weren't changed after loading the indexing of the tables had only to be done once after loading. Table 5.4 lists some indices and their creation times.

Tablename (indexname)	No. of records	Indexed attribute	Attr Type	Index type	Indexing time
Xfio_boundary (_0b)	10044511	Bbox (polygon	Polygon	GiST	5:41:17
Xfio_boundary (_0)	10044511	shape	Linestring	GiST	5:28:18
Xfio_boundary (_1b)	10044511	object_id	Integer	B-Tree	0:28:17
Xfio_line (_0)	3502313	bbox	Polygon	GiST	1:51:33
Xfio_line (_1)	3502313	Ogroup	Integer	B-Tree	0:10:34
xfio_parcel(_3)	3820699	x_akr_objectnummer	char(17)	B-Tree	0:13:56

Table 5.4: Index creation times some indices.

The PostgreSQL documentation advises to perform a VACUUM ANALYZE on the database after the creation of indices. It isn't exactly clear to me what this command does, but the documentation states: VACUUM ANALYZE: *Updates column statistics used by the optimizer to determine the most efficient way to execute a query* [5]. Anyway, this is a very time consuming operation. When all the tables were loaded it took over three entire days to complete a VACUUM ANALYZE. A test without VACUUM ANALYZE revealed that for the simple type of queries that were used in the testing, it isn't necessary to execute this command, since it didn't yield any advantages in query time.

5.4 Clustering

Besides indexing PostgreSQL also supports the clustering of tables. During clustering the order in which the records are stored is altered to represent that of a certain index. According to the PostgreSQL manual, clustering should mostly affect queries performed on tables which contain lots of duplicate values (in the attribute where the index and the subsequent clustering was based on).

Clustering was only tested on the xfio_boundary table. During the clustering the entire table is copied to a new table and therefore temporarily twice the disk space normally used by the table (without indices) is needed. Queries performed after clustering were performed slightly faster than before clustering (see chapter 7, table 7.2).
PostgreSQL allowed the indexing to be based upon the spatial GiST index without any problem, this in contrast to Oracle, Informix and Ingres which currently do not support clustering based on spatial attributes [7][1].

The following command was issued to create a clustering on the xfio_boundary table based on the xfio_boundary_0 index [5]:

```
CLUSTER xfio_boundary_0 ON xfio_boundary;
```

[1] According to a co-author of that article, this remark needs to be checked for the Informix DBMS.

6 Querying

The most frequent use of databases is in the querying by users. Queries should be handled as quickly as possible, but this should never be at the cost of the correctness of the result. The dataset used in this research is mainly used for its spatial attributes, therefore the queries that were performed during testing, are aimed at testing the performance of the database in this area. There are many different spatial queries possible so first there will be a discussion of some frequently used spatial operators. Paragraph two will discuss the queries used to test the DBMS and the results that were returned. The results are compared to the results of a previous research, which used exactly the same data and the same queries. The content of the results should therefore be the same. However with regards to the performance there can be differences. These will most notably be in the sense of longer or (hopefully) shorter duration of the computations.

6.1 Spatial operators

The following spatial operators are available in Postgis [3]:

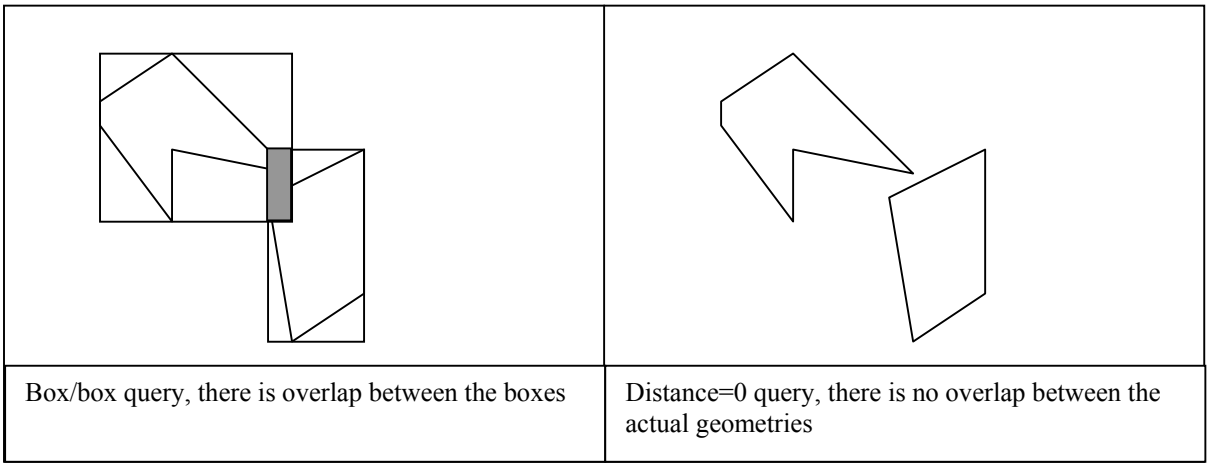
A <& B	returns true if A's bounding box overlaps or is to the right of B's bounding box.
A &> B	returns true if A's bounding box overlaps or is to the left of B's bounding box.
A << B	returns true if A's bounding box is strictly to the right of B's bounding box.
A >> B	returns true if A's bounding box is strictly to the left of B's bounding box.
A ~= B	is the "same as" operator. It tests actual geometric equality of two features. So
A ~ B	returns true of A's bounding box is completely contained by B's bounding box.
A && B	is the "overlaps" operator. If A's bounding box overlaps B's bounding box the operator returns true.

It is interesting to notice that this set doesn't seem to be complete. The first four operators use left and right but there isn't another set of operators that use above and below. Besides these operators which all utilise the bounding boxes of objects, there are some additional functions that can be useful in queries:

TRULY_INSIDE(A,B)	returns true if any part of B is within the bounding box of A.
DISTANCE(A,B)	return the cartesian distance between two geometries in projected units.

Some queries to be performed had to use the actual shape of the objects and not just the bounding boxes. The difference between a bounding box comparison and a comparison that takes the exact geometry into account is visualised in figure 6.1. The first attempts to use these two functions resulted in disappointing results. Search times of over 30 minutes weere encountered when utilising the truly_inside or distance functions. This compared to search times of less than 1 second for Oracle and Ingres. Fortunately the results were correct. An investigation into the source of this problem gave to following cause: instead of using the created indices the database performed a sequential scan. This means that the DBMS checks the validity of every record in a table for a given query. This is especially a problem with large tables, e.g. xfio_boundary. The documentation of PostGIS suggested to set the PostgreSQL variable ENABLE_SEQSCAN to false, but this didn't work, the DBMS still performed a sequential scan of all records. When the && overlap operator from the first set was used an index scan was performed. This leads to the assumption that some operators and

functions do not support index scans. From the operators and functions used in the queries, the && overlap operator supports the use of indices in both PostgreSQL as in PostGIS. The PostGIS DISTANCE function does not use the indices and neither does the PostgreSQL <-> distance operator, which I couldn't get to return the expected answer.
The queries to be used in the testing should therefore at least use some of the first set of operators to limit the number of records that have to be processed by the other functions. Some testing indicated that when queries are constructed which use of a combination of the bounding box and other functions the results will be correct and returned in a time span that is comparable to that of the reference DBMS's.



Box/box query, there is overlap between the boxes	Distance=0 query, there is no overlap between the actual geometries

Figure 6.1: Influence of bounding box use in queries.

6.2 Queries used in testing

The queries used in testing are the same, as the ones used in [6]. At least the intention of the queries is the same, that is, they are made to produce the same result. Since the available operators are somewhat different then the ones used in the reference dataset and given the previously described problem with the use of indices, the actual queries are a little different from the ones used to test the reference databases. The queries are adapted in such a way that they produce the same results with as high a performance as possible. This is achieved by combining the operators that use indices and therefore provide speed, with operators and functions that require sequential scanning of all candidate records but give a more accurate result. That is, they don't use the bounding boxes, but the actual shapes of the objects.
The index operators quickly limit the number of candidate records by comparison of the bounding boxes. These candidates are subsequently checked by more precise operators and functions to arrive at the correct answer to the query. This was implemented in the queries by using the distance function. If the distance between two geometries is equal to zero the two geometries overlap or touch. The selection of records for which the distance function is to be performed, is than made by the standard PostGIS && overlap operator. The general layout of these queries is:

SELECT COUNT (*) FROM *TABLE_NAME* WHERE ((*SEARCHGEOMETRY* && *GEOMETRY*) AND (DISTANCE (*SEARCHGEOMETRY*,*GEOMETRY*)=0));

Where *GEOMETRY* are the records stored in the database and *SEARCHGEOMETRY* is the users region of interest. Without any additional commands the DBMS decides to only calculate the

distance function when the && operator returns true. This query layout has the following consequence: when the selection by the && operator returns a small number of records the total time of the query will be small since few distances are to be calculated, but when a large number of records is returned the query time will increase dramatically over the pure overlap query since a lot of distances have to be calculated. If the query shape differs very much from its bounding box the difference in the number of records returned between a overlap only query and a DISTANCE = 0 query can be very large. Figure 6.1 shows why there can be differences in the result of a queries using only the bounding boxes and a queries using the actual geometries.

The same technique was tried using the PostgreSQL <-> distance operator in combination with the PostgreSQL && overlap operator, but this only resulted in counts that returned 0 as the result. Also no alternatives where found to perform those queries within PostgreSQL.

Both PostgreSQL and PostGIS do not seem to support the use of tolerances in spatial queries. The use of tolerances in spatial data can be important since the measurements that are used to create the data are subject to stochastic variations. The Oracle DBMS supports tolerances and to achieve results equal to Ingres these were set to zero during that research. Since the count results of PostgreSQL/PostGIS are equal to those results, it can be assumed that no tolerances are used, otherwise they are set to zero by default. Also no references to the use of tolerances are made in either the PostgreSQL or the PostGIS online manual.

7 Results

In this chapter the results of the performance testing are presented. These results consist mainly of durations of queries. The timing results are compared to the ones of Oracle 8i Ingres and Informix 8.11 as presented in [6][2]. The numeric results of the queries are fortunately identical to the ones given in [6] and can also be found in appendix F.

The timing results were obtained by creating time stamps in the log files that were created when the queries were run. This was done through the use of SELECT NOW(); SQL queries. The result of this query is a line like 2002-01-30 14:45:00+01 which accompanies every query result. The length of the runtime of a query was obtained by subtracting the time stamp before the query from the one after. Since this had to be done for a lot of queries an Excel spreadsheet was created to automate part of this process. But still all the time stamps had to be copied one by one to the spreadsheet. It would have been nice if this process was automated but that would have taken to long to develop for this research.
To achieve an honest comparison to the other DBMS's tested in [6] one test run was made just after the server had been rebooted. This test also took place in a weekend so few other activity was present on the server. This led to the overall timing results as shown in table 7.1 which are also compared to the results of Oracle, Ingres and Informix as presented in [6]. The entire results of the query timings are given in appendix E. The influence of rebooting the server just before the queries are performed is rather small. The queries are performed approximately 7% faster on a duration of about 2h40m. The increase in speed by clustering is about 10% with regards to timing before clustering, and both without rebooting. These figures are based upon timings as indicated in table 7.2.

As can be seen in table 7.1 the overall performance of PostgreSQL/PostGIS is far behind the performance of the other DBMS's with the exception of the box/box queries. This is the query which only uses the && overlap operator. As soon as more detailed queries are to be performed, that have to take the exact shape of the geometries (both the ones in the query as the ones in the database) into account, the performance of PostgreSQL drops dramatically. This is a consequence of the way these queries were implemented in this research. Distance calculation between two geometries is a computationally expensive function. Especially since the function used is designed to return the exact distance. A cheaper implementation of the function could be designed that does answer the question whether the distance is equal to zero, but without calculating the exact distance. For example by omitting some square rooting that could be used in the function. Of course the distance = 0 was only a workaround for the lack of an overlap function in PostGIS that takes the exact geometries in to account. In the future there will probably be a better, faster function to perform this type of query.
It must be noted that although Oracle and Ingres are both much faster than PostgreSQL/PostGIS, they are still very far from the speed of Informix 8.11. This shows that a dramatic improvement in speed is possible when intelligent search algorithms are implemented, as is clearly the case with Informix 8.11. Informix 8.10 shows results similar to Oracle and Ingres.
Unfortunately no results for PostgreSQL only could be shown in this table, since not all queries could be implemented. Therefore no totals could be calculated for comparison with the other databases. Timings for queries that could be performed are shown in appendix E,

[2] The timing results for Informix are not available in [6].

and they indicate the PostgreSQL only can achieve a better performance than the PostgreSQL/PostGIS combination.

Besides the aspects of the database that were used in the research, additional parameters could be used to influence the performance. An interesting parameter could be the SHARED_BUFFERS variable which is used to assign the number of buffers available. More buffers means that the chance that records can be retrieved from the buffers, instead from disk, is greater [5]. This could improve query performance since memory access is faster than disk access. During performance testing the database was located on a hardware raid5 disk set, but in a single directory. The PostgreSQL documentation only states how to locate an entire database at a specific location whereas other DBMS sometimes allow locations to be set per table and per index. This later option could improve performance since index and table can be accessed simultaneously. A posting on the internet mentioned that this could also be done for PostgreSQL, but it involves manually moving database files to other directories and creating symbolic links. If the data stored is very important, this probably isn't the way you want to go since it can give problems if the database crashes and has to be restored, since you altered files which are normally handled by the DBMS.

Type	No	PostGIS	Oracle	Ingres	Informix 811
Box/Box	1-16	0:00:59	0:01:04		
	17-27	0:42:56	0:40:02		
	Total	**0:43:55**	**0:41:06**		
Bnd/box	1-16	0:01:31	0:01:32	0:00:40	0:01:20
	17-27	2:33:14	0:11:57	0:50:07	0:00:55
	Total	**2:34:45**	**0:13:29**	**0:50:47**	**0:02:15**
Bnd/shp	1-16	0:01:51	0:00:45	0:00:32	0:00:32
	17-27	1:47:34	0:09:42	0:49:47	0:00:24
	Total	**1:49:25**	**0:10:27**	**0:50:19**	**0:00:56**
Line/box	1-16	0:00:21	0:00:12	0:00:06	0:01:10
	17-27	0:45:44	0:06:37	0:11:49	0:01:01
	Total	**0:46:05**	**0:06:49**	**0:11:55**	**0:02:11**
Text/loc	1-16	0:00:05	0:00:05	0:00:02	0:00:07
	17-27	0:04:03	0:01:56	0:04:11	0:00:09
	Total	**0:04:08**	**0:02:01**	**0:04:13**	**0:00:16**
Gcpnt/loc	1-16	0:00:01	0:00:00	0:00:01	0:00:01
	17-27	0:00:12	0:00:05	0:00:10	0:00:01
	Total	**0:00:13**	**0:00:05**	**0:00:11**	**0:00:02**
Sym/loc	1-16	0:00:07	0:00:06	0:00:03	0:00:08
	17-27	0:05:29	0:02:34	0:04:59	0:00:11
	Total	**0:05:36**	**0:02:40**	**0:05:02**	**0:00:19**
Par/box	1-16	0:00:36	0:00:35	0:00:16	0:00:48
	17-27	0:55:51	0:05:15	0:15:28	0:00:59
	Total	**0:56:27**	**0:05:50**	**0:15:44**	**0:01:47**

Table 7.1: Timing comparisons between PostgreSQL, Oracle, Ingres and Informix.

Query	bnd/box	bnd/shp	ln/box	txt/loc	gcp/loc	sym/loc	par/box
Before reboot							
Sum 1-16	0:01:33	0:01:58	0:00:22	0:00:08	0:00:01	0:00:08	0:00:37
Sum 17-27	2:44:03	1:56:45	0:48:37	0:04:25	0:00:12	0:05:43	0:57:45
Total	**2:45:36**	**1:58:43**	**0:48:59**	**0:04:33**	**0:00:13**	**0:05:51**	**0:58:22**
After reboot							
Sum 1-16	0:01:31	0:01:51	0:00:21	0:00:05	0:00:01	0:00:07	0:00:36
Sum 17-27	2:33:14	1:47:34	0:45:44	0:04:03	0:00:12	0:05:29	0:55:51
Total	**2:34:45**	**1:49:25**	**0:46:05**	**0:04:08**	**0:00:13**	**0:05:36**	**0:56:27**
After clustering							
Sum 1-16	0:01:17	0:01:29					
Sum 17-27	2:28:36	1:40:19					
Total	**2:29:53**	**1:41:48**					

Table 7.2: Influence of rebooting and clustering on performance.

8 Conclusions

This chapter comes back to the question presented in chapter one and an answer is given based upon the results and experiences gained in the research. The question to be answered is:

> *How does the PostgreSQL DBMS perform in the loading and querying of geometrical spatial cadastral data in comparison with other DBMS's such as Oracle spatial, Ingres and Informix?*

It must be noted that the research was limited to the spatial part of the dataset. The answer to this question has to be split into two aspects. The first of which is the loading of the data.

Natively the PostgreSQL DBMS does support all the data types present in the LKI table definitions and accidentally the textual format in which the data was available was compatible with PostgreSQL, which gave a good performance on the loading process. However the DBMS lacks in support for spatial data when it comes to indexing and querying. Therefore it was necessary to use the PostGIS extension to PostgreSQL. Using this extension it is possible to successfully perform all necessary actions on the spatial attributes, but the loading of the data became a more complicated procedure. With this loading procedure, loading is performed slower than Oracle, but still faster then Ingres. The GiST indexing method used by PostGIS is claimed to be created faster than PostgreSQL's native R-Tree indexing method, but this couldn't be confirmed in this research. In fact the opposite was observed; GiST indices were created slower than R-Tree indices.

With regards to the performance of querying, the PostgreSQL/PostGIS combination is much slower than Oracle and Ingres on all queries except the pure bounding box queries, which are performed slightly faster. If only PostgreSQL's native spatial abilities are used, the bounding box queries are performed much faster than with PostGIS or Oracle or Ingres. Assuming that PostGIS is used, most of the improvement in performance will have to come from this part. It is important to note that both PostgreSQL and PostGIS are continuously being developed to provide more functionality and better performance. Especially the PostGIS software is expected to provide more functionality, amongst others because they're planning to be OpenGIS compliant in the future, which does require more functionality than presently implemented. It would be interesting to see how well the combination performs when that is the case.

In order to achieve the best performance on queries PostgreSQL/PostGIS allows the clustering based upon the GiST indices on spatial attributes, although this only results in a minor increase of query performance. Another way to increase performance could be the use of multiple physical locations for tables and indices, but this isn't supported by PostgreSQL.

References

[1] OpenGIS website http://www.opengis.org

[2] PostGIS website http://postgis.refractions.net

[3] PostGIS online documentation http://postgis.refractions.net/docs

[4] PostgreSQL website http://www.postgresql.org

[5] PostgreSQL interactive documentation http://www.postgresql.org/idocs

[6] T.P.M. Tijssen, C.W.Quak and P.J.M. van Oosterom. Spatial DBMS testing with data from the Cadastre and TNO NITG. Delft 2001

[7] T.P.M. Tijssen, C.W.Quak and P.J.M. van Oosterom. Testing current DBMS products with realistic spatial data. Delft 2001

[8] M.F.Worboys. GIS, a computing perspective. London 1995

[9] J.M.Hellerstein, J.F.Naughton,A.Pfeffer. Generalized Search Trees for Database Systems. Berkely 1995

Appendix A: Test Configuration

1 Hardware and OS [6]

- Sun Enterprise E3500 server with Sun Solaris 7
- Two 400 MHZ UltraSPARC CPUs with 8Mb CPU cache each
- 2Gb main memory
- two mirrored internal disks of 18.2 Gb, fiber channel;
- two internal software RAID0 sets (3*18.2Gb each), fiber channel
- four external, hardware controlled, RAID5 sets (6*18.2 Gb each)
  all disks are 10,000 rpm

2 Software

- PostgreSQL 7.1.3
- PostGIS 0.6
- gcc (GNU C++) 2.95.2
- perl 5.005_03
- gunzip 1.3

3 My own computer

- Athlon 900 MHZ processor
- 196 Mb main memory
- Asus A7v mainboard
- Suse Linux 7.2 NL
- PostgreSQL 7.0.3
- 40 Gb IBM 305040 harddisk

Appendix B: PostgreSQL only loadscripts

This is an example of the scripts used to load data into a PostgreSQL-only database. Since the files are directly compatible no additional conversions have to be made. The /echo statements are to provide comments in the logfile, so the progress can be monitor. The select now(); statements are present to be able to calculated run times of commands.

```
--loads data from files into db
\echo '***start copy_data.sql***'

select now();
\echo 'copy xfio_boundary'
copy                            xfio_boundary                                   from
'/export/home/postgres/tempdata/rhenen/xfio_boundary.dat';

select now();
\echo 'copy xfio_gcpnt'
copy                             xfio_gcpnt                                     from
'/export/home/postgres/tempdata/rhenen/xfio_gcpnt.dat';

select now();
\echo 'copy xfio_line'
copy xfio_line from '/export/home/postgres/tempdata/rhenen/xfio_line.dat';

select now();
\echo 'copy xfio_parcel';
copy                             xfio_parcel                                    from
'/export/home/postgres/tempdata/rhenen/xfio_parcel.dat';

select now();
\echo 'copy xfio_parcelover';
copy                           xfio_parcelover                                  from
'/export/home/postgres/tempdata/rhenen/xfio_parcelover.dat';

select now();
\echo 'copy xfio_sympnt';
copy                             xfio_sympnt                                    from
'/export/home/postgres/tempdata/rhenen/xfio_sympnt.dat';

select now();
\echo 'copy xfio_text';
copy xfio_text from '/export/home/postgres/tempdata/rhenen/xfio_text.dat';

select now();
\echo '***end copy_data.sql***'
```

Appendix C: PostGIS loadscripts

This is an example of a load scripts as it was used to load into a PostgreSQL/PostGIS database. The data is first loaded into a temporary table, then the spatial attributes are converted and finally the appropriate records are inserted into the main table which has been created before.

```
\echo 'convert geometry content xfio_boundary_tmp, copy to xfio_boundary
and drop'

select now();                                    --create textual time stamp
\echo 'drop any existing xfio_boundary_tmp, including geometry columns'
drop table xfio_boundary_tmp;
delete from geometry_columns where f_table_name = 'xfio_boundary_tmp';
select now();

create table xfio_boundary_tmp
(
  --oid                serial(1)             ,
 ogroup          integer       not null,  -- Group Id (KEY.1) ->CLASS.1
 object_id       integer       not null,  -- Line object Id (KEY.2)
 slc             integer       not null,  -- slc code
 classif         integer       not null,  -- Object class code ->CLASS.2
 interp_cd       smallint      not null,  -- Line interpolation
 shape_txt       text      not null,  -- Line coordinates (2D)
 height          integer       not null,
 node_cd         smallint      not null,  -- Node code
 status_cd       integer       not null,
 fl_line_id      integer       not null,  -- Line Id left side 1st p.->LINE
 fr_line_id      integer       not null,  -- Line Id right side 1st p.->LINE
 ll_line_id      integer       not null,  -- Line Id L side last pnt->LINE
 lr_line_id      integer       not null,  -- Line Id R side last pnt->LINE
 l_obj_id        integer       not null,
 r_obj_id        integer       not null,
 accu_cd         integer       not null,  -- Accuracy code
 bbox_txt        text          ,  -- Bounding box
 --abox          box           ,  -- Area box
 linelen         integer       not null,  -- Length of line
 object_dt       integer       not null, -- Date (of measurement; user time)
 tmin            integer       not null,  -- Time created/last updated
 tmax            integer       not null,  -- Time deleted/last updated
 sel_cd          char(3)              ,  -- Belongs to map: cadast, GBKN...
 source          char(5)              ,  -- Source of data
 quality         char(2)              ,  -- Data quality: method, accuracy
 vis_cd          char(1)       not null,  -- Visibility code
 l_municip       char(5)              ,  -- Left Mun. code ->ALT-TEXPGN.1
 l_section       char(2)       ,  -- Left Section code ->ALT-TEXPGN.2
 l_sheet         char(4)              ,  -- Left Sheet code ->ALT-TEXPGN.3
 l_parcel        char(5)              ,  -- Left Parcel code ->ALT-TEXPGN.4
 l_pp_i_ltr      char(1)              ,
 l_pp_i_nr       char(4)              ,
 r_municip       char(5)              ,  -- Right Mun. code ->ALT-TEXPGN.1
 r_section       char(2)       ,  -- Right Section code ->ALT-TEXPGN.2
 r_sheet         char(4)              ,  -- Right Sheet code ->ALT-TEXPGN.3
 r_parcel        char(5)       ,  -- Right Parcel code ->ALT-TEXPGN.4
 r_pp_i_ltr      char(1)              ,
 r_pp_i_nr       char(4)              not null   -- Visibility code
```

```
);
--Create PostGis fields
\echo 'Create postGis fields'
SELECT AddGeometryColumn ('lki7','xfio_boundary_tmp','shape',-
1,'LINESTRING',2);
SELECT AddGeometryColumn ('lki7','xfio_boundary_tmp','bbox',-
1,'POLYGON',2);

select now();
\echo 'copy data from xfio_boundary.dat';
copy xfio_boundary_tmp from '/export/home/postdata/tmp/xfio_boundary.dat';

\echo 'Update PostGIS fields'
Select now();
\echo 'xfio_boundary_tmp.shape'
update xfio_boundary_tmp set shape = geometryfromtext(('LINESTRING '
||shape_txt)::geometry,-1);

Select now();
\echo 'xfio_boundary_tmp.bbox'
update xfio_boundary_tmp set bbox =

--envelope returns bounding box of a geometry as a polygon
Envelope(geometryfromtext(('MULTILINESTRING '||bbox_txt)::geometry,-1));


Select now();
\echo 'insert content xfio_sympnt_tmp into xfio_boundary'
insert into xfio_boundary
(
      ogroup,
      object_id,
      slc,
      classif,
      interp_cd,
      height,
      node_cd,
      status_cd,
      fl_line_id,
      fr_line_id,
      ll_line_id,
      lr_line_id,
      l_obj_id,
      r_obj_id,
      accu_cd,
      linelen,
      object_dt,
      tmin,
      tmax,
      sel_cd,
      source,
      quality,
      vis_cd,
      l_municip,
      l_section,
      l_sheet,
      l_parcel,
      l_pp_i_ltr,
      l_pp_i_nr,
      r_municip,
      r_section,
```

```
      r_sheet,
      r_parcel,
      r_pp_i_ltr,
      r_pp_i_nr,
      shape,
      bbox
) select
      ogroup,
      object_id,
      slc,
      classif,
      interp_cd,
      height,
      node_cd,
      status_cd,
      fl_line_id,
      fr_line_id,
      ll_line_id,
      lr_line_id,
      l_obj_id,
      r_obj_id,
      accu_cd,
      linelen,
      object_dt,
      tmin,
      tmax,
      sel_cd,
      source,
      quality,
      vis_cd,
      l_municip,
      l_section,
      l_sheet,
      l_parcel,
      l_pp_i_ltr,
      l_pp_i_nr,
      r_municip,
      r_section,
      r_sheet,
      r_parcel,
      r_pp_i_ltr,
      r_pp_i_nr,
      shape,
      bbox
from xfio_boundary_tmp;


select now();
\echo 'drop any existing xfio_boundary_tmp, including geometry columns'
drop table xfio_boundary_tmp;
delete from geometry_columns where f_table_name = 'xfio_boundary_tmp';
```
select now();

Appendix D: Query scripts

Two types of queries where used:

1, queries with only the && overlap operator

```
select now();
\echo  'Query 1 box Scheveningen'

select count (*) from :x_tablename where  ( GeometryFromText(
'POLYGON     ((78550000     457900000,78550000     458025000,78650000
458025000,78650000 457900000,78550000 457900000))'
,-1 ) && :x_attribute );

select now();
```

2, queries with both the && overlap operator and the DISTANCE = 0 function

```
select now();
\echo  'Query 1 box Scheveningen'

select count (*) from :x_tablename where  ( GeometryFromText(
'POLYGON     ((78550000     457900000,78550000     458025000,78650000
458025000,78650000 457900000,78550000 457900000))'
 ,-1 ) && :x_attribute )
AND
distance ( GeometryFromText(
'POLYGON     ((78550000     457900000,78550000     458025000,78650000
458025000,78650000 457900000,78550000 457900000))'
 ,-1 ),:x_attribute )=0;

select now();
```

Herein :x_tablename and :x_attribute are variables that are replace by the table and the attribute to be queried. This way two scripts are sufficient to perform all the required queries.

The script that calls these queries has the following layout:

```
\set                    x_tablename                    'xfio_boundary'
\set x_attribute  'shape'
\i                                                  'querie_general.sql

\set                      x_tablename                        'xfio_text'
\set x_attribute  'location'
\i                                                  '/query_general.sql
```

Whereby query_general.sql is a script that contains queries as described above.
The overall script is run through the PostgreSQL client psql by a UNIX shell script:

```
#!/usr/local/bin/tcsh

psql     -f     /home/user/allqueries.sql     lki7     postgres     >&!
/home/user/query_lki.txt
```

Which starts the client and directs the output to a logfile for later analysis.

Appendix E: Timing results

Results of run *without* reboot (h:mm:ss)

Query	&&1	&&2	bnd/box	bnd/shp	ln/box	txt/loc	gcp/loc	sym/loc	par/box	Totals
1	0:00:00	0:00:01	0:00:01	0:00:01	0:00:01	0:00:01	0:00:00	0:00:00	0:00:00	**0:00:04**
2	0:00:00	0:00:01	0:00:01	0:00:02	0:00:01	0:00:00	0:00:00	0:00:00	0:00:01	**0:00:05**
3	0:00:01	0:00:01	0:00:02	0:00:02	0:00:02	0:00:01	0:00:00	0:00:01	0:00:01	**0:00:09**
4	0:00:00	0:00:01	0:00:01	0:00:01	0:00:01	0:00:00	0:00:00	0:00:00	0:00:01	**0:00:04**
5	0:00:05	0:00:08	0:00:10	0:00:13	0:00:04	0:00:01	0:00:00	0:00:02	0:00:05	**0:00:35**
6	0:00:16	0:00:38	0:00:56	0:01:16	0:00:06	0:00:03	0:00:00	0:00:04	0:00:19	**0:02:44**
7	0:00:00	0:00:00	0:00:01	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	0:00:01	**0:00:02**
8	0:00:00	0:00:00	0:00:00	0:00:01	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	**0:00:01**
9	0:00:00	0:00:01	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	**0:00:00**
10	0:00:00	0:00:01	0:00:02	0:00:02	0:00:01	0:00:00	0:00:00	0:00:00	0:00:01	**0:00:06**
11	0:00:01	0:00:00	0:00:00	0:00:01	0:00:00	0:00:01	0:00:00	0:00:01	0:00:00	**0:00:03**
12	0:00:00	0:00:00	0:00:00	0:00:00	0:00:01	0:00:00	0:00:00	0:00:00	0:00:01	**0:00:02**
13	0:00:00	0:00:00	0:00:01	0:00:01	0:00:01	0:00:00	0:00:01	0:00:00	0:00:00	**0:00:04**
14	0:00:00	0:00:01	0:00:02	0:00:02	0:00:00	0:00:00	0:00:00	0:00:00	0:00:01	**0:00:05**
15	xxxxxxx	0:00:03	0:00:07	0:00:06	0:00:02	0:00:00	0:00:00	0:00:00	0:00:03	**0:00:18**
16	xxxxxxx	0:00:03	0:00:09	0:00:10	0:00:02	0:00:01	0:00:00	0:00:00	0:00:03	**0:00:25**
Sum		**0:00:59**	**0:01:33**	**0:01:58**	**0:00:22**	**0:00:08**	**0:00:01**	**0:00:08**	**0:00:37**	**0:04:47**
17	0:00:07	0:00:10	0:00:20	0:00:17	0:00:05	0:00:01	0:00:00	0:00:02	0:00:09	**0:00:54**
18	0:00:10	0:00:16	0:00:31	0:00:26	0:00:16	0:00:02	0:00:00	0:00:34	0:00:11	**0:02:00**
19	0:02:03	0:03:41	0:10:39	0:08:37	0:03:05	0:00:26	0:00:01	0:00:00	0:03:48	**0:26:36**
20	0:00:14	0:00:23	0:01:11	0:00:50	0:00:19	0:00:02	0:00:00	0:00:01	0:00:26	**0:02:49**
21	0:02:44	0:04:13	0:14:56	0:10:53	0:03:15	0:00:33	0:00:01	0:00:43	0:05:10	**0:35:31**
22	xxxxxxx	0:03:52	0:19:03	0:13:04	0:05:21	0:00:24	0:00:00	0:00:42	0:06:33	**0:45:07**
23	0:01:44	0:03:24	0:12:13	0:09:10	0:03:25	0:00:18	0:00:01	0:00:26	0:04:05	**0:29:38**
24	0:04:26	0:08:03	0:26:23	0:19:21	0:06:22	0:00:38	0:00:03	0:00:49	0:09:41	**1:03:17**
25	0:04:42	0:09:07	0:37:28	0:24:31	0:12:41	0:00:59	0:00:04	0:00:59	0:13:11	**1:29:53**
26	xxxxxxx	0:04:54	0:22:53	0:17:03	0:03:33	0:00:18	0:00:00	0:00:46	0:07:52	**0:52:25**
27	xxxxxxx	0:04:53	0:18:26	0:12:33	0:10:15	0:00:44	0:00:02	0:00:41	0:06:39	**0:49:20**
Sum		**0:42:56**	**2:44:03**	**1:56:45**	**0:48:37**	**0:04:25**	**0:00:12**	**0:05:43**	**0:57:45**	**6:37:30**
Total		**0:43:55**	**2:45:36**	**1:58:43**	**0:48:59**	**0:04:33**	**0:00:13**	**0:05:51**	**0:58:22**	**6:42:17**

All timings are based upon PostgreSQL/PostGIS queries except the &&1 query which is based upon a query using only the PostgreSQL native overlap operator &&. xxxxxxx indicates that that query wasn't performed. The &&2 column shows results of the PostGIS overlap operator &&. Both && operators use only bounding boxes.

Results of run *with* reboot (h:mm:ss)

Query	bnd/box	bnd/shp	ln/box	txt/loc	gcp/loc	sym/loc	par/box	**Totals**
1	0:00:01	0:00:01	0:00:01	0:00:00	0:00:00	0:00:00	0:00:01	*0:00:04*
2	0:00:01	0:00:01	0:00:01	0:00:00	0:00:00	0:00:00	0:00:01	*0:00:04*
3	0:00:02	0:00:02	0:00:02	0:00:01	0:00:00	0:00:00	0:00:01	*0:00:08*
4	0:00:02	0:00:01	0:00:01	0:00:00	0:00:00	0:00:01	0:00:01	*0:00:06*
5	0:00:10	0:00:13	0:00:03	0:00:01	0:00:00	0:00:02	0:00:05	*0:00:34*
6	0:00:52	0:01:10	0:00:06	0:00:02	0:00:00	0:00:04	0:00:18	*0:02:32*
7	0:00:01	0:00:01	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	*0:00:02*
8	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	*0:00:00*
9	0:00:01	0:00:01	0:00:01	0:00:00	0:00:01	0:00:00	0:00:00	*0:00:04*
10	0:00:01	0:00:02	0:00:00	0:00:00	0:00:00	0:00:00	0:00:01	*0:00:04*
11	0:00:01	0:00:00	0:00:00	0:00:01	0:00:00	0:00:00	0:00:00	*0:00:02*
12	0:00:00	0:00:01	0:00:01	0:00:00	0:00:00	0:00:00	0:00:01	*0:00:03*
13	0:00:02	0:00:01	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	*0:00:03*
14	0:00:02	0:00:02	0:00:01	0:00:00	0:00:00	0:00:00	0:00:01	*0:00:06*
15	0:00:06	0:00:06	0:00:02	0:00:00	0:00:00	0:00:00	0:00:03	*0:00:17*
16	0:00:09	0:00:09	0:00:02	0:00:00	0:00:00	0:00:00	0:00:03	*0:00:23*
Sum	**0:01:31**	**0:01:51**	**0:00:21**	**0:00:05**	**0:00:01**	**0:00:07**	**0:00:36**	**0:04:32**
17	0:00:19	0:00:18	0:00:05	0:00:02	0:00:00	0:00:02	0:00:09	*0:00:55*
18	0:00:29	0:00:26	0:00:14	0:00:02	0:00:00	0:00:02	0:00:11	*0:01:24*
19	0:09:48	0:07:59	0:03:02	0:00:19	0:00:01	0:00:31	0:03:41	*0:25:21*
20	0:00:57	0:00:47	0:00:18	0:00:02	0:00:00	0:00:01	0:00:26	*0:02:31*
21	0:13:59	0:10:30	0:03:11	0:00:27	0:00:01	0:00:41	0:05:05	*0:33:54*
22	0:18:16	0:12:25	0:05:12	0:00:24	0:00:01	0:00:41	0:06:20	*0:43:19*
23	0:11:16	0:07:40	0:03:23	0:00:17	0:00:00	0:00:25	0:04:01	*0:27:02*
24	0:25:27	0:18:22	0:06:07	0:00:36	0:00:03	0:00:47	0:09:21	*1:00:43*
25	0:33:32	0:21:04	0:11:56	0:00:55	0:00:03	0:00:55	0:12:52	*1:21:17*
26	0:21:43	0:16:10	0:03:01	0:00:17	0:00:01	0:00:45	0:07:20	*0:49:17*
27	0:17:28	0:11:53	0:09:15	0:00:42	0:00:02	0:00:39	0:06:25	*0:46:24*
Sum	**2:33:14**	**1:47:34**	**0:45:44**	**0:04:03**	**0:00:12**	**0:05:29**	**0:55:51**	**6:12:07**
Total	**2:34:45**	**1:49:25**	**0:46:05**	**0:04:08**	**0:00:13**	**0:05:36**	**0:56:27**	**6:16:39**

Appendix F: Query counts

Count results returned by queries

Query	&&	Bnd/box	Bnd/shp	Ln/box	Txt/loc	Gcp/loc	Sym/loc	Par/box
1	423	423	415	58	78	0	0	231
2	271	271	267	274	71	0	0	115
3	2118	2118	2118	1890	389	0	169	623
4	784	784	782	409	40	0	473	301
5	17250	17250	17241	4181	1400	56	6053	7594
6	142405	142405	142405	8656	9186	51	13398	51951
7	1	1	1	1	0	0	1	10
8	0	0	0	0	0	0	0	0
9	508	508	507	187	126	1	129	198
10	2114	1692	1680	401	227	103	171	649
11	440	396	392	103	89	27	44	198
12	378	300	293	113	46	0	22	137
13	919	462	442	57	39	12	42	227
14	1082	573	549	260	125	2	93	257
15	8472	5189	5159	900	892	8	0	2291
16	13699	10065	10035	864	673	0	0	3891
Sum	**190864**	**182437**	**182286**	**18354**	**13381**	**260**	**20595**	**68673**
17	27911	340	267	14	0	0	0	488
18	50488	242	125	5	0	0	0	248
19	814919	1647	1133	159	0	0	0	2176
20	94963	799	670	47	0	0	0	878
21	956608	1411	931	206	0	0	0	1899
22	895891	1247	884	164	0	0	0	1677
23	735999	1011	792	89	0	0	0	1425
24	1932126	2851	1753	314	0	0	0	3507
25	2027970	2216	1530	740	0	0	0	2914
26	1028776	3305	2385	297	0	0	0	3880
27	1211695	1739	1426	634	0	0	0	2290
Sum	**9777346**	**16808**	**11896**	**2669**	**0**	**0**	**0**	**21382**
Total	**9968210**	**199245**	**194182**	**21023**	**13381**	**260**	**20595**	**90055**