Front Matter

Reconsidering t he Pascal Language

Data Processing for the small organization and hobbyists

Terry A. Haimann

Copyright

I want to thank Dave Wolz for helping me with my editing. Without his help this would be a much less readable book. I would also like to thank Cecil Cook for helping me turn this into an epub. I would also like to thank the Free Pascal community, for providing the tool set which I have found so useful.

Preface

This book is best suited for a hobbyist or the person in charge of data processing for a small organization. This person may have some scripts or even web sites that are creating bottlenecks in their data processing systems. On the other hand this person may find themselves trapped in a canned solutions that are no longer meeting their needs. The tools that I present here are not canned solutions and will require some knowledge and work to make them applicable in your environment.

T

Table of Contents

Front Matter
Preface
- 1. Why Pascal
- 1a. Near c++ performance
- 1b. The code is nearly self-documenting
- 1c. Pascal and the Part-time Programmer
- 1.d Security

2. Embracing MySQL/MariaDB
- 2a. MySQL Performance
- 2b. Defining a database for the beginner
- 2c. Compiler/Interpreter performance
- 2d. CLI library to access MySQL
- 2e. A basic CLI Pascal program
- 2f. A program with multiple queries
- 2g. Comma Separated File
- 2h. Performance Revisited

3. Don't throw those old Binary Files away
- 3.a Security
- 3.b MySQL and Binary Access

4. Doing Pascal the Lazarus way
5. Graphing
6. Capture that data
7. Date Math
8. Math
9. EMail
- 9a. Installing the Library
- 9b. Reading Email
- 9c. Sending Email

10. Summary

1. Why Pascal

Dr. Nicklaus Wirth originally designed the Pascal language about 1969-1971. It was a tiny compiler with limited built-in functionality, and was designed to teach programming to computer science students. He had become disenchanted with what academia was doing with their compiler, Algol. Algol required huge amounts of resources for the day and compilation was slow. He took the best ideas from Algol and simplified them and ended up making the first popular language based on Algol. He was basically making an academic argument for small, efficient tools! By making his compiler small, it compiled quickly. This is still true today and additional functionality was eventually added as libraries. In 1971 this was new and Pascal quickly became a popular programming language and was implemented on most mini-computers and mainframes of the day.

With the arrival of Turbo Pascal in the 1980s, Pascal became an even faster compiler and came with standard string handling routines. Turbo Pascal also streamlined the language getting rid of some obtuse structures like packing and unpacking arrays and after version four added units which became libraries.

Today Pascal is still a very fast compiling language with a lot of functionality added in the libraries which is now pretty full featured. Also, Lazarus which is built on Free Pascal makes a very nice visual programming environment which can easily make GUI type programs.

Pascal has both *procedures* and *functions*. A function returns one value that is assigned to an external variable outside of the function using Pascal's assignment operator. Procedures can update multiple values versus functions returning a single value. By keeping the two types of modules different it makes it clearer to someone reading the source code what the functionality is.

In truth Pascal is all about making the source code readable and maintainable. Pascal found a balance point between overly wordy languages like COBOL and the overly brief. Today's programmers really don't want languages like APL (no one can figure out what the code did, so they throw it away and start over) or COBOL (compiles slowly and obfuscates logic with wordiness).

1a. Near c++ performance

c++ is considered a Tier 1 language and generates extremely fast programs while Free Pascal is classified as a Tier 2 language and is considered to be ***just*** fast. Let's be honest, c and c++ are harder to program in than Object Pascal. Technically speaking c and c++ have higher level of indirection and abstraction. What does this mean in English? With c and c++ it is harder to see the forest because of the trees. c and c++ are much more detail oriented for just a modest performance gain. c and c++ have been described by some authors as closer to the machine. One author even said it is a low-level language with some high-level constructs. There are pluses and minuses to using such a language, but it needs to be used with caution and understanding.

Why is this bad? The programmer can get lost in the detail of trying to get the code right, that he or she can lose the overall sight of what they are trying to accomplish. The coding difficulty level of Object Pascal is somewhat more difficult than Python but gives a considerably higher level of performance.

Let's look at some of these c versus Pascal issues. c requires the programmer to manipulate pointers through out the source codes, while Pascal hides a lot of the pointer manipulation and in the process makes the algorithms appear simpler. By Pascal doing this, many bugs never have the opportunity to occur and an application comes to fruition much sooner. In Pascal when one passes a variable to a procedure or function, it is passed it by value if it doesn't need to be modified. If it needs to be changed, it is passed by reference and that procedure or function can then modify that variable and it will stay changed after exiting the procedure, but if one passes it by value, the change only occurs within the view of that procedure or function and the rest of the program will not see the change. To pass by reference in Pascal, we just put the Pascal reserved word ***var*** in front of the variable name in the procedure declaration.

In c and its descendants, a variable is passed by value just like in Pascal. To pass by reference, instead of passing a variable, the address of the variable is passed and must be treated as a pointer to the variable inside the called function. Inside the function, it can now only be accessed as a pointer accessing a memory location. This makes a void function somewhat more complex than the corresponding Pascal procedure. This may not seem like a big deal, but in a large application, one may have hundreds of functions (c calls procedures void functions), possibly thousands. To a part-time programmer it can become overwhelming, so instead of looking at a forest, all they are seeing are the trees.

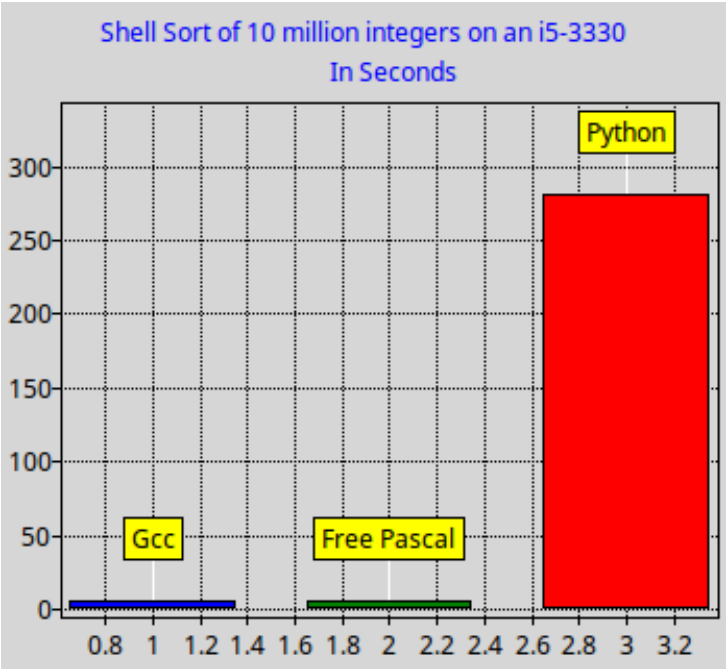
	Example of passing Integer by reference deceleration
Pascal Example	Procedure PassByRefernceExample(Var MyInteger: Integer);
c Example	void PassByReferenceExample(MyInteger *Int);

This can get dangerous, let's say a programmer accidentally forgets the asterisk in front of the pointer and adds or subtracts a value to it. What the programmer meant to add or subtract a value to the passed variable, what he or she has done is changed what the pointer is pointing at. If this programmer is lucky, he or she will just crash his or her program. If the programmer is unlucky the program could have an unpredictable result which may not show itself until much later in it. This can be very difficult to figure out, because he or she will probably be looking at another complete section of the program.

The part-time programmer really doesn't have time to mess with this!

Additionally, in Pascal if the programmer needs a large array, it just can be declared. In c, one has to declare a pointer to a large array and then allocate the memory for it. Then to process it, we have to use pointer math to move through the array, and finally after we are done, we are supposed to release the memory back to the operating system. This is how a beginning or part-time programmer can get lost in the details. Pointer math is another whole level of detail that we can get lost in, while in Pascal we just use an index to the element. All this for a couple percent better efficiency and a smaller amount of RAM. This may have made sense back in the 1970s when a minicomputer would have had less than one megabyte of RAM and only executed about 400,000 instructions per second, today a PC has at least 4 gigabytes[1] of RAM and executes hundreds of billions of instructions per second. A gigabyte is a thousand times larger than a megabyte, and today's PC is running somewhere above one million times faster than that 1970s minicomputer. My point here is that this excessive messing around with pointers creates bugs and with the speed and memory of today's computers it is really unnecessary. Be it typos or logical blunders, c's reliance on pointers for general application programming is dangerous and will cause errors that will never occur in Pascal. Yes, one can use pointers in Pascal, but in Pascal, pointers are usually limited to just a few routines usually working with trees or linked lists. For general business applications they are usually not necessary.

Most benchmarkers state that a Pascal program will run 2 - 3 times longer then a comparable c++ program but your mileage may vary. I find that business type programming will perform nearly identically (with the exception of SQL) especially taking into account they are usually I/O bound, while scientific or mathematical applications will probably be closer 1:3 performance ratios of some of these benchmarks. Even so, that isn't as big of an issue as it once was. Most computers are so fast today that you are unlikely to notice the difference. Let's face it, most of us are writing business applications anyway.



I ran a little benchmark and compared a Shell sort between the three. All three used random data and of course was the same number of items. Free Pascal ran 4% longer then the c program, while the Python application ran 53 times longer then the c program, that is 5,200%. If you are curious, Python ran 51 times longer then Free Pascal or 5,000%. I grabbed the source code for all three off of the web (https://rosettacode.org), so I didn't even muck with the source code to get the performance I wanted. For Raspberry Pi users, Free Pascal is also available. I ran the same Python script and Pascal programs and Python script ran 43 times longer. **Free Pascal can turn a Raspberry Pi into a *tool* instead of just a *toy*.**

One will occasionally hear c++ programmers say something to the effect of "*Real* Programmers don't code in Pascal!" This is really a bunch of macho ***bull shit!*** Back in the late 1980s it became macho to write in c instead of Pascal. The thinking was that c was used to write the UNIX operating system, so why can't I use it to write my application. ***Really???*** The needs of someone writing an operating

1 Billion is $1x10^9$ or a 1,000 times a million

system are considerably different from someone writing business applications. Let's not forget, one can write an operating system in Pascal; the original Mac operating system was written mostly in Object Pascal.

At the end of the day the only thing that matters is the result. Does it do what is needed in a timely fashion? Pascal delivers programs with reasonable run times and efforts in coding. It's a middle of the road solution while c++ delivers fastest possible run times with a somewhat more difficult coding effort. To make a GUI type application, the effort required to program c++ is considerably higher. Historically, Pascal and c were sibling compilers. Dennis Ritchie and his crowd disliked Pascal due to the fact that it couldn't pass variable-sized tables between procedures and functions. Object Pascal removed that limitation many years ago. Remember that with Pascal, there is a lot of power there if it's needed,. but one doesn't have to reach down into the language primitives to get ordinary stuff done, while with c one does!!! One last point: Pascal compiles a lot faster than c or c++, giving an almost interpretive-like access to running programs. There is a price to be paid if one throws everything into the compiler including the kitchen sink, and that is exactly what c++ has done.

Python is also tempting, and many articles and books about it can be found, but the truth of the matter is, it is very slow and is classified as a 4th tier language. I have seen web sites indicating 20-40 times longer than c++, which is a significant difference and is probably not going to be hidden by ever faster machines. Python is an interpreter and interpreters are generally slower than compilers, but even for an interpreter it is slow. Java will run rings around Python.

To put that in a realistic comparison, an i7-8700 at 3.3GHz running Python will give no better performance then c or Pascal running on an Intel Core2 Duo U7500 @ 1.06GHz, which was a processor from 2009. Does one want to spend hard-earned income to buy a fast computer only to have it run like a 10 year old dog of a computer? Maybe that’s OK for a one-time script, but not something used daily or weekly.

How I came up with this result.

When comparing computers I use the Passmark website, and they report that an i7-8700 has a benchmark of 15,273, so CPUs falling between 1/20 – 1/40 of that i7’s performance would give a rating of 381.8 – 763.7, making the midpoint would be about 570. We can go back to the Passmark website, find a computer at about 570, and choose one. It’s not really the closest choice, although fairly close, but it’s a well known CPU, so I went with it.

Any program will perform fine when it is first set up and moved to production, but data grows over time. As the data grows, run times will grow something more than linearly for c++ and Pascal. Python applications' run time could expand exponentially without rewriting many of Python's inner loops in c. That is why Python found its niche as a scripting tool, not a production language, and when it is used for production, it is often excruciatingly slow. Those who use Python for scientific applications use it as a wrapper around c routines, so one actually may have to learn two languages in order to get reasonable run times.

1b. The code is nearly self-documenting

c and c++ make it easy for the programmer to slap code down in a very compressed format. In fact I have seen a **for** loop within a **for** loop all coded within a single line with a few other instructions thrown in, all with ***NO*** spaces. Authors are out there right now imploring c++ programmers to toss this style of programming in favor of a more Pascal-like and readable source code. c and their descendant languages make this and other types of code compression sins easy to write. But this type of code is notoriously hard to maintain or even figure out.

This is a joke, but it's not far from the truth:

In c, one can do this:

```
for(;P("\n").R-;P("|"))for(e=3DC;e-;P("_"+(*u++/8)%2))P("| "+(*u/4)%2);
```

In Pascal, one *can't* do this:

```
for(;P("\n").R-;P("|"))for(e=3DC;e-;P("_"+(*u++/8)%2))P("| "+(*u/4)%2);
```

Pascal doesn't force the programmer to use meaningful names and good code structure, but if a programmer buys into the Pascal ethos, the structured code should follow. A programmer who buys into the c (and descendant languages) ethos often buys into that highly compressed and unreadable coding structure. Yes, one can ram a **for** loop into one line, but even at Pascal's worst, it will be easier to read then c or c++ due to the fact it forces the programmer to use *some* spaces. Pete Goodliffe in his book *Becoming a Better Programmer* stated:

“Favor clarity over brevity. Names don't need to be short to save you key presses -- you'll read the variable name far more times than you'll type it.” and then he goes on to say (My summary). It's all about communicating with future programmers or even ourselves, possibly years from now.

This goes for procedure and function names as well as variable names and program logic. A program written using structured coding principles and pretty printing (such as Allman or GNU) is much easier to maintain. Also, a variable name that actually describes what the variable is used for is much better than a one letter variable name. I am assuming that any programs that the readers of this book will be used to support a small business or proprietorships. It should be understood that once an application is put into production, it often outlives its author. So, one's child and or grandchild could end up supporting this application fifty years from now. There are computer programs that have been running on mainframes since the 1960s. A program will get bigger and more complex with time, but it won't usually go away. The programmers who originally wrote those programs are long retired or dead and somebody else has to maintain them. It costs time and money to replace those programs and until it becomes cheaper to do a complete rewrite (probably never) those programs will not be rewritten, so even clunky old COBOL will still be running for a very long time.

Let's take that statement, “Favor *clarity* over *brevity*.” and talk about the lowly assignment operator. I know a lot of programmers hate Pascal's assignment operator, but it is very clear what it does, it’s easy to read because it stands out. In Pascal it is “:=”, while in c and c++ it is an “=”. Too many times

beginning c programmers forget the difference between “=” and “==” and end up using the “=” when really want the “==”. The first is c's assignment operator while the second is checking for equality. With the Pascal way it is absolutely clear, while a c programmer can waste hours looking for their elusive bug, which is just a wrong operator. The c short hand operators if used in the compressed manner at some point starts to look like gobbledygook and can be murder to decipher what is going on. “Favor *clarity* over *brevity*”, and Pascal makes it much easier. To restate Pete Goodliffe, “***You will read it many more times than you will type it.”***

1c. Pascal and the Part-time Programmer

Again, I am assuming many of the readers of this book are either hobbyists or supporting a small business, proprietorship, or even a non-profit. Part-time programmers need to wear many hats: Management, Bookkeeper, Marketer, laborer, Database Admin, Network Admin, and programmer. To be a c++ programmer today is a full-time job. The Standard Library is huge and it is evolving. The Part-time programmer would also need to be on top of many concepts, while Pascal was designed to teach programming to students. One could get by with one or two used books, such as a book on Turbo Pascal and another one on Delphi; these are easily found in used-book stores or Amazon. To put it simply, Pascal is just easier to grasp. I will confess that I also look up a lot of things on the Internet, as there is a lot of information to be had for free:

Source	Format
Essential Pascal http://www.umass.edu/preferen/Class%20Material/Essential%20Pascal.pdf	PDF by Marco Cantu
http://www2.dcc.ufmg.br/disciplinas/pc/source/essential_delphi.pdf	PDF by Marco Cantu
http://www.tutorialspoint.com/pascal/index.htm	Online tutorial
http://www.efg2.com/Lab/Library/Delphi/	Website

I should inform the reader that Lazarus and Delphi are very similar, but there are differences. The files that Lazarus generates has different naming convention than their Delphi counterparts. The two GUIs look similar but are not identical. When reading *Essential Delphi* or any other Delphi book, please realize that the general concepts are the same but some details may be different.

When one tries to Google it, I find it often helps to throw the word Delphi in with the other search terms, and Delphi is so close to Free Pascal that whatever is said about Delphi usually works. Then things I end up looking up more than once, I throw into a CRM application. Sadly, a lot of the specific Free Pascal documentation online is pretty dry; the Delphi documentation is often better.

1.d Security

There is one additional reason to use a compiled language over an interpreter like Python. If programmers have developed GUI applications in an interpreted language and have clerical staff working for them, the clerical staff (or crackers) in theory could get access to the actual source code and nefariously alter it. although one can take precautions against this like only giving them rights to execute only, security can and has been broken. Python and web programming is now taught in most high schools and secondary school systems. Average clerks may have learned a little of it by helping their children with their school work or when they were in school themselves. Also, one never knows who is a cracker these days since they won't advertise the fact.

If, however, the programs are written in Pascal, they can be disseminated as executables only. It is much more difficult to alter an executable then a text file; it can be done but takes another whole level of expertise beyond what your average clerk or script kiddie would posses. We then can then keep the source files on a private directory, an encrypted drive or even on a separate network. Additionally, one could then check date, ownership and file size on the executables periodically. Changes on any of these would be indicators of someone altering your program files. We could even write our own GUI application to check all of the executables' file sizes and dates and alert us to changes, and no one would know that we had this program. This argument is doubly true for web applications. Every time a bug fix is introduced, a new hole is found. For those using web applications, it would seem prudent to be running periodic security audits.

2. Embracing MySQL/MariaDB

There are many advantages to using a database:

- Sharing data
- Data integrity
- Consistency
- Security
- Smaller footprint.

Pascal did not originally have any support for databases. This isn't all that surprising, since the original Pascal compiler came out about the same time as Dr. Codd published his work on relational databases.

I really like the user interface to MySQL. Getting database and table descriptions from command line is much more intuitive than in competitive products like PostgreSQL. I understand that for large organizations PostgreSQL may be the better solution, but for small business and hobbyist use, MySQL is just easier to work with. It doesn't matter which database one uses, Free Pascal and Lazarus can use most of the free databases and some of the non free ones too. In this book I only will be talking about MySQL, however.

Generally speaking, one should do as much as can be done on the server level, trying not to select individual records and then updating on that row. Sometimes this can't be avoided, but it is very costly in application run time, especially on a network. It is much more run-time-friendly to do a group update.

2a. MySQL Performance

MySQL/MariaDB has a reputation for being very fast. The truth of the matter is MySQL is geared to doing very fast selects, but considerably slower at inserts and updates. This is by design since the designers of MySQL realized their customers were spending much more time performing selects then doing updates and inserts.

2b. Defining a database for the beginner

I started looking around for something on the Internet to describe relational database concepts for a beginner. I am not seeing something that is simple and to the point from a MySQL point of view. I am sure it is out there, but I am not finding it. I also wanted it to be Pascal-oriented or at least language-neutral. I have seen books, but some of them are quite expensive. This is just an introduction, but it could get one started.

Let's start out with a simple concept, a basic payroll system. First, we collect every piece of information we might need and put it into one large table:

Payroll Table	
EmplId	**String(10)**
EmplLast	String(25)
EmplFirst	String(15)
EmplMiddle	String(15)
EmplAddr1	String(40)
EmplAddr2	String(40)
EmplCity	String(15)
EmplState	String(25)
EmplDept	String(25)
EmplPosition	String(25)
EmplHrs	Array[1..26] of Double
EmplPayRate	Array[1..26] of Double
EmplGross	Array[1..26] of Double
EmplNet	Array[1..26] of Double

- **I have put the table keys in bold.**

MySQL has two string types, Char and VarChar. Char uses a fixed number of bytes no matter what, while VarChar uses only what it needs. So if we declare a VarChar of 50, but only use 10 bytes, only about 10 bytes are actually used. The Char can be a tad bit faster, though I tend to use just VarChars. The speed difference is virtually insignificant.

OK, this is an unnormalized table and has multiple problems with it. Unnormalized is considered bad. To move it to First Normal Form, we need to remove repeating data items. **Simply put, that means the *arrays*!** An array data item in a table is not even legal. What we do is create a second table with EmplId, Period as an index. This type of linkage is known as one-to-many. It is a more flexible design with no limit to the number of pay periods, so if a pay period gets split between two consecutive years, one year can have 27 pay periods defined. On the downside, arrays are much faster, relational theory considers data consistency more valuable then speed. The Int type can hold a fairly large integer, so we could store an integer consisting of yyyyww, for example 201801, so multiple years could be stored this way. I know the Address lines could be put in a sub-table, but I am pretty sure that two lines will

suffice. If there were any more than two, I would change it.

So our First Normal Form will look like the following:

Payroll Table	
EmplId	**VarChar(10)**
EmplLast	VarChar(25)
EmplFirst	VarChar(15)
EmplMiddle	VarChar(15)
EmplAddr1	VarChar(40)
EmplAddr2	VarChar(40)
EmplCity	VarChar(15)
EmplState	VarChar(25)
EmplDept	VarChar(25)
EmplPosition	VarChar(25)

PayrollPrd Table	
EmplId	**VarChar(10)**
Period	**Int**
EmplHrs	Double
EmplPayRate	Double
EmplGross	Double
#tocEmplNet	Double

Now when we need to add a new payroll pay amount to an employee, we could just add a record to PayrollPrd. To select the data for an employee, we would do something similar to:

Select Payroll.EmplLast, Payroll.EmplFirst, Payroll.EmplMiddle, PayrollPrd.EmplGross, PayrollPrd.EmplNet from Payroll, PayrollPrd where Payroll.EmplId = 'HaimannT' and Payroll.EmplId = PayrollPrd.EmplId order by PayrollPrd.Period;

“Order by” is how we get our SQL Server to sort our data. We can also add the key word **“Desc”** after the variable we are sorting on and it will sort in descending order. If we wanted the data for all of our employees, we would have dropped our first **where** clause, like the following:

Select Payroll.EmplId, Payroll.EmplLast, Payroll.EmplFirst, Payroll.EmplMiddle, PayrollPrd.EmplGross,
PayrollPrd.EmplNet from Payroll, PayrollPrd where Payroll.EmplId = PayrollPrd.EmplId order by Payroll.EmplId, PayrollPrd.Period;

To move on to Second Normal Form, we need to find data items that are not directly dependent on the primary key of the table. What jumps out to me the most here, is the EmplDept and EmplPosition. EmplDept and Position should be set up as look-up tables. This could prevent inconsistent data across our database. If we put the department name in our primary table, each department could be in our table 100 times or more. By having a data item in our table multiple times misspellings or typos can and usually do occur, but if we use a lookup function and pull the index from a table, no typos will occur. Also, if a change needs to be made, alter the item in the lookup table and it will then propagate throughout the database. Another benefit is that our database will be physically smaller and still hold the same information.

So this is my 2nd Normal Form version of the database:

Payroll Table	
EmplId	**VarChar(10)**
EmplLast	VarChar(25)
EmplFirst	VarChar(15)
EmplMiddle	VarChar(15)
EmplAddr1	VarChar(40)
EmplAddr2	VarChar(40)
EmplCity	VarChar(15)
EmplState	VarChar(25)
EmplDeptIdx	VarChar(10)
EmplPositionIdx	VarChar(10)

PayrollPrd Table	
EmplId	**VarChar(10)**
Period	**Int**
EmplHrs	Double
EmplPayRate	Double
EmplGross	Double
EmplNet	Double

Dept Table	
DeptIdx	**VarChar(10)**
DeptName	VarChar(25)

Position Table	
DeptIdx	**VarChar(10)**
PosIdx	**VarChar(10)**
DeptName	VarChar(25)

Third Normal Form is a fuzzier concept, mostly it talks about functional dependencies. In my opinion, to move to Third Normal Form, anything that can be a lookup table, should be made such. One should try to separate any data that could be in the database multiple times into its own table. This eliminates more possibilities for misspellings and typos that could get into it. Here State and possibly City jumps out. City could go either way, but if it is a small business and all of its employees are within 10 or 15 miles of the business location, one should go ahead and do it!

Once we have things in Third Normal Form, I think we are finished. If it has been done correctly, we should already be in Fourth Normal Form. Fourth Normal Form occurs in just a few special cases and it is not likely to come up.

The final step is actually declaring the data definitions. One will see some fancy tools out there like PhpMyadmin, but they are really not necessary. All of this can be done from the command line and in an editor. I usually type up the table description in an editor and than save that file to a MySql script directory. It will look something like the following:

```
Create Table Payroll (
        EmplId          VarChar(10) Not NULL,
        EmplLast        VarChar(25) Not NULL,
        EmplFirst       VarChar(15) Not NULL,
        ...
        EmplPositionIdx VarChar(10));

Create unique Index PayIdx on Payroll(EmplId);
```

Go on and define every table that you came up with. Each Index has to have a unique name and the Index fields have to be defined. For PayrollPrd, it would look something like this:

Create unique Index PPdIdx on PayrollPrd(EmplId, Period);

Secondly, we then sign in to our MySQL command prompt (probably will need to use the MySQL root user for these steps), if it is running local on your machine, the following command will work:

mysql -u <user> -p

Then we issue our create database command:

Create Database payroll;
use payroll;

Now to execute our script, let's say we called it Payroll.sc, from the command prompt we just type

\. Payroll.sc;

and press enter.

What I have talked about mostly here are one to many relationships and that should cover about 95% of the data relationships you will have. Occasionally though a many to many relationship jumps up. An example here could be Books and Authors. A single book can have multiple authors and an Author can have written multiple books. So what is required is a Linkage Table:

Book Table	
BookIdx	**VarChar(10)**
BookName	VarChar(25)

Link Table	
BookIdx	**VarChar(10)**
AuthorIdx	**VarChar(10)**

Author Table	
AuthorIdx	**VarChar(10)**
AuthorName	VarChar

Now for every book/author combination, there has to be a linkage record.

Book Table	
BookIdx	**Book Name**
P101	Pascal 101

Link Table	
BookIdx	**AuthorIdx**
P101	**NW**

Author Table	
AuthorIdx	**AuthorName**
NW	N. Wirth

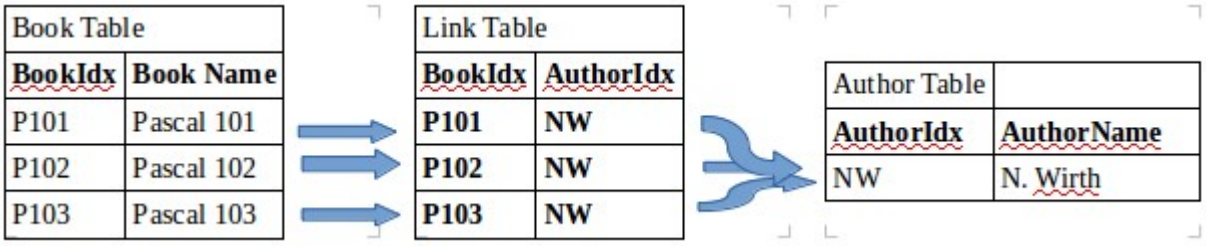
If one wants multiple authors for a book, that book will have multiple linkage records.

Book Table	
BookIdx	**Book Name**
c101	C Lang. 101

Link Table	
BookIdx	**AuthorIdx**
c101	**BK**
c101	**DR**

Author Table	
AuthorIdx	**AuthorName**
BK	B. Kernighan
DR	D. Ritchie

But multiple books can also point to the same author by each having a linkage record.



A full book on MySQL or database concepts should cover most of these concepts in more detail.

2c. Compiler/Interpreter performance

The c library cannot be beat. I have found that using the Pascal Library, I only get about 60% the performance as if I had written it in c. But even with that 60% performance, access time is acceptable. What I am talking about here is just the time required for library source code to execute, not the time required for the query to run in MySQL. The time for the query to execute in MySQL should remain nearly identical across the languages. I have a GUI pascal app that loads stock prices and volume into a chart. The display time is well less than a second and it doesn't matter if I am loading data for the last month, quarter or whole year. In Free Pascal's defense, the libraries are general use allowing a programmer to use one library to hit multiple relational database servers. Also the database objects are data aware, which makes programming easier, but that comes at a cost.

I have run some tests and found Python has a modestly faster MySQL library, taking only about 80% as much time to run the same query. But Pascal's array manipulation occurs about 50 times faster on Intel processors and 40 times faster on Raspberry Pi's. Tables or arrays are the most common data structure used in computer programming and it does not take very much table processing to "***turn the tables"*** on Python. I suspect, but don't have proof that Free Pascal also has a faster GUI library than does Python.

Case in point. I have some MySQL tables with common first names, last names and street names. I loaded each of these into an array. I then used these arrays to create random data by selecting a random integer and using that as an index to each table and then building insert statements from those lookups. I then duplicated this in Python. The Python run time was almost double what Pascal's was. If we do anything but simple input and output, one will get better run times from Free Pascal/Lazarus.

2d. CLI library to access MySQL

OK, this isn't my source code. I found a simple version of this on line somewhere and I improved it a bit. In truth, I could compile this into my own unit, it's small, but I wouldn't gain that much. I just include the file that contains this in my CLI Pascal programs:

```
function GetQuery(Conn: TSqlConnector; Tran: TSqlTransaction) : TSQLQuery;
  var MyQuery : TSQLQuery;
begin
  MyQuery := TSQLQuery.Create(Tran);
  MyQuery.Database := Conn;
  MyQuery.Transaction := Tran;
  GetQuery := MyQuery;
end;

Procedure CreateTransaction(Conn: TSqlConnector;Var Tran: TSqlTransaction);
Begin
  Tran := TSQLTransaction.Create(Tran);
  Tran.Database := Conn;
End;

Function CreateConnection(CType, HostAddr, Database, User, Password: String):
TSqlConnector;
Var
      Conn: TSqlConnector;
Begin
      Conn := TSQLConnector.Create(nil);
      Conn.ConnectorType      := CType;
      Conn.Hostname           := HostAddr;
      Conn.DatabaseName       := Database;
      Conn.UserName           := User;
      Conn.Password           := Password;
      CreateConnection        := Conn;
End;
```

2e. A basic CLI Pascal program

This program just runs a query and prints a report. This is very much a toy and not much beyond the famous Hello World program.

```
Program PlanetOrbits;
Uses SysUtils, Classes, db, sqldb, mysql55conn, Math;

{$I /home/terry/Documents/fpc/MysqlConnLib.inc}

Const
      KKDaysInYr = 365.242189;

Var
      MyConn:           TSqlConnector;
      MyTran:           TsqlTransaction;
      MyQuery:          TSqlQuery;
      PlanetName:       String;
      AU, OrbTime:      Double;



Begin
      // Connect to the MySQL Database and run Query
      MyConn  := CreateConnection('MySQL 5.5', '127.0.0.1',
            'MyDB', 'MyUser',  'MyPass');
      CreateTransaction(MyConn, MyTran);
      MyQuery     := GetQuery(MyConn, MyTran);
      MyQuery.SQL.Text := 'select PlanetName, AU from Planet order by AU';
      MyConn.Open;
      MyQuery.Open;
      // Display Heading Line
      WriteLn(Format('%-22s %-5s %s', ['Planet N.', 'A.U.',
            'Days in Period']));
      If MyConn.Connected Then
      Begin
            While Not MyQuery.eof Do
            Begin
                  // Read the values in from the table.
                  PlanetName :=
                        MyQuery.FieldByName('PlanetName').AsString;
                  AU := MyQuery.FieldByName('AU').AsFloat;
                  OrbTime     := Power(AU, 1.5);
                  WriteLn(Format('%-20s %6.2f %9.2f',[PlanetName,
                        OrbTime, OrbTime  * KKDaysInYr]));
                  MyQuery.Next;
            End;
      End Else WriteLn('Failure');
      MyQuery.Close;
      MyConn.Close;
      MyQuery.Free;
      MyConn.Free;
      MyTran.Free;
End.
```

Some notes:

- The uses clause are standard libraries that this CLI program will use, see green source code. The libraries db, SqlDB, and MysqlNNconn are required libraries to access MySql databases. The NN in MysqlNNconn needs to correspond to the Mysql version you are using.
- The 3rd line with curly brackets and the $I includes the file from 2c, see orange source code. This file will need to be entered in and saved in a standard place where all of your programs can access it, alternatively it could be compiled into its own unit.
- These are common database routines and could be used to hit PostgreSQL or many other open-source databases by altering the uses clause and the first parameter in the CreateConnection function.
- There needs to be a transaction object before there can be a Query object, and there needs to be a connection object before there is a transaction object. Therefore, we must make the connection object first, then we must create the transaction object, and then finally, we can make a query object. See source code in light blue source code.
- Getting a column value from the query can be done in one of two ways (See purple source code Arrow):
  VariableName := QueryName.Fields[nn].VaribleType;
  VariableName := QueryName.FieldByName('FieldName').VariableType;
- Fields[nn], where nn is a number which indicates its field position in the query, is sort of like an array of fields. This is kind of dangerous and not very descriptive. I always use the second method, since I always know exactly what field I am getting, even if it is a bit wordier ***but*** it is clearer to the reader what is going on. Variable type will usually be AsString, AsFloat, or AsInteger.
- For movement within a query, we have the procedures First, Next, Prior, and Last. See dark blue source code. Note on the while statement the check for “End of File”. “QueryName.Eof**”** is a boolean value noting if end of file has been reached.
- The rest of this is basic Object Pascal.

2f. A program with multiple queries

Sometimes we need to access multiple queries in one program. I know my example could be done in a single query, but humor me. In truth, you may run into cases where one big query is impractical or additional data is used to update one of the tables coming from an exterior source such as the Internet. Also I have seen cases where several smaller queries will run faster than one larger one.

Let us say we have a salesman table with salesman id and name, a sales table with salesman id, customer id and sale amount, and finally a Period table with salesman id and total sales for the period. The program we want to write will create a salesman history record for each salesman.

Salesman Table	
Field	Description
SalesmanID	VarChar(15)
SalesmanName	VarChar(60)

Sales Table	
Field	Description
SalesmanID	VarChar(15)
CustomerID	VarChar(15)
SaleAmt	Double

Period Table	
Field	Description
Period	VarChar(10)
SalesmanID	VarChar(15)
Sales	Double

The program will of course need one TsqlConnector, but then will need a separate TsqlTransaction and TsqlQuery for each query. In truth it isn't that hard.

Notes:

- ParamCount has the number of parameters coming from the command line and ParamStr has each item; note ParamStr is an array that uses parentheses instead of brackets. I am getting the Period Name as a parameter from the command line. Getting variables from the command line has been done this way since early days of Turbo Pascal; see green source code.
- Note each separate Transaction and Query. See blue source code.
- Notice how I formatted the queries, there is another way but I find it is more trouble. I discovered the QuotedStr function while researching this book and it makes the source code much easier to read. It places single quotes around a variable. See red source code.

```
Program CreatePeriodData;

Uses SysUtils, Classes, db, sqldb, mysql57conn, Math;

{$I /home/terry/Documents/fpc/MysqlConnLib.inc}
{$I /home/terry/Documents/fpc/DbConst.inc}

Var
      SalesConn:  TSqlConnector;
      SalesPsnTran, SalesTran, PeriodTran:     TsqlTransaction;
      SalesPsnQry, SalesQry, PeriodQry:        TSqlQuery;
      Period, SalesId, SalesSumStr:            String;
      SalesSum:                                Double;

Begin
```

```
If ParamCount <> 1 Then
Begin
      WriteLn('Wrong number of parameters ... Halting');
      Halt(4);
End;
Period := ParamStr(1);

// We need to make a single Connection to MySQL
SalesConn := CreateConnection('MySQL 5.7', '127.0.0.1',
      'ExampleDB', MyUser, MyPass);

// For each Query, we have to create a transaction and query object
// This one is for SalesPsn
CreateTransaction(SalesConn, SalesPsnTran);
SalesPsnQry := GetQuery(SalesConn, SalesPsnTran);

// This one is for Sales
CreateTransaction(SalesConn, SalesTran);
SalesQry := GetQuery(SalesConn, SalesTran);

// This one is for Period
CreateTransaction(SalesConn, PeriodTran);
PeriodQry := GetQuery(SalesConn, PeriodTran);

// Open the Sales Person Query
SalesPsnQry.Sql.Clear;
SalesPsnQry.Sql.Add('Select * from SalesmanTable order by SalesmanID');
SalesPsnQry.Open;
If SalesPsnQry.RecordCount > 0 Then
Repeat
      SalesId := SalesPsnQry.FieldByName('SalesmanID').AsString;

      // Open SalesQry;
      SalesQry.Close;
      SalesQry.Sql.Clear;
      SalesQry.Sql.Add('Select * from Sales Where SalesmanID = '
            + QuotedStr(SalesId));
      SalesQry.Open;

      // Sum Sales for salesman
      SalesSum := 0;
      If SalesQry.RecordCount > 0 Then
      Repeat
            SalesSum := SalesSum +
            SalesQry.FieldByName('SaleAmt').AsFloat;
            SalesQry.Next;
      Until SalesQry.Eof;

      // Create Sql for Insert
      SalesSumStr := Format('%6.2f', [SalesSum]);
      PeriodQry.Sql.Clear;
      PeriodQry.Sql.Add('Insert into Period(Period, SalesmanID, ' +
            'Sales) Values(' + QuotedStr(Period) + ', ' +
            QuotedStr(SalesId) + ', ' + SalesSumStr + ')');

      // We don't actually have to open PeriodQry, just
      // execute the Sql
      PeriodQry.ExecSql;
```

```
            PeriodTran.Commit;
            SalesPsnQry.Next
        Until SalesPsnQry.Eof;
        SalesPsnQry.Close;
        SalesQry.Close;
        SalesPsnQry.Free; SalesPsnTran.Free;
        SalesQry.Free; SalesTran.Free;
        PeriodQry.Free; PeriodTran.Free;
End.
```

2g. Comma Separated File

CSV or Comma Separated Values files are a common method of moving data from a spreadsheet or another relational database into a MySQL table. It used to be wonderfully easy to move data from one application to another. We used the Load Data Infile command, but somewhere around MySQL 5.5 the powers that be decided this was a security issue. So now, you have to start your mysql client with the switch:

local-infile=1

This is a royal pain in the neck and I generally don't do it, mainly because one can't do it programmatically anymore. There comes a point where the added level of security starts to make the system unusable. OK, we aren't there yet, but I really don't like this. For large business organizations, I can see this being a valid issue, but MySQL is more of a small-organization database. For small businesses, organizations, or personal systems, however, this is just a headache. What I have been doing of late is using awk. Most of my csv files are separated by semicolons and awk manipulates these sort of files very easily, so I do something like the following:

```
#!/bin/bash
cat /somewher/somefile | awk -F\; '{printf("\nInsert into work
(Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7,");
printf(" Field8, Field9, Field10, Field11, Field12, Field13, Field14,
Field15, Field16) Values ");
printf("(\"%s\", \"%s\", \"%s\", \"%s\", \"%s\", \"%s\", \"%s\", \"%s
\", \"%s\", \"%s\", \"%s\", \"%s\", \"%s\", \"%s\", \"%s\", \"%s\", \
"%s\");", $1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13,
$14, $15, $16, $17)}' | /usr/bin/mysql --user=MyUser --
password=MyPassword MyDatabase
```

This is dangerous in its own right, especially if one turns it into a script, which I have done here. It turns out there is a Pascal way that works just as well and one can hide ones password inside an executable. Hiding a password inside an executable is not 100% safe either. I assume that disassemblers still exist and someone could disassemble the file. It would take a trained eye to find the password and a script kiddie would probably not be able to do it, but someone familiar with assembly language could. In truth, stopping the script kiddies prevents 95% of the hacking. Very few hackers today really know what they are doing. They have taken scripts developed by others and use those to attack insecure sites. If they are unsuccessful, they move on to an easier target.

I don't think Oracle's fix has improved things. I have found two workarounds (there are probably dozens of others), and they each have security issues. There is no such thing as perfect security, so all one can do is limit the risk. One must not use the root user and passwords on business servers for program access. Instead one should set up a MySQL users for your programmatic access with the minimum access rights to get the job done. The same thing should be done with one's users. One mustn't give them any more rights than they need to get the job done. Even for oneself, one should set up users with increasing security rights and then use the one with the minimum rights to get the job done. That way, if someone finds your commands that were issued the damage will be localized. Also if a user or a program just needs select privileges, don't give them insert, delete or update privileges. This

way at most they will get access to a single database or possibly just a few tables **or even** just limit them to select access.

Now, Pascal starting with Turbo Pascal has always had good string-handling capabilities. This is probably not the best solution for a special job to import a CSV just once. Then again, one can use this source code over and over again with just minor alterations, and compilation only takes a few seconds. If one has a repeating process where one need to update a table weekly or so, this would be a great method. It is also possible that the awk method is faster, but it is also less secure.

There are two functions we are going to look at here, ExtractWord and WordCount, and they are both in the StrUtils library. WordCount returns the number of substrings in a string separated by a separator character:

Cnt := WordCount(CsvBuffer, [';']);

Here, Cnt is an integer, CsvBuffer is the string and ';' is the separator character. Notice the brackets. That allows us to have multiple separator characters, like the following example:

Cnt := WordCount(CsvBuffer, [';', ',']);

which would separate on both semicolons and commas. I personally find commas to be dangerous separator characters, as they find their way into the actual text. When that happens, we get messed-up table columns, where a single column will be split up between two columns. Semicolons are pretty safe, but Tab separated files are about the safest and a chr(09) should work:

Cnt := WordCount(CsvBuffer, [Chr(09)]);

Now after one gets the number of columns in your string, one can use the ExtractWord function to extract each column:

MyStr := ExtractWord(i, CsvBuffer, [';']);

“i” is an integer that refers to the column to extract, CsvBuffer is of course our string, and the array of separator characters works just like it does in WordCount. So the following program therefore should be self-evident:

```
Program ImportCsv;

Uses SysUtils, Classes, db, sqldb, mysql57conn, StrUtils;

{$I /home/terry/Documents/fpc/MysqlConnLib.inc}

Var
     ImpConn:  TSqlConnector;
     ImpTran:  TsqlTransaction;
     ImpQry:   TSqlQuery;
     CsvFile:       Text;
     CsvBuffer:     AnsiString;
```

```
     Qry:      AnsiString;
     i, Cnt:   Integer;
     CsvArray: Array[1..20] of String;

Function SemiColonProblem(s: String): String;
{
     This function is necessary since I found when two
     separator characters were not separated by another
     character, these routines wouldn't see them as a
     separate word.  This situation did occur in actual
     csv files created by LibreOffice Calc.
}
Var
     i:   Integer;
Begin
     For i := Length(s) Downto 1 Do
          If Copy(s,i,2) = ';;' Then
               Insert('" "',s, i+1);
     SemiColonProblem := s;
End;

Begin
     ImpConn := CreateConnection('MySQL 5.7', '127.0.0.1',
          'ExampleDB', 'MyUser', 'MyPass');
     CreateTransaction(ImpConn, ImpTran);
     ImpQry := GetQuery(ImpConn, ImpTran);

     Assign(CsvFile, '/home/terry/Downloads/MyFile.csv');
     Reset(CsvFile);
     While (Not Eof(CsvFile)) Do
     Begin
          ReadLn(CsvFile, CsvBuffer);
          CsvBuffer := SemiColonProblem(CsvBuffer);
          // Clear Array
          For i := 1 To 20 Do
               CsvArray[i] := '';
          // Load single row columns into array
          Cnt := WordCount(CsvBuffer, [';'];
          For i := 1 To Cnt Do
               CsvArray[i] := ExtractWord(i, CsvBuffer, [';']);
          // Build Insert statement from array
          Qry := 'Insert into work ' +
               '(f01, f02, f03, f04, f05, f06, f07, f08, f09, f10) '
               + 'Values(';
          For i := 1 To 9 Do
               Qry := Qry + QuotedStr(CsvArray[i]) + ', ';
          Qry := Qry + QuotedStr(CsvArray[10]) + ')';
          // Insert row into table
```

```
          ImpQry.Sql.Clear;
          ImpQry.Sql.Add(Qry);
          ImpQry.ExecSql;
          ImpTran.Commit;
     End;
End.
```

One last point: text fields are usually surrounded by double quotes; one really doesn't want that in one's table. We can remove the double quotes with the copy function:

```
     MyString := Copy(MyString, 2, Length(MyString)-1);
```

which should remove both the left- and rightmost characters in the above string. If there are other characters you need removing, Pascal also has the pos function:

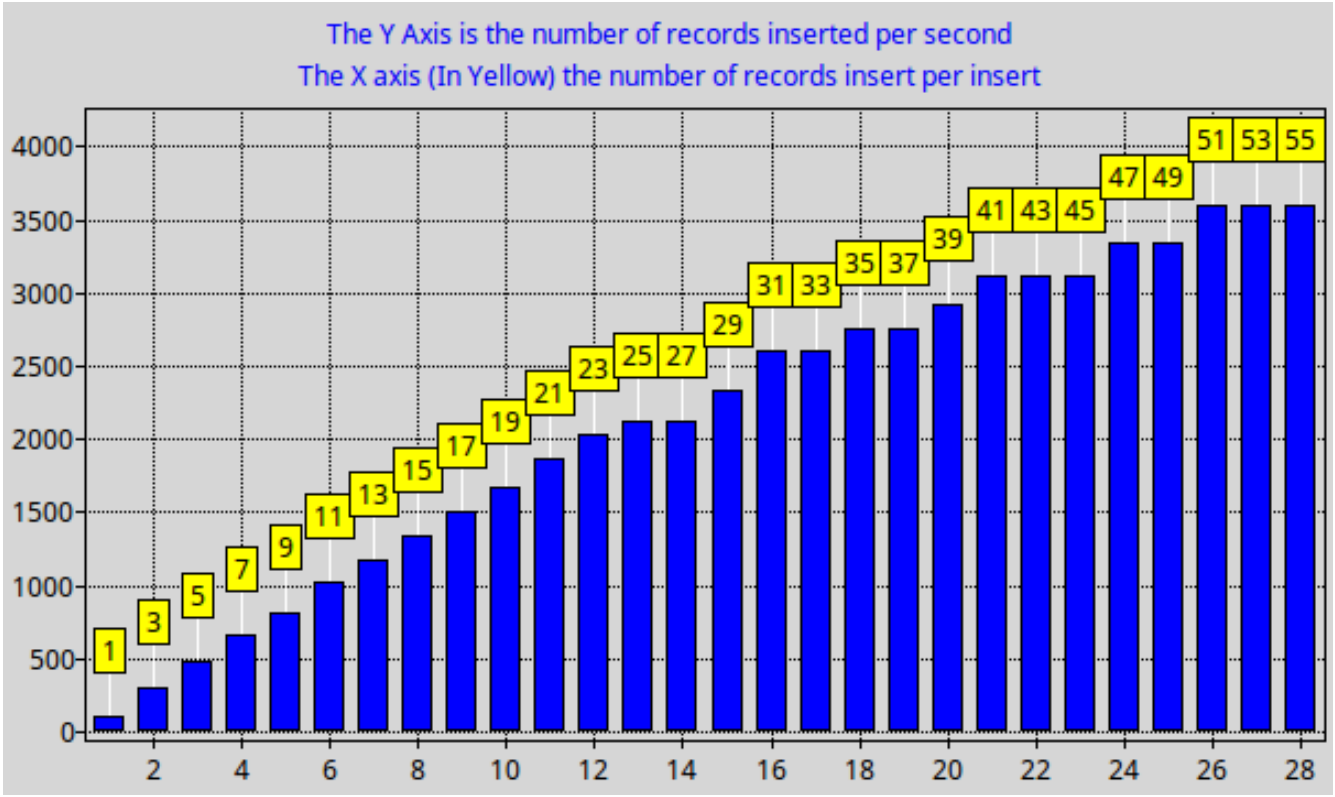
```
     i := Pos('~', MyString);
```

If i is zero, the substring wasn't found; otherwise it will be the position in the string of the character. Both the Copy and Pos functions have been in Object Pascal since the early Turbo Pascal days.

We can improve the performance here considerably by writing multiple inserts as a single SQL statement, something like the following:

```
Insert into MyTable (Field1, Field2, ..., FieldN) Values ('Value1-1',
'Value1-2', ...), ('Value2-1', 'Value2-2', ...);
```

I modified my previous program to let me enter from the command line how many rows per insert to run at one time. The following graph shows the number of logical inserts per second being executed when they were run in blocks:



As one can see it can be driven pretty far. Others have reported similar results and have pushed it well above what I have done here. I saw one web page where they were exceeding 1000 rows per insert. My impression was that they had to have had a high-end server with voluminous amounts of memory. I am running a more modest setup and do not have a corporate budget for hardware. I pushed it up to 55 rows per insert, but I don't think the throughput had topped out and I could have added even more rows. For the small business this is good news. The program to generate multiple rows per insert is somewhat more complex; see below.

```
Program ImportCsv;

Uses SysUtils, Classes, db, sqldb, mysql57conn, StrUtils, DateUtils;

{$I /home/terry/Documents/fpc/MysqlConnLib.inc}
{$I /home/terry/Documents/fpc/DbConst.inc}

Var
      ImpConn:          TSqlConnector;
      ImpTran:          TsqlTransaction;
      ImpQry:           TSqlQuery;
      RecPerInsert:     LongInt;
      RecCnt, GrpCnt:   LongInt;
      Secs:             LongInt;
      CsvBuffer, Qry:   AnsiString;
      CsvFile:          Text;
      BTime, ETime:     TDateTime;

Function SemiColonProblem(s: AnsiString): AnsiString;
{
```

```
      This function is necessary since I found when two
      separator characters were not separated by another
      character, these routines wouldn't see them as a
      separate word.  This situation did occur in actual
      csv files created by LibreOffice Calc.
}
Var
      i:    Integer;
Begin
      For i := Length(s) Downto 1 Do
            If Copy(s,i,2) = ';;' Then
                  Insert('" "',s, i+1);
      SemiColonProblem := s;
End;

Function FormValues(CsvBuffer: AnsiString): AnsiString;
{
      This function just creates a single row for the
      Insert statement.  To gain efficiency multiple rows
      can be inserted in one SQL statement. Note the use
      of Ansi String throughout this program. Ansi (null
      terminated) strings can be quite long while
      standard Pascal Strings may have a limit.
}
Var
      CsvArray:   Array[1..20] of AnsiString;
      i, Cnt:     Integer;
      Q:          AnsiString;
Begin
      Q := '(';
      For i := 1 To 20 Do
            CsvArray[i] := '';
      Cnt := WordCount(CsvBuffer, [';']);
      For i := 1 To Cnt Do
            CsvArray[i] := ExtractWord(i, CsvBuffer, [';']);
      If Length(CsvArray[3]) < 4 Then CsvArray[3] := '0';
      If Length(CsvArray[4]) < 4 Then CsvArray[4] := '0';
      If Copy(CsvArray[14],1,1) = '"' Then CsvArray[14] := '0';
      For i := 2 To 13 Do
      Begin
            If ((i = 5) or (i = 6) or (i = 12)) Then
            Begin
                  Q := Q + QuotedStr(CsvArray[i]) + ', ';
            End Else
                  Q := Q + CsvArray[i] + ', ';
      End;
      FormValues := Q + (CsvArray[14]) + ') ';
End;


Begin
      If ParamCount <> 1 Then
      Begin
            WriteLn('Wrong Number of Parameter ... Aborting.');
            Halt(4);
      End;
      RecPerInsert := StrToInt(ParamStr(1));
```

```
      ImpConn := CreateConnection('MySQL 5.7', '127.0.0.1',
            'TestStars', MyUser, MyPass);
      CreateTransaction(ImpConn, ImpTran);
      ImpQry := GetQuery(ImpConn, ImpTran);
      BTime := Now();
      Assign(CsvFile, '/home/terry/Documents/MyStars/NearStars3.csv');
      Reset(CsvFile);
      ReadLn(CsvFile, CsvBuffer);
      ReadLn(CsvFile, CsvBuffer);
      RecCnt := 0;
      While (Not Eof(CsvFile)) Do
      Begin
            Qry := 'Insert into NearStars3 ' +
                  '(Hip, Hd, Hr, Gliese, Bayer, Proper, Ra, Decl,
                  Distance, VMag, AMag, Spec, ColorIdx) ' +
                  'Values';
            GrpCnt := 1;
            Repeat
                  CsvBuffer := SemiColonProblem(CsvBuffer);
                  Qry := Qry + FormValues(CsvBuffer);
                  If GrpCnt >= RecPerInsert Then
                        Qry := Qry + ''
                  Else
                        Qry := Qry + ', ';
                  GrpCnt := GrpCnt + 1;
                  ReadLn(CsvFile, CsvBuffer);
            Until ((GrpCnt > RecPerInsert) or (Eof(CsvFile)));
            RecCnt := RecCnt + GrpCnt - 1;
            If GrpCnt <= RecPerInsert Then
            Qry := Copy(Qry,1,Length(Qry)-3);
            ImpQry.Sql.Clear;
            ImpQry.Sql.Add(Qry);
            ImpQry.ExecSql;
            ImpTran.Commit;
      End;
      ETime := Now();
      Secs := SecondsBetween(ETime, BTime);
      Secs := RecCnt Div Secs;
      WriteLn('Rec Per Sec: ', Secs);
End.
```

2h. Performance Revisited

So if one already has one's applications in Python, how much does one have to gain moving it to Object Pascal?

It depends on whether one's applications are core bound (meaning the program spends most of its time in computation and very little in I/O), it could be quite a lot, 20 - 40 times faster, meaning a one-hour job could run in three minutes or less. I am assuming that any non-trivial program will do some array processing and/or number crunching, therefore I do expect that the Free Pascal will give better overall run times. However if all they are using the Python for is as a wrapper to execute SQL, then the Python could run faster.

I should note that I have a Free Pascal program with a complex query in it. It had about five left-handed joins in it. I removed the left-handed joins and loaded those lookup tables into arrays before the main query. I then did array lookups (binary searches) and the program run time dropped to 56% from the original. They were small lookup tables and the binary searches didn't help that much; the standard array lookups were giving me about 58% of the original run time. I really don't have the time right now to redo this in Python but suspect it will not improve the run time that much based on my previous experience of Python and arrays.

If your Python scripts have any list processing (what every one else calls arrays or tables), Free Pascal will blow the Python away. I am seeing 50 times longer to do list processing in Python than array processing in Pascal and I use arrays everywhere. In fact, in my first test comparison between Pascal and Python, I thought Pascal had the faster SQL library because I foolishly had an array in those programs.

Is it worth the time and effort? Possibly. One will have to cost that out oneself. It could delay the need for expensive hardware upgrades or just upgrading hardware may be the cheapest option.

3. Don't throw those old Binary Files away

As I stated earlier, on MySQL inserts and updates are slow. On the i5 I spoke of earlier, I can run only about 150 inserts per second running it through as a single process. With more RAM, I probably could get better results. But the old obsolete Pascal binary files are fast—really fast. While I can only insert about 150 rows per second using a MySQL library, I can write about 180,000 records per second using the old binary format. Yes, that is more than a factor of 1000 to one!

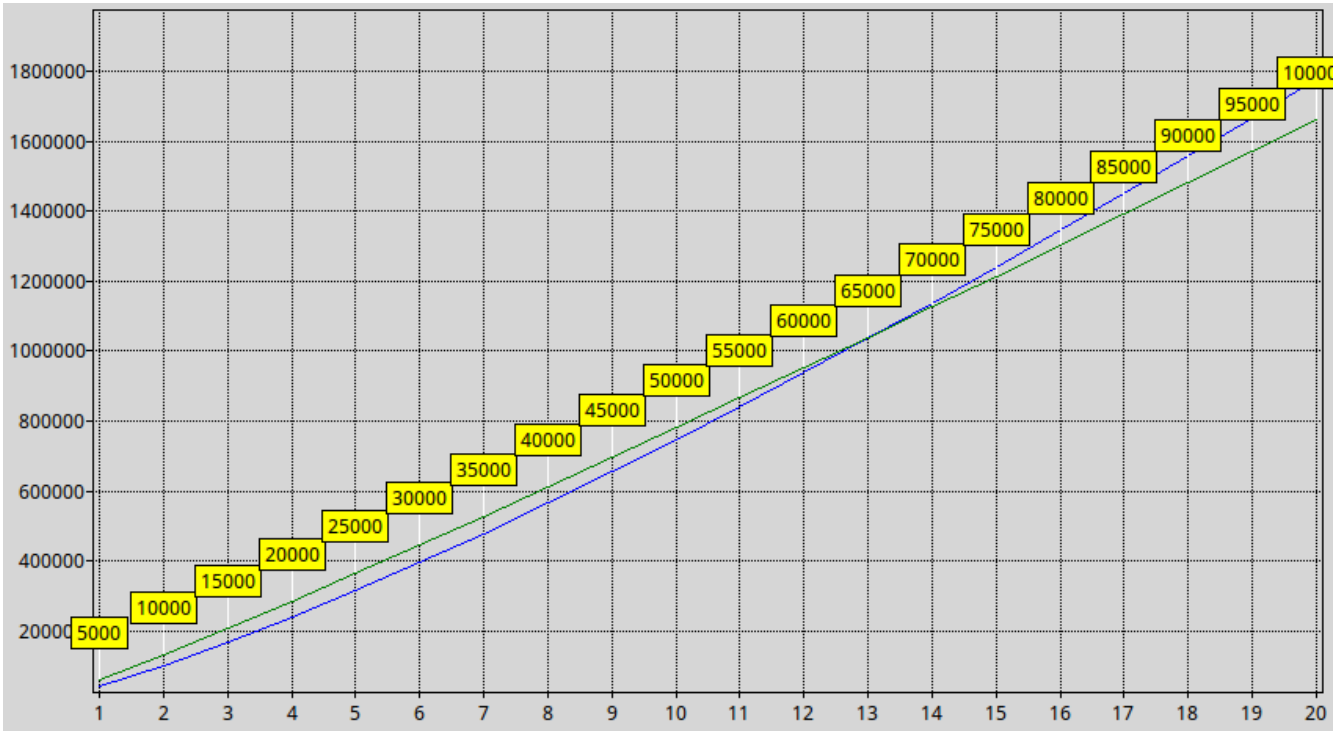
So it seems logical to extract the data from MySQL and write it out to a Pascal binary file. Then each time we have to massage the data along the way to finishing our job, we would will be able to write or update our data with the superior speeds of binary files. Even jobs that require only a few massaging steps could see their run times drop by 50% or more, the bigger the extract or the more massaging steps involved, the more one has to gain. Now if there aren't that many records involved, the MySQL way is more convenient. But if we have thousands of records involved and it is a recurring process, we could easily turn a 1 hour job or more into just a few minute job. By the way, in industry many jobs have hundreds of steps to complete in generating daily reports or batch updates. As the application ages, one may find him- or herself adding additional steps to their jobs. One will need to choose one's battles here, and only do this where the run times are significant. One will never cost out doing this on a job that runs three minutes, 30 minutes possibly.

BinSort on i5-3330 with 128 byte record	
Records	Time in seconds
10,000	0.35
50,000	2.81
100,000	6.57
500,000	50.80

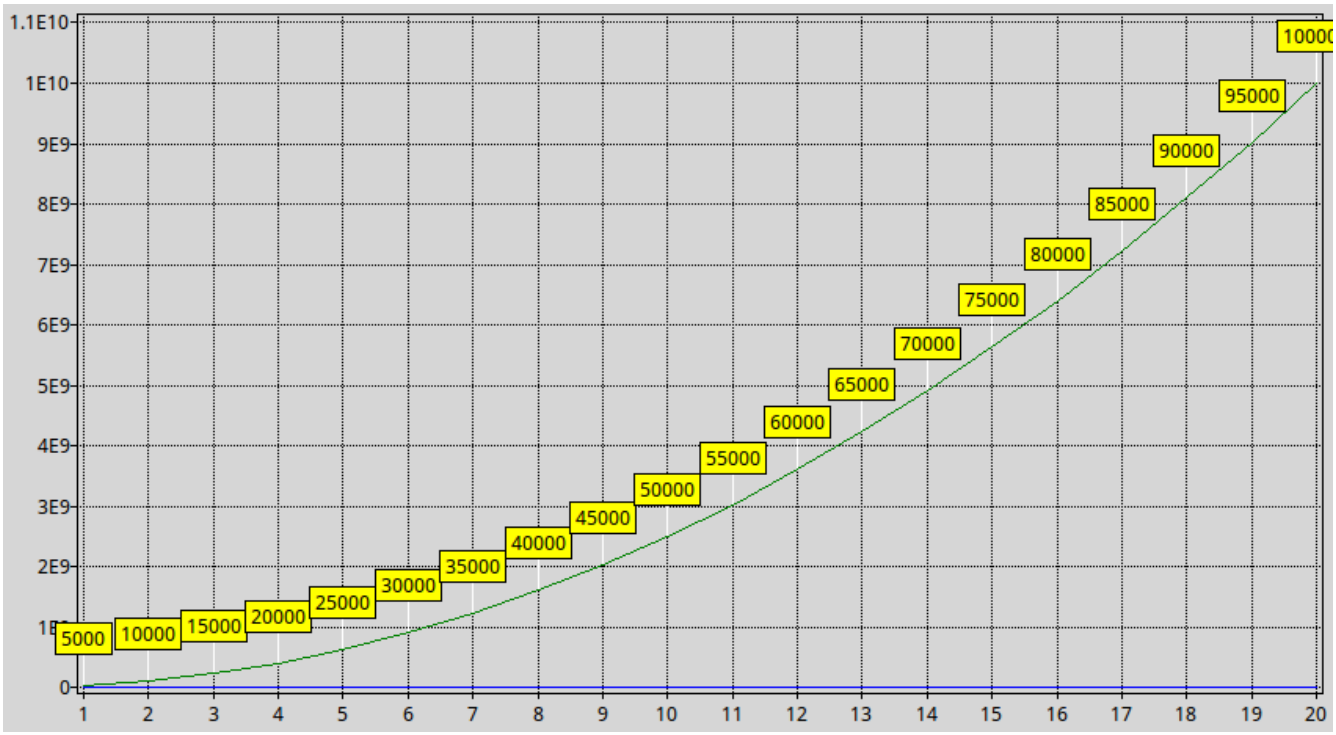
I used the Bubble sort as my sub-sort for this utility. If I had used the Insert sort instead for my sub-sort, it probably would have outperformed the Quick-Sort in the sub-70,000 record range. But we are only talking about saving tenths of seconds and it's not really worth the effort.

Of course any massaging of data usually requires a sort. I don't know of any general sorts still available for Pascal binary files. I don't believe there is a way to use the standard Linux sort utility without converting your binary files to text files. This is of course an option. So I have gone ahead and written my own general sort utility for binary files: http://www.haimann.us/SortDownload.html . It just uses a Shellsort algorithm, but the sort times are really quite good and should be adequate for most small organizational use. Of course it has its limits and only sorts ascending and only supports keys of the type of String, Integer, LongInt, Single and Double. It is also set for a maximum record size of 500 bytes and 50,000 records, but this can be increased by changing a couple of constants in the program and recompiling. Use at own risk. I have tested it but I am not going to guarantee there are no bugs. Read the comments in the source file on how to use it. Theoretically speaking, for small datasets, the Shellsort is the better algorithm. The Quicksort isn't supposed to start outperforming the Shellsort until around 70,000 records. My tests indicate the Quicksort does do a little better, though they are in the same ballpark with sub-70,000 records. That is of course with random data, and data is seldom truly random. The Quicksort can go N-squared (N-Squared indicates the number of comparisons a sort routine will complete which is the square of the number of records. Efficient sorts usually have $N*Log_2(N)$ comparisons which as N grows is a much lesser number and therefore faster. See graphs on next page), with nonrandom data, the programmer can do certain things to prevent that from occurring, but can not completely remove the possibility. The Shellsort can never go N-squared.

If we need to have a reverse or descending sort on a Ordinal Primitive type(Strings, chars and integers), recalculate the field as High(Ordinal Type) – variable, then sort on the new variable. We can recalculate back to the old values after the sort is done



Shellsort(Blue) vs Quicksort(Green)



Quicksort(Blue) vs N-Squared Sort. Note, these graphs to the eye look similar, but the high numbers are considerably different. The first graph has a high number of 1.8 million, while the second one has a high number of 1e10(or 10 billion).

by then repeating the formula. High(Ordinal Type) – NewVariable. The two are inverses of each other. I know a full- featured sort could do this all in one step instead of three, but the price isn't that high. Since strings aren't really an ordinal type, but an array of type **char**, we will have to step through each **char** variable separately. Strings take a few more instructions to invert; see below:

```
Var
     i:   Integer;
     b:   Byte;
     ...

          For i := 1 To Length(MyRecord.MyString) Do
          Begin
               b := Ord(MyRecord.MyString[i]);
               b := High(Byte) - b;
               MyRecord.MyString[i] := Chr(b);
          End;
```

You may have noticed there is no date type. I could have added a date type to the sort but a Julian date is sortable. There is a standard function to convert a TdateTime to and from Julian date. These versions convert into a Double Float, and Julian dates do sort correctly. If one wants to invert the sort,

just subtract the Julian date from 10,000,000, but calculate it to a new field. Converting back and forth is dangerous with float types, since errors will creep in. There are some algorithms around to calculate Julian dates as Long Integers and that would be safer to convert back and forth.

Here is one I have:

```
Const
     MonthArray: Array[1..2, 1..12] of Integer =
          ((31,28,31,30,31,30,31,31,30,31,30,31),
          (31,29,31,30,31,30,31,31,30,31,30,31));

Function ToJulian(DateIn:    TDateTime): LongInt;
{
     Functiuon converts a Gregorian(TDateTime)) to
     Julian Format.
     Gregorian will be in the Format: mm/dd/yyyy
     A Julian date named after Julius Caesar will be
     in the format yyyyddd.  Benefits of Julian Format
     is that it uses less space and is sortable.
}
Var
     dt: LongInt;
     i, j: Integer;
     y, m, d: Word;
Begin
     DecodeDate(DateIn, y, m, d);
     if (y Mod 4) <> 0 Then
          j :=1
     Else
          j := 2;
     dt := Y * 1000;
     If M = 1 Then
     dt := dt + d;
     If M > 1 Then
     Begin
          For i :=1 To m-1 Do
          dt := dt + MonthArray[j,i];
          dt := dt + d;
     end;
     ToJulian := dt;
end;

Procedure ToTDateTime(JDate: LongInt; Var GDate: TDateTime);
{
     Procedure converts a Julian to Gregorian(TDateTime)) Format.
     Gregorian will be in the Format: mm/dd/yyyy
     A Julian date named after Julius Caesar will be
     in the format yyyyddd.
```

```
}
Var
     Yr, Day, mm, dd, i, j:        Word;
     YrWork:                       LongInt;
Begin
     Yr := JDate Div 1000;
     YrWork := Yr * 1000;
     Day := JDate - YrWork;
     If ((Yr Mod 4) <> 0) Then
          j := 1
     Else
          j := 2;
     mm := 0; i := 1;
     while (mm = 0) Do
     Begin
          If Day <= MonthArray[j, i] Then
          Begin
               mm := i;
               dd := Day;
          End;
          Day := Day - MonthArray[j, i];
          i := i + 1;
     End;
     GDate := EncodeDate(Yr, mm,  dd);
End;
```

There are a couple of downsides to using binary files:

3.a Security

One's operating system provides one wall of security, but MySQL has its own security system. So if one's data is hidden in one's database an attacker will have to break multiple levels to get at it But if one leaves data lying around in flat files however, and an attacker has broken into the OS, he or she could conceivably capture some of one's data. Minimally, the temporary binary files should be deleted when done. Possibly, execution of a safe delete should be done after the jobs are run.

One can programmatically delete files when you are done with them with the DeleteFile function. To do a safe delete, one will have to choose one of the many programs available; most are free.

3.b MySQL and Binary Access

There is a second problem doing this though. I'm not sure why, but a program that accesses a MySQL database can't write to a Pascal binary file simultaneously. There is a simple solution though: write the data out in fixed structure format to standard output and then pipe that into another Pascal program which can write it to a binary file. When it is time to insert the completed data back into a MySQL database, one could print it out from the first Pascal program and pipe it into a second Pascal program which would actually execute the inserts. With the amount of data one is likely to pull, one could just dump it into an array, close the SQL connection, and then open the binary file. With modern computers, one can have very large arrays (millions of records) without running out of memory.

I'm not suggesting for one to do away with using an SQL server. The benefits of using such a server are considerable. I'm just saying that there can be a large time savings by storing intermediate files in the old binary format. With the exception of reports at some point one will have to import your data back into the SQL server preferably in a summary format.

A final note on performance of database applications. If the Pascal programmer uses both SQL and Pascal binary files in combination, the Pascal program should perform as well as, if not better than, a c++ program doing SQL alone. Again one's mileage may vary. For small datasets, there is no advantage to using Pascal binary files; the additional work is not offset by much better run times. For large datasets, however, there are gains to be had. Oh, and did I mention, data grows over time? What five or ten years ago could have run perfectly fine as pure SQL, now could be holding things up.

4. Doing Pascal the Lazarus way

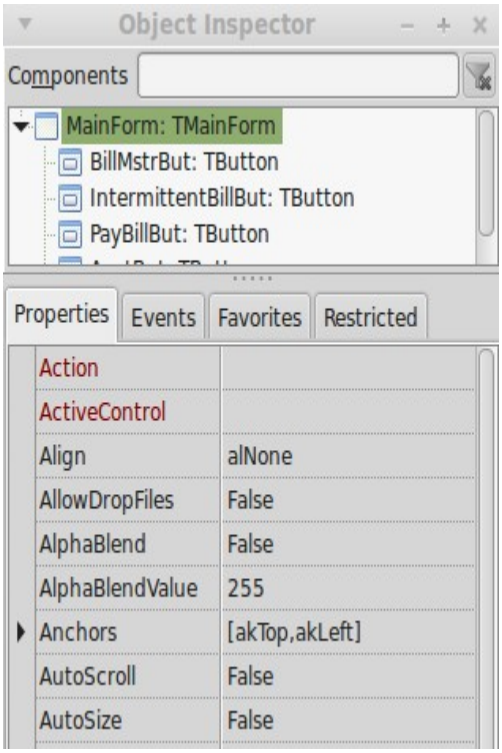
Lazarus is a front end to Free Pascal that gives a programming experience very similar to Delphi. It is what is known as a rapid application development system. It is also a form based system with code closely linked with forms. For every form there is also a unit or source file created.

When I create a new application project, five windows appear:

A. The Lazarus IDE

Lazarus Object Inspector

B. Object Inspector



The Object Inspector allows you to change specific objects parameters after you place them on a form, one can also alter these parameters programmatically.

C. The Form Window

D. A message window, which is where the compiler messages appear.

E. Source Code Window.

On the Lazarus IDE there are tabs going across with the different sections of objects. There is actually a nice selection of objects to choose from. I'm going to just look at a few in this book.

To begin with let's look at a very simple object, the “button”. Under the “Standard” Tab, the button is usually the third from the left. If we click on it and then anywhere on the form, a button will be placed there. We are now able to drag it around to position or resize it as we please. I normally change several values on the Object Inspector immediately. The first is the name of the button to something meaningful, usually ending in “But” to signify its function. The second is the field “Caption”. “Caption” is the word that appears on the button, and I change it to something descriptive.

Making a joke

Yes, it is possible to make a joke out of the button variable names. **But Don't!** This is *extremely poor* programming practice. It can make maintenance more difficult because the programmer has to look beyond the name and try to figure out what the variable is used for. Doing so **purposely adds a level of indirection for no good engineering or practical reason.**

Now if we click on the “Events” Tab in Object Inspector, we will see an event “OnClick”. If we double-click on that, we will be dropped into a source code window within a procedure where you can put the action one would want to happen when the button is clicked.

Simple things that can be done here are “Close;” which will close the window. If it is the only window for this app, it will close the app. In a specific application, we can create multiple forms. We can start an additional form in a button by adding “FormName.ShowModal;” Of course, we will have had to already defined this form and added its Unit to the Uses clause under implementation in the present source code. FYI, pressing the second button from the left on the Lazarus IDE Window, creates a new form. After creation, change the form's name with the Object Inspector and then do a save all. Lazarus will ask what to save the Unit as, give it a meaningful name.

Now I am going to talk about doing a database application with Lazarus. The first decision we have to make is do we use the standard components which are under the SQLDB or do we use the ZeosDBO components which have to be installed. The SQLDB components should already be installed on most Lazarus installations, but the ZeosDBO components are more flexible. On the downside I have run tests and ZeosDBO is considerably slower than the SQLDB library, but I have not noted a performance problem with my internal GUI apps. I had to install the ZeosDBO source code on my system and then had to put it in my path. After that it's a no-brainer and I haven't touched SQLDB in years except for my batch programs. One can read this site, last I knew it wasn't quite up to date so one may have to muck around a bit:

http://wiki.freepascal.org/Zeos_tutorial

Why two libraries?

I realize I am adding complexity by using two separate libraries to access MySQL, one for batch and another for GUI applications. There are good reasons to do such. The standard library that comes with Lazarus is faster and that is an important consideration for batch processing, but the icons that come with Lazarus are hard-linked to the MySQL version and MySQL has a fairly fast upgrade process. Therefore each time MySQL upgrades to a new version, one would have to yank out the icons and replace them with new versions. And remember, it's not just the icons, but the settings within those icons which must be changed, while ZeosDBO seems more flexible and does not require changing a MySQL version parameter. If one decides one needs the extra speed of SqlDB, one would be better off programming it in a similar way such as I did with my batch programs and then just changing the version parameter.

I create a database form that is never visually shown when the application is running. On this form I put the following Icons from the Zeos Tab

- TzConnection
- TzQuery
- TDatasource (From the Data Access Tab)

I usually rename TzConnection to the database I am accessing. I also change protocol to MySQL, user to the database user ID and password to the corresponding User ID and password. For testing I usually set the database to active. We will need a TzQuery and TDatasource for every table or query we need to access. TzQuery is the logical connection to each of our database tables. Our visual objects that show the row level values connect to the query through the TDatasource, so we need one TzConnection for each database, and one TzQuery and TDatasource for each table or query one is using. In the Object Inspector I usually change the name of TzQuery to a Table Name usually followed by “Qry.” I set Connection to the TzConnection variable name. I then put the actual query into the SQL field of the TzQuery object and then set it to active for testing. I finally give a name to TDatasource and set its Dataset name to the Query Variable name. We then have to set dataset field in the TDatasource to point to the query. I know this sounds complex, but it isn't really that bad.

Now we move back to our Data entry form. In its Unit one will have to add to the Uses clause (the one right under the word Implementation) the unit name that is connected with the Database Form. Now place a TDBNavigator icon on the form, it will be on the DataControls Tab. Now go to the Object Inspector Properties and change the datasource field to the datasource name on one's database form. If the Database and Query are active, so will be the Navigator. Now the third icon on the Data access tab is a TDBEdit icon. Place that somewhere on your form. Set its Datasource Field to the same datasource we used in the navigator. Now clicking on the datafield should get a list of fields from the select, choose one. Now if we compile the app, using the navigator buttons, one can page through the table. The TdbEdit field will display the field we chose at the current row. If one hits the “+” button, one can add a row. Clicking the “-” will delete a row. The pencil allows one to edit the current row. If one hits the checkmark, it will post one's edit. It's all very powerful, and with these tools, one can create a custom database application.

There is another database-aware object I want to discuss, TdbLookupComboBox. Place one on the form and change its values in Object Inspector as we did with the TdbEdit. Also give it a meaningful name; I usually end the variable name with DBCB. Let's call it ApprovalDBCB for simplicity's sake. Now click on the form and then the Object Inspector and finally the Events tab. Double-clicking on the OnActivate will drop one into a new procedure. One should add the following code:

```
ApprovalDBCB.Items.Clear;
ApprovalDBCB.Items.Add('Maybe');
ApprovalDBCB.Items.Add('No');
ApprovalDBCB.Items.Add('Yes');
```

Now if we compile this, it will be a drop-down box; upon clicking one of the selections it will fill the column row with the chosen selections. I often put a whole lookup table into these drop-down boxes like this:

```
StateDBCB.Items.Clear;
DbForm.StateTable.First;
Repeat
    StateDBCB.Items.Add(DbForm.StateTable.FieldByName('StName').AsString);
    DbForm.StateTable.Next;
until DbForm.StateTable.EOF;
```

It's fast and makes data entry easy. There are other types of database-aware objects that are also useful. There's are listbox, memobox, radio button, and checkbox objects and at one time or another I have used most of them.

Now let's say we have a tech-savvy playboy who wants to keep records on all the ladies that he is or has dated. He wants a fully relational database that keeps track of their hobbies, educational level, schools they attended, politics, when and where their dates were, likes and dislikes and such. We are going to look at just at his main form. Often he starts out with just a name and city, since a lady may not give out much personal information until she feels safe to do such.

Thus on the main data-entry form he wants their name, address, city, state, etc., but he also wants to record their level of attractiveness, perceived intellect and how easy it will be or is to have sex with them.

When a record is created, the fields he needs access to are her name and city, and the first date status. It would be silly to have access to all of the fields of the main form until the first date actually has occurred. So, we set a database field to indicate if the first date has taken place. Then we can define actions to take depending on what button has been pressed on our TDBNavigator bar. To do that, we select our TDBNavigator button, then go to ObjectInspector window and click on the “Events” tab. We should then see an “OnClick” event, then just double-clicking in the right hand column of the OnClick event. We will then be dropped into a source-code window and it will have automatically created a procedure for us. This is what my completed procedure looks like:

```
procedure TLadyForm.LadyNavClick(Sender: TObject; Button:
TDBNavButtonType);
begin
     If (Button = nbInsert) Then SetNoDateYet(Sender);
     If ((Button = nbFirst) or (Button = nbLast) or (Button = nbNext)
```

```
          or (Button = nbPrior) or (Button = nbPost)) Then
          Begin
               If DbForm.LadyQry.FieldByName('FirstDt').AsString =
                    'Yes' Then
                    SetHadDate(Sender)
               Else
                    SetNoDateYet(Sender);
          end;
end;
```

NbInsert, NbLast, NbFirst, NbNext, NbPrior, NbPost are predefined constants which corresponds to database actions on the TDBNavigator Bar. SetHadDate and SetNoDateYet are procedures I have written.

Why, well, those procedures are going to have to be executed in multiple places and having duplicated source code throughout a program is considered a very bad programming practice. If a bug or fix has to be put in, all of them would have to be updated, while if we create a procedure, correct the one procedure and they are all fixed. These procedures aren't ordinary procedures, they are object oriented procedures and we have to be careful how we set them up. Is it worth the time and effort to do this? **Yes it is!**

We will have to go in and define the procedure first; this will be under the tform definition in the Type Section. If we have already placed buttons or other object on your form, there are already several procedure and possibly function definitions there. Place this new procedure definition under those in the format of:

```
Procedure MyProcedureName(Sender: TObject);
```

The Sender variable is the data in the form and this will allow it to update your form variables. Now we put the actual procedure underneath the implementation. Below that we look for the form name in bracketbraces; place our actual procedure underneath that. I usually do a copy and paste from the procedure name defined in the type section.

```
Procedure MyProcedureName(Sender: TObject);
```

Now in front of the MyProcedureName, put the tform name followed by a period. It is going to be the same as what is just a couple lines below the word implementation in braces. In this case, it was TLadyForm and the SetHadDate procedure will look like this:

If one can't tell, I like Pete Goodliffe's book *Becoming a Better Programmer*. The book is often humorous and just drips with the experiences of a professional programmer.

He states therein:

“Unnecessary code duplication is evil. We mostly see this crime perpetrated through the application of *cut and paste programming*: when a lazy programmer chooses not to factor repeated code into a function, but physically copies it from one place to another in their editor. Sloppy. This sin is compounded when the code is pasted with minor alterations.

When you duplicate code, you hide the repeated structure, and you copy all of the existing bugs. Even if you repair one instance of the code, there will be a queue of identical bugs ready to bite you another day. Refactor existing code sections into a single function.”

In his book, he seems to be talking mostly to c++ programmers, but it is mostly language neutral. So IMHO, you can change the word Functions to also mean Procedures.

```
Procedure TLadyForm.SetHadDate(Sender: TObject);
Begin
     LadyIdxEdit.Enabled         := True;
     LNameDBEdit.Enabled         := True;
     FNameDBEdit.Enabled         := True;
     MNameDBEdit.Enabled         := True;
     FirstDateDBCB.Enabled       := True;
     Addr1DBEdit.Enabled         := True;
     Addr2DBEdit.Enabled         := True;
     CityDBEdit.Enabled          := True;
     StateDBCB.Enabled           := True;
     ZipDBEdit.Enabled           := True;
     EmailDBEdit.Enabled         := True;
     LadyPhoneDBEdit.Enabled     := True;
     AttractDBCB.Enabled         := True;
     IntelectDBCB.Enabled        := True;
     SluttyDBCB.Enabled          := True;
end;
```

Each of he fields has been set to Enabled as *true*, which means the user will be able to access them after this procedure runs. If we set these to *false*, the user will not be able to access them. One saw where I called these procedures from the TDBNavigattor click, but we also need to call them at form activate. We can define this event, by clicking on the main form and going to the ObjectInspector window and click on the “Events” tab. We should then see an “OnActivate” item, then just double-click in the right-hand column of the OnClick item, then just add these procedures in the new one provided.

Another option, instead of enabling and disabling each field, is that there is a Visible Object for each field, which also is a Boolean type.

The point worth considering here is, with an hour or two, I can take a basic idea and turn it into a simple GUI application. Given a few more hours, I can buff and fluff it into something nice. These applications are lean enough that we can run them on older i3 desktops or laptops and still get reasonably good performance.

There is one last topic I want to cover on databases, which is setting up a one-to-many relationship. Again this works very similar to Delphi and most used books on Delphi will cover this. Let's say we have two queries set up, one called AquariumQry, and another called FishQry. Each query has the following columns in it:

Name	AquariumQry
Column1	AqrIdx VarChar(20)
Column2	AqrName VarChar(40)

Name	FishQry
Column1	AqrIdx VarChar(20)
Column2	FishIdx VarChar(20)
Column3	FishName VarChar(40)
Column N	Additional data on fish ...

Now go to your Database form and highlight your FishQry. Go to the Object Inspector and look for the field “MasterSource”, which is a drop-down box, and choose the DataSource for the AquariumQry. Now select the MasterField and click in there, and one will get all of the columns from AquariumQry; choose the field AqrIdx. Finally go to Linked Fields and click on it, one should see all the columns from FishQry; select AqrIdx.

Now if we have a data-entry form with a DbNavigator which is pointed to the DataSource for AquariumQry, let's assume also that there are DbEdit set of its data entry fields, and below that is another DbNavigator which is pointed to the DataSource for FishQry and below its data entry fields. If we try clicking through all of the fish, it will only show the fish that are linked to the row that is selected by the Aquarium DbNavigator.

Move the first DbNavigator to another aquarium, the second DbNavigator will select the first fish in that aquarium. Inserting a new fish automatically will add a new row with FishQry.AqrIdx set to the current AquariumQry.AqrIdx row's value.

5. Graphing

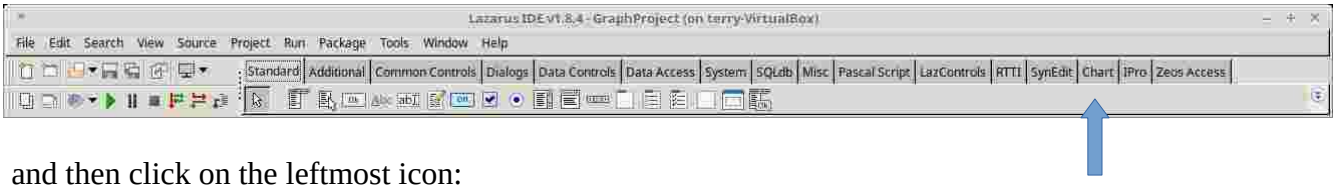
Star Spectral Types

Spectral type used to be a lot easier to remember than it is today, we remembered them by the mnemonic, “Oh be a fine Girl kiss me.” Yeah, it's a bit sexist, but I did not invent that mnemonic and it is in all the astronomy books, some people now replace the Girl with Guy. Type “O” stars were the hottest and type “M” were the coolest. If you are curious our Sun is of a G Spectral class and has a middling star temperature. There are more letters now and each one has many subdivisions; changing the select to remove the substring and grouping on that produces almost 3,000 separate spectral classes.

None of these is very hard to write. I created a new form for each graph type I was using. The data I am using for this demonstration is “nearby stars” that I downloaded and put in a MySQL table. I then selected the data based on star spectral type using the following SQL statement:

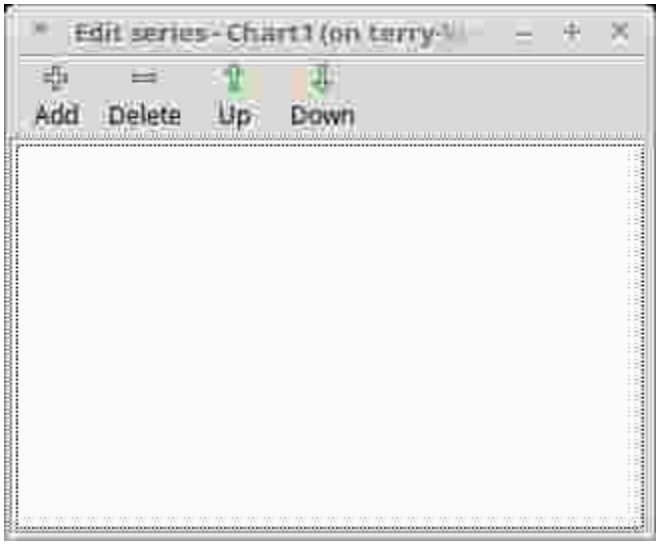
```
select substring(Spec,1,1) as s, count(*) as c from NearStars group
by substring(Spec,1,1) order by c desc;
```

First I just placed a graph object onto my form. Just click on the Chart Tab of the Lazarus IDE



and then click on the leftmost icon:

Now just clicking on the form should place the graph object there. This is resized to one's preference. Now if we have our chart object selected, we can then go to the object inspector window and search for the item “Series”. Clicking in the rectangle just to the right of that, should bring up a “...” button, which is clicked.



Clicking on “Add” will give us a whole bunch of Series types that can be added. The only ones I am going to use for this book are Pie, Line and Bar. One is free to experiment with the others. In this case, I am going to choose “Pie”.

After adding it, it will give us a name like “ChartPieSeries1; we can select it and go to the Object Inspector Window and rename it to something meaningful or leave the name as is. I will rename mine to MyPieSeries. While we are at it, click on Marks->Style and change SmsNone to SmsLabel.

At this point, I would put a standard button on the bottom of the form and rename it to CalcBut, which will create an OnClick event for it. This is what that finished procedure looks like:

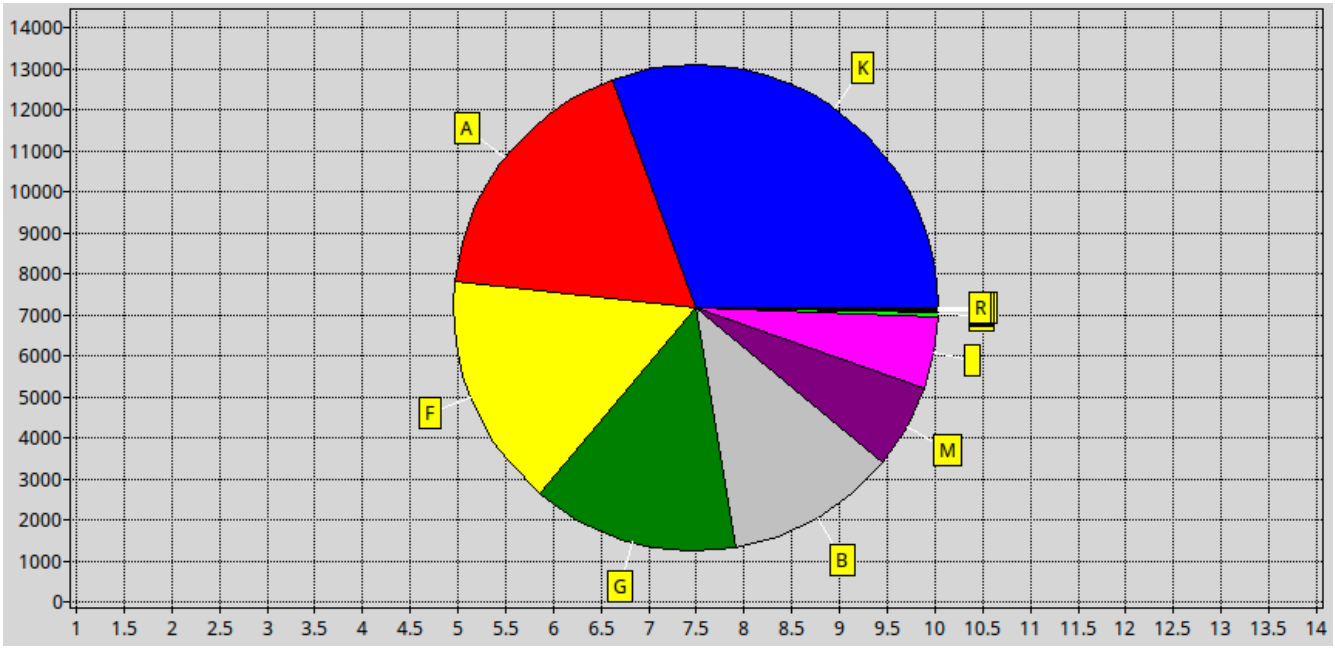
```
procedure TPieForm.CalcButClick(Sender: TObject);
Const
  ColorArray: Array[1..10] of Integer = (ClBlue, ClRed, clYellow,
     ClGreen, clSilver, clPurple, clFuchsia, clLime, clNavy,
     clOlive);
Var
  DVal:       Double;
  iVal:       LongInt;
  s:          String;
  i, Cnt:     Integer;
  ColorCnt:   Integer;
begin
     If MyPieSeries.Count > 0 Then
        For i := MyPieSeries.Count Downto 1 Do
            MyPieSeries.Delete(i-1);
     DbForm.MyStarsDB.Connected := True;
     DbForm.MyStarsQry.Active := True;
     DbForm.MyStarsQry.First;
     ColorCnt := 1;

     If DbForm.MyStarsQry.RecordCount > 0 Then
        Repeat
          iVal := DbForm.MyStarsQry.FieldByName('c').AsInteger;
```

```
            DVal := iVal;
            s := DbForm.MyStarsQry.FieldByName('s').AsString;
            MyPieSeries.Add(DVal,s,ColorArray[ColorCnt Mod 10]);
            DbForm.MyStarsQry.Next;
            ColorCnt := ColorCnt + 1;
          until DbForm.MyStarsQry.EOF;
end;
```

Guess what--a pie chart looks crappy if all the slices are the same color. Therefore the constant array is used to set the colors for each slice; see source code in red. Also, if we want an integer to start over after it goes through an array, divide the index with Modulus by the count in the array. Modulus gives remainder, not quotient.

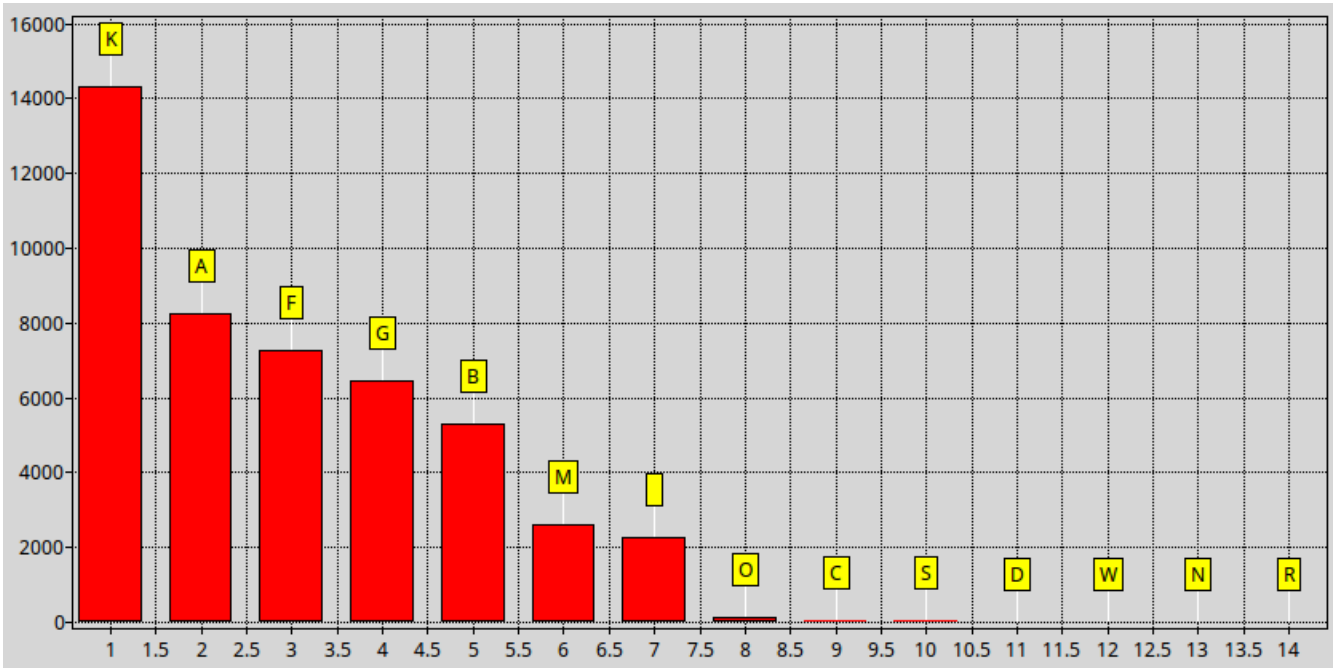
The MyPieSeries.Add function requires the data element to be a double float, but the count I got from my select was an integer. I thus had to convert my integer to a double float; see blue source code. Also, don't use the Clear (MyPieSeries.Clear); it will remove style settings. Instead I executed a **for** loop to delete any previously added pie slices, see green source code. This source code will generate a pie chart like the following:



The pink wedge's label actually did not have any letter associated with it. Wedge labels do not have to be limited to one character however; it is just that this example worked out that way. LineCharts and BarCharts works almost identical. The only difference is that they look pretty good all in one color, so we won't need the ColorArray. Here is my BarChart example and the resulting graph:

```
Procedure TBarChartForm.CalcButClick(Sender: TObject);
Var
  DVal:   Double;
  iVal:   LongInt;
  s:      String;
  i, Cnt: Integer;
begin
     If MyBarSeries.Count > 0 Then
        For i := MyBarSeries.Count Downto 1 Do
            MyBarSeries.Delete(i-1);
     DbForm.MyStarsDB.Connected := True;
     DbForm.MyStarsQry.Active := True;
     DbForm.MyStarsQry.First;
     If DbForm.MyStarsQry.RecordCount > 0 Then
        Repeat
          iVal := DbForm.MyStarsQry.FieldByName('c').AsInteger;
          DVal := iVal;
          s := DbForm.MyStarsQry.FieldByName('s').AsString;
          MyBarSeries.Add(DVal,s,clred);
          DbForm.MyStarsQry.Next;
        until DbForm.MyStarsQry.EOF;
end;
```

As can be seen, the source code here is even simpler to look at than the Pie Chart. These charts build fast, basically in the blink of an eye. We also can alter the Chart Bottom or Left axis to pretty this up and/or give it a title.



One last point. It is very easy to capture these graphs and put them on the clipboard. I just dropped Button on form and added a one line to the provided procedure:

```
procedure TBarChartForm.CopyButClick(Sender: TObject);
begin
     MyChart.CopyToClipboardBitmap;
end;
```

MyChart is the name that the TChart is defined as under your TForm name. Charts aren't the only objects that can be copied onto the clipboard; many other objects can too. I often copy TStringGrids onto the clipboard which makes it easy to drop the data into a spreadsheet.

6. Capture that data

Linux is endowed with many command-line utilities. It's possible to get stock quotes, weather, system monitoring, and things I probably haven't heard of. Occasionally we may want to display the output of these in a GUI application or capture the information to put into a database.

Free Pascal has the ability to run other programs in the background and to then capture the data and then we can do anything with it we want to. It's not that difficult; the following web site documents the process:

http://wiki.freepascal.org/Executing_External_Programs#The_Simplest_Example

Basically, one adds a process to the uses clause, then we declares a variable of type Tprocess

```
MyProcess: TProcess;
```

Then the following can be added to the logic:

```
MyProcess := TProcess.Create(nil);
MyProcess.Executable := 'SomeProgramName';   // This can be a CLI program or a script

// Now add your parameters
MyProcess..Parameters.Add('Some Parameter');

// Tell it to wait for exit
MyProcess.Options := MyProcess.Options + [poWaitOnExit];

// And Finally tell it to execute
MyProcess.Execute;
```

Now if we want to capture the output and it isn't that big, we would need to declare a TStringList then modify your MyProcess.Options to:

```
MyProcess.Options := MyProcess.Options + [poWaitOnExit, poUsePipes];
```

Then after your process exits:

```
MyStringList := TStringList.Create;
MyStringList.LoadFromStream(MyProcess.Output);
```

Here is an example of a fully fuctional program that executes an external process. The concept here is pretty simple and kind of silly, it just runs the "ls" command prints the output and exits.

```
Program MyLs;

Uses Classes, Sysutils, Process;

Var

     MyProcess:              TProcess;

     MyStringList:           TStringList;

     i:                      Integer;

Begin

     // Create our Stringlist

     MyStringList := TStringList.Create;

     // Create the tprocess object

     MyProcess  := TProcess.Create(Nil);

     // Populate the tprocess object

     MyProcess.Executable := '/bin/ls';

     MyProcess.Parameters.Add('-la');

     If ParamCount = 1 Then

          MyProcess.Parameters.Add(ParamStr(1));

     MyProcess.Options := MyProcess.Options + [poWaitonExit,
          poUsePipes];

     // Run our sub process and grab the output

     MyProcess.Execute;

     MyStringList.LoadFromStream(MyProcess.Output);

     // Display the output

     For i := 0 To MyStringList.Count-1 Do

          WriteLn(MyStringList[i]);

     MyStringList.Free;
```

```
    MyProcess.Free;

End.
```

Now we can do anything with your stringlist we want; it will have the captured data that our background process wrote to its StdOut. This is what I have done to download my stock and volume numbers after the markets close. My program runs from crontab (Unix method to schedule batch processes) and it executes a sub-program (a bash script) which then loads the closing values into a database, which I then can view at my convenience. Yes, batch processing is still useful!

It is possible to push this technology further. Instead of declaring just one variable of tprocess, we can declare an array of tprocess and remove the poWaitOnExit. We then loop through the array and start all of the processes, then go through it again as often as needed and checking each process state:

```
    If ProcessTable[i].Process.Running = False Then
```

If this is true, we then just capture the data and start an additional process. With this it is possible to keep ten, fifty or even 100 processes running at once. Why would we ever want to do this? I can think of several scenarios where this could be useful. Let's just talk about just one, the slow MySQL insert. I found that if I can break down the inserts into batches and run multiple batches as processes against each other there is very little penalty. Each process will be able to insert about 150 rows per second, and with ten running, I can push the insert rate up to 1000 per second. I found the safe limit on my i5 was about 75 processes running together and at that level I was getting about 3000 inserts per second.

7. Date Math

I find this occasionally comes up, but periodically we need to calculate new dates, and guess what, Free Pascal can do math on dates. We use the TDateTime. I have used it once when I needed to create labels for computer tapes with current date and the return date. I used the standard routines that Free Pascal provides.

To load values into a TDateTime, we have the EncodeDate Function and it is used something like this:

```
Var
     MyDate:   TDateTime;
     Month, Day, Yr, Hr, Min, Sec, MSec:    Word;
...
Yr := 2018; Month := 5; Day := 21;
Hr := 5; Min := 10; Sec := 0; MSec := 0
MyDate := EncodeDateTime(Yr, Mo, Day, Hr, Min, Sec, MSec);
```

The parameters to EncodeDate have to be of the type **Word**. A **Word** is an Integer that can't have negative values; The opposite of EncodeDate is the DecodeDate which is a procedure. Year, month, and day variables also have to be **word**s and it is used something like the following:

DecodeDateTime(MyDate, Yr, Month, Day, Hr, Min, Sec, MSec);

Now to add time to TDateTime, we can use one of the following functions:

IncYear

IncMonth

IncWeek

IncDay

They each have two fields, the first being a TDateTime, and the second field being the number of time units to add as an integer. Each function returns a TDateTime, so IncDay would be used something like:

MyDate := IncDay(MyDate, 5);

which would add 5 days to MyDate; to subtract we can:

MyDate := IncDay(MyDate, -5);

The other functions work similarly.

I also found it was occasionally necessary to know what day of the week a future date would land on; Free Pascal makes this easy with the DayOfWeek Function.

MyWord := DayOfWeek(TDateTime);

It will return 1 for a Sunday, 2 for a Monday, up to 7 for a Saturday.

8. Math

Pascal uses “*” for multiplication, “+” for addition, and “-” for subtraction like most languages do today. Unlike most other languages, it uses a different identifier for real division *versus* “integer” division. To divide real numbers, we use the “/” like most every other language does, but for integer values, the reserved word **div** must be used. To get a remainder from an integer divide, we must use the reserved word **mod** for modulus. If one is not a computer or math person, it is possible that not to know the difference between reals and integers. Reals have a fractional component to them, while integers are whole numbers only.

Turbo Pascal was a long time ago. Back then I was an underemployed guy who had an Associates Degree from a local community college with a number of COBOL classes under my belt. I bought a cheap personal computer and the only compiler I could afford was Turbo Pascal. It was an eye-opening experience and totally different from COBOL. COBOL was wordy, while Pascal was elegant and still is.

To raise a variable to a power in COBOL, they used a double asterisk “**” and Python still does this, but Pascal never had a character that raised a variable to a power. I had a friend with a math degree who had taken a little Pascal in college, and I had to ask him how I should do it.

He explained in Pascal, one has to take the natural log of the variable (the Ln function), multiply the returned value by the power we are raising it to and then take the exponent (exp function) of that value. So to raise a real value to *pi (π)*, the following code would work:

MyReal := Exp(Ln(MyReal) * 3.141592);

This was one of Dr. Nicklaus Wirth's simplifications of Algol to make the compiler smaller and more efficient. I believe Algol used the “^” character to signify raising to a power. Try using the Ln and Exp functions; they still work. Today we also have the Power function, and it would be used like this:

MyReal := Power(MyReal, 3.141592);

Or to simplify it and make it even more accurate, we have the **pi** function, which returns *pi* to the highest level of precision for a variable type.

MyReal := Power(MyReal, Pi());

What are Logarithms anyway, and are they magic?

Logarithms (often called just logs) are basically exponents. Exponents have some very interesting properties. If we add two exponents of the same base number, the result is the same as multiplying the two original numbers. Example:

8 * 16 = 128

But $8 = 2^3$ and $16 = 2^4$, so we can represent it as $2^{(3+4)}$ which = 2^7 which also equals 128.

Also, if you take that 2^3 and multiply the exponent by 2, which is 2^6, that equals 64 which is the square of 8. Take that 2^3, multiply the exponent by 3, and it becomes 2^9, which is equal to 512, and that is the cube of 8.

Therefore, multiplying an exponent is the same as raising that base number to that power, and that multiplier does not even have to be an integer. Also, to get that number's *nth* root, you just divide the exponent by *n*.

So around about 1600, a real nut job by the name of John Napier (there is a Wiki page about him) got the brilliant idea that any real positive number can be represented as an exponent of a specific base number. The number *e* (2.718281828...) is a common base number to use and that is what a natural logarithm (Ln) is based on.

Fortunately for us, the algorithm to calculate what a logarithm is for a specific number is built into the math co-processor of most of our CPUs today.

One must be careful because the Power function is just a simple wrapper around Ln and Exp functions; it doesn't do any error checking. If MyReal has a negative value, all of these examples above would crash with a runtime error of a negative argument to a function whose domain is only positive ones. If one wants the truth, I still prefer using the Ln and Exp method. It is the way I learned to do it and it corresponds to how we did it back in algebra class. Back then we had a logarithm table and used this to raise values to powers, take root values, and similar operations. This was before computers made it into the classroom.

One may think this is ridiculous; why would one ever do it that way? Well to be honest, Pascal is not the only language to do it this way; c and their descendant languages do something very similar. Back when these compilers were originally designed, computers were much slower, resources were expensive and this compiled more rapidly.

Back when I had this Turbo Pascal compiler and I wanted to to write my own amortization program, unfortunately I wanted it to be accurate and the reals of that version of Turbo Pascal were not accurate to the penny. Back then there was only the single real type, there were no double or extended types (it was a nonstandard real of 48 bits and did not use a math co-processor). I knew what the formula was, but... There was another version of Turbo Pascal that came with my compiler, Turbo Pascal BCD which would be accurate to the Penney, but unfortunately it didn't have any math functions and couldn't talk to the other compiler, and by NO math functions I mean **no LN** or **EXP** functions. Well, my mother

had an old book on the history of mathematics that gave me a clue on how to write my own Ln function; and it worked and I learned something. My math friend told me how I could write my own Exp and Sqrt functions. Guess what——I still have that book and the algorithm.

The Turbo Pascal I originally used only had just a few built in trig functions, there was no standard math library. Today Free Pascal has an extensive math library, including one of my favorites the StdDev function. For now back to the trig functions.

My old TI calculator would calculate all of the trig functions and return the value in degrees. Most computer languages I have seen return the value in radians, and Pascal is no different. Most humans are used to degrees; we can wrap our heads around degrees, not radians. Back then to convert the output into degrees, one had to know to multiply 180/Pi by the value in radians. Today, Pascal has the RadToDeg and DegToRad functions. Pascal isn't the only language that does it this way; in fact if one tries to use a trig function in a spreadsheet it also will output the value in radians and then one would have to convert them to degrees. Just a word to the wise.

I took a statistics class in college and loved it. We spent a lot of time working with standard deviations. Standard deviation is a formula used to gauge variance in a random or psudorandom dataset, or more simply put: how tightly packed the data is around the average. It is used heavily in both science and business. Everyone knows how to calculate the mean (or average), but the mean doesn't tell us a whole story; knowing the variance in the data tells us a great deal more, and combined with the mean, it can tell us nearly the whole story. Basically it tells us how tightly bunched our data is. I will issue this warning: not all datasets follow what is known as a normal distribution. If they don't, don't use the Standard Deviation functions.

All data that falls within one Standard Deviation of mean (the statistical word for average), should cover 68% of your data, 2 deviations cover 95% and 3 deviations should cover 99.97%. So if we need to know what the Standard Deviation for a dataset is, we can load it into an array and use Free Pascal's standard routines:

* StdDev

* MeanAndStdDev

But assuming the data is already in your database, we can also select it in a MySQL Query:

```
select Avg(MyRealColumn), Std(MyRealColumn) from MyTable;
```

To know the actual percent of a specific range, we have to do some calculations. So here is my scenario:

We have melons for sale; the mean weight of our melons are 9.5 lbs with Standard Deviation of 3.3 lbs and we know that melons between 6 and 12 lbs sell the best. No, I don't know this is a fact, I am making these numbers up as I go. We have to calculate the *z* value for both of those points

```
z point = (Point - Average)/Standard Deviation
```

So, we get (6-9.5)/3.3 and (12-9.5)/3.3 which equals -1.06 and .76 respectively; now we just run that

range through the following function and we get 63%. That wasn't so bad. It should be fairly easy to put ranges in that are useful to one in one's own programs. By the way, this is the main method teachers used to use to grade on the curve! In that system, the teacher would have taken the mean and standard deviation of the scores of a test.

Grade	z Values	Percent of class
Grade of A	>= 2.0	2%
Grade of B	>= 1.0 and < 2.0	14%
Grade of C	>= -1.0 and < 1.0	68%
Grade of D	>= -2.0 and < -1.0	14%
Grade of F	< -2.0	2%

About the functions below, I got the formula for the line of the bell curve from my professor; it's also on Wiki. We next need to determine the area under the curve. I used the hand method to calculate the area. The hand method is not that efficient, but today's computers are really fast, so it's not a problem.

The hand method tries to break the area down under the line into little rectangles and add up all the rectangle's areas to get an approximation of the total area. If the rectangles are skinny enough, we will get very good results, I was getting three digits of accuracy. Using the hand method, there is a small rounded area above each rectangle, so to get a better number I calculated that as a triangle. A few algorithms I have checked agree to four digits, most do not;, they have a smaller number. If anything I am still underestimating that area, since there are some rounded areas I am still not getting. Note, for non-math geeks: The function will give a result somewhere between 0 and 1; we will need to multiply that number by 100 to get it in percent ranges.

```
Function CalculateHeight(sv, lv, x: Double): Double;

Begin

   CalculateHeight := sv * Exp(lv * 0.5 * x * x);

end;

Function BellCurveArea(BeginVal, EndVal: Double): Double;

{

         This function calculates the area under

         the Bell Curve using the hand method.
```

```
         The line for the curve is equal to:


         (1/(2 * Pi)^.5) * e^(1/x^2)
}
Var
   x, y1, y2, YVal, Sum, SqrtVal, LogVal: Double;
Begin
   SqrtVal := 1.0 / Sqrt(2.0 * Pi());
   LogVal  := Ln(1.0 / Exp(1));
   Sum := 0; x := BeginVal;
   Repeat
      y1 := CalculateHeight(SqrtVal, LogVal, x);
      y2 := CalculateHeight(SqrtVal, LogVal, x + 0.01);
      If (y1>=y2) Then
         YVal := y2
      Else YVal := y1;
      Sum := Sum + YVal * 0.01;
      // This adds the triangle .5(h * w)
      // Width = .01 and multiply that by .5 = .005
      Sum := Sum + Abs(y2-y1) * 0.005;
      x := x + 0.01;
   Until (x > EndVal);
   BellCurveArea := Sum;
End;
```

9. EMail

It can be really convenient having an email object or a procedure in your applications. It can be used for sending out warning emails, reports and even receiving Ebay orders. In 2008 I did some work as registrar for a not for profit convention, where we accepted orders and signed people up via Paypal. I ended up sucking in a lot of that data via email into a database.

The last time I checked, Delphi had an email object that was really easy to use and was called Indy. There have been several attempts to port Indy over to Lazarus, but it has never made it into the code base to my knowledge. Instead, in Lazarus or Free Pascal, I end up using the Synapse Library, which is not object oriented, but is a totally procedural library. What it looses in ease of use, it gains in in efficiency, however. One can download the library at:

http://www.ararat.cz/synapse/doku.php/download .

To be fully honest it is more than just an email library. It is a full TCP/IP networking library, but the only functionality I have ever used is the email.

OK, I have never had any luck connecting to Gmail using these libraries. I have used one of these cheap LAMP servers like HostGator without much trouble however. There are literally hundreds of these companies. Personally, I think one is better off choosing a larger company like HostGator. I chose a small company first and for a few years they gave me pretty good service, but then it all fell apart and I couldn't even recover my domain name. Companies like HostGator usually let one set up as many email accounts as one would like for one small monthly fee.

Finally a request for responsibility. With these tools it is possible to send out huge volumes of email. Put a million addresses into a MySql table and one can send out a million emails in no time flat. But with **great** power comes **great** responsibility, so please use this for valid legal and acceptable use only. And finally, this would make a terrible spamming system. I have only managed to send out text emails and I do not believe that would have a big impact on the general public. I have not managed to send out any HTML emails using this technology, I am not going to say it isn't possible, because it probably is. But one would have to go to a lot of effort to get it to work.

9a. Installing the Library

The easiest thing to do here, is to download the library at:

http://www.ararat.cz/synapse/doku.php/download

About halfway down the page, one should see a download link for “synapse.zip”. Unzip that into a directory and then copy the following files into any directory of a project that needs email.

- blksock.pas
- smtpsend.pas
- ssfpc.pas
- synacode.pas
- synafpc.pas
- synaip.pas
- synautil.pas
- synsock.pas

9b. Reading Email

The following additional libraries will be required to read emails in one's programs:
synautil, synacode, blcksock, pop3send, smtpsend

If one is going to be working with attachments, these two libraries will also need to be added:
mimemess, mimepart

These libraries were added in our previous step.
If one will wants to define these variables, one can change the variable names to something meaningful:

EMail: TStringList;
pop: TPOP3Send;

Now one would need to set up out TPop3Send data structures. These settings may vary from to server to server and one may have to play around to get them right for one's server. More than likely one's LAMP server will have a guide on setting up one's email client, one can use this to help figure out settings. A TStringList, is basically an array of string with some additional procedures, functions and variables connected with it.

Pop.UserName := '<myuser>';

This might just be one's first name, or one's first name at one's domain. When you see me put a <....>, its value is a variable and can vary from installation to installation.

pop.password := '<MyPassword>';
pop.TargetHost :='<MyHostname>';

Hostname will probably be one's domain.

pop.AutoTLS :=True;

Will either be True or False;

pop.AuthType := Pop3AuthAll;

This can be one of three values, Pop3AuthAll, Pop3AuthLogin, or Pop3AuthAPop. Then something like the following will check to see if you signed on correctly:

```
Ok := pop.login;
If Not Ok Then
Begin
  WriteLn('Email LogIn Failed: ' + TimeToStr(Time));
  WriteLn('UserName: ' + pop.username);
  WriteLn('Password: ' + pop.password);
```

```
  WriteLn('Host: ' + pop.TargetHost);
  If pop.AutoTLS = True Then
        WriteLn('AutoTLS: True');
  If pop.AutoTLS = False Then
        WriteLn('AutoTLS: False');
  If pop.AuthType = Pop3AuthAll Then
        WriteLn('AuthType: Pop3AuthAll');
  If pop.AuthType = Pop3AuthLogin Then
        WriteLn('AuthType: Pop3AuthLogin');
  If pop.AuthType = Pop3AuthAPop Then
        WriteLn('AuthType: Pop3AuthAPop');
  halt;
End Else
Begin
     If CountSw = 'Yes' Then
     Begin
          pop.stat;
     Cnt := pop.statcount;
     WriteLn(Cnt);
End;
```

In the above logic, if successful, the integer Cnt would have the number of emails on one's server. Now one should be able to use the following logic to read and dispose of one's email:

```
For i := 1 To Cnt Do
Begin
     pop.retr(1);
          EMail := pop.FullResult;
     // Do something with the Email...
     pop.dele(1);
End;
```

After each email is successfully read, the TStringlist “Email” would contain the contents of said email. Once one figure out the settings for your server, one would be able to hide the logic in a procedure and use it over and over again in many of one's apps.

9c. Sending Email

Sending email is a lot easier then reading it. To send email only one additional library is required:

SmtpSend

And only one procedure is required:

SendToEx(FromAddress, ToAddress, Subject, Server, TStringList, user, password) ;

Before running the procedure, one would need to load the email content into the TStringList. One can do something simple like:

Email.LoadFromFile('MyFile.txt');

There isn't really anything more to sending email than this.

10. Summary

For the average person who needs to do some programming, there are many options. The two most popular choices are not really obtainable for the majority of people.

c++ is very technical and cryptic, but gives the best performance. Also creating a GUI type application in c++ is not really feasible by a nonprofessional programmer. Python is relatively easy to code, but performs very poorly. Early on after one's application is first completed, it will seem to run okay. Data grows over time, however, and Python's run-times can increase exponentially as one produces more data.

Free Pascal is fairly easy to code, on a par with Python, but gives performance closer to c++. c++ programming is nearly a full-time job and can require a small library of reference material, while the Pascal programmer can get by with just a few books.

Yes, the original Pascal compiler was primitive. When Turbo Pascal came out in the 1980s, it was a huge improvement, with an integrated development environment and some nice built-in functions and procedures. It was also still extremely limiting by today's standards. The language has become more useful over time, and with libraries and objects it has become very powerful.

In summary this is not your parents' Turbo Pascal. Yes, the core language is still there and if you still had their Turbo Pascal programs from the mid to late 1980s, with some very minor modifications they could still be compiled and one could run them on your current computer. There is a lot more here now: Free Pascal is object oriented, has the ability to make GUI applications, and to access and modify SQL databases. By combining old ideas with newer ones, the modern Pascal programmer can push the performance and obtain excellent results. Pascal is still fast and can solve almost any problem you want to throw at it.