CONNECTION POOLING IN POSTGRESQL

by

Michael J. Shelton

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

June 2004

The project presented by *Michael J. Shelton* entitled *Connection Pooling in PostgreSQL* is hereby approved.

Amit Jain, Advisor	Date
John Griffin, Committee Member	Date
Jyh-haw Yeh, Committee Member	Date
John R. Pelton, Graduate Dean	Date

dedicated to my wife Cassie – without your sacrifice, this would never have been possible

ACKNOWLEDGEMENTS

The biggest thank you goes to my family. They've probably wondered where their husband and father has been the last seven months (especially these last eight weeks). I love you and hope I can repay what you've given up so that I could finish this.

Thank you to Ben R. Shelton, my dad, for helping me with the analysis. I never would have realized the possibilities of what I've begun with this experiment. It's one thing to try something out, it's a completely different thing to understand the impact of what you've done and to see new opportunities for further research with exciting potential for improvement.

A big thank you to Amit Jain for being a great teacher who inspired me when I took my first Data Structures class at Boise State University. Everything just "clicked" for me and I've been loving computer science and programming ever since. I'd also like to thank Amit for helping me achieve something so difficult and encouraging me to keep going. This experience will be invaluable for my next step.

Thanks to Tim Vance for helping me with the PHP scripting, it's nice to have someone pull through in a pinch like that.

I'd also like to thank my neighbors and friends for shunning me so I would focus and not get distracted (oh, and thanks to Cassie for asking them to do so in the first place).

AUTOBIOGRAPHICAL SKETCH

I was born in Boise, ID in 1973. My father joined the US Army in 1975 and as a result I had the opportunity of living in Washington, Georgia, Oklahoma, Germany, Colorado, Oklahoma, and Korea.

Computers were always a large part of my life. In elementary school in Colorado I helped the teachers with the Commodore PET and would take it from class to class showing all the students how to use it (teachers included).

I graduated from Lawton High School in Oklahoma and moved to Boise, Idaho to attend Boise State University (because they had blue Astroturf). After starting my degree in Computer Engineering in the University of Idaho Boise Coop program I left between my freshman and sophomore year to serve a two year mission for the Church of Jesus Christ of Later-day Saints in Manchester, England.

When I returned and was in my second semester of my sophomore year I took Circuits and Data Structures at the same time. I disliked Circuits and loved Data Structures and so I switched my degree from Computer Engineering to Computer Science and I've never looked back.

I graduated in 1997 and started my Master's program at the University of Idaho (distance Program) in Computer Science in Spring of 1998. After joining HP I switched to California State, Chico. I left HP for a two year Internet startup venture during the Internet boom years, but got the pink slip like almost everyone else and headed back to HP. After returning to HP I decided to finish my Master's at Boise State.

I married Cassie in 1994 and now have three beautiful girls, Sadie (6), Abigail (4), and Clara (8mos).

ABSTRACT

The purpose of this project was to implement a to-do item for the open source database project, PostgreSQL. The item chosen was adding connection pooling on the backend of the database. The experiment performed not only included implementing the feature to show that it could be done, but also testing and analysis to show whether it was a help or hindrance to the database's performance.

Most databases like DB2, Oracle, MySQL, MS SQL Sever, and PostgreSQL make use of persistent or pooled connections (e.g. a web server using PHP's persistent connections, ADO's pooled connections, or DB2's connection pooling agents). These are not implemented in the database itself though, rather they reside in a layer between the client and the actual database.

Database providers claim that creating connections is resource intensive and can have a higher cost in comparison to simple queries like those performed by merchant websites. The results in both the original PostgreSQL database server and the modified "pooled" PostgreSQL database server show that this is not the case. The analysis of the test data showed that the connection times were 4 to 500 times faster than the query times.

Since the connection times were not as expensive as first thought, the improvements made through pooling when measured in terms of connection initialization are not as significant as were initially hoped. However, a pleasant side effect occurred.

In the “pooled” version of PostgreSQL there was an improvement in total connection time. The results showed a 200-300% speedup overall. This increase in efficiency came during the query time rather than the initialization time.

The results of this experiment show that there is value added with the implementation of connection pooling in the PostgreSQL database. Through additional research and testing it is hoped that this feature will be accepted by the core developers for inclusion into the main development trunk of the PostgreSQL source tree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 What is PostgreSQL?

PostgreSQL : The most advanced Open Source database system in the world.

PostgreSQL is an enhancement of the POSTGRES database management system (and is still sometimes referred to as simply “Postgres”), a next-generation DBMS research prototype. While PostgreSQL retains the powerful data model and rich data types of POSTGRES, it replaces the PostQuel query language with an extended subset of SQL. PostgreSQL is free and the complete source is available[1].

1.1.1 A Short History of the PostgreSQL Global Development Group

PostgreSQL development is performed by a team of developers who all subscribe to the PostgreSQL development mailing list. The current coordinator is Marc G. Fournier. This team is now responsible for all development of PostgreSQL. It is a community project and is not controlled by any company.

The authors of PostgreSQL 1.01 were Andrew Yu and Jolly Chen. Many others have contributed to the porting, testing, debugging, and enhancement of the code.

The original Postgres code, from which PostgreSQL is derived, was the effort of many graduate students, undergraduate students, and staff programmers working under the direction of Professor Michael Stonebraker at the University of California, Berkeley.

The original name of the software at Berkeley was Postgres. When SQL functionality was added in 1995, its name was changed to Postgres95. The name was changed at the end of 1996 to PostgreSQL[2].

1.1.2 How Do You Say PostgreSQL

PostgreSQL is pronounced Post-Gres-Q-L. An audio file is available at `http://www.postgresql.org/postgresql.mp3` for those who would like to hear the pronunciation.

1.2 Frontend Connection Pooling

Frontend connection pooling is the method currently used by most database systems. As shown in Figure 1.1 the client interfaces with the connection pool rather than the database for opening, using, and closing connections. Initially, when a new connection is made by the client, the pool opens the connection to the database and passes any queries from the client through this link. When the client closes its connection it is actually closing the connection with the pool rather than the database. The real connection with the database is kept open and held by the pool until the next client opens another connection (see Figure 1.2). As long as a valid connection is available in the pool, any future client transactions will be handled by the existing connections rather than new ones.

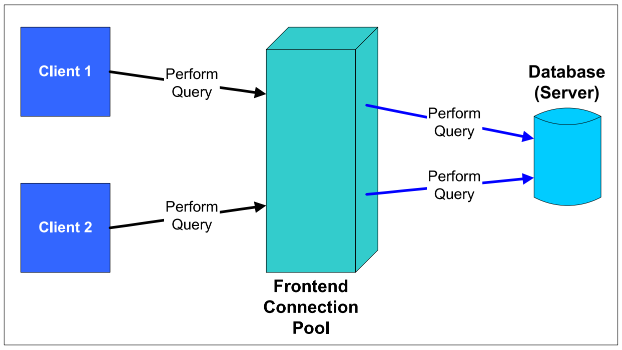

Figure 1.1. Performing Queries with Frontend Connection Pooling.

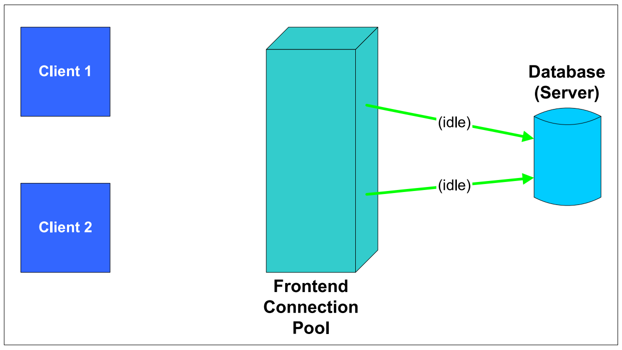

Figure 1.2. Closing Connections with Frontend Connection Pooling.

1.3 Backend Connection Pooling

Backend connection pooling is different from frontend pooling in a number of ways. The most obvious is that the pooling functionality resides inside the database itself versus being implemented by a middle layer application, client application, or database driver. Figure 1.3 shows the process that occurs in PostgreSQL when a new connection is opened. In the standard implementation of PostgreSQL, clients request a connection and the postmaster forks a new backend to handle queries (see Section 3.1.6 for more details). In the pooled version of PostgreSQL a sleeping backend is reactivated by the postmaster instead.

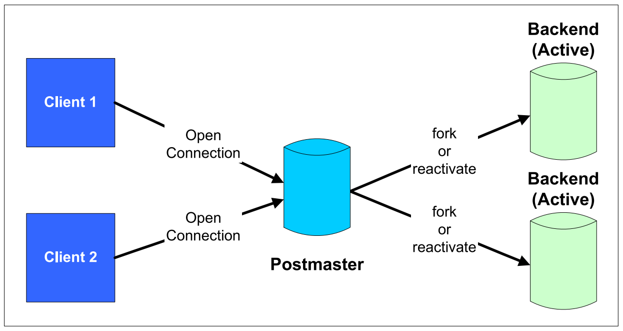

Figure 1.3. Opening Connections with Backend Pooling.

Figure 1.4 shows the communication path between clients and backends when the queries are being performed. This process was not changed for backend pooling.

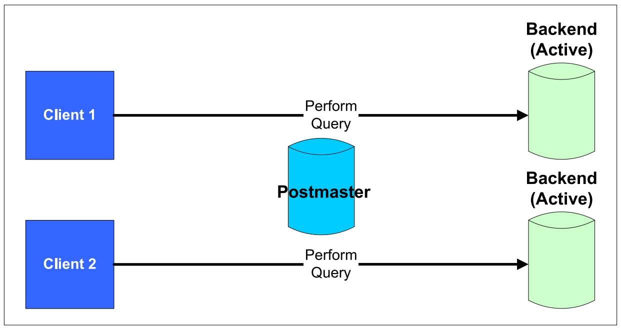

Figure 1.4. Performing Queries on Backends in PostgreSQL.

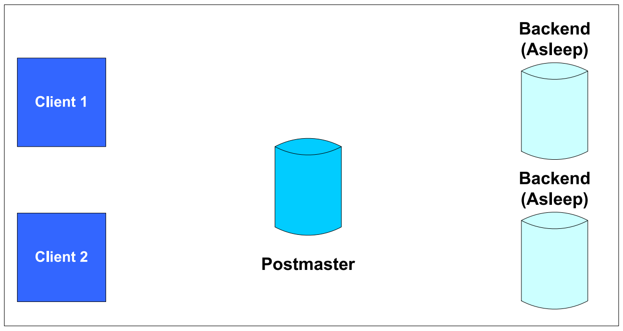

Figure 1.5. Closing Connections with Backend Pooling.

Figure 1.5 represents the state of the database system after the clients have closed their connections with the backends. The backends perform a queisce, notify the postmaster of their new state, and put themselves to sleep awaiting the next wakeup call from the postmaster.

Chapter 2

WHY CONNECTION POOLING IN POSTGRESQL?

2.1 My Path

The following outlines my path to databases and why I eventually chose PostgreSQL connection pooling as my Master's project.

2.1.1 Work History

After graduating from Boise State University in 1997 I worked for HP in embedded firmware on color LaserJet printers. I enjoyed the mathematical challenges of the imaging team I worked on (it had been a while since I last used geometry and it was one of my favorite subjects in school). After a year I was hired on full time with HP and joined the Enterprise Storage group working on RAID disk arrays. The team was switching to C++ for their source code and we were signed up for various classes (e.g. UML, OO, etc.) to aid us in the design of the new backend. I found the challenge of this environment very exciting and looked forward to using these skills on a day-to-day basis.

After two years with this group opportunity knocked and I was invited to lead a team with an Internet startup by a company called Sawtooth Investments. Our main business focus was on-line futures and options trading (think *E*TRADE*[13] for futures traders). Of course on-line commerce means one major component – a database. This was my first exposure to databases and I loved it. We called on an expert in the field of databases, Don Burkes, one of the authors of IBM's DB2. He had also spent some time working for a Japanese bank helping them design and implement a worldwide database transaction system. He helped us come up with the original database design and pointed out pitfalls to avoid and design considerations to be aware of.

2.1.2 Xpit

Once the initial design of the system was complete we began the development of what would soon be called the Xpit trading system. The environment was made up of entirely Microsoft products. MS SQLServer was the backend on a fail-over system using MS Advanced Server and load balanced front ends with IIS on MS Advanced Server also. It was a dream come true to be able to work with so many different technologies. Besides SQL, I learned COM, DCOM, ADO, more C++, and so on. It was great to go to work everyday and learn something new or use something on the edge of technology.

2.1.3 The End of the Dream

Unfortunately, the Internet success was not meant to be (so much for retiring early). In 2001 our venture capital firm went bankrupt and we were sold to RateXchange who laid off half of us and sold the rest to a company called CQG.

I headed back to HP and worked for one of their contracting agencies, Manpower Professional. The job was boring, I was bored, and I missed the excitement of the new technologies, the fancy features, the control of design decisions, etc. To help with the boredom I decided to do some side projects for my current job using a database. However, I knew that it would be difficult to get approval for a setup like I had at Xpit so I decided to look into Open Source and see what was available.

2.1.4 It's Free and Powerful

That's when I found PostgreSQL. It had features that MySQL didn't have at the time (like Transaction processing – a rather important feature). PostgreSQL would be the better choice. I began using it and though the front end management software wasn't as flexible as MS SQLServer, I found the performance and usability of the database itself to be more than a match. The project went well and I was hooked.

2.1.5 Which Feature to Choose?

Anyone can add/fix a feature in PostgreSQL, they just need do the work, test it, and send the source code to the core developers for multi-platform testing. After approval it is usually included in the next major release. To determine which feature to choose I went through the FAQ[2]. It said to subscribe to the developers mailing list for at least 6 months to get an idea of the development cycle, how things are done, and who does what. After a few months I found that I could take a look at the list of to-do items[3] and see if anything caught my eye. I liked two features. The first was transforming the backend from multi-process to multi-threaded. The second was connection pooling in the backend. I looked at the archives for both and asked Tom Lane (one of the main developers[5] of PostgreSQL) which he thought would make a decent Master's Project. He stated that multi-threading would be too large to take on by myself and suggested I look into connection pooling and that I might be able to get it done in two semesters (hah, if I only knew better back then).

2.1.6 Connection Pooling it is then

I spent the next week or so trying to decipher the contents of the thread of emails from the mailing list that related to connection pooling[4]. I would like to point out that at the time I started the investigation of my project the list of emails was half the size it is today (everything up to the subject *Connection Pooling, a year later*). The later emails add more light to the discussion behind pooling and it will be interesting to see what my experiment will add to this discussion.

After reading all I could on the current debate and ideas surrounding connection pooling I approached Amit and got his approval. I submitted my proposal and continued to involve myself in discussions around the development group for additional understanding of what would be required.

Chapter 3

IN THE BEGINNING

Starting this project was nearly as difficult as finishing it. Facing a thousand source files two to three thousand lines long, with little to no documentation, the feelings of inadequacy and self doubt were quite high. There's no simple way to overcome this, nor is there a magic method, but put in the simplest of terms, diving in head first to the deep end begins the sink or swim experience. Fortunately after some initial choking and splashing of water, *swimming* did take place after all.

3.1 Getting Started Version 1.0

The first task to undertake was to determine how the backend worked. That would only come slowly and painfully.

3.1.1 Documentation? What documentation?

In a large software or firmware project (e.g. LaserJets at HP) there are many layers that make up a single system. Each layer needs to be able to talk to the next (e.g. an application layer using a driver in the operating system to talk to an engine). In order to successfully accomplish this large documents are created that have highly detailed specifications. These documents outline command formats, sequence diagrams, state machines, interfaces, etc. They aren't always up to date or completely correct, but they are close. They serve as a base to start with and build on when designing and implementing new features or integrating new hardware.

Unfortunately the exact opposite is true when it comes to PostgreSQL. This was not only surprising it was flat out discouraging.

3.1.2 What There Is and Is Not

An initial search for documentation (with hopes for a nice O'Reilly "In a Nutshell" book) led to a link in the FAQ[2] that described a list of books to read. Unfortunately they were all about SQL and transaction processing. As an example, the famous Gray book[16] is a staple in database classes and programming. However it did not contain information on pooling. Nor did any of the others. Recall that *backend* connection pooling has not been done, only frontend pooling.

What was really needed was a book on developing core code for PostgreSQL. Continued searching prompted the purchase of *PostgreSQL Developer's Handbook*[17], *PostgreSQL Essential Reference*[18], and *PHP and PostgreSQL Advanced Web Programming*[19].

Unfortunately these books were nothing more than guides for the frontend developer (i.e. the person who is writing SQL, creating tables, administering the DB, writing stored procedures, etc), rather than a guide for a backend developer. The reference was useful in setting up tables and the PHP guide was handy for testing when finished with the implementation, but those were the easy parts, it was everything in between those two tasks that required documentation.

3.1.3 What This Means

What this means is that gaining the knowledge necessary to design and implement this feature would require a thorough reading of the source code. There was no other way to get an understanding of the backend.

3.1.4 The Source Code

Most programmer's have had the *pleasure* of reading someone else's code (or in this case, an entire group of developers' code). Within a large code base like PostgreSQL it is kind of like starting a foreign language class only to find out that there is no teacher, only a book.

3.1.5 Size of Code Base

For a better idea of what was being faced Figure 3.1 is a snippet of a command line and the number of files that were the starting point.

```
/home/mshelton/bsu/master/src
-> find . | egrep ‘‘(\.h|\.c)’’ | wc -l
    926
```

Figure 3.1. Number of Source Files.

That is correct, almost a thousand files. And those thousand files represented 405,898 lines of code! What was the next step? Start at `main()` and go from there.

3.1.6 How is PostgreSQL Split up?

Before continuing with the next step, it will be helpful to explain what is meant by the *backend* of the database. PostgreSQL is split up into three main areas. Though this is undoubtedly an oversimplification it will certainly suffice for the purposes of this document.

- *Interfaces*: The interfaces portion of the database deals with the extensible languages (e.g. C, C++, Python, JAVA, TCL, etc) that allow a developer (think client, not core) to write a "stored" procedure. These built-in procedures can then be run over and over again and are usually compiled and thus faster than normal textual SQL statements passed via socket communication from the client to the database[1].

- *Backend*: Here's where the core of PostgreSQL actually exists. It has things like authorization, cache, commands, parser, communication, disk access, regular expression handler, postmaster, etc. The primary areas of my focus ended up being, the postmaster, communication, storage, authorization, and process handling.

[1]There are times where the *frontend* of the database is used interchangeably with the interfaces as well as the clients like PHP, etc. My apologies in advance for any confusion that creates.

- *Contrib*: These are utility type programs that are not compiled into the core PostgreSQL system. They generally are utilities to do things that the database doesn't handle internally yet (e.g. backups, re-indexing, XML, etc). The biggest reason they are here and not in the database is that the code for these tasks aren't portable, or they're written in a language other than C (e.g. C++ is inappropriate for the core code base according to the primary developers).

3.1.7 Using the ddd Debugger with gdb

The problem with reading source files directly and bouncing from file to file tracing each function call is that there is no way of knowing what the data values are or how they are used. After a couple of months the source code was peppered with little `// MJS -- blah, blah, blah` comments everywhere (to help track what had already been read, what was understood, and what was not). But unfortunately things were starting to look a little too much like Figure 3.2.

```
// MJS
//     Should this be reset with each connection
//     or would this be part of the required ‘‘match’’
//     for connections in the pool?
// MJS
//     Good question.  Probably not for at least
//     the first iteration of getting this to work.
// MJS
//     Only do once
av[ac++] = ‘‘postgres’’;
```

Figure 3.2. Typical MJS Comment Set.

As seen in this example, there ended up being three, sometimes four sets of comments for a single line of code. This comment series was trying to determine whether or not this single line of code needed to be repeated in a pooled backend startup or only done once for the original fork. The answer may seem obvious with a single line of code like this, but in reading through all the lines of code nothing was obvious right away. Things weren't obvious the third, fourth, and sometimes tenth time. It just took a lot of time (and other helpful tricks that will be shared later).

It should be painfully clear at this point that just reading the code was not going to lead to a full understanding of the code, nor was it helping to alleviate the fears that this may be too difficult to accomplish single handedly. It was time to try a new route. Being able to see what the data was like during the execution of the code would require the use of a debugger. A bit difficult to get going, but very useful, even invaluable once it was setup and running.

3.1.8 Setup

Using the debugger for viewing the source code, stepping through function calls, setting breakpoints, etc. was similar to most any other C or C++ program debugging experience. Other than manually modifying the makefiles to remove `-O2` for simpler process stepping (it's not removed even though `--enable-debugging` is used during `configure`), there were a couple of tricks used to make things simpler.

- To attach to the postmaster (the main process) start the process in gdb without any command line arguments.
- Common functions to break at in the postmaster are `ServerLoop()` and `BackendStartup()`.
- To attach to a backend (child process) the code was changed to check for a file in the working directory. If it was there the function entered a loop that would sleep, check a debug var, sleep, etc. Once attached to the process simply `finish` the current function until in the sleep loop, reset the var, set breakpoints, and then continue to wherever desired.
- Have the log file print out the child pid after `fork()` for faster attaching.

Once past the initial learning curve due to the newness of the PostgreSQL source code, I found `ddd` and `gdb` to be my best friend over the next 6 months or so.

One of my favorite books[15] has a quote in it that I use a lot, especially when I get really frustrated: *"line upon line, precept upon precept, here a little and there a little."* This was exactly how I would come to know the backend of PostgreSQL almost as well as the back of my own hand (as the saying goes).

And so, each time I would start up the postmaster and fork a backend I would step through the function calls a little bit here and a little bit there. And over the next five or six months I'd put a nice little `// MJS` comment here and one there. In the beginning of course they all pretty much said the same thing – *What's this do?*

or *Why's this called*. But after a few months things began to be more clear and a lot of my question comments were followed by two or more answer comments.

3.2 Getting Started Version 2.0

What? You're not writing code yet?

After well over a semester of stepping through code I was no where near being able to come up with a solution yet. It was about four weeks into the second semester of my project that I asked Amit for another semester to finish (which he kindly replied that most people found two semesters to not be enough and it would be fine). Well, even with another semester to go I was beginning to feel panicked again. Things were progressing too slowly for me to be able to understand the system fast enough to come up with a basic design let alone start writing code. I needed a quicker method of describing the sequence of function calls and the basic flow of a backend query. That's when I turned to my good old friend `printf()`. Sorry `gdb`, no hard feelings but I needed some serious speed and a whole lot of data to go with it.

3.2.1 Log Overload

To handle the large number of files a quick script was utilized that started the postmaster with a unique log file name each time (appending the date and time worked fine). These files were also kept in a single directory for easy browsing as well as comparison, which became useful when actually writing the real code. By the time the project was finished there were 377 log files which represented over 5GB of ASCII data! There definitely was no lack of data.

3.2.2 Function Calls

Another trick to following the sequence of function calls for the postmaster and the child backends was to create a macro that printed the name of the function on entry (which gives a rudimentary listing of the function calls throughout the processes' lives). When more functions were needed (e.g. functions were being called that I hadn't looked at yet and weren't printing to the log) those would be added and the backend would be run again[2].

3.2.3 Variable Values

Another macro was used to print things out via `printf()`, which created the ability to print various variable values throughout the code. At times it was much quicker to add a couple of macros in the source code, recompile and run a backend than to step through, break in multiple places and either display or print the values each time. It made for quicker variations through the frontend and quick comparison of the logs on the backend.

[2]And again. And again. You get the idea.

3.2.4 What to look for?

Remember that the actual purpose of all this work is to provide connection pooling in the backend of the PostgreSQL database. In order to do this, a few things needed to be understood and mastered:

- How the Postmaster Handles a Connection.
- What are the tasks prior to forking?
- Post fork processing by postmaster.
- Post fork processing by backend.
- Where does authentication occur?
- How does the client socket get used?
- When does the backend process commands?
- What happens when the backend exits?

It would be simple to answer these questions now. However, it would belie the difficulty faced in actually gaining the insight and knowledge necessary to be able to answer them in the first place. So rather than give it away here (not to mention a whole other list of things learned along the way that are also interesting) it is important to continue with the chronological path that was taken. This will help you see what was learned as it was learned versus what is known now.

Chapter 4

INITIAL DESIGN

The initial design always centered around the idea that somehow the backends would be put in a waiting state, the postmaster would hand some data to it, wake it up, and let it handle the query from the client as if it had been forked from the postmaster versus reactivated. But what wasn't really understood yet was where this would occur in the process, what data would be handed to the backend, what initialization would need to re-occur, as well as how to communicate between the postmaster and backend.

4.1 My First Diagram

Figure 4.1 was the first design diagram drawn. It isn't the final design nor does it include all the details that went into the first implementation. However it does show the direction taken from the beginning, which is useful for determining where things went from there and how the final design came to be.

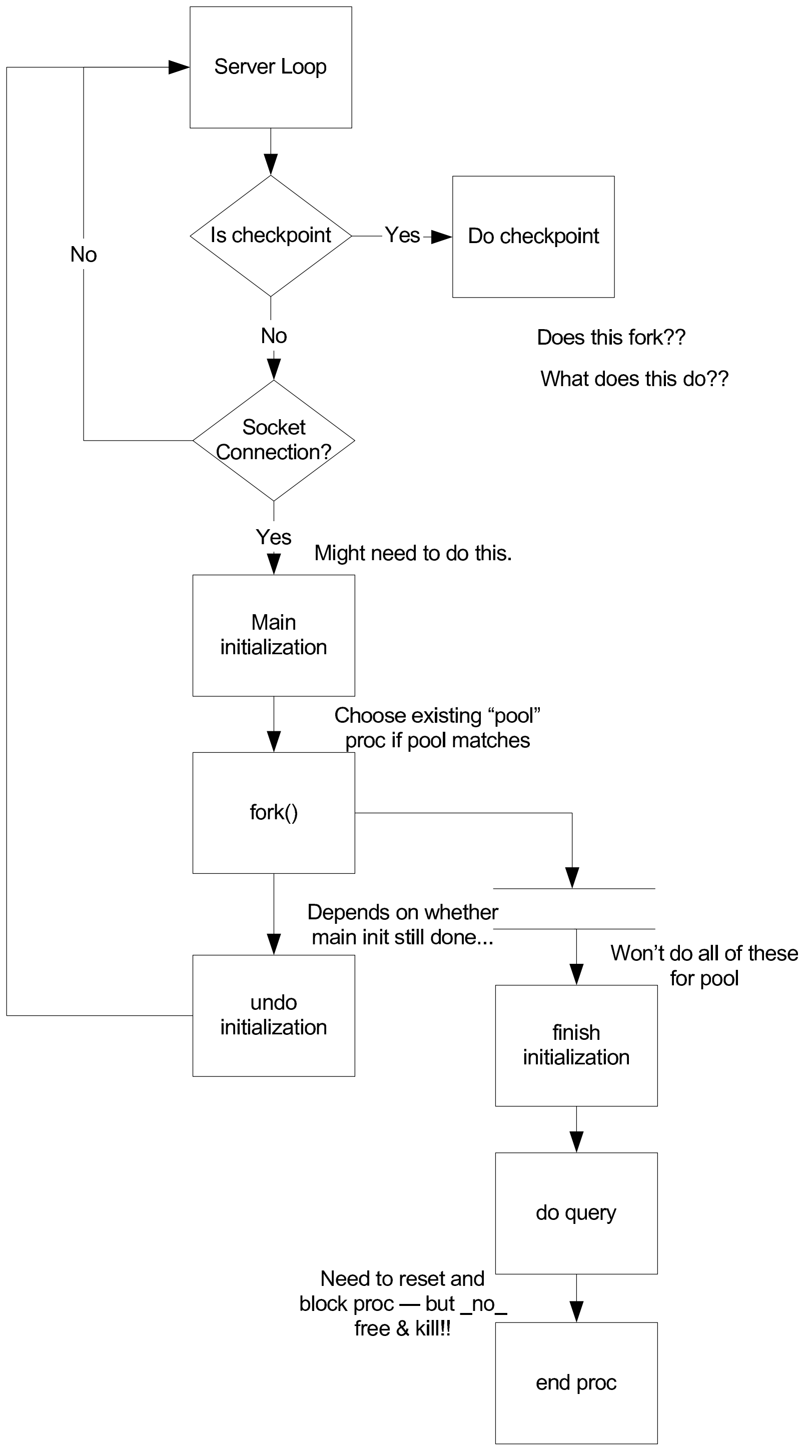

Figure 4.1. Initial Design Diagram.

4.2 Initial Questions

Before writing code there were some initial questions that came from debugging and log reading that needed to be answered before coming up with a basic solution. The major ones were listed in Section 3.2.4. The answers are described in the following sections.

4.2.1 How the Postmaster Handles a Connection

The postmaster goes through some initialization of its own, e.g., MemoryContexts, command line arguments, socket creation, etc. But once that is done it goes into the main loop in the function `ServerLoop()` in `backend/postmaster/postmaster.c`. This is what Figure 4.1 is representing. The postmaster checks to see if a timeout has occurred and if so it forks a checkpoint process for periodic database cleanup and maintenance and then checks the socket to see if a connection is pending. If there is one, it does some minor initialization (sets random seed, salt value, etc) , calls `BackendStartup()` which does more initialization, then forks the backend. After the fork it undoes the previous initialization because it was only for the child via fork but is not needed by the postmaster and must be reset between children. Finally it frees the client connection on the postmaster side (the child will own it now) and continues another iteration of the `ServerLoop()`.

4.2.2 Connection from Client

PostgreSQL (via the postmaster) opens a socket either in the UNIX domain[21] or via TCP anywhere on an intranet or the Internet. Once this socket is available, clients (e.g. PHP, psql, etc) can start to send connection requests to the postmaster. The postmaster will attach to the socket when it is ready and fork a backend to handle communication with the client. This includes a PostgreSQL version compatibility check, authentication, and finally the commands (i.e. SQL queries, stored procedure calls, etc.).

4.2.3 What are the Tasks Prior to Forking?

There are various signal handlers that get set and reset throughout the entire process for both the postmaster and the backend. This section does not contain any detail about those here, though there was a major problem with signal handling and pooling that is discussed later in Section 7.1.

The postmaster selects a random seed prior to forking so that the children do not get the same seed and end up with identical random values with their sibling backends. The postmaster also precomputes password salt values for each connection. The comments in the code mention, “This is annoying to do long in advance of actually knowing whether they’ll be needed, but there’s no choice here due to getting identical salt values in the backends.”

In `BackendStartup()` the postmaster creates a cancel key (for transmitting to the frontend) and then it uses `malloc` to get space for the backend structure that will store data regarding the backend about to be forked. It flushes `stdout` and `stderr` and then forks the backend.

4.2.4 Post Fork Processing by Postmaster

Once the backend has been forked, assuming there were no fatal errors, the backend pid and cancel key are stored in the backend `struct` and added to a doubly-linked list of all the backend processes in use. After that the postmaster returns to the server loop where it closes the socket that was attached and returns to looping.

4.2.5 Post Fork Processing by Backend

The backend (child) calls `DoBackend()` where postmaster specific memory and variable values are freed or reset (including the main postmaster sockets). After that it communicates with the frontend client and requests version information for backwards compatibility (client could be different version than postmaster – not all versions are compatible).

The backend[1] requests user, database name, argument list, and password (if necessary). If approved it calls `PostgresMain()`, which handles the argument list (options from both the postmaster which are considered secure and options from the client

[1]At this point the backend takes care of authentication – see Section 4.2.6.

which are non-secure). The client doesn't necessarily send options to the postmaster, but this is where they'd be checked and processed if necessary.

4.2.6 Where Does Authentication Occur?

The true authentication happens in `ClientAuthentication()`. This is in `DoBackend()` right before `PostgresMain()`.

Initially it was thought that it would be necessary to move all of the authentication into the postmaster prior to the fork or pool reactivation. But it turned out that `ProcessStartupPacket()` communicates with the client first (this performs the backwards compatibility check in Section 4.2.5), and gets the user name and database name very early on. This allows authentication to remain in the backend.

This was an important find during the investigation process. A major concern was determining what would be needed in order to identify a valid pooled backend. Or rather, how would the postmaster know a backend matched a new request from the client.

Before this discovery the initial idea was to do something similar to Microsoft's ADO Connection Pooling in COM+[14]. The ADO pool uses any number of credentials to match a connection, but the three main ones are *user name*, *password*, and *database name*. The first pass at implementing the pool contained these three variables in the credentials struct. However, it was later found that PostgreSQL authentication code mentioned not storing it anywhere for security reasons. There was no reason to doubt the experience and wisdom of the core developers so the *password* was dropped from the credentials in the pool.

4.2.7 How Does the Client Socket Get Used?

The client socket is not opened by the postmaster. It is only used by the backend (at least it was until the pool implementation changed this behavior). It is used for compatibility check, user, database, and authentication (see Section 4.2.6). Once complete and approved by the backend then the socket is used to read commands from the client. These can be SQL commands, stored procedures, exit commands, cancels, etc.

4.2.8 When Does the Backend Process Commands?

As mentioned in the previous section the backend performs startup processing, then authentication then it enters a loop in `PostgresMain()` where it continues to read commands from the client until either EOF (unexpected socket closure), a quit command is received, or an error occurs.

4.2.9 What Happens When the Backend Exits?

There are several ways in which a backend can exit. This was one of the more difficult scenarios to take care of when writing the pooling code. They include:

- Timed out (there are spots in the backend where signal alarms are set for fear of hanging on the client socket for example).

- Failed authentication.
- Client socket closed unexpectedly (for whatever reason).
- Too many backends already (default is 32).
- Not enough reserved backends (default is 2 – these are used for superuser access, it helps guarantee a super user would have access to the database under high load conditions).
- Error occurred (backend would still exit gracefully).
- Fatal error occurred (backend would exit quickly and not well – could have corrupted shared memory).
- Client finished sending commands.
- Backend received a signal (such as an interrupt). This could happen for any number of reasons, but the most common would be from the postmaster trying to kill its children and reset itself or shut down immediately.

When one of these occurs, a call to `proc_exit()` is made and signals are ignored, `shmem_exit()` calls are made and then `proc_exit()` calls are made. When the postmaster starts up as well as when the backend starts up they go through various initialization stages. Some of these stages do things like create memory contexts, initialize shared memory, create sockets, open files, etc. Most of these things are related to system resources. In order to help prevent the wasting of these resources and to decouple multiple areas in the code from the shutdown/exit code an interface for storing callbacks exists in the form of the `on_shmem_exit()` and `on_proc_exit()` functions. Both of these functions basically take a pointer to another function that will do whatever is needed in an exit scenario. Any function can store one or more callbacks. The backend or postmaster calls the `shmem_exit` functionality first and then the `proc_exit` functionality second.

This is all important to the pool because it needs to intercept some of these calls and either wait to call them until the pooled backend really needs to exit, go ahead and call them each time it gets pooled, or do part of what would be done when the proc is pooled[2].

[2]If you're thinking, *"Boy, that sure sounds tough."*, then you've hit the nail on the head – it was tough.

Chapter 5

FINALLY, THE CODE

The heading says it all. There is a sense of euphoria[1] when a developer finally reaches the knowledge base required to implement a feature in an open source project. Having experienced this, it is now clear why so many have flocked to the "movement" of open source, why it is so popular, and why it is successful. There is a high price to pay for entry into this exciting club, but once paid, it is well worth it.

5.1 Master and Project Versions

General discussion of the *Master* version of PostgreSQL refers to the unaltered source code of 7.3.4 as was received after downloading from the main tree via `CVS`. During the testing phase a few macros were added in key places to get timestamps for measurement and comparison purposes.

The *Project* version, also known as the *Pooled* or *Pool* version, is based on the same 7.3.4 version downloaded from `CVS`. However it also contains all of the changes necessary to implement the backend connection pooling for this project. The *Project*

[1]The most memorable experience was when I was poised to implement the last steps for pooling a backend in the postmaster. I was so giddy I couldn't think straight and had to take a 10 minute break to calm down before I could finish.

version also contains similar timing macros as the *Master* to enable the comparisons to be as close as possible.

5.2 Basic Flow

The following items all refer to Figure 5.1.

- The credentials are the *user name* and the *database name* from a client connection. In the *master* version the client socket is attached and a new backend is forked to handle all communication. The first communication is to request the PostgreSQL version (for backwards compatibility) and the credentials. However, as shown in Item 1, the responsibility of requesting the credentials is moved to the postmaster (pre fork/reactivation).

- The credentials are required in order to do a lookup on the pool and see if a valid sleeping match is available (Item 2).

- If the credentials match a valid sleeping backend the postmaster will pass the backend required data via shared memory, pass a wakeup signal to the backend, then send the open file descriptor to the child (Item 3). Once completed the postmaster relinquishes any resources used for this connection and returns to its main listening loop for new client connections.

- Item 4 designates what was thought[2] to be the primary improvement possibility when designing this implementation. Instead of forking a new process each time the postmaster received a client connection request (which is costly by itself) and then running through the entire initialization sequence, the pooled backend would be awakened and run through *some* of the initialization. Thus, hopefully saving time during initialization.

- When a backend (either the *master* or *project* version) is finished, `proc_exit()` is called. In the *project* version (Item 5), a series of conditions[3] is checked to determine whether it should quiesce or exit.

- If quiescent is approved the backend will put itself to sleep (Item 6) and await either a reactivation signal (Item 3) or a kill signal (Item 7) from the postmaster.

- When the pool does not have any matching valid sleeping backends available it will free (Item 8) an existing backend using an algorithm similar to a Least Recently Used (LRU) cache algorithm.

- Either the pool was full and a backend needed to be freed (Item 8) or an empty slot was available. In either case the backend is added to the pool (Item 9) by storing shared memory client data, identification data, and state information.

[2]See Chapter 9 for actual results.

[3]Is backend poolable, is postmaster aborting, is backend handling a signal interrupt, and was client cmd/query successful

After which the new backend is forked (almost similar to the *master* version – but credentials have already been received from the client).

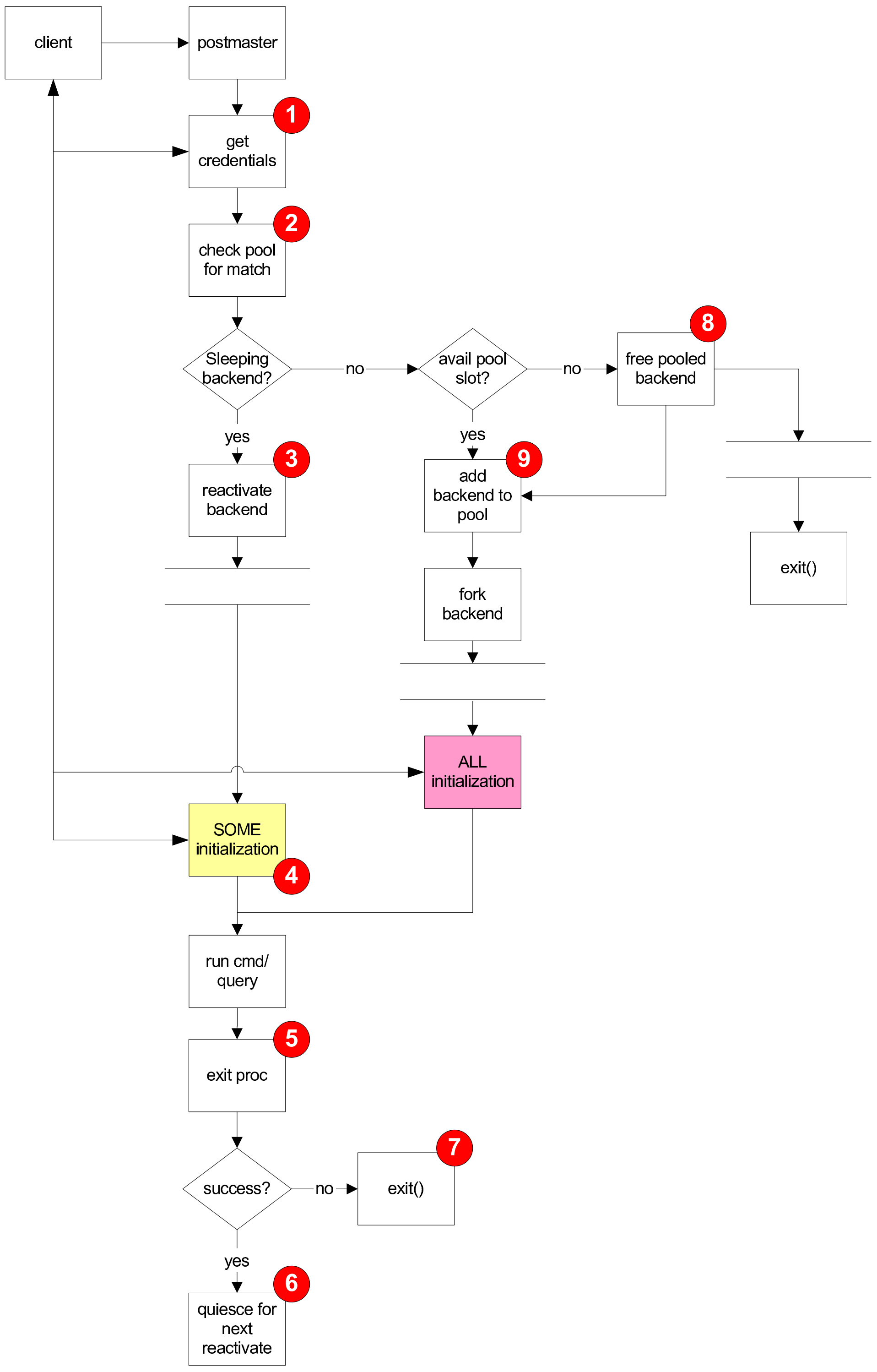

Figure 5.1. Basic Flow.

5.3 Inter-Process Communication

Inter-process communication (IPC) is achieved through signals, shared memory, and in the case of the *project* version, sockets[4]. The pool uses all three of these methods to communicate between the postmaster and the children backends.

During reactivation (see Figure 5.2) the postmaster sets state variables and process information (see Section 5.4), resets the timestamp for the LRU algorithm, then sends the signal to the process to wakeup.

```
// Set state variable to run
// NOTE: No need for spinlock, already asleep and
//      no other process
//      will access this state var in shmem.
p->list[poolid].state = POOL_STATE_RUN;

// Copy the port data needed to shmem, then send
// fd for the client socket
pool_passProcInfo(poolid, port);

// Indicate backend is no longer sleeping
p->list[poolid].sleeping = false;

// NOTE: new "lastused" date (for proper
//      pool replacement)
p->list[poolid].lastused = time(NULL);

// Send wakeup signal to pid
ProcSendSignal(p->list[poolid].procid);
```

Figure 5.2. Reactivation Code Called by Postmaster.

[4]The socket usage is not discussed here, but rather in Section 6.13.

In some cases of IPC between the postmaster and the child, there existed a need for using a semaphore to prevent simultaneous access to shared memory. This was primarily used for protecting the valid and sleeping variables (see Section 6.6 for additional details).

As can be seen in the code example of Figure 5.2, care was taken to prevent over usage of semaphores by shifting write responsibility from child and postmaster to only postmaster in the case of the state variable for example. There were similar cases for the other variables that are currently protected (they could have been only written by the postmaster). However, out of order reading of the variables is still possible, so the semaphore acts as a synchronization technique as well as a protective one.

The shared memory model in PostgreSQL (which was not changed for the *project* version), had wrappers for creation, initialization, access, and removal[5].

5.4 Data Structures

Figure 5.3 is for passing data that is received during startup processing due to the need to gather the credentials for pool matching. This direction of the data is from the postmaster to the backend. The struct is a paired down set of members from the `Port struct` described in Section 6.10. A new struct was created in the pool to restrict the amount of data used in shared memory and because not all of the data

[5]Shared memory is never destroyed by the postmaster. It gets reset whenever the postmaster resets itself, but other than that it is initialized during the postmaster init and used for the rest of the life of the postmaster and backends.

needs to be passed from the postmaster to the backend each reactivation (it does not change due to the nature of the "matching" connection types).

```
typedef struct pool_port {
    SockAddr laddr;
    SockAddr raddr;
    ProtocolVersion proto;
    char options[SM_OPTIONS+1];
    char tty[SM_TTY+1];
    char auth_arg[MAX_AUTH_ARG];
    char md5Salt[POOL_MD5_SALT_SZ];   /* Password salt */
    char cryptSalt[POOL_CRYPT_SALT_SZ]; /* Password salt */
} pool_port;
```

Figure 5.3. Client Communication Related Data [shared memory].

Figure 5.4 is used only by the postmaster, which is why the data is stored on the heap. Each time `pool_addBackend()` is called the current connection credentials are stored by the `poolid` associated with that slot in the pool assigned to the new backend. When the postmaster searches the pool container for a valid sleeping backend it compares the credentials to determine if a match is found or not.

```
typedef struct pool_credential {
    char user[SM_DATABASE_USER+1];
    char dbname[SM_DATABASE+1];
} pool_credential;
```

Figure 5.4. Client Credentials for Backend Matching [heap].

Figure 5.5 represents each backend in the pool or an available slot in the pool for a new backend. Most of the data is self explanatory or the comments provide enough information for understanding. However, it is important to clarify the use of the pid in this node. When a backend is freed (see Section 5.2) the state, sleeping, and valid variables are reset by the postmaster prior to sending the appropriate signal to the child. The subtlety of this is that there are two reasons for the postmaster to do this. First because the postmaster is resetting itself and all sleeping backends must be signaled – this poses no issue to shared memory integrity. The second reason is when the pool is full and no matching backend is found an existing one must be freed. Once the old one is freed (via shared memory changes and a process signal) a new backend immediately takes its place in shared memory when `pool_addBackend()` is called by the postmaster. Then the fork occurs and the shared memory is now changed and owned by the new backend. Occasionally the dying backend does not shutdown fast enough and may want to access shared memory (e.g. to see if it should exit or pool itself) prior to exit. In order to prevent this the pid is immediately invalidated by the postmaster and then reset later by the child. Once reset by the postmaster the dying backend's pid will no longer match and it ignores shared memory and exits (nor would it match the new backend's pid – the operating system handles that).

Figure 5.6 is the main container struct for the pool itself. It contains the node list, credentials, size (for looping), an aborting flag (when postmaster needs to shutdown or reset), the spinlock shared by all of the backends and the postmaster for pool related semaphore actions, and the `BackendId` of the postmaster. If the backend

```
typedef struct pool_node {
    bool sleeping; // If valid, then this indicates active
                   // or not
    bool valid; // Whether a backend belongs to this slot
                // in the pool
    time_t lastused; // Timestamp for LRU algorithm
    BackendId procid; // Used for signal based IPC
    int state; // When reactivated, indicates what should
               // be done
    pool_port portdata; // Subset of Port members for client
                        // communication
    bool reloadPasswds; // Indicates database password file
                        // has changed
    pid_t pid; // Used to prevent multiple backends from
               // changing this node
} pool_node;
```

Figure 5.5. Pool Node [shared memory].

does not signal the postmaster it is shutting down it won't be freed from its master list of backends and the postmaster can't fork new backends (see Section 7.6)[6].

Figure 5.7 is used to store some of the callbacks made by the postmaster and backends for exiting a backend. Not all of the real callbacks to `proc_exit()` and `shmem_exit()` are used by the pool because they are too destructive to the backend and prevent it from being reused. The `shmem_exit` callbacks are made prior to the proc exit ones just as the postmaster does.

[6]This is functionality that exists in the *master* database and was not created for the pool.

```
typedef struct pool_container {
    pool_node *list; // In shmem also
    pool_credential *cred; // postmaster heap
    int size; // maxBackends
    bool aborting;
    slock_t slock;  // For protecting shmem access
                    // between backends
    BackendId postmasterid;  // So child can signal postmaster
                             // done with shmem
} pool_container;
```

Figure 5.6. Pool Container [shared memory].

```
static struct POOL_ONEXIT
{
    void        (*function) ();
    Datum        arg;
} pool_on_proc_exit_list[POOL_MAX_ON_EXITS],
  pool_on_shmem_exit_list[POOL_MAX_ON_EXITS];
```

Figure 5.7. On Exit Callback Lists [heap].

5.5 External Interfaces

Figure 5.8 shows the external interfaces from the new pool code. These functions are used throughout the postmaster and the backends.

5.6 PostgreSQL File Changes

Figure 5.9 is a list of all the files that were changed from the original *master* version of PostgreSQL in order to implement the pooled *project* version. The first three files are new and contain the majority of the pool functionality.

```
/* Postmaster */
extern void pool_init(int size);
extern int PoolShmemSize(int size);
extern int pool_getFreeMatch(char *user, char *dbname);
extern void pool_startBackend(int poolid, Port *port);
extern int pool_addBackend(char *user, char *dbname);
extern void pool_postmasterPostFork(void);
extern void pool_postmasterSignalChildren(void);
extern void pool_notifyBackendsPasswdChange(void);
/* Boolean functions */
extern bool pool_isProcFromPool(void);
extern bool pool_isProcPoolable(void);
extern bool pool_isPoolInProgress(void);
extern bool pool_isAborting(void);
/* Backend */
extern void pool_enterPoolAndSleep(Port *port);
extern void pool_beginQuiesce(void);
extern void pool_on_shmem_exit(void (*function) (), Datum arg);
extern void pool_on_proc_exit(void (*function) (), Datum arg);
extern void pool_childPostFork(void);
extern void pool_baseInitLocalBuffer(void);
extern int pool_numSleepingBackends(void);
extern void pool_ifSleepingHandleInterrupts(void);
extern void pool_setBackendId(int procid);
extern void pool_die(void);
extern void pool_querySuccessful(void);
extern void pool_resetShmemPointer(void);
```

Figure 5.8. External Interfaces Provided by the Backend Pool.

```
src/backend/pool/Makefile
src/backend/pool/pool_backend.h
src/backend/pool/pool_backend.c
src/backend/libpq/pqcomm.c
src/backend/utils/init/postinit.c
src/backend/postmaster/postmaster.c
src/backend/tcop/postgres.c
src/Makefile.global
src/backend/bootstrap/bootstrap.c
src/backend/storage/smgr/smgr.c
src/backend/storage/ipc/ipc.c
src/backend/storage/lmgr/proc.c
src/backend/storage/lmgr/deadlock.c
src/backend/storage/ipc/sinvaladt.c
src/backend/utils/cache/relcache.c
src/backend/utils/mmgr/portalmem.c
src/backend/commands/trigger.c
src/include/postgres.h
src/backend/main/main.c
src/backend/utils/misc/ps_status.c
src/backend/commands/async.c
src/backend/Makefile
src/include/Makefile
src/backend/storage/ipc/shmem.c
src/backend/storage/ipc/ipci.c
src/backend/storage/ipc/sinval.c
src/include/storage/pg_sema.h
src/include/storage/spin.h
src/include/storage/proc.h
src/backend/storage/file/fd.c
src/include/libpq/libpq-be.h
src/backend/utils/init/globals.c
src/include/libpq/libpq.h
src/backend/utils/mmgr/mcxt.c
src/backend/libpq/hba.c
src/include/utils/memutils.h
src/backend/utils/cache/inval.c
src/include/utils/inval.h
```

Figure 5.9. Source Files Created or Modified for Connection Pooling.

Chapter 6

DEVELOPMENT ISSUES

The following sections outline some of the issues faced during the implementation phase of pooling the backends. Some were easily solved, others took a few days of tinkering and constant focus. More often than not, the solutions would come the next day after a night of sleep – it's amazing how the mind can quickly process various scenarios and either come up with a solution or methods that would yield more data to help determine what was really wrong. Finding the problem usually led to a quick solution, so fixing these issues was generally the easy part – finding what was really wrong was not.

Before moving on it is important to point out the value of keeping notes through this process. The following issues were found during actual coding. They were not part of the initial design, nor were they bugs found after finishing the implementation (those are in Chapter 7). Most of the time it was one step forward, two steps back. In coding a single function, two, sometimes three, issues would arise. Sometimes it was possible to interrupt the current function to think up a solution and implement it immediately (those issues never made it to the issues list). Generally, the issue would simply be added to the list and after the current function was completed the list would be used to determine where to go next.

The second reason for valuing notes is exemplified by this report. Without specific details the more interesting issues that were found may easily have been lost. The issues that follow are the more interesting or difficult challenges faced during this experiment.

6.1 Which Backend to Replace When no Match is Found?

One of the first issues was what method to use in order to determine which backend would be replaced when the pool is full and no match was found. It could be number of times accessed, could be last accessed, or could be a combination of both. The decision made was to go with the oldest backend not in use. To facilitate this a timestamp was attached to each backend. This gets set when a backend is added to the pool and reset whenever the backend is reactivated.

The irony in this is that this was one of the first issues faced and also one of the bugs to be discovered and fixed last (because it was easy enough to fix, it never made it into the official bug list). All the other issues had been ironed out along with all of the current bugs when during some final verification testing it was found that the backends were being chosen in an order that was not expected. As it turned out the timestamp was not getting reset when the backend was reactivated.

6.2 What Types of Backends can be Pooled?

The backend types are regular client connections or checkpoints (also known as dummy procs). The initial expectation was that there would be additional backend types. Actually, there are two other backends, the startup process and the shutdown process. The startup process is forked before the pool is initialized so there is nothing to worry about for pooling. And the shutdown process is not created until shared memory has been released and that call takes care of releasing all the backends at the same time, so the pool is in an aborted state and would not try to pool the shutdown process anyway.

To prevent the pooling of the checkpoint the logic for determining which processes can be pooled is set by the postmaster itself. Thus a backend doesn't need to determine whether they are poolable or not. There's a single location for forking connection backends and that's where the postmaster makes the `pool_addBackend()` call.

6.3 What Happens if a Backend Corrupts shared memory?

The backends can possibly corrupt shared memory. When that's detected `quickdie()` is called (`postgres.c`) and needs to know if the postmaster will try to correct the shared memory, or if it releases it and then recreates. If the postmaster is going to release and recreate then no need to do anything since non-shared memory data will be reset anyway when procs are released and shared memory will simply be reset for the pool along with other shared memory clients' data.

As an illustration of the thought process used to investigate and solve problems the following Sections are taken directly from notes kept during the development of the pool functionality. They are primarily concerned with the need for reset functionality in the pool and where that reset should or could occur.

6.3.1 Initial Direction

Initially I was concerned that I might need a reset_pool function. I thought maybe everything would be fine if `reset_shared()` just reset shared memory (including the pool), but if the pool needed to be reset then it should actually free the backends then reset – more than just memory involved. The pool reset call checks to see if the pool container is valid (might have backends asleep) and if so it goes ahead and frees any sleeping backends. Then let `reset_shared()` recreate the memory which would also initialize the pool. The "valid container" global is outside shared memory since `reset_shared()` is only called by the postmaster (and thus not needed by any of the child backends).

6.3.2 Update1

The function `quickdie()` (`postgres.c`) indicates it is used to quickly shutdown a backend immediately (`exit(1)`) without calling `proc_exit()` and does not clean up the transaction. The comments say that it's called because shared memory might be corrupted. Definitely need a reset function. However, it's also possible the system will simply release all shared memory and start over (in fact, the system might just shutdown for all I know). Need to keep looking.

6.3.3 Update2

Turns out the postmaster basically resets itself. I've replaced `pool_reset()` with `pool_shmemCleanup()` and added it to the `on_shmem_exit` queue that gets called when `shmem_exit()` is called, which happens before `reset_shared()` is called and everything works great.

6.3.4 Postmortem

The solution I implemented works fine but I should point out I learned a little more about this (in an unfortunate way, see Chapter 10 for more). There is a case where the postmaster can shutdown without resetting itself. If a stuck spin lock occurs then the postmaster must assume that shared memory is in an invalid (corrupted) state and it immediately exits – nothing nice about it. I can't say that I fully understand why or how this happens, but it occurred in both the master PostgreSQL installation as well as in my pool project installation. As a result I didn't spend any further time to investigate (there were much more pressing issues at the time – I hadn't even pooled my first backend yet).

6.4 Avoiding Stack Overflow in the Backend

In the process of stepping through the debugger and tracing the various `on_shmem_exit()` and `on_proc_exit()` calls a look at the current stack showed that it was 8 calls deep. This meant that every time a backend was pooled in the current implementation the stack would end up with an additional four levels.

Realizing the need to prevent overflow (which would inevitably happen in this case) the code was hacked[1] to see what would happen to the system as the stack was unwound (i.e. exit each function as high as possible up the stack). It was determined that the stack could be safely popped until `BackendStartup()` where the call to `DoBackend()` is made. All that was needed was to put in a check to see if the backend was quiescing and if so enter a loop that would first go to sleep and wait for a signal from the postmaster and when awakened it would enter `DoBackend()` again.

6.5 Signal Interrupts When Quiescing or Asleep

In reading my notes it appears initially I felt this was a bug on my part rather than an issue. However, looking back I think it was a bit of both.

When the backend receives an interrupt a couple of functions get called. The first is `die()`, which checks several global variables to see if it's okay to handle interrupts at the present time or to see if interrupts are already being handled. What had been

[1] This was only done to see where to go on the stack for sleep and reactivation – once found the hacks were removed.

done was to copy some code from one of the `on_proc_exit()` calls (because not all of the function needed to be executed – it would have killed the backend) into a custom `pool_on_exit()` call. However, the usage of one of the variables was misinterpreted and the initial thought was that it would be prudent to set it to 1 instead of 0 like the original code did. After spending about six hours debugging this issue it was found that all of the signal handlers were looking at this variable and not doing anything because it was greater than 0[2].

Although a bug had been introduced into the system, having done so helped in the recognition of certain cases where signal interrupt handlers would need to know about the pool. They would need to know what state it was in and how to tell the backend it needed to abort instead of pool the next time it was shutdown (which is generally immediate, but could be a "next signal" scenario as well – there were several cases to handle). In any case, an abort variable was instrumented and after long debate[3] it was decided to put it into the pool container rather than the backend pool node. These signals that were occurring were fatal and would result in the resetting of the system so the abort variable prevented any more backends from entering the pool if they weren't already asleep in the pool. This worked fine because later in the process the postmaster signals all the children and that just gets caught and frees any sleeping backends.

[2]Imagine my surprise and the lovely vocabulary I expressed at my monitor when I realized what I had done to myself.

[3]Yes, with myself.

6.6 Semaphore Usage

To use a semaphore or not. And if semaphores were used, then how many? One per process? One for the whole pool? And since there is already a semaphore for each process can that one be used for the pool as well?

The reason this is such a tough question is deadlock. Sounds like elementary Computer Science but PostgreSQL (and other databases no doubt) has very complex deadlock detection logic. Effort would not be spent on such things if deadlock was not a possibility. And if it is that much of a problem already, adding to the complexity with the pool functionality was not a good idea. Needless to say caution was used in using semaphores.

Initially the `poolid`, `sleeping`, `valid`, `state`, and `resetPassword` variables were protected. However, after a bit of thought the protection around the `poolid` was removed because only the postmaster controls the setting and resetting of that variable and since it's a single process there was nothing to worry about. The backend would read the `poolid` but only after the postmaster had set it anyway (since the postmaster creates the backends the `poolid` was being set before that occurred and it was being reset after the backend was freed).

The purpose for protecting the other variables is that the postmaster needed to either get a snapshot of the pool all at once (checking which of all the pools are sleeping or valid for example) or because if a backend is freed and the variables have been reset they cannot be touched again by that freed backend[4]. As a result the backends check their pid against the one in the pool node before making any changes. That way the postmaster or new backend can make changes atomically by using the semaphore.

For technical clarification, the semaphore used is implemented by a Spin Lock[16]. This is useful in a multi-process environment because it checks for a brief period of time to see if the other process that's active happens to release the lock and then it can get it instead of blocking and being context switched by the operating system. If the other active process was not the one who had the lock then a block would occur like normal.

6.7 MaxBackends

The initial concern for maximum backends relates to Section 6.2. If there were a significant number of backends that weren't poolable, then there might be a need to keep a reserve of backends available for those tasks. However, it turned out to work in the favor of the pool. The only non-poolable backends found were the startup, shutdown, and checkpoint processes. The startup and shutdown occur before the pool is active and after it's aborted respectively. That left the checkpoint process.

The checkpoint process is created in a special backend called the `DummyProc` and does not count towards the total number of `MaxBackends` in the system. Its initial-

[4]The backends are not always killed immediately, they shut themselves down unfortunately.

ization and processing was completely independent of the regular backends which required no change for pooling.

That leaves the actual reserved backends themselves. These are for super-user activities on the database (e.g. security, maintenance, VACUUM, etc). The default is 2 and the startup of a backend (pooled or forked) exercised the same code to check and see if the number of backends exceeded the reserve availability and whether the current user was a super-user. The only logic added to this process was to count the number of valid, sleeping backends in the pool as free backend slots in the system. Otherwise once the pool was full no one but the super-user would have been able to start a new backend, even if it was from the pool.

6.8 When to Call addBackend?

Should the call to `pool_addBackend()` be done in the postmaster? If so, is it better before the fork or after the fork? Or should it be in the backend after authentication?

The decision was made to put `pool_addBackend()` in the postmaster prior to the fork. The postmaster is aware of what types of backends to add to the pool and this way it prevents the overuse of semaphores in the pool[5]. The only thing that this decision resulted in by way of additional work was the inclusion of a way to determine whether a backend came from a fork or from the pool. This was crucial for

[5]Something I am still wary of (see Section 6.6 for more details).

determining what initialization should take place on the backend versus what should be skipped.

6.9 Backend Linked List

The postmaster keeps a doubly-linked list that contains `PGProc structs`. Each of these structures contain data about each backend. Things like pid, semaphores, backendids, list pointers, etc. The concern was what would happen to this list if the backend were kept alive? Would the list get out of order somehow? Would it become disconnected? Would it free the node related to the sleeping backend? Each of these would be more and more disastrous for the system as a whole (if they were to actually occur).

As it turned out, there were really only two issues with this system of tracking backends. The first, mentioned earlier in Section 6.7, was that the postmaster reported no more connections available when the pool was full even though there were sleeping backends that could be reactivated and used. The solution was simply to add the count of the sleeping backends to the count of the empty slots in the backend count (minus any reserved backend slots).

The second issue was more difficult to resolve. The issue wasn't fully understood until deep into bug tracking and fixing[6]. Basically the postmaster was never being

[6]The actual bug is hardly worth mentioning, but the problem was using a function pointer instead of a function call to determine the value of a conditional statement (forgot the '`()`' at the end of the function name).

alerted when a backend was freed (either through a signal, replacement, or aborted transaction, etc.), and so the postmaster was unable to create a new connection to replace the freed pooled connection once the pool was full.

6.10 Passing Data to Pooled Backend

This issue ended up being the longest open issue for this project. It had been assumed that there would need to be some data passed between the postmaster and backend, but what? In the beginning the thought was that it would have to do with the credentials used to match the backend with the new client connection, but that turned out to not be necessary at after all.

The main reason is that if the postmaster uses the user name and database name to compare against the sleeping backends, then the sleeping backends would already have this information if a match were to occur. If it already had the information then what's the point in passing it to the child from the postmaster? In fact, as you'll see in Section 6.11 this type of criteria was what led each data structure to be stored in either shared memory or the heap.

So if most of the data were already stored by the backend, what was needed from the postmaster? Well, what was unique between each connection? The client socket for one thing. That was where the data would come from that makes up each query or command. Turns out this was easily verified because the data passed from the postmaster to the backend (that wasn't basic proc info like pid, etc.) was stored in a single struct called `Port`. See Figure 6.1 for the members of the struct.

```
typedef struct Port
{
  int       sock;          /* File descriptor */
  SockAddr  laddr;         /* local addr (postmaster) */
  SockAddr  raddr;         /* remote addr (client) */
  char      md5Salt[4];    /* Password salt */
  char      cryptSalt[2];  /* Password salt */
  /* Information that needs to be held during the fe/be
   * authentication handshake. */
  ProtocolVersion proto;
  char      database[SM_DATABASE + 1];
  char      user[SM_DATABASE_USER + 1];
  char      options[SM_OPTIONS + 1];
  char      tty[SM_TTY + 1];
  char      auth_arg[MAX_AUTH_ARG];
  UserAuth  auth_method;
} Port;
```

Figure 6.1. struct Port.

The other members that needed to be passed from the postmaster to the backend were `laddr`, `raddr`, `md5Salt`, and `cryptSalt`. The reason was because these were initialized or set (via communication with the client) in the postmaster, especially once the postmaster took on initial communication with the client for user name and database name verification purposes.

Before moving on to the next section there is one more thing to consider. How would the socket's *open* file descriptor be passed from the postmaster to the backend when `fork()` could no longer be used (which enables the backend to automatically inherit it)? That problem alone took over seventeen hours of work to research, solve, and implement a working test case for (see Section 6.13 for the gory details).

6.11 Where to Put Data (shared memory versus heap)

Although not a serious issue there was no reason to waste space in the shared memory allocation pool with unnecessary data `struct` members. If it didn't need to be in shared memory it would be better to get it from the heap or statically as a global or static variable. Plus there is a performance hit for shared memory access not to mention possible synchronization issues (which are the worst of all the reasons).

So the basic criteria used for deciding was *"who needed to read/write the data?"*. Where "who" means the postmaster and/or the child backend. If the data was accessed only by one or the other (but not both) then it was static or heap. If it was accessed by both, but by postmaster pre-fork and child post-fork and never by postmaster again, then static or heap storage was also chosen. However, if it was needed post fork by both the child and the postmaster then it was necessary to store the data in shared memory.

6.12 How to Put a Child to Sleep

This may seem like a trivial issue, but the important thing to remember is that while implementing the new functionality for enabling pooled backends it was always the goal to follow the current PostgreSQL method of doing things. So the question might be worded *"How should the child be put to sleep in the PostgreSQL fashion."* There were two options to consider (once the options were found, of course).

The first was to use `ProcSleep()` and `ProcWakeup()`. However, upon further investigation they required the caller to already hold the Locktable's `masterLock` at entry to the call. This was a major red flag due to the lack of knowledge about what the `masterLock` was, who used it, and what the side effects of using it for pooling might be. Though additional investigation was certainly an option (and one used quite frequently for most of the other issues), it simply led to a focused searching for a simpler solution.

The second method of putting a process to sleep and waking it up turned out to be the simpler solution and was therefore employed. The functions used were `ProcWaitForSignal()` and `ProcSendSignal()`. No new locks or semaphores were necessary. They are used, but by the functions being called rather than the calling functions (the pool), which results in a higher likelihood of using them correctly.

6.13 Passing an Open File Descriptor to Another Process Without Fork()

This issue was saved for last because not only was it the most difficult it also was the most gratifying (not to mention interesting).

6.13.1 Description

Throughout the development process a lingering question remained at the top of the issues list: *What data needs to be passed from the postmaster to a pooled backend?* The design called for a function to pass the data to the backend. A void pointer was put in the `pool_node` struct as a marker and reminder. Multiple files had several comments throughout the code indicating, *"This might be possible data required for the backend."*

Eventually a single struct was found that held the data needed. It was initialized by the postmaster but was used by the backend. This was the `Port struct` (see Section 6.10). How simple could that be? A single struct contained all of the data (even if some was redundant) required by the backend and the postmaster could easily set the values in shared memory and the backend read them upon reactivation.

However, this was not to be as simple as it seemed. Unfortunately, the issue faded from the forefront to the back while other issues were tackled. Eventually the other issues had been resolved and all that remained were a couple of issues, one of which was passing the data from postmaster to backend. The implementation quickly began and then suddenly came to a screeching halt. *One of the data members was not only a file descriptor, but had been opened by the postmaster and was required by the backend without any interruption to the client.* Things could not be more dire. Pure panic turned into mad determination which can only lead to one thing – Google[7][8].

[7]The actual keywords searched for were: "pass open socket multiple process".

6.13.2 Resolution

After much searching, a well built Linux man page website[9] offered up the first glimpse of a possible solution[8]. *Passing open file descriptors can be handled via socket messaging.*

That's correct – in order to pass an open file descriptor for a socket connection a second socket pair would be needed. Only this time the pair would be between the postmaster and the backends (one pair per backend, created each time a new backend is forked). This may seem simple, but implementing it was tricky.

The only example usage of this functionality was in the `cmsg(3)` man page[22]. It contained an example of looking for data in an ancillary buffer[9] as well as an example of passing an array of file descriptors over a Unix socket.

Although these examples were half the information needed to formulate the solution, unfortunately *they were half the information needed.* The other half came through trial and much error as well as many readings of several related man pages on sockets and ancillary data messages. During research it was found that only Unix sockets (`AF_UNIX`) allowed these messages to be passed (versus Internet sockets or `AF_INET`).

[8]The total time taken from the point of the first search to completion of code was 17 hours over two days.

[9]Ancillary data are also known as control messages. These messages are not a part of the socket payload. This control information may include the interface the packet was received on, various rarely used header fields, an extended error description, a set of file descriptors or UNIX credentials.

This turned out to be an advantage, for while reading about `AF_UNIX` sockets the function `socketpair()` was found (see the socketpair(2) man page[23] for more information). Additional searching even led to a better example[10] that included the use of `socketpair()`.

Combining the three examples resulted in successful creation of the sockets, success in passing data from one process to another, yet no amount of trial and error led to the successful passing of an open file descriptor from the parent process to the child. It was confirmed through debugging that the parent was successfully loading the correct amount of data onto the socket but it was never received by the child.

With frustration at its peak, all that was left to do was go to sleep. The very next morning a re-reading of the unix(7) man page[25] found the slightly vague comment in the last paragraph shown in Figure 6.2.

```
The PF_UNIX (also known as PF_LOCAL) socket family is used to
communicate between processes on the same machine efficiently.
Unix sockets can be either anonymous (created by socketpair(2))
or associated with a file of socket type. Linux also supports
an abstract namespace which is independent of the file system.

Valid types are SOCK_STREAM for a stream oriented socket and
SOCK_DGRAM for a datagram oriented socket that preserves message
boundaries. Unix sockets are always reliable and don't reorder
datagrams.

Unix sockets support passing file descriptors or process
credentials to other processes as ancillary data to datagrams.
```

Figure 6.2. Description from unix(7) Linux Man Page.

The reason this is so vague is because in the socket(2) man page[24] for the `SOCK_STREAM` socket type parameter value it states, *"An out-of-band data transmission mechanism may be supported."* This was the initial assumption followed in attempting the successful passing of an open file descriptor using socketpairs. However, by simply switching a single parameter to *socketpair()* (from `SOCK_STREAM` to `SOCK_DGRAM`) the test program built to validate this new functionality was successfully executed.

In conclusion, in order to successfully pass an open file descriptor from one process to another without the use of `fork()`, a unix datagram socket must be employed in conjunction with ancillary data payloads.

Chapter 7

BUGS, BUGS, AND MORE BUGS

After finishing all of the development raised issues the next step is runtime debugging. Most of these bugs were found through simple exercise of the newly created connection pooling functionality. These included tasks like: starting the database, shutting down the database, attempting the first connection, attempting the first pooled reactivation, attempting to fill the pool, freeing a backend from the pool, etc.

In almost every case one or two things would go wrong with each action. The following is not a complete list of each bug found, but rather a list of the more difficult or more interesting problems run into.

7.1 Postmaster Will not Shutdown

This was an early problem that had two defects associated with it. The first was that `pool_closePostmasterSockets()` was being called by the postmaster's reset functionality. However, the shared memory in the pool had yet to be initialized but it was trying to access it anyway to close all of the socketpairs used for passing the open file descriptors (see Section 6.13). A simple flag indicating the validity of shared memory provided the solution.

The second defect was either a simple typo or a misunderstanding of the Memory Contexts used by PostgreSQL. The pool was using `free()` to deallocate memory instead of `pfree()` as required.

7.2 SIGSEGV in Postmaster During ProcessStartupPacket()

The postmaster was crashing with a `SIGSEGV` status in `secure_read()` during `ProcessStartupPacket()`, where the credentials are retrieved from the client by the postmaster versus the backend. The initial hypothesis was that there was additional initialization that needed to occur before communication with the client would be possible. This indeed turned out to be the solution, `pq_init()`[1] must be called prior to any communication functions being used. This also necessitated the move of `pq_close()` via the `on_proc_exit()` usage to the postmaster instead of the backend as well as relocation of `MyProcPort` initialization (the pointer to the all important `Port` struct).

7.3 Shutting Down Postmaster Causes Hang

This defect is similar to the one in Section 7.1, but the behavior and cause were different. This was also more difficult to root cause. The following list represents the various ideas and steps used in debugging and solving this problem.

[1]The `pq_*()` function list comprises PostgreSQL's client communication library.

- The initial assumption was that the postmaster was waiting for some type of response from one or more child backends. This was verified by the `ps` utility that listed not only the postmaster but also one or more backends after the shutdown attempt had been made but timed out and failed.

- In `postgres.c` there is a signal handler called `die()`. It is called in response to a `SignalChildren()` call in the postmaster. The initial thought was that there needed to be logic in the function to know about and handle the pooling functionality. However, this led to a question of where exactly to put the logic in order to make it a clean solution (less coupling, for example). The dilemma was whether to change `SignalChildren()` in the postmaster or whether to change something in the backend itself. The simplest solution (up front) would have been to put it in the postmaster (simply free any sleeping backends before signaling the active children). However, there is a timing issue here – what if an active child sleeps before the postmaster can signal? Also, by putting the logic into the children, then not only were the children prepared to handle signals from the postmaster, but they were able to handle other signals as well.

- The initial attempt to solve this was to put a wakeup call in the backend signal handler if the backend were sleeping. All backends actually got a signal from the postmaster as it turned out, but the sleeping ones weren't handling it properly. The global variable `InterruptHoldoffCount` had a value of 1 and it wasn't immediately clear where that was coming from or why it had been set.

- It turns out the bug was a misunderstanding of what the variables were to be used for. When creating the pool's shutdown code (based on PostgreSQL's `proc_exit()` original function) some of the code was modified to prevent the backend from actually shutting itself down. Unfortunately this modification happened early on in the development process when understanding of the system wasn't as deep as it was at the end. The modified code had set the value to 1. By setting it to 0 instead the signal handler would now attempt to handle the signal instead of ignoring it as previously directed.

- Although this change enabled the backends to handle the signal which resulted in a call (after other things) to `proc_exit()` but instead of shutting down, the backend was pooling itself again. To fix this issue two functions were added to the pool. The function `pool_die()` is used for when the backend gets a signal (whether from the postmaster or elsewhere – an important distinction). The function `pool_postmasterSignalChildren()` is used for when `SignalChildren()` is called. Both functions set various state variables that prevent the current backend and all backends respectively, from pooling themselves when `proc_exit()` is called.

7.4 Postmaster Crashes After 21 Pool Reactivations on a Single Backend

Fortunately this problem was easily solved because one of the authors of PostgreSQL was kind enough to use `ASSERTs` to enforce constraints in their code. An `elog` (PostgreSQL's logging system) was generated that clearly stated the problem and where it occurred. The problem was the `on_proc_exit_list` that stored callbacks was full and no additional callbacks could be added. A quick investigation turned up the maximum value of the static array at 20 callbacks. It definitely didn't make sense to make the array dynamic especially since there was no knowing what would happen when two or more of the same callback were to be made (releasing resources, affecting shared memory and semaphores, etc). The solution was simple since only one callback was being added multiple times, `pq_close()`. It was being added by `pq_init()` which was called each pool reactivation for client communication initialization. A quick conditional check to prevent more than one entry into the callback list fixed the problem.

7.5 hba Pre-Parsed Data is NULL in Pool

Figure 7.1 shows the comments from the `hba.c` source file that describes what it is and how it is used.

```
* hba.c
*       Routines to handle host based authentication (that's the
*       scheme wherein you authenticate a user by seeing what IP
*       address the system says he comes from and possibly using
*       ident).
```

Figure 7.1. Source File Description for File hba.c.

The `hba` functionality relies on pre-parsed files (e.g. user and group files) stored in doubly-linked lists. They are used during `ClientAuthentication()` called in the backend to verify the client's authority to use the database. During reactivation of a pooled backend `ClientAuthority()` is called again to be certain the client is authorized to use the same database as before. However, by the time the reactivated pool used it again, the pointer to the list was `NULL`.

Initially it was thought that the problem was a memory stomp. The reason for this is that the data was still valid when the backend went to sleep but during initialization something would change the value of the pointer and invalidate it. No calls to `free()` could be found for the memory allocated. Then the "stomp" was narrowed down to a raw IO call to a data file in `InitPostgres`. During the call the memory changes. The `read()` call had a block size of 8192 bytes. At this point it was assumed that the problem was inherent to the PostgreSQL *master* code base. It would be submitted as a bug and a call to `hba_load()` would be made at each reactivation (which would be unfortunate since hard disk access is expensive).

Even though this was marked as fixed in the master bug list, it still remained an open issue to the subconscious, and eventually a new hypothesis was generated as a result. PostgreSQL uses MemoryContexts for most of its memory management. If the linked list belonged on a context, then the memory would be allocated in a large chunk from the heap, used and then returned to the context manager. When the next context gets that memory it may not even use all of it (since smaller calls to `palloc()` are made for the actual memory used by the system). So it may take awhile before the physical memory is changed, thus changing the value seen in the debugger (which a 8192 byte IO read could certainly do). More than likely the memory context that the linked list was allocated on was released to back to the context manager.

The new solution, therefore, was not call `hba_load()` each time, but rather to change the memory context to the top context (which is never released) before `palloc()` call is made for the `hba` list. Thus it will remain in the backend for the life of the process. In researching how to do this and where, it was also determined that calls to `load_user()` and `load_group` would need to be made by pooled backend during reactivation if indicated by the postmaster through shared memory (this would occur whenever the `hba` files are changed on disk – either manually or through the database SQL interface).

7.6 on_proc_exits() not Resetting Server Properly

In Section 7.3 a method was described wherein a solution was provided for proper handling of signals to backends that are pooled – asleep or otherwise. The `elog` function allows for several severities – the highest two are `ERROR` and `FATAL`. `ERROR` will shutdown the current backend whereas `FATAL` signals the postmaster to shutdown all backends after which it either resets or aborts (depending on what the cause is).

What wasn't apparent at the time of the initial solution was that `FATAL` `elog` calls in a backend does not generate an actual signal in the backend (and it ignores other signals because it knows it is "handling" it already). Instead it communicates the error to the postmaster and then calls `proc_exit()` but was sending an exit code of 0. The initial design of condition checking in `proc_exit()` would not pool if the exit code was not 0. This was a problem for `FATAL` `elog` calls though.

This led to an investigation of what other calls to `proc_exit()` use an exit code of 0. The total came to 15 calls that either were an actual 0 or could be a 0 depending on other conditions. Looking at all of the other calls forced the question, "Which call to `proc_exit()` was really a valid or successful exit scenario?" The answer turned out to be fairly simple – the case where the commands and/or queries from the client were successful and did not result in an error or an aborted transaction. To implement this additional criteria a flag was added to the pool that is false when a backend reactivates and is only ever true after a successful command or query sequence. In all other cases the backend gracefully shuts itself down (including the initial case of a `FATAL` `elog` within a backend).

7.7 Out of Cache Callback Slots

This is another great example of using `ASSERTs` to enforce constraints (similar to Section 7.4). During one of the test runs (early on when things were much simpler than the examples described in Chapter 8) a `FATAL` `elog` was called with the following message, *"Out of cache_callback_list slots."* With a message as clear as that it took a few seconds to look the error up using `glimpse`. After reading the source file comments it took just a few minutes to create the functionality in the pool to handle the problem – the `cache_callback_count` variable needed to be reset after each backend is quiesced.

Figure 7.2 shows the comments from the source file associated with the constant that defines the a maximum size of the callback list (actually an array – that was part of the problem).

```
/* Dynamically-registered callback functions.  Current implementation
 * assumes there won't be very many of these at once; could improve
 * if needed. */
#define MAX_CACHE_CALLBACKS 20
```

Figure 7.2. `MAX_CACHE_CALLBACKS` Comment.

This is an important comment. In Section 8.4.4 this comment was used to help develop the hypothesis that attempts to explain the memory leak in the backends that are pooled and repeatedly reactivated (a leak related to the current design of the backends rather than the new pool functionality directly).

Chapter 8

TESTING

The following types of testing falls under three categories: regression, verification of the proof of concept, and performance. The regression was a must because it's required in order to merge code back into the main trunk of PostgreSQL. The verification was also a must. It was needed to show that the project itself was successfully implemented. The performance testing on the other hand wasn't really a requirement. However, it turned out to provide the more interesting part of this entire project. As will be seen in Chapter 9 the assumed improvements weren't even the interesting "find" after all. It turned out to be something entirely unexpected.

8.1 Regression

Regression testing is an important aspect of any large multi-cycle development project. Before any new code or bug fix for PostgreSQL may be submitted to the core developers the submitter must be able to successfully run the regression suite. A successful run of the regression tests indicates that everything currently working and tested by the suite is still working as expected.

Figure 8.1 shows that the *project* version of PostgreSQL successfully passed the regression tests.

```
test conversion           ... ok
test truncate             ... ok
test alter_table          ... ok

======================
 All 89 tests passed.
======================

/local/mjsbsu/project/src/test/regress
->
```

Figure 8.1. Regression Test Results on *Project* version of PostgreSQL

8.2 Proof of Concept Verification

To verify the actual implementation of connection pooling in the backend of the PostgreSQL database a simplified version of the TPC-W[7] test specification was used[1]. Five of the tables were employed with most of the relational constraints and primary key relationships intact. The queries run were a series of inserts wrapped within a transaction (see Appendix B for complete details). There were four users in use for the testing, the Linux user account (which acted as a super-user account), and three generic regular users. Each user would spawn X number of background processes that would run Y number of queries in serial to the same database.

[1]This table and query setup was also used for the psql performance testing in Section 8.3.

This combination was used in the following order:

- Spawn 32 processes running 10 queries each as user `mshelton` to fully load the pool.
- Spawn 16 processes running 10 queries each as user `dbuser1`. Half of the pool backends will belong to `mshelton`, half to `dbuser1`.
- Spawn 8 processes running 10 queries each as user `dbuser2`. Eight processes will belong to `mshelton` still (it is the oldest used and will be replaced first), 16 to `dbuser1`, and 8 to `dbuser2`.
- Spawn 4 processes running 10 queries each as user `mshelton`. The number of processes per user will not change, but the timestamp on 4 of the `mshelton` backends will have been updated to become the most recently used for the whole pool.
- Spawn 8 processes running 10 queries each as user `dbuser3`. 4 processes will belong to user `mshelton` (the ones just updated), 12 to `dbuser1` (the other 4 to be replaced for `dbuser3` will come from `dbuser1`), 8 to `dbuser2`, and 8 to `dbuser3`.

After each of these steps, `ps` was used with `grep` to isolate the backends that are in the pool. The status contains the last user to access the backend and a report is generated that is used to verify that the scenarios expected match the current state of the pool. The output from the test script is shown for the final test case in Figure 8.2.

8.3 Performance

The performance testing was the most difficult of the three. The issue was the lack of experience with analysis or performance testing in general. The first runs generated almost a gigabyte of data. However, the results weren't useful for showing whether an improvement had been made or not. Also, one of the setups involved the comparison of PHP Persistent Connections[12] and results were unavailable (see Section 8.4 for details).

Without Ben R. Shelton's[26] help the test output and resulting analysis would have been shoddy at best. But in complete contrast they have become the highlight of this experience. The results were sometimes expected, other times slightly disappointing, and then finally completely surprising. So much so that they are more than just worth reading about, they are worth additional research that may lead to possible advancements. However, before getting to the analysis, the test process must first be understood.

8.4 PHP

The test alternatives for PHP consisted of two *master* versions of PostgreSQL and one *project* version. The first *master* was queried with PHP via traditional connections (the backend is released when the connection is closed by the client). The second *master* was queried by PHP using persistent connections (the backend is still running waiting for additional requests from the client). PHP holds the connection open during an entire session and reuses it for each open/close cycle that needs it in the PHP scripts. The third alternative was non-persistent connections using the *project* version of PostgreSQL. The connection pooling was built into the database rather than through the client.

8.4.1 Desired Testing Capabilities

The hope was to be able to compare the initialization and total command/query times between the three alternatives. The initialization is the time between when the postmaster receives a new connection on its main socket and when the forked backend finishes authentication and initialization and is prepared to read the first command or query from the client. The total time is the time between when the postmaster receives the new connection and when `proc_exit()` is called at the end of a command/query.

8.4.2 Test Scenario

A series of scripts were created by Tim Vance[27] to perform parallel accesses to the database using multiple users and processes. Each access would execute identical queries to the database, thus providing similar load requirements on the database itself. The idea was to focus on the initialization time, assuming the query time would be similar for each scenario, since the *project* database had no changes in any of this functionality compared to the *master* version.

The actual tests run and queries used will not be provided, Sections 8.4.3 and 8.4.4 explain why. However, the basic setup was a series of simple, medium, and complex queries. Each of these series were looped through ten times to provide a sustained load on the database. This loop was put into a single script and called in parallel seven times for each of four different user names. Finally this script of parallel runs was called one hundred times in a row to help isolate any environmental anomalies.

8.4.3 Results

The 100 iterations were run five different times with a complete wipe of the database before each run. Each of the 100 iterations took about 51-55 seconds on both the non-persistent and persistent alternatives using the *master* version of the database. When the same test scenario was run against the *project* version, it took seven hours to run 7 iterations in the first test time.

The log files for the persistent *master* database alternative contained only the initialization checkpoints but no ending checkpoints. The log file for the non-persistent alternative was almost a gigabyte in size. The log file for the 7 iteration run on the *project* database contained over 35,000 entries.

8.4.4 Analysis

This testing scenario was a complete disaster. The persistent connections did not have finish times, therefore it was not only impossible to determine the actual total time, it was only possible to gather the initialization times on the first few connections to the database (that's the nature of persistent connections). Without serious modification to the database driver/frontend this will remain the case.

Unfortunately changing the driver and frontend of the database is outside the scope of this experiment. It is possible that the effort required may be equal to that of this actual experiment (in difficulty, size, and time). Suggestions for future developers could include any of the following (or variations thereof):

- Unique byte sequences that indicate the "closing" of the connection (or the end of its use by the current client, but that it will be used again later).
- An additional command in the database that would take a snapshot of the current time, run the query, take another snapshot of the time and return the values to the client with an additional command (which would be the last one in the query set). Note that in order to pair these with parallel runs identifying data would also need to be supplied. This is probably the simplest of the options.
- A final idea[2] would be to create the client queries in a deterministic size (e.g. a single transaction containing two inserts and four selects). Then if possible, determine the number of query commands run and when the limit is reached mark that as the ending time.

[2]This idea was conceived while writing this document. Although it may be tempting to think that this may have been implemented within the given time frame of this project (nothing is ever as easy or as quick as initially "thought"), it's worth pointing out that the analysis and results of the final tests easily overshadow this notion.

Another problem, that is even bigger than the previous, is the database performance on the *project* when such a large test scenario is executed against it. In using `top` to view the resources on the test server it was apparent that all of the available memory (only 11MB left from 4GB of installed RAM) was being used by the backends of the postmaster. This means one thing – memory leak. This is a serious defect and leads to the question, *"Why wasn't this listed as a bug in Chapter 7?"* The reason is because this bug was not directly caused by the pool functionality. Rather, (after much thought, worry, and consideration), it has been concluded that the leak is due to the non reusable nature of the backends themselves. They were never designed to be reinitialized for a new connection. And in the seven hours the 32 backends did run they achieved over 30,000 queries – most of which finished within the first few minutes of the test process. This conclusion was reached based upon information gathered from the source code itself. It can be imagined that code authors took no thought to leaving memory allocated knowing full well that it was only allocated during initialization (thus unaffected by PHP persistent connections for example) and would never be allocated again (and the original data "lost") and once the backend exited all memory would be free.

8.5 psql

Since PHP was no longer a viable option for testing the performance of the new pooled backends, the next alternative was psql, a client application that comes with

PostgreSQL and is capable of running queries on the command line or via a file. Initial testing resulted in too much data about the wrong types of checkpoints in the system (like trying to compare apples and oranges).

After conferring with a professional analyst a new more focused plan of action was developed. The many queries were simplified into a set of 52 queries (inserts into a constrained relational table set) wrapped in a single transaction. From this new test scenario it was possible to extract valuable information and then analyze it for some interesting results.

8.5.1 Understanding the System

In determining which checkpoints in the query process needed a timestamp the following questions needed to be answered:

- Is the reactivation faster than fork? The guess was that fork was an expensive process and waking a sleeping process simply involved a bit of shared memory and semaphore access, which "should" be faster.
- How long is the initialization time of each connection (fork versus reactivation). The assumption here was that since the *project* database was skipping some of the initialization steps during reactivation as long as the reactivate process were faster, then the initialization as a whole should be faster.
- How long does the actual work portion of the request take? This is the query or command part of the entire connection life cycle. Since nothing should have

changed for the parsing and data access portions of the database these values should be similar between the two databases (they weren't – keep reading).

- What is the total time of the connection life cycle? Was there an improvement? Did the initialization really cost more than the query/command?

Once these questions were asked and answered the following checkpoints were used:

- `STARTUP` [both] - the time at which the socket request was received by the postmaster.
- `PREFORK` [master] - the time right before the `fork()` occurred in the postmaster.
- `POSTFORK` [master] - the time after the `fork()` is completed in the child process.
- `PRECANREACT` [project] - the time before the pool is scanned for a match, a free slot, or in the case of a full pool with no match, a backend is freed and the slot made available.
- `PREREACT` [project] - the time before the reactivation starts in the postmaster.
- `POSTREACT` [project] - the time after the reactivation completes in the backend.
- `PRECMD` [both] - the time right before the first data is read from the client socket for the query or command.
- `FINISH` [both] - the time when `proc_exit()` is called.

Once this was implemented two series of tests were run. The first was a serial test. The setup involved a single user making a single connection at a time and running the query described in Section 8.5. One thousand of these connections were run at three separate times (to alert of any discrepancies caused by possible environment variances) on both the *project* and *master* databases. The second series was a parallel test. Thirty parallel connections in separate processes were run using a single user and the same query as before. One thousand of these were also run at three separate times. However, it appears the multiple connections caused the memory leak to be more apparent and skewed the data too much. It was recommended that the test size be reduced to 120 queries.

8.5.2 An Interesting Observation

Prior to the data being entered into the spreadsheets for analysis, an immediate observation was made. The tests were all run with the `time` command for curiosity purposes and an exciting development occurred. The time taken by the *master* database ranged from 883.230s to 886.463s. However, the time taken by the *project* database ranged from 102.218s to 104.618s. This was exciting – the connection pooling was having an effect on the database performance after all. And it is a positive one as well[3].

[3]I know what you're thinking, *"But I thought you said the connection time wasn't the most expensive part of the operation? If that's the case then why the discrepancy in times?"* Read on for the exciting conclusion.

```
mshelton 16971 16872 postgres: mshelton bsu [local] pooled
mshelton 16974 16872 postgres: mshelton bsu [local] pooled
mshelton 16976 16872 postgres: mshelton bsu [local] pooled
mshelton 16983 16872 postgres: mshelton bsu [local] pooled
mshelton 17364 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17366 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17371 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17373 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17379 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17381 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17382 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17383 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17384 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17385 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17386 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17393 16872 postgres: dbuser1 bsu [local] pooled
mshelton 17670 16872 postgres: dbuser2 bsu [local] pooled
mshelton 17673 16872 postgres: dbuser2 bsu [local] pooled
mshelton 17683 16872 postgres: dbuser2 bsu [local] pooled
mshelton 17684 16872 postgres: dbuser2 bsu [local] pooled
mshelton 17685 16872 postgres: dbuser2 bsu [local] pooled
mshelton 17686 16872 postgres: dbuser2 bsu [local] pooled
mshelton 17687 16872 postgres: dbuser2 bsu [local] pooled
mshelton 17688 16872 postgres: dbuser2 bsu [local] pooled
mshelton 17872 16872 postgres: dbuser3 bsu [local] pooled
mshelton 17875 16872 postgres: dbuser3 bsu [local] pooled
mshelton 17885 16872 postgres: dbuser3 bsu [local] pooled
mshelton 17886 16872 postgres: dbuser3 bsu [local] pooled
mshelton 17887 16872 postgres: dbuser3 bsu [local] pooled
mshelton 17888 16872 postgres: dbuser3 bsu [local] pooled
mshelton 17890 16872 postgres: dbuser3 bsu [local] pooled
mshelton 17933 16872 postgres: dbuser3 bsu [local] pooled

Num procs for mshelton:      4
Num procs for dbuser1:      12
Num procs for dbuser2:       8
Num procs for dbuser3:       8
```

Figure 8.2. Output from Test Script `showpool` for Last Test Case.

Chapter 9

ANALYSIS

Initially two claims were made about performance. The first was concerning connection time versus query time. The general understanding[1] was that connection time is more expensive than simple query times. The second claim was a hypothesis on the effects of connection pooling to the performance of connection initialization time as well as overall time. The hope and purpose of this experiment was to show that there is an improvement in the connection time as a result of Connection Pooling in the *backend* of a database (in this case PostgreSQL).

The following sections show the results of the testing performed in Section 8.5. They explain the results as they apply to each claim and whether the claims still hold true. The final section concerns a surprising find regarding a measurable performance improvement with repeatability.

9.1 Are Connections Expensive?

Initial understanding of the cost relationship of connections to small queries was weighted more heavily to the connection itself. To verify that this was still the case

[1]With current references to back it up[11].

with modern databases and hardware a quick lookup on the Internet revealed the following statement about Connection Pooling in conjunction with IBM's DB2[11]: *"Establishing and severing connections to the database server can be a very resource intensive process that adversely affects both database server and DB2 Connect(TM) server performance."*

However, by comparing the *Init* and `FINISH`[2] times in the Tables under Section 9.3 the *Init* time is at worst 4 times faster than the `FINISH` time and at best 500 times faster. Or to put it another way the *Init* time was anywhere from 23.39% to 0.20% of the total time of the connection. Keeping in mind that none of these tests were meant to measure the connection times compared to query times, these results still warrant additional investigation into the real data concerning connection times versus query times.

9.2 Did Pooling Provide an Improvement in the Initialization Times?

Yes and no.

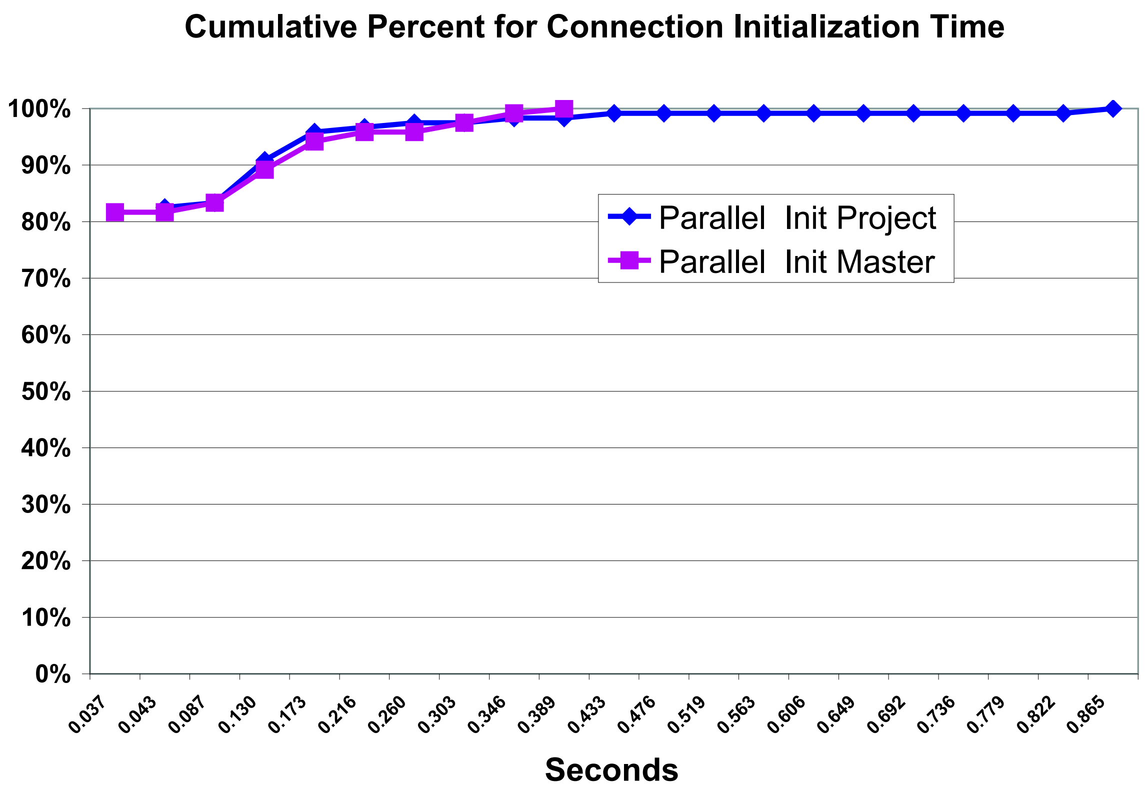

Figure 9.1. Parallel Initialization Times.

9.2.1 Parallel Results

For the parallel tests (see Figure 9.1), the *Init*[3] times were almost identical between the *project* and *master database*. Ninety-eight percent of the connections had *Init* times of 0.303 seconds or less. It is safe to say that connection pooling in the backend

[2]The `FINISH` time is the time it takes to execute the command or query sent by the client.

[3]The *Init* time is the time elapsed from the `STARTUP` checkpoint to the `PRECMD` checkpoint. See Section 8.5.1 for an explanation of the various checkpoints.

does not adversely effect the performance of initialization while there are parallel connections occurring.

9.2.2 Serial Results

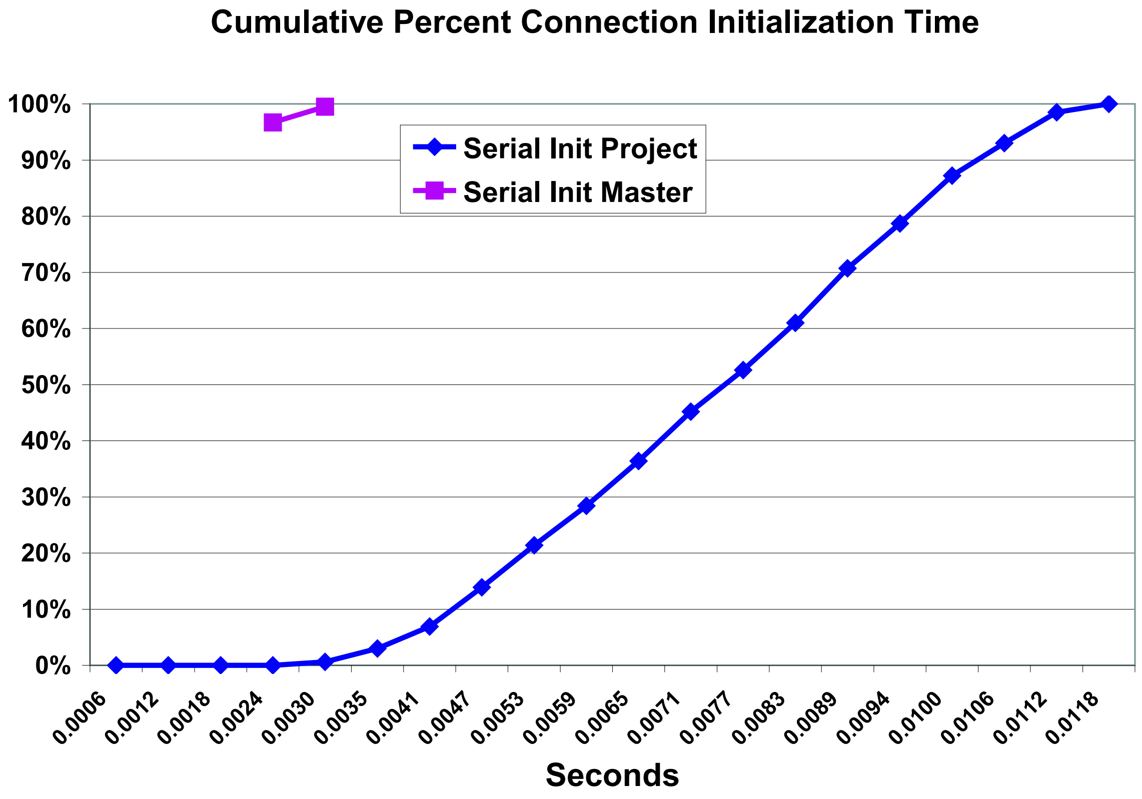

Figure 9.2. Serial Initialization Times.

The *Init* times (see Figure 9.2), for the serial tests were a disappointment for the *project* database. The *master* database had 97% of its times measuring under 0.0024 seconds whereas only 1% of the *project* times were under 0.0030 seconds. The 80%

mark for the *project* measured at 0.0094 seconds and 100% of the times fell under 0.0118 seconds.

Although all of these times (both databases) still measured under the parallel times by an order of magnitude, it's still surprising to see the data so different between the two. It could also be argued that since the common environment for database systems is one of parallel connections, that the serial test case though interesting is not necessarily a detractor for the overall performance measurement of the backend connection pool.

9.3 The Surprise

What is the surprise? The surprise is that somehow the changes in the backend because of the implementation of the pool resulted in a performance improvement in the query/command processing time. It is a surprise because the purpose of the pool was to improve everything up until this processing takes place. Nothing else should have been effected. However, even though this is an unexpected side effect the following results show that it has potential to be a very positive one.

9.3.1 Parallel Results

The chart in Figure 9.3 says it all – and in a big way! This is an unexpected side effect of the connection pooling implementation. The *project* database reached 100% of its times by 3.865 seconds versus only 38% of the *master* at the same time length.

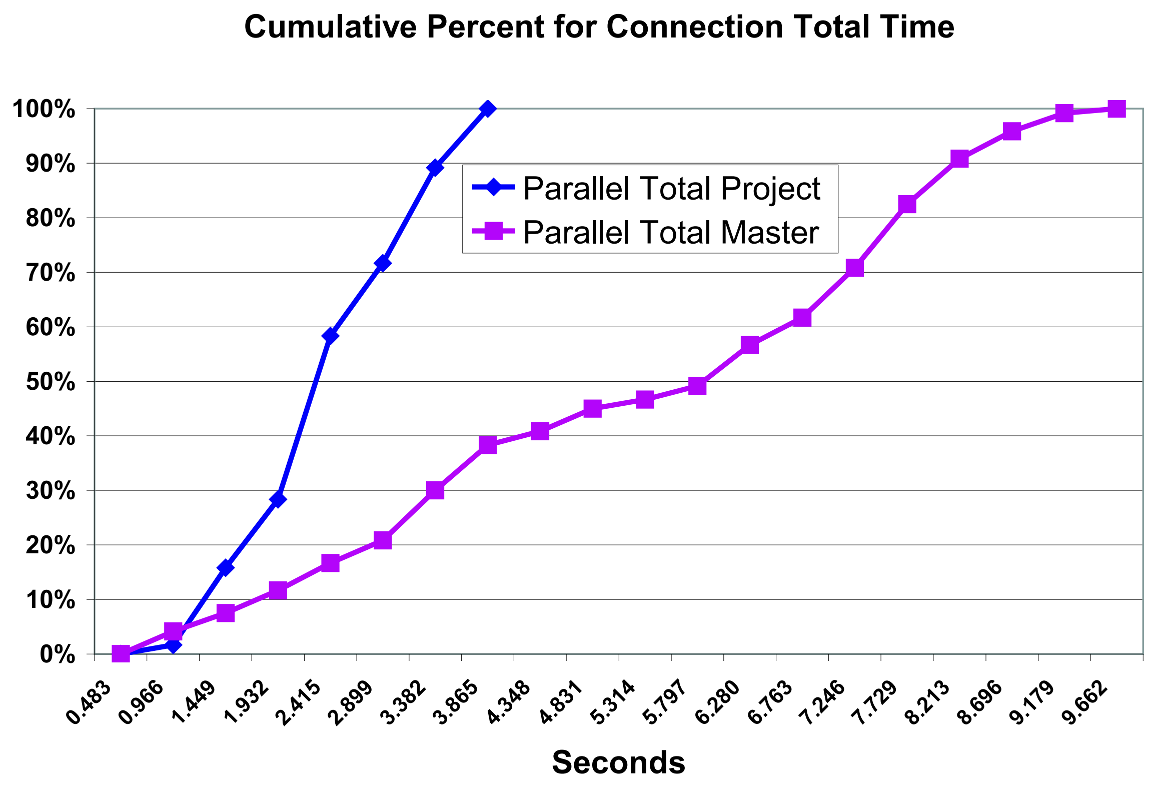

Figure 9.3. Parallel Total Times.

The *master* had 80% of its times under 7.729 seconds (almost two times as long) and 100% under 9.662% (almost three times as long).

The next question is why did this happen? The answer is still unknown and requires additional research, experimentation, testing, and analysis. An interesting thing to remember as well is that in the results of Section 9.2.1 the *master* and *project* were nearly identical. So none of the improvement can be attributed to a speedup in initialization times due to avoiding `fork()` or skipping some of the initialization steps. Furthermore, Section 9.4 demonstrates that these results are not an anomaly.

Tables 9.1 and 9.2 have been provided as a brief summary of the data used to formulate the charts that are under analysis. For curiosity sake the time required to fork a process in the *master* versus the time to reactivate a process in the *project* was also measured. The reactivation was on average two times faster than the fork. However when the time measured is in the micro-seconds it's probably not much of an issue overall (especially in terms of total times ranging from 0.10 seconds to 9.66 seconds).

Parallel Master		*PREFORK*	*POSTFORK*	*Fork*	*PRECMD*	*Init*	*FINISH*	*Total*
Mean		0.00001	0.00038	0.00038	0.03088	0.03126	5.01103	5.04229
Median		0.00001	0.00028	0.00029	0.00199	0.00233	5.80967	5.91204
Range		0.00001	0.00591	0.00592	0.36405	0.36406	9.55822	9.55830
Minimum		0.00000	0.00022	0.00022	0.00161	0.00185	0.10139	0.10355
Maximum		0.00001	0.00613	0.00614	0.36566	0.36591	9.65961	9.66185
Count		120	120	120	120	120	120	120

TABLE 9.1 Parallel *Master* Statistical Summary

Parallel Project	*PRECANREACT*	*PREREACT*	*POSTREACT*	*Reactivate*	*PRECMD*	*Init*	*FINISH*	*Total*
Mean	0.00005	0.00006	0.00007	0.00018	0.03548	0.03566	2.27578	2.31144
Median	0.00005	0.00005	0.00003	0.00013	0.00186	0.00200	2.22107	2.22856
Range	0.00048	0.00085	0.00291	0.00424	0.86403	0.86406	3.02624	3.02614
Minimum	0.00000	0.00003	0.00002	0.00005	0.00121	0.00132	0.67178	0.67331
Maximum	0.00048	0.00088	0.00293	0.00429	0.86524	0.86538	3.69802	3.69945
Count	120	120	120	120	120	120	120	120

TABLE 9.2 Parallel *Project* Statistical Summary

9.3.2 Serial Results

The serial results are similar in comparison to the parallel results in that they also show a large improvement in total time for the connection life cycle. Worth pointing out though, is that unlike the serial performance results for the initialization times where the *project* under performed the *master*, the results this time are more aptly described as stellar. The chart in Figure 9.4 shows an almost exponential result in the locale of the times for the serial runs. Twenty-seven percent of the times were under 0.08 seconds (remember this is *total* time, not just initialization time) and 100% were under 0.15 seconds! Only 15% of the *master* times were under 0.15 seconds. Whereas 80% were between 1.30 seconds and 1.38 seconds with 100% under 1.53 seconds. This is an order of magnitude in improvement!

Tables 9.3 and 9.4 have been shown in order to provide a similar summary for comparison reasons like the parallel result tables were. The data and ratios are similar between the parallel and serial runs and therefore nothing new is being presented.

Serial Master		*PREFORK*	*POSTFORK*	*Fork*	*PRECMD*	*Init*	*FINISH*	*Total*
Mean		0.00001	0.00024	0.00025	0.00184	0.00209	0.80054	0.80262
Median		0.00001	0.00024	0.00025	0.00182	0.00207	1.16935	1.17142
Range		0.00001	0.00040	0.00041	0.00217	0.00244	1.50651	1.50657
Minimum		0.00000	0.00020	0.00020	0.00168	0.00191	0.02272	0.02471
Maximum		0.00001	0.00060	0.00061	0.00385	0.00435	1.52923	1.53128
Count		1000	1000	1000	1000	1000	1000	1000

TABLE 9.3 Serial *Master* Statistical Summary.

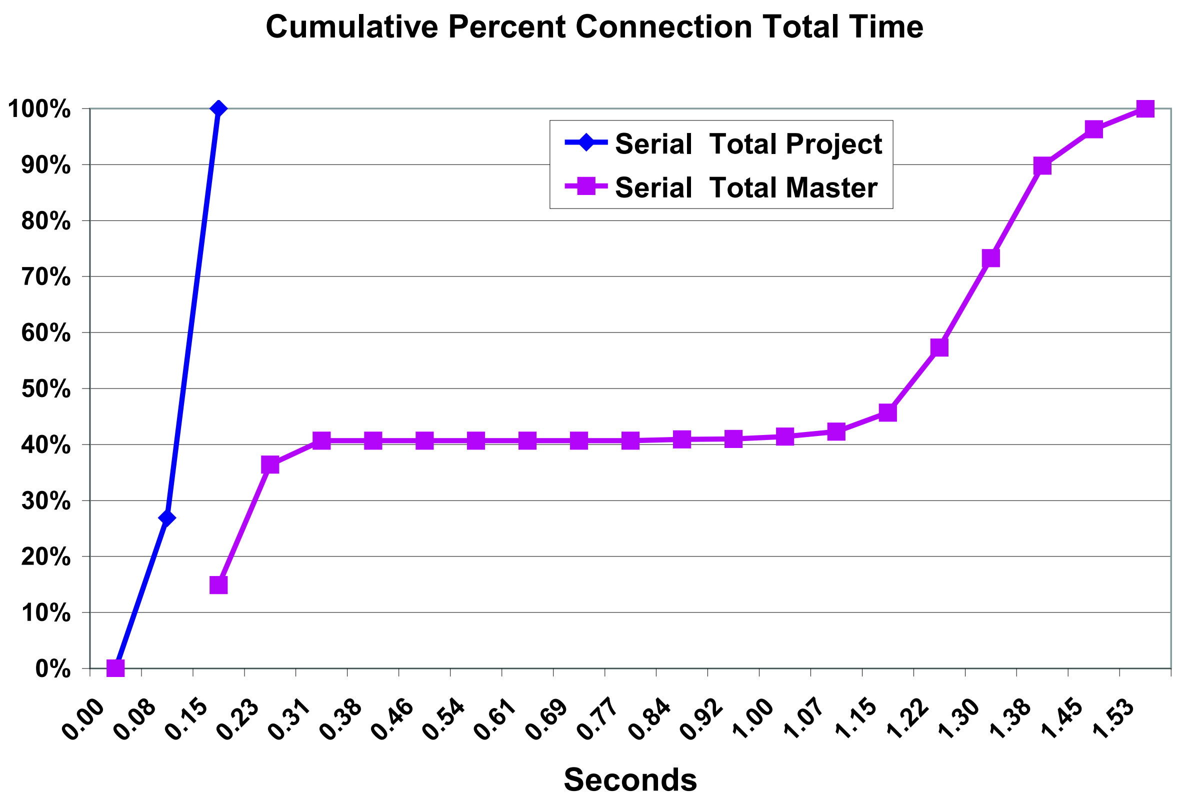

Figure 9.4. Serial Total Times.

9.4 Are the Results Reproducible?

All of the tests were rerun on different days and different times to verify that the results were consistent and not related to an anomaly in the computing environment. When the tests were complete the count of the rows in each of the tables used (see Appendix B) were compared between the *master* and *project* runs – they were identical. A final check was performed that compared the output from the runs by dumping the tables into a text file and comparing them – the output was as expected.

Serial Project	*PRECANREACT*	*PREREACT*	*POSTREACT*	*Reactivate*	*PRECMD*	*Init*	*FINISH*	*Total*
Mean	0.00004	0.00005	0.00002	0.00011	0.00728	0.00739	0.07485	0.08224
Median	0.00004	0.00005	0.00002	0.00011	0.00740	0.00749	0.07641	0.08370
Range	0.00006	0.00005	0.00003	0.00014	0.00905	0.00905	0.11444	0.12148
Minimum	0.00000	0.00003	0.00002	0.00005	0.00268	0.00277	0.02159	0.02468
Maximum	0.00006	0.00008	0.00005	0.00019	0.01173	0.01182	0.13603	0.14616
Count	1000	1000	1000	1000	1000	1000	1000	1000

TABLE 9.4 Serial *Project* Statistical Summary.

Chapter 10

CONCLUSION

"I've thought of a perfect ending for my book, and he lived happily ever after, till the end of his days." – Bilbo to Gandalf upon leaving Bag End[20].

10.1 Effects on Initialization Time

The purpose of providing connection pooling in a database is to avoid the "costly" expense of making a database connection for small queries. It's strange to think that had the true "cost" of a connection to a database been known then there would have been no reason to undertake this experiment. Fortunately the time differences were assumed bad and the project was attempted.

The results for the initialization times are disappointing in a way, since that was the original reason for implementing this functionality. Although, given the data on the initialization times for the *master* database (i.e. they were fast to begin with), they are not a surprise either.

10.2 Effects on Query Time

The improvement on the query/command time was a surprise. It is still a surprise. It is exciting – this is what one hopes to achieve when trying an experiment.

It will be interesting to see what further research unfolds. Will the results be reversed when additional understanding of the entire system is achieved? Will the results change in a different testing environment? Or was there truly something new discovered that may change the design and implementation of databases of the future? Time will tell.

10.3 Where to Go from Here?

Though this is the end (literally – this is the *last* section in this thesis) it is also a beginning. I've had a hard time concentrating on finishing the writing of this report because of the exciting new possibilities. No doubt I am naive in my experience of these things and nothing may come of this after all. But nevertheless I look forward to the next experiment. I enjoy open source and now know what's involved, the preparation required, and the satisfaction it provides. And the strangest result from all of this (at least to me) is that I enjoy writing.

I hope you've enjoyed reading this. Hopefully it's given insight into the possibilities of developing on an open source project, as well as the unexpected hidden treasures that can come from a single experiment.

REFERENCES

[1] Josh Berkus. *PostgreSQL advocacy website.*
`http://advocacy.postgresql.org/about`

[2] Bruce Momjian. *PostgreSQL FAQ.*
`http://www.postgresql.org/docs/faqs/FAQ.html#1.1`

[3] Bruce Momjian. *PostgreSQL To-do List.*
`http://developer.postgresql.org/todo.php`, Startup.

[4] Bruce Momjian. *Connection Pooling To-do Thread History.*
`http://momjian.postgresql.org/cgi-bin/pgtodo?pool`

[5] Josh Berkus. *Individual Members of the PostgreSQL Global Development Group.*
`http://advocacy.postgresql.org/about/#individuals`

[6] Tom Lane. *Memory Contexts.*
`http://www.geocrawler.com/archives/3/10/2000/4/0/3669856/`

[7] TPC Administrator. *TPC-W.*
`http://tpc.org/tpcw/`

[8] Google. *Web Search Engine.*
`http://www.google.com`

[9] die.net. *On-line Linux Man Pages.*
`http://www.die.net/doc/linux/man/`

[10] Santa Cruz Operation, Inc. *Socketpair sample code (UNIX only).*
`http://osr5doc.ca.caldera.com:457/netguide/`
`dusockD.socketpairs_codetext.html`

[11] IBM. *Connection Pooling.*
`http://publib.boulder.ibm.com/infocenter/DB2help/`
`index.jsp?topic=/com.ibm.DB2.udb.doc/conn/c0006170.htm`

[12] PHP Hypertext Preprocessor. *PHP: pg_pconnect - Manual.*
`http://us3.php.net/function.pg-pconnect`

[13] E*TRADE FINANCIAL. *E*TRADE Home.*
`https://us.etrade.com/e/t/home`

[14] Microsoft Knowledge Base. *COM+ ADO Connection Pool.* `http://support.microsoft.com/support/kb/articles/Q266/6/90.asp&NoWebContent=1`

[15] Mormon. *The Book of Mormon.* `http://scriptures.lds.org/2_ne/28/30#30`

[16] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, First Edition, 1993.

[17] Ewald Geschwinde and Hans-Jurgen Schonig. *PostgreSQL Developer's Handbook.* SAMS Publishing, Second Edition, 2002.

[18] Barry Stinson. *PostgreSQL Essential Reference.* New Riders Publishing, First Edition, 2001.

[19] Ewald Geschwinde and Hans-Juergen Schoenig. *PHP and PostgreSQL Advanced Web Programming.* SAMS Publishing, First Edition, 2002.

[20] J.R.R. Tolkien. *The Lord of the Rings, the Fellowship of the Ring.* Houghton Mifflin Co, 2nd edition, 1974.

[21] Andi Kleen. *socket(7).* Linux Man Page, 1999.

[22] Unknown. *cmsg(3).* Linux Man Page, 1998.

[23] Unknown. *socketpair(2).* BSD Man Page, 1993.

[24] Unknown. *socket(2).* Linux Man Page, 1999.

[25] Andi Kleen. *unix(7).* Linux Man Page, 1999.

[26] Ben R. Shelton. *Personal Communication.* May 2004.

[27] Tim Vance. Manpower Professional. *Personal Communication.* May 2004.

Appendix A

DEVELOPMENT ENVIRONMENT

- HP XW8000 with dual Intel Xeon 3.2GHz processors, 4GB RAM, 2-36GB hard drives, and Red Hat 9.0 Linux Workstation for the operating system.
- GNU gcc compiler and other GNU development tools and libraries.
- Graphical Vi IMproved (`GVIM`). Worth mentioning because it has been my best friend through all of this – thank you Bram Moolenaar, it's fantastic.
- Data Display Debugger (`ddd`) with GNU Debugger (`gdb`) for debugging (oh, and let's not forget `printf()` either).
- LaTeX 3.14159 (of course).
- Last but not least, PostgreSQL 7.3.4.

Appendix B

TEST ENVIRONMENT

The following figures contain the SQL commands used for setting up the test database. These tables are a subset of the TPC-W[7] specification.

```
CREATE SEQUENCE address_seq INCREMENT 1 START 1;
CREATE TABLE address (
    addr_id        numeric(10,0)
        DEFAULT NEXTVAL('address_seq'),
    addr_mykey    varchar(20)
        CONSTRAINT address_pk PRIMARY KEY,
    addr_street1    varchar(40),
    addr_city    varchar(30),
    addr_state    varchar(20),
    addr_zip    varchar(10)
);
```

Figure B.1. Table address.

```
CREATE SEQUENCE customer_seq INCREMENT 1 START 1;
CREATE TABLE customer (
    c_id        numeric(10,0)
        DEFAULT NEXTVAL('customer_seq'),
    c_mykey        varchar(20)
        CONSTRAINT customer_pk PRIMARY KEY,
    c_uname        varchar(20),
    c_addr_mykey    varchar(20),
    CONSTRAINT customer_fk1 FOREIGN KEY (c_addr_mykey)
        REFERENCES address (addr_mykey) MATCH FULL
);
```

Figure B.2. Table customer.

```
CREATE SEQUENCE author_seq INCREMENT 1 START 1;
CREATE TABLE author (
    a_id        numeric(10,0)
        DEFAULT NEXTVAL('author_seq'),
    a_mykey        varchar(20)
        CONSTRAINT author_pk PRIMARY KEY,
    a_fname        varchar(20),
    a_lname        varchar(20)
);
```

Figure B.3. Table author.

```
CREATE SEQUENCE item_seq INCREMENT 1 START 1;
CREATE TABLE item (
    i_id        numeric(10,0)
        DEFAULT NEXTVAL('item_seq'),
    i_mykey        varchar(20)
        CONSTRAINT item_pk PRIMARY KEY,
    i_title        varchar(60),
    i_a_mykey    varchar(20),
    i_subject    varchar(60),
    i_cost        numeric(15,2),
    CONSTRAINT item_fk1 FOREIGN KEY (i_a_mykey)
        REFERENCES author (a_mykey) MATCH FULL
);
```

Figure B.4. Table item.

```
CREATE SEQUENCE orders_seq INCREMENT 1 START 1;
CREATE TABLE orders (
    o_id        numeric(10,0)
        DEFAULT NEXTVAL('orders_seq')
        CONSTRAINT orders_pk PRIMARY KEY,
    o_i_mykey    varchar(20),
    o_addr_mykey    varchar(20),
    o_c_mykey    varchar(20),
    o_total        numeric(15,2),
    CONSTRAINT orders_fk1 FOREIGN KEY (o_i_mykey)
        REFERENCES item (i_mykey) MATCH FULL,
    CONSTRAINT orders_fk2 FOREIGN KEY (o_addr_mykey)
        REFERENCES address (addr_mykey),
    CONSTRAINT orders_fk3 FOREIGN KEY (o_c_mykey)
        REFERENCES customer (c_mykey) MATCH FULL
);
```

Figure B.5. Table `orders`.

```
CREATE USER dbuser1;
CREATE USER dbuser2;
CREATE USER dbuser3;
CREATE GROUP users USER dbuser1, dbuser2, dbuser3;
GRANT ALL ON TABLE address TO GROUP users;
GRANT ALL ON address_seq TO GROUP users;
GRANT ALL ON TABLE author TO GROUP users;
GRANT ALL ON author_seq TO GROUP users;
GRANT ALL ON TABLE customer TO GROUP users;
GRANT ALL ON customer_seq TO GROUP users;
GRANT ALL ON TABLE item TO GROUP users;
GRANT ALL ON item_seq TO GROUP users;
GRANT ALL ON TABLE orders TO GROUP users;
GRANT ALL ON orders_seq TO GROUP users;
```

Figure B.6. Users and permissions.

Appendix C

GLOSSARY OR "WHAT WAS THAT AGAIN?"

This appendix describes some of the terms that I'm very familiar with but might be foreign to a first time user of PostgreSQL or even a more advanced user but not necessarily a backend developer.

SQL Structured Query Language.

postmaster This is the main PostgreSQL process. It receives frontend client socket connections and forks a backend to handle authentication and command processing.

backend This is the child process that's forked by the postmaster. It does most of the work.

frontend Literally think of this as *"Not the backend."*

Memory contexts PostgreSQL uses Memory Contexts[6] as a way to simplify the prevention of memory leaks. PostgreSQL gets large chunks of memory from the heap via the OS and then acts as the heap owner for the postmaster and backend processes. Contexts can be put together in a tree structure (thus you could free a tree via single leaf context or somewhere higher up that would recursively free all children contexts).

Stored procedure Most databases have a way for the user to store SQL commands in a procedural format. The purpose being the ability to pre-parse the commands once and then be able to re-use them each time without the need to re-parse them (time savings). In MS SQLServer the language was known as T-SQL and resembled SQL very closely. In PostgreSQL it can be done in psql (also SQL like), C, TCL, JAVA, or Python to name a few (not to mention the interfaces can be extended to work with other languages as needed).

Callback Used to put a function call in a queue that will be handled later. Usually relating to a future event or possible event. Allows flexibility in determining what will be called and when, without having to call all possible callback owners every time an event occurs.

shmem `SH`ared `MEM`ory. A method whereby separate processes can easily share data and/or communicate.

master This is PostgreSQL version 7.3.4 from the `postgres.org` CVS archives.

project Built on the “master” database source code, this contains the additional source files as well as changes to existing files used for implementing the connection pooling system.

Appendix D

PROJECT ARCHIVE

The following website is the location for the archive of this project:
`http://cs.boisestate.edu/∼amit/research/connectionpooling/`

Among other things it contains the following items.

- The full source for PostgreSQL version 7.3.4.
- A patch that when applied to 7.3.4 provides the connection pooling functionality implemented for this experiment.
- A patch for 7.3.4 that provides the timing Macros for comparing the original source with the pooled source.
- Test files used in performing the analysis.
- Several spreadsheets that contain the data gathered from the test results for the analysis.
- A copy of this Thesis (in LaTeX, PostScript, and PDF formats).
- Contact information for the author.
- Any additional results or papers from continued research on this topic.