Multiple Buffer Pools and Dynamic Resizing of Buffer Pools in PostgreSQL

by

Nailah Ogeer

A thesis submitted to the
School of Computing
in conformity with the requirements for the
degree of Master of Science

Queen's University
Kingston, Ontario, Canada
April, 2004

Abstract

An autonomic database management system is a self tuning, self optimizing, self healing and self protecting database management system (DBMS). Since expert database administrators (DBAs) are scarce, introducing a DBMS that is self tuning will decrease the total cost of ownership of the system. In this thesis we present a first step to incorporating self tuning capabilities in the PostgreSQL DBMS.

The buffer area is the main memory management area of the DBMS. Effective use of this area ensures efficiency of the DBMS. Some DBMSs split the buffer area into multiple buffer pools and this has led to performance increases in some cases. Once multiple buffer pools are supported it is up to the DBA to size them based on each of their needs. Optimal sizing leads to good performance, therefore, as the workload changes the DBA has to adjust the sizes of the buffer pools.

We extend PostgreSQL (Version 7.3.2) to support multiple buffer pools. We remove the dependency on the DBA and automatically adjust the sizes of the buffer pools to changes in the environment. We present a number of experiments to verify our approach. These experiments compare throughputs of both the original and modified versions of PostgreSQL running a TPC – B workload and other specifically designed workloads, under various conditions.

Acknowledgements

I would like to thank my supervisor, Dr. Pat Martin and for his guidance and advice throughout the years. His expertise exceeds himself and he has proven himself to be a great professor. He has always supported my work and research at Queen's University.

I would also like to thank our database lab expert, Mrs. Wendy Powley for her advice during my experimental design. Her suggestions were very rewarding. I would like to also thank her for reviewing this thesis. Special thanks also go to my lab mates who have lent an ear many a times when needed.

Thanks also to the School of Computing and the School of Graduate Studies at Queen's University for letting me have the opportunity to pursue this degree and providing financial support throughout the years. I also want to thank IBM Canada Ltd. and CITO for their continuing support to the database laboratory at Queen's University.

I would like to thank my parents and my family for encouraging me to pursue this degree and supporting me all the way. Special thanks to my husband Mr. Shiva Bissoon who has never lost faith in me and encourages me to pursue my dreams.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

Generally there are three areas that can affect the performance of the overall database system, the operating system, the database management system (DBMS) and the database itself. Adjusting the DBMS software installation and configuration to better interface with the operating system is crucial [26]. To obtain the desired performance from the DBMS multiple system dependent parameters need to be tuned. The choice of values for these parameters is based on several factors including the current workload and available resources. Typically, the tuning of these parameters is the job of the database administrator (DBA). This can be a tedious and sometimes redundant task if the DBMS executes the same queries every day. Furthermore, with the increasingly more diverse workloads fed to DBMSs from applications such as e-Commerce, it has become more difficult to tune DBMSs [13] [15]. Hence, database professionals have committed their resources to find a preferable method of tuning these parameters.

Autonomic computing is a popular research area because it attempts to free the DBA from the task of setting system dependent parameters. The term autonomic computing stems from the human autonomic nervous system. The autonomic nervous system

automatically reacts to stimuli from its external environment and frees the conscious brain from the burden of dealing with vital, but lower level, bodily functions [13].

With the increase in complexity of computer technologies, there is a need for similar behavior from computing systems. This need for autonomic computing can be portrayed with a simple present day example. A modern home central heating system and air conditioning unit typically has a simple regime for morning, day and night temperature settings. Normally, the system operates untended and unnoticed; however, users can readily override these settings any time if it is too hot, too cold or they are just trying to save energy. If the system is instrumented with a sensor and knowledge of a family's calendar, the temperature and energy consumption can be optimized to allow for in-house climates and late workdays. Extending this example to more complex systems is quite a challenge as decisions generally include many more variables [31].

Autonomic computing applied to DBMSs presents such a challenge. An Autonomic Database Management System (ADBMS) should have the ability to configure, optimize, heal and protect itself. Systems should automatically adapt to dynamic environments. They should monitor and tune resources automatically to maximize resource utilization and meet users' needs. When a client needs a certain resource the DBMS should automatically provide it to the best of its ability to ensure maximum throughput. An ADBMS should recover, diagnose and react to unexpected disruptions. It should also anticipate, detect and protect itself from outside attacks [13] [15].

Self configuration is a fundamental feature of ADBMSs that focuses on efficiency. In current DBMSs, outside intervention is sought from DBAs who manually tune parameters to provide better performance. ADBMSs should eliminate this dependency.

An ADBMS should automatically determine the settings of these parameters based on the environment and dynamically adjust them to changes in the environment. PostgreSQL is a full feature open source DBMS that is free of charge and readily available for download [25]. The PostgreSQL software itself was developed in 1986 at the University of California at Berkeley as a research prototype, and has moved since then to a globally distributed development model. There is even a commercially-supported version of PostgreSQL which is shipped as part of RedHat Linux [17]. PostgreSQL also supports a large number of programming interfaces, including ODBC, Java (JDBC), Tcl/Tk, PHP, Perl and Python and for this reason has become prevalent in industry and academia.

There are some parameters to tune in PostgreSQL to ensure optimal performance; these include buffer cache size and sort size [26]. The flexibility of choosing the *optimal* sizes may be an advantage for a highly skilled database administrator, but generally the task of choosing the optimal values of system dependent parameters remains cumbersome and error prone.

Of the many parameters that can be tuned in PostgreSQL, memory management is the most important. The buffer pool acts as the memory management unit of the DBMS. Data stored in the buffer pool can be accessed faster than data on disk. Idealistically, frequently accessed data should be kept in the buffer pool so that the average time to retrieve requested data is minimal [11]. However, predicting the exact pages that will be accessed by upcoming queries from an unstable workload is not viable. Several page replacement algorithms have been presented [8] [9] that introduce several approaches to page replacement. Perhaps changing PostgreSQL's least recently used page replacement

algorithm could be one way of keeping the right pages in the buffer pool and improving overall performance.

PostgreSQL provides no support for multiple buffer pools. In this research, we transparently segment the buffer area into multiple buffer pools and maintain a page replacement scheme for each buffer pool. The aim is to reduce the DAT and to increase the utilization of the buffer area. The effects and overhead of segmenting the buffer area into multiple buffer pools will be studied.

Currently PostgreSQL allows the user to increase the total size of the buffer pool. If more buffer area is allocated, there is still a chance that a frequently accessed page can be replaced. For example, when a very large query is executed and the data objects requested by this query occupy the entire buffer pool, pages need to be written out to disk to cater for incoming queries. The question remains of how large of a buffer area should be given to a particular workload. Should more buffers be allocated whenever the system has to execute a query requiring an inordinate amount of memory and if so, what limit is there for such an on demand approach.

The approach implemented in the DB2 Universal Database (DB2/UBD) [14] and Oracle [24] to improve overall performance is to split the single buffer area into multiple buffer pools and to assign each data object in the database system to a particular buffer pool. A query that accesses a set of data objects can be restricted to a buffer pool thus reducing its effects on other queries.

It has been shown that performance can be improved by grouping similar objects into the same buffer pool [11]. The method of grouping data objects into buffer pools was studied by Xu et al [32] and an approach based on clustering was proposed. Another

important aspect of buffer pool configuration is choosing the sizes of each buffer pool. Performance improvement could be achieved by adjusting the sizes of buffer pools based on each of their needs. The optimal sizes of the buffer pools was determined by Tian [30] using a greedy algorithm. Both of these approaches are implemented as stand-alone tools for the DBMS using data collected while running a typical workload.

The availability of the source code for PostgreSQL allows the possibility of new self-tuning techniques into the DBMS itself. The objectives of this thesis are to explore methods of implementing multiple buffer pools in PostgreSQL, incorporating self-tuning technology to dynamically resize the buffer pools in PostgreSQL and then to evaluate the effectiveness of this technology using designed workloads.

The remainder of the thesis is organized as follows. Chapter two outlines the related research conducted in the area of buffer pool management. Chapter three and four describe the design and implementation changes made to PostgreSQL. Chapter five presents a set of experiments to verify our approach. The thesis is summarized and future work is discussed in Chapter six.

Chapter 2 Background and Related Work

Our work in this thesis is a subset of a large set of research on automatic resource management in DBMSs. In Section 2.1, we present background information in this area and introduce autonomic computing and autonomic DBMSs. In Section 2.2, we highlight the importance of the DBMS buffer area and study the role of the buffer manager in the DBMS. We present multiple buffer pools in Section 2.3 and study various algorithms that use multiple buffer pools to increase overall DBMS performance. In the final section of this chapter, we consider the PostgreSQL DBMS and highlight the possibilities for incorporating these ideas into the DBMS.

2.1 Autonomic Computing

Imagine a world where computers can configure themselves and fix their own problems before administrators even know something is wrong. This is the aim of autonomic computing as proposed by IBM [14]. These systems reap a variety of benefits for companies in terms of reduced complexity for informational technology skills and

reduced dependency on human intervention. These benefits lead to accelerated implementation of new capabilities, improved decision making and overall cost savings for businesses [13].

Autonomic computing is a popular research area because it focuses on developing systems that are self tuning, self optimizing, self protecting and self healing. Such systems require less human intervention because they are ‘smart’ enough to configure and protect themselves [13] [15] [31].

Systems have been developed that support autonomic features. One such system is the interactive executive (AIX*) operating system [15]. A logical partition (LPAR) in an IBM pSeries symmetric multiprocessor is a subset of the hardware that can host an operating system instance. The AIX* operating system supports multiple LPARs. Each operating system instance can be managed separately and this leads to increased system performance. Further performance increases can be achieved when spare resources from a particular LPAR are shifted to another LPAR that requires more resources. The first release of LPAR support was static in nature, that is, the reassignment of a resource from one LPAR to another could not be made while the operating system was running.

Currently, the IBM pSeries 690 supports dynamic reassignment of resources across LPARs running AIX*. Dynamic logical partitions (DLPARs) allow reassignment of resources without rebooting the machine. This offers a great deal of flexibility and optimizes the performance and usage of system resources [15]. The ideas of autonomic computing have successfully been applied to the AIX* operating system and have reaped

benefits. The question arises whether improvements to other computing fields can be expected from autonomic computing.

Autonomic computing in the field of DBMSs has become a popular research area. DBMSs such as IBM's Universal Database (DB2/UBD) [14], Oracle [24] and Microsoft's SQL Server [20] have adopted various autonomic features that encompass the four major characteristics of autonomic computing systems; self configuration, self optimization, self healing and self protecting. Our work focuses on the self configuration capability of ADBMSs.

2.1.1 Self Configuration of DBMSs

The performance of a DBMS depends on the configuration of the hardware and software underlying the DBMS. An *autonomic database management system* (ADBMS) should dynamically adapt its configuration to provide acceptable performance. It should also be able to reconfigure itself without severe disruption of online operations [10]. The tools that currently exist assist with the initialization of system parameters that affect performance but little support is provided for dynamic reconfiguration.

Effective memory management, and more specifically management of the buffer area, is of utmost importance in DBMSs. Self configuration of the buffer area is thus crucial to DBMS performance. Microsoft's SQL Server [20] and Oracle [24] both provide some degree of automatic memory management. These systems allocate memory on demand when the system's physical resources run low [10].

One option provided by IBM's DB2/UDB [14] and Oracle [24] is multiple buffer pools. Here the buffer area is segregated into multiple areas or buffer pools. The purpose

of these multiple buffer pools is to restrict data objects to specific buffer pools so that other buffer pools are not accessed or manipulated when a particular data object is referenced. Currently, these multiple buffer pools are manually configured. Autonomic features would map data objects into buffer pools and automatically size the buffer pools as well as remap the buffer pool configuration in face of changes to the workload.

Studying the benefits of segregating the buffer area into multiple buffer pools (similar to the divisions of the hardware into LPARs) is one idea worth examining. The best approach to clustering data objects into multiple buffer pools must also be considered.

It was noticed that shifting resources in the AIX* operating system did yield a performance increase. Dynamically adjusting the sizes of the buffer pools involves shifting buffers from one buffer pool to another, which is analogous to reallocating resources amongst DLPARs. In this thesis, we determine if there is a similar increase in PostgreSQL after adjusting resources according to demand.

2.2 The Role of the DBMS Buffer Manager

DBMSs use a main memory area as a buffer to reduce the number of disk accesses performed by a transaction. A collection of buffers is referred to as a buffer pool. The buffer manager controls the allocation of buffers in the buffer pool and maintains an effective page replacement policy. It decides which pages should be kept in the buffer pool so that the number of accesses to disk is minimized.

A group of data objects reside in a data page. A request for a page by a process is called a *logical read*. If the page does not currently reside in the buffer area, a page fault

or “miss” occurs. In this case, the page is brought into the buffer pool from disk into a buffer as shown in Figure 2- 1.

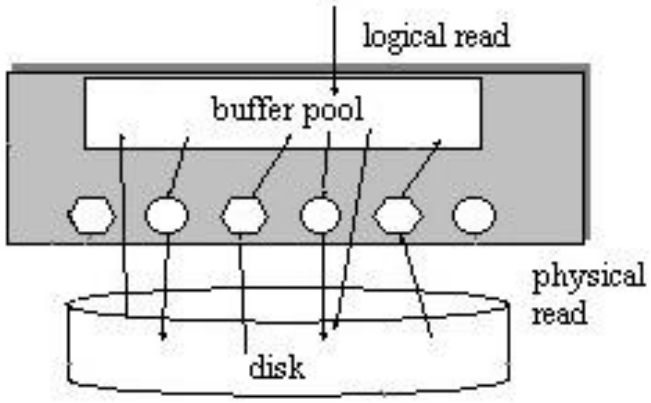

Figure 2- 1 Accessing Requested Pages via a Single Buffer Pool

Retrieval from disk is called a *physical read*. A physical read is very costly because a disk access requires more time than a read from main memory. A physical read may also require page replacement in the buffer pool, thus adding additional cost. Database administrators try to minimize the number of physical reads. This causes an increase in the number of buffer pool ‘hits’ in a given time period (buffer pool *hit rate* increases) and lowers the *Data Access Time* (DAT), or time taken to retrieve the page needed for the execution of a query [30].

When a page replacement occurs, a victim page is selected from the buffer pool and is replaced by a requested page. If the page in the buffer pool is dirty, that is, its contents have been changed, it must be written out to disk. If the page is not dirty, then writing to disk is not necessary and the incoming page replaces the victim page in the buffer pool [8] [9].

Several replacement policies have been developed that try to minimize the DAT. Some keep track of the most frequently and/or recently accessed buffer pages. These

include the least recently used (LRU), most recently used (MRU), least frequently used (LFU) and most frequently used (MFU) [8] [9]. For the LRU and MRU replacement policies, the page that is chosen for replacement is either the least recently accessed or most recently used page, respectively. If a requested data page has a high probability of being accessed again within a short period of time, LRU is useful as the most recent pages are kept in the buffer pool and the least recent pages are replaced. If there is a low probability that a page is accessed right after it has been previously accessed, then MRU is a better replacement algorithm. The same is true for the relationship between LFU and MFU, with the quantitative measure being frequency. Each page has a counter that monitors how frequently it has been used. The least frequently used page is replaced first in LFU and the most frequently used page is replaced first in MRU [8] [9].

One possibility to minimize disk accesses is to use a more robust page replacement algorithm. Several algorithms have been proposed that consider both the recently and frequently used pages. One of these is the LRU-k page replacement algorithm, which keeps track of the last k references to popular database pages [23]. This algorithm measures how recently accessed and frequently used the most popular k pages are and chooses pages that are ‘older’ or less frequently accessed for replacement.

PostgreSQL currently uses the LRU replacement strategy. Presently there is an attempt to integrate LRU-k in hopes of reducing disk contention [25]. Another possibility is to consider multiple buffer pools whereby the buffer manager controls allocation of buffers and page replacement within each buffer pool.

More complex buffer allocation policies have also been proposed. For example, Davison and Graefe attempt to put a price on available buffers [7]. The principle of

maximum profit from economics is analogous to the buffer manager's allocation duties in these policies. A broker assigns currency to certain resources and processes can buy these resources. If two processes request the same resource, the broker has to determine which process is more deserving. If the contesting process can afford to purchase this resource and there is maximum profit in the system after execution of this process, the broker can sell this resource to the process.

Query processing in DBMSs requires the purchase of certain resources in order to execute queries. It is the job of the buffer manager, or 'buffer broker' to enhance the throughput of the system by efficiently utilizing the buffer pool area [7]. Users attempt to execute queries and buffer pages are allocated by the buffer manager to fulfill the needs of the queries.

The number of buffers chosen by the buffer allocation policy to execute a query is based on the demand factor, the buffer availability factor and the dynamic load factor [11]. The demand factor takes into account the space required by the query and assigns more buffers to the query as needed. The buffer availability considers the number of buffers that can be spared for the query and the dynamic load factor determines the characteristics of the queries currently in the system. A good buffer allocation strategy should consider all these factors.

Buffer allocation policies aim to reduce the number of disk accesses by allocating buffers to queries intelligently. Trivial buffer allocation policies such as first in first out (FIFO), least recently used (LRU), random, clock and waiting set [8] [9] only consider the number of buffers available. They fail to take advantage of the specific access patterns of the queries (the demand factor).

Allocation strategies that consider the demand factor include the hot-set model [19] and the DBMIN buffer management algorithm [5]. While the strength of these algorithms lies in their consideration of query access patterns, they are oblivious to buffer availability. The marginal gains algorithm (MG-x-y) proposed by Faloutsos et al [22] is similar to DBMIN proposed by Chou and DeWitt [5] except that the number of available buffers at load control time is taken into consideration. Hence it considers both the demand factor and the number of available buffers.

Faloutsos extends his work with an algorithm that takes into consideration all three factors, which he calls '*predictive load control for flexible buffer allocation*' [11]. This algorithm is different from the others in that it uses predictive methods that use dynamic information so that the minimum number of buffers assigned to a query is not static, but rather a function of the workload.

This dynamic allocation of buffer pages is desirable but the predictive load control approach considers the buffer pool as being divided among concurrently executing queries. In other words, each *query* can be assigned to a buffer pool in the DBMS. Each query is characterized as either a random, sequential, looping or hierarchical reference. A reference to a relation is a sequence of references to pages belonging to the relation. A query that, for example, simply accesses one tuple from a relation is referred to as a random access. In a sequential reference, such as a selection using a clustered index, pages are referenced and processed one after another without repetition. When a sequential reference is performed repeatedly, the reference is called a looping reference. A hierarchical reference is a sequence of page accesses that form a traversal from the root to the leaves of the index. The expected number of page faults, or number of accesses to

disk, caused by a particular reference is a unique mathematical equation for each reference category and combines the probability of a page fault and the number of faults that have been observed [11].

An alternative to classifying the buffer pools by query reference type is classification by *data object* type. This was suggested by Martin et al in the dynamic reconfiguration algorithm (DRF) [18]. Classification by data object type is preferred as queries often share data objects. By segregating queries into the four classifications mentioned earlier (random, sequential, looping and hierarchical) a "wall" is put into place and different queries may have to jump over this wall to access data objects that are also being requested by another query belonging to a different classification.

DRF groups data objects according to their access patterns. There are three types of access patterns - sequential, re-reference and random accesses [32]. Sequential accesses occur when a set of data objects is accessed in sequence, that is, one after the other. An object that is classified as re-reference has been referenced prior to the current access and there is a high probability that it may be accessed again. A random reference is classified as anything other than a sequential or a re-reference access.

The concept of *fragment fencing* was introduced by Brown et al [2]. A fragment is a statistically determined set of queries that have uniform access probabilities. The goal is to achieve response time goals for a fragment by individually controlling hit rates on the fragments. Each fragment has a target residency and the sum of all target residencies is referred to as the resident volume. The aim of the fragment fencing algorithm is to dynamically adjust the target residencies such that fragments in deficit are given more buffers and fragments in surplus give up buffers. The problems with the fragment fencing

prototype are that it exhibits unanticipated overheads with certain DBMSs, it lacks data sharing and it is difficult to expand it to include wider ranges of workloads.

Brown et al [3] revised their approach to compensate for the problems of fragment fencing. The *class fencing* approach allows for data sharing between classes as opposed to the passive fences of fragment fencing. This approach caters to a wide range of workloads (classes) and produces less overhead as less information has to be stored about the classes themselves.

2.3 Multiple Buffer Pools

Segmenting a buffer area into multiple buffer pools has been studied previously. Chou et al [5] separate pages into access types and refer to this as domain separation. Each type is separately managed in its associated domain. When a page of a certain type is needed, a buffer is allocated from the corresponding domain. If none is available, a buffer is borrowed from another domain. This method increases system performance as each buffer pool is managed separately by the system. Pages do not have to compete with dissimilar pages for space in the buffer pool as page replacement is done locally within the buffer pool.

Multiple buffer pools are available in DB2 Universal Database (DB2/UBD) [14] and Oracle [24]. In DB2/UDB, a buffer pool is automatically created for each database. The administrator can create additional buffer pools for each database if required. Oracle provides the option of generating multiple buffer pools but the default is a single buffer pool. Two special purpose buffer pools are available, namely the keep pool and the recycle pool. If there are certain objects that are referenced frequently, they are assigned

to the keep buffer pool. Data objects placed into the recycle buffer pool are those that are scanned rarely or are not referenced frequently. It is the job of the user to decide on how many buffer pools are used and how they are configured.

Figure 2- 2 shows accesses of requested pages. These pages are index or data pages and are being read from or written to disk. An index page is used to retrieve data pages.

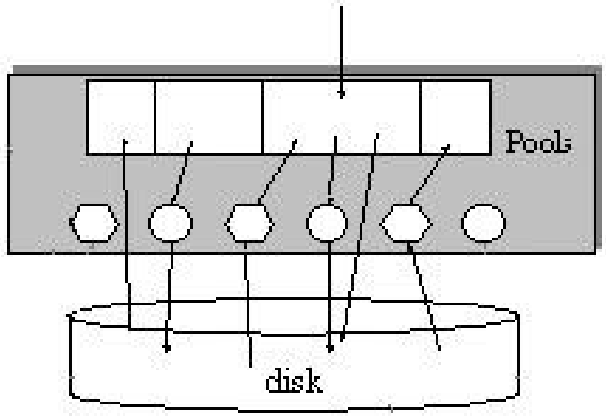

Figure 2- 2 Accessing Requested Pages via Multiple Buffer Pools

Each data object (data table or index) in the database system is assigned to a particular buffer pool. A query that accesses a set of data objects can be restricted to a buffer pool thus reducing its effects on other queries. It was shown that there is reduced disk contention using multiple buffer pools in such cases [14].

PostgreSQL provides no support for multiple buffer pools. In this research, we transparently segment the buffer area into multiple buffer pools and maintain a page replacement scheme for each buffer pool. The aim is to reduce the DAT and to increase the utilization of the buffer area. The effects and overhead of segmenting the buffer area into multiple buffer pools will be studied.

When using multiple buffer pools, the question arises as to how these buffer pools should be formed, that is, which data objects should be assigned to which buffer pools. The mapping of data objects to buffer pools is a clustering problem that has been addressed in previous work [32]. Another topic that needs to be considered is the optimal sizes of these buffer pools. An algorithm that calculates these sizes has been developed [30]. Both of these algorithms have not been integrated into a DBMS.

2.3.1 Clustering Data Objects into Buffer Pools

Configuring multiple buffer pools involves 'clustering' data objects such as data pages and relational index pages together into particular buffer pools. There are several approaches to perform this clustering but it has been shown that performance can be improved by grouping similar objects into the same buffer pool [32]. Data objects can be assigned to different buffer pools based on their similarities with respect to relative size, type (data or index) and typical access patterns (sequential, re-reference or random access).

Segmenting data objects into domains is a clustering problem. Xu et al [32], use the k-means clustering algorithm, which takes k as an input parameter and partitions a set of objects into k clusters. It begins by randomly selecting k objects to initially represent the clusters' center. It then assigns the remaining objects to the most similar cluster measured with respect to the mean value of objects within the cluster. Ideally, there should be a small distance (below a certain threshold) between the object that is placed in the cluster and the current cluster's mean. The k- means algorithm works well for clusters that are separated from one another in terms of their average mean values. However, the

disadvantage is the foresight in choosing the right number of clusters (k). The k- means algorithm is also sensitive to noise and outlier objects since these values can drastically increase or decrease the clusters' mean [32].

Xu studied various clustering algorithms including k-means, partitioning around medoids (PAM) and the divisive hierarchical clustering algorithm and determined that the k-means algorithm was the most robust in terms of clustering data objects into buffer pools. This algorithm was implemented as a stand-alone tool for DB2/UDB using trace data collected while running a typical workload.

2.3.2 Calculating Optimal Sizes of Buffer Pools

A good buffer manager considers the demand factor of the data objects and carefully tunes the buffer pools associated with these data objects so that there is neither a deficit nor a surplus of buffers for any one buffer pool. The buffer pool sizing problem has been addressed by Tian [30] who uses a greedy approach to calculate the optimal sizes of the buffer pools.

Intuitively, buffers that reside in one buffer pool that are not being used can be devoted to another buffer pool where there is a greater demand for buffers. Tian [30] attempts to determine the optimal sizes of the buffer pools provided that the total number of buffer pools is known. His algorithm depends on performance data collected by the DBMS and uses performance measures such as DAT and hit rate.

2.4 Possibilities using PostgreSQL

A truly autonomic DBMS should automatically determine the appropriate number of buffer pools and automatically determine which data objects are grouped together into a buffer pool. Appropriate sizes of these buffer pools should also be chosen. The ADBMS should monitor the performance and dynamically change the buffer pool configuration in face of changing workloads.

To incorporate the ideas proposed above, the algorithm proposed by Xu [32] can be incorporated into the DBMS source code so that the clustering is done automatically and transparently. Furthermore, the data objects assigned to each cluster may change as the workload changes, thus requiring an internal feature that detects a workload change and dynamically adjusts the contents of the buffer pools.

Tian's sizing algorithm was also implemented as a stand-alone tool. To be truly autonomic, the resizing should be done internally by the DBMS. The optimal sizes of each buffer pool may change as the workload varies. Building an internal feature that detects this change in a workload and dynamically resizes the buffer pools should lead to better overall performance.

PostgreSQL is an open source DBMS and, for this reason, it is an ideal candidate for incorporating the autonomic features described above. The properties of the data objects can be collected while the DBMS is running. These algorithms can be integrated into the DBMS and new configurations suggested by these algorithms can be installed while the

system is running. Also changes in workload can be monitored and dynamic adjustments in the DBMS can be made to reflect on variations in workload.

Dynamically calculating the sizes and resizing multiple buffer pools would be one step further to a DBMS that is self configuring. For this thesis, we examine the dynamic resizing of multiple buffer pools within PostgreSQL. For this work, we assume that the clustering of data objects within buffer pools is known and constant.

Chapter 3 Design and Implementation for Integration of Multiple Buffer Pools in PostgreSQL (V 7.3.2)

In Section 3.1 we present an overview of the internals of the PostgreSQL DBMS and in Section 3.2 more specifically examine memory management in PostgreSQL. We then present a method to create multiple buffer pools in Section 3.3 and show how we are able to maintain them.

3.1 PostgreSQL Internals

PostgreSQL is an open source DBMS comprised of several components including the *postmaster*, the *postgres server* and *shared memory*. The postmaster manages system wide operations such as startup, shutdown and periodic checkpoints. The postmaster does not do these operations but forks out commands to other PostgreSQL internal

components. The postmaster also creates shared memory components such as the buffer pool and the statistics collector. The relationship between the postmaster and other internal components is shown in Figure 3- 1.

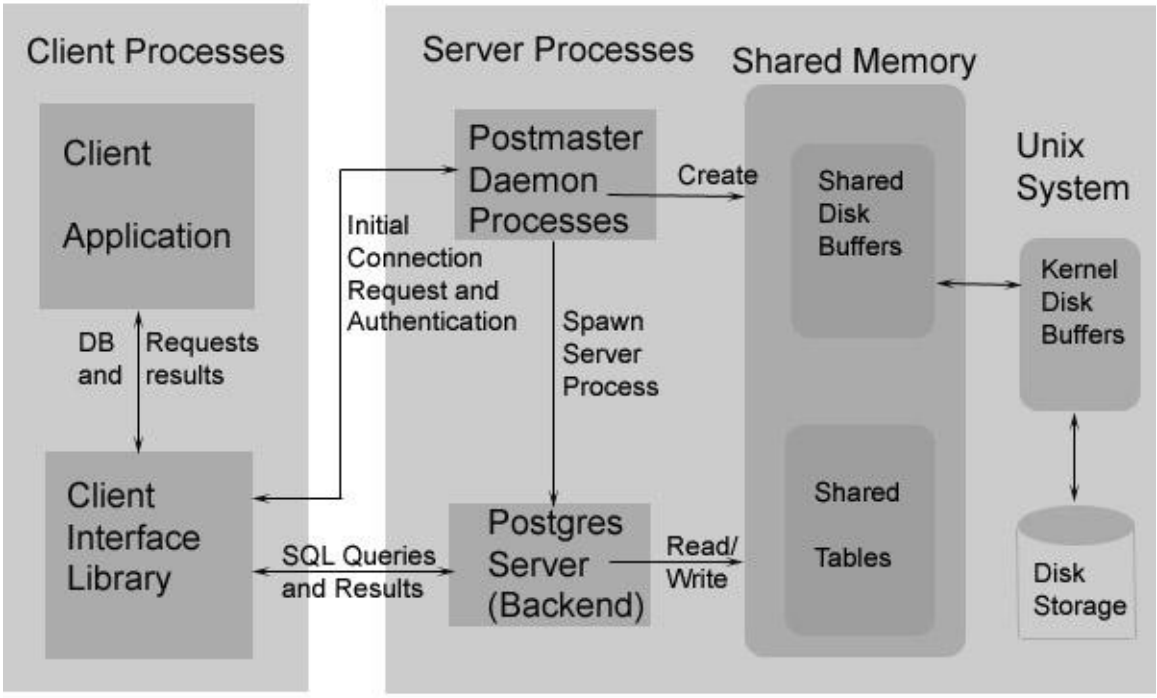

Figure 3- 1 PostgreSQL Client/ Server Communication [25]

The Postgres Server is also referred to as the *backend*. The backend is responsible for ensuring that SQL queries passed from a client are valid. It also generates results of these queries by accessing shared tables and passes the results back to the client.

The backend is composed of four major stages. These stages make up the 'compiler' of the DBMS and as can be seen in Figure 3- 2, include the *parser*, *rewriter*, *planner* and *executor*. The parser analyzes the syntax of the query and reports if there are any syntactical errors. It also looks up object definitions that may be used at this stage. The

parser creates a *parse tree* using these object definitions and passes it to the rewriter. The rewriter retrieves any rules specific to tables or views accessed by the query. It rewrites the parse tree using these rules and passes the rewritten parse tree to the planner. The planner or the 'optimizer' finds the most optimal path for the execution of the query by looking at statistics collected on relations accessed by the query. A plan for execution of the query, called the *plan tree,* is passed to the executor. The main function of the executor is to fetch data needed by the query and pass this data to the client.

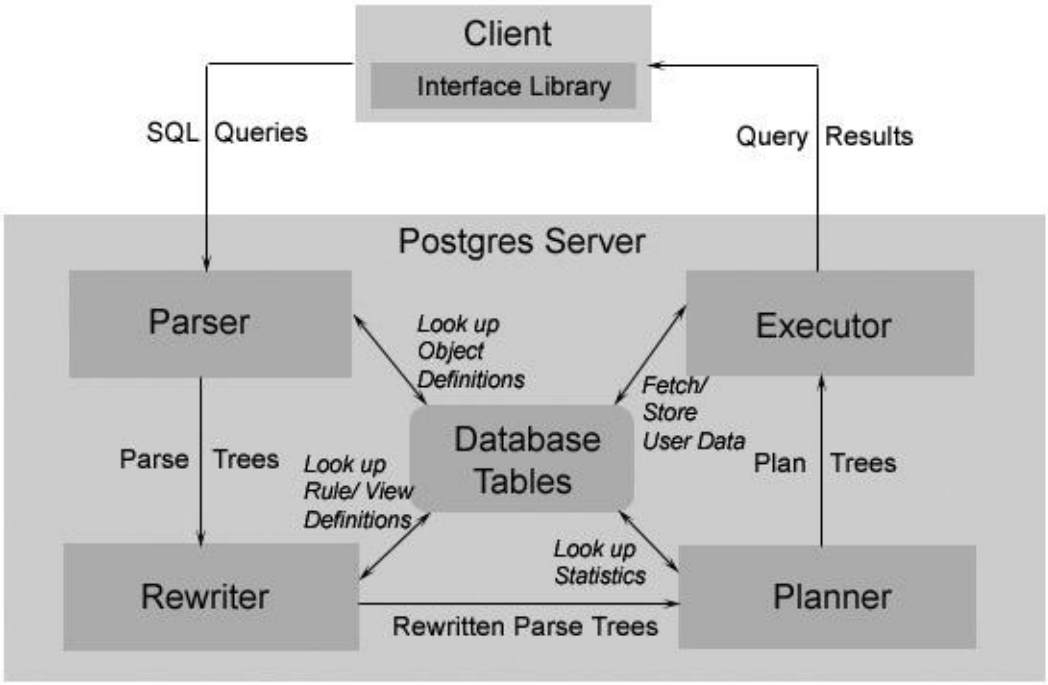

Figure 3- 2 Postgres Server [25]

3.2 PostgreSQL Memory Management

Similar to other DBMSs, PostgreSQL stores large amounts of data on disk. This data has to be brought into the buffer area to be accessed by queries. Reading from and

writing to the disk is expensive and database administrators try to minimize this as much as possible. A read from disk, referred to as a physical read, is made when a referenced page is not currently in the buffer area. A write to disk occurs during a page replacement when the page being replaced has been ‘dirtied’ and must be updated on the disk.

PostgreSQL uses two types of memory, local and shared memory. Local memory is recyclable; if a particular data object is no longer needed, its local memory location can be freed and assigned to another object while the system is running. On the other hand, a shared memory location is not recyclable; once it has been requested by and assigned to a data object this memory location is specific to that object until the system is restarted.

Shared memory is used for data objects that are to be analyzed by internal components while the DBMS is running. There are several hash tables used by system components that make the retrieval of information in shared memory more efficient. Changes can also be made to shared memory by these internal components. The changes affect all components as well as all clients.

An example of the use of shared memory is the collection of statistics. At initialization, an area in shared memory is set aside to maintain statistics on all relational objects. Information such as the number of tuples fetched, the number of tuples deleted and the number of tuples inserted is kept for each data object in the database. At predefined intervals, statistics are summarized and stored in shared memory. If statistics were kept in local memory, they would be over written when one process finishes and another one starts. PostgreSQL’s statistics collector is studied in a later section.

PostgreSQL also uses shared memory for the buffer area. The buffer area is made up of a number of shared buffers. The buffers are called shared buffers because they are

allocated in shared memory and are accessible by multiple clients. The minimum number of shared buffers is 16, each 8 kilobytes in size, and the maximum is dependent on the shared memory settings of the system kernel.

3.2.1 Kernel File Cache Storage

There exists an intermediary device in PostgreSQL called the kernel file cache which consists of *kernel disk buffers* [25]. The main purpose of this device is to act as a cache for buffer pages. Instead of writing directly to disk, pages are written to the kernel disk buffers. These 'temporary' kernel disk buffers act as an extension to the buffer pool. If a page is not in the buffer pool, the kernel file cache is searched. Only if the page does not reside in a kernel disk buffer is a read from disk necessary. This infrastructure increases system performance but increases the dependence on the operating system.

3.2.2 The PostgreSQL Buffer Pool

The buffer pool (or the memory management unit of the DBMS) is governed by the buffer manager, which is responsible for the allocation of data pages to buffers. The buffer pool consists of a number of shared buffers that store blocks of data or pages requested by queries. PostgreSQL (V 7.3.2) uses a single buffer pool. The buffer pool's size is initialized as a constant at postmaster start up. This size cannot grow while the DBMS is running. If the buffer pool is full and more buffers are needed by a query, the buffer manager must decide what page(s) to replace. The buffer manager in PostgreSQL uses a LRU page replacement policy where the least recently used page is replaced by a

new page from the disk when no free pages are available in the buffer pool to satisfy a query request.

The buffer pool consists of *pinned* and *unpinned* shared buffers. The pinned buffers are those that are currently being used by the DBMS. These buffers can not be selected as victims for replacement. The remaining buffers, the unpinned buffers, are kept in a circular doubly linked list called the *free list*. The free list can be accessed via the head of the free list queue. Pinned buffers must be unpinned and placed in the free list to be accessible by the replacement algorithm. Most recently used buffers are added to the tail of the free list which means that the first buffer selected by LRU is always the buffer closest to the head of the list.

In Figure 3- 3, there are 8 buffers in the buffer pool numbered 0-7. Buffers 1, 2 and 6 are currently being used by the DBMS and are thus pinned. Buffers 0, 3, 4, 5 and 7 are not being used so they are stored in the free list, which is maintained in a doubly linked list. The next page selected as a victim by the page replacement algorithm is the page that currently resides in buffer 0. If this buffer contains a ‘dirty’ page (page that has been updated by the DBMS), it must be written out to disk before a new page replaces it.

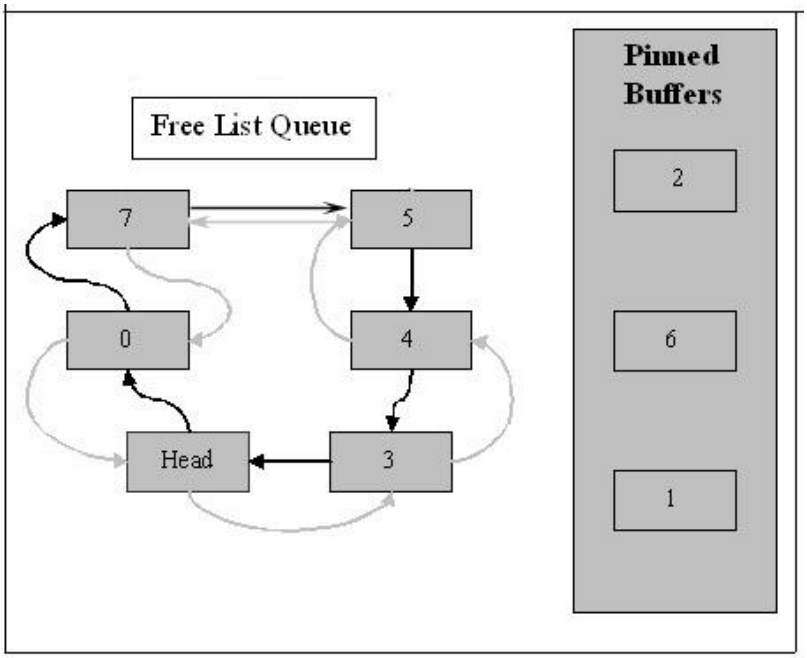

Figure 3- 3 Pinned and Unpinned Buffers of the Buffer Pool

Data is segmented into equally sized *blocks* or pages in the DBMS. When a page is referenced, it has to be brought into a buffer in the buffer pool. To keep track of the buffers an array called *Buffer Descriptors* is maintained in shared memory. It is essential that this array resides in shared memory because it must be accessible from all internal components.

Each entry of the Buffer Descriptors array is called a *Buffer Descriptor* and maintains information on the contents of a buffer. The main contents are a pointer to the data page originally brought in from disk and a *buffer tag*, which identifies the data object and the page identification associated with this buffer. Other entries include the buffer identification, the number of times this buffer has been referenced, whether the buffer is currently being accessed and whether the buffer is 'dirty' and has to be written to disk.

Next and previous pointers in the free list queue are also stored. The structure definition of a Buffer Descriptor is given in Appendix A.

For easy lookup of a block of data by the buffer manager, a *shared buffer hash table* is maintained. The hash table determines whether or not a requested page is already in the buffer pool. Figure 3- 4 shows the sequence of steps taken by the buffer manager on page look up and retrieval. The shared buffer hash table is used to look for a requested page. If the page is found in the hash table, it is present in the buffer pool and the buffer block or page associated with the buffer can be retrieved. If it is not present in the hash table, it has to be retrieved from disk.

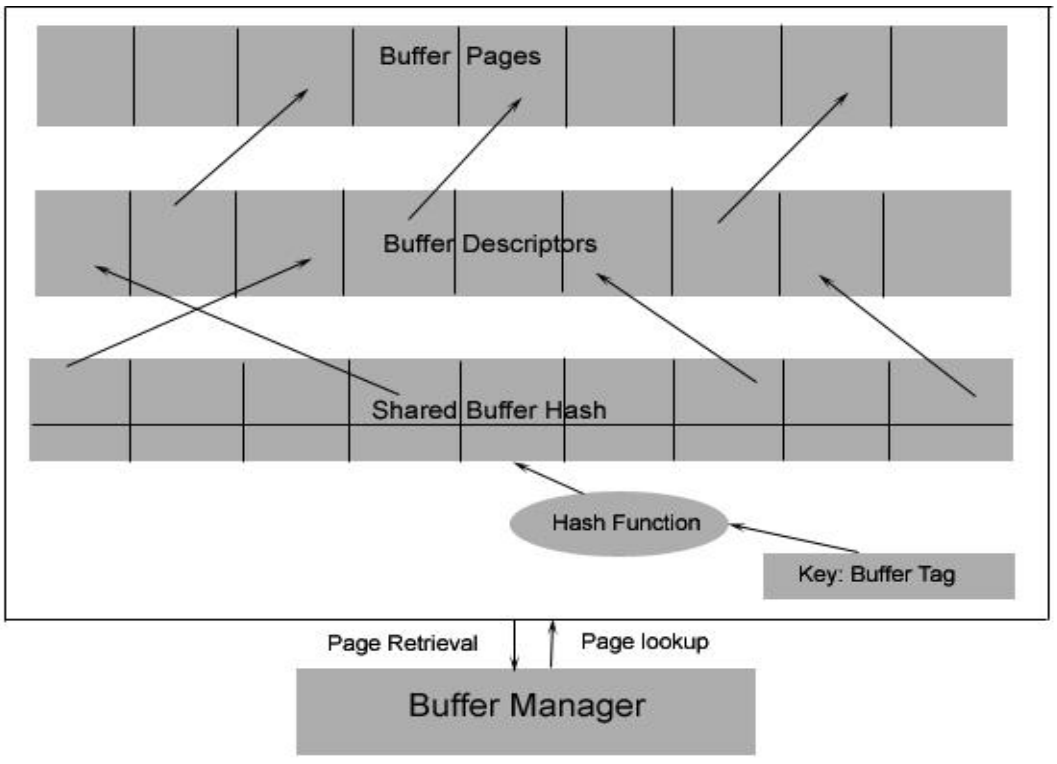

Figure 3- 4 Buffer Manager Look Up and Retrieval of Requested Pages

3.3 Maintaining Multiple Buffer Pools in PostgreSQL

To create multiple buffer pools, all the buffer pages have to be divided into multiple groups of buffer pages. To keep track of the buffer pools, a new shared memory array, called the Buffer Pool Descriptors, is created at postmaster startup. The size of this array is the number of buffer pools requested by the system. Each Buffer Pool Descriptor (BPDesc) (see Appendix A) has an integer representing the buffer pool identification, an integer variable storing the number of buffers in the buffer pool, and a pointer to the head of the buffer pool free list. A shared memory linked list is also created for each buffer pool that stores the identification of all data objects belonging to this buffer pool. This is created so that statistics summarizing all the data objects in the buffer pool can be maintained. We refer to it as the Object Queue (see Appendix A). The linked list must reside in shared memory so that it can be accessible from all internal components.

The database consists of a number of data objects, namely tables and related indices, and each data object is mapped to a particular buffer pool and buffer usage is restricted to the assigned buffer pool. Many objects may share a buffer pool. Clustering of objects together into a buffer pool is based on a number of characteristics including relative size of the object, typical type of access (random, sequential or re-reference) and object type. Relations and their indices are assigned to buffer pools when they are created in the DBMS. For the purpose of our work, we statically assign objects to buffer pools based on our knowledge of the workload. In an autonomic system, a clustering algorithm such as that of Xu [32] could be implemented to dynamically cluster the objects and to adjust the clustering if necessary. The implementation of such an algorithm is beyond the scope of this thesis.

A data object in the DBMS is either a relation or an index page. An Object Descriptor (see Appendix A) stores information about a data object that has been accessed. This information includes the object identification number, statistics about the object, and the number of times the object has been accessed. To keep track of which objects are assigned to which buffer pools, a buffer pool identification number is added to each Object Descriptor.

Every data page is associated with a buffer in the buffer pool (refer to Figure 3- 4). Information is kept on the buffers using the Buffer Descriptor (BufDesc) data structure (see Appendix A). To identify which buffers belong to which buffer pool, a new field is added to the Buffer Descriptor data structure. This field stores the buffer pool identification and is assigned at the start of the DBMS. This field is not constant and can be changed later on by the DBMS.

To create k buffer pools, we split the single buffer area into k groups of buffer pages. Each buffer pool has its own free list queue as well as a list of pinned buffers. Figure 3- 5 shows the procedure taken by the buffer manager to initialize the buffer pools. The total number of buffers and the number of required buffer pools are parameters of the algorithm. The number of descriptors in each buffer pool is a distribution of the number of buffers in the total buffer area. Each buffer has to be assigned a buffer pool and a previous and next buffer in its associated buffer pool free list. Initially all buffers are free as they are not being used to retrieve data, so they all belong to the free lists of their respective buffer pools. If the buffers can not be evenly distributed the buffer pool free list is broken and the remaining buffers are added to the free list of the last buffer pool.

```
InitializeBufferPools (Number of buffers, Number of buffer pools) {
    BufferDesc *buf = BufferDescriptors;      // first buffer in the buffer descriptor array
    assign space for BufferPoolDescriptors ('BPDescriptors');
    GNumBuffers= floor (number of buffers/ number of buffer pools);
    for (j=0; bp = start of BPDescriptors to end of BPDescriptors, j++){
        for (i = 0 to GNumBuffers, buf ++){
            set next and previous pointers of i^th buffer in the free list queue;
            buf->buf_id = i + counter;
            buf->bp_id= j;          //set the buffer pool associated with this buffer
            initialize other fields;}
        close the circular queue;
    bp->bp_id = j;                              // initialize buffer pool descriptor
    allocate new object queue;
    number of descriptors in buffer pool = GNumBuffers;
    initialize the statistics for the buffer pool;
    counter = counter + GNumBuffers;}
    if(counter < Number of buffers){            // put extra buffers in the last buffer pool
        break the last buffer pool's circular queue;
        for (i = last buffer assigned to a buffer pool to Number of buffers)
    set next and previous pointers of i^th buffer in the free list queue;
    close the circular queue;}
}
```

Figure 3- 5 Algorithm that Initializes the Buffer Pools

If a single buffer area consists of 16 buffers and we split the buffer area into four buffer pools, each buffer pool will have 4 buffers. If there was a total of 17 buffers, the first three buffer pools will have 4 buffers and the last buffer pool will have 5 buffers.

The algorithm in Figure 3- 5 has a time complexity of θ (N_b) where N_b is the number of buffers and a space complexity of θ (N_{bp}) where N_{bp} is the number of buffer pools. If we increase the number of buffer pools no extra time is spent creating the additional buffer pools but more space is required to store extra buffer pool information such as statistics on the buffer pools.

We also expect CPU overhead because the buffer pool descriptor associated with each data object has to be determined every time a data object is requested so that the buffer manager knows the buffer pool designated for that object. The algorithm in Figure 3- 6 outlines the procedure taken by the buffer manager to determine the buffer pool descriptor associated with a requested data object.

```
BPDesc FindBufferPoolDescriptor (data object) {
    Integer dobjbp = data object-> buffer pool id; // assigned when data object is created
    for (BPDesc bp = start of BPDescriptors to end of BPDescriptors){
       if (dobjbp = = bp->buffer pool id)
                return bp;
  }
}
```

Figure 3- 6 Algorithm to Find the Buffer Pool Descriptor Associated with a Data Object

The time complexity associated with determining the buffer pool descriptor for a data object is $\theta(N_{bp})$ where N_{bp} is the number of buffer pools. If the number of buffer pools increases, the time taken to determine the buffer pool descriptor associated with the data object also increases.

Once the buffer pool descriptor is found, the buffer manager transfers the data page associated with the data object from disk to this buffer pool. Upon subsequent requests for the data object, the buffer manager searches only in that buffer pool to find the requested data object. As the number of buffer pools increases, the number of buffers per buffer pool decreases (if the total number of buffers is fixed). Therefore, the time complexity required to search for a requested data object decreases by a factor of N_{bp} (for N_{bp} number of buffer pools) for every additional buffer pool. Therefore, during a search for a data object, the overhead caused by the first algorithm is counterbalanced by the improvements in the second algorithm.

Table 3- 1 summarizes the overall complexities for initializing the buffer pools and finding a data object in the buffer area. N_b is the number of buffers, N_{bp} is the number of buffer pools and N_{doBA} is the number of data objects in the buffer area.

Procedure	**Time Complexity**	**Additional Space Complexity**
Initialize Buffer Pools	$\theta (N_b)$	$\theta (N_{bp})$
Find a Data Object	$\theta (N_{bp}) + \theta (N_{doBA}/ N_{bp})$	0

Table 3- 1 Complexities of Algorithms used To Support Multiple Buffer Pools

Chapter 4 Design and Implementation of Dynamic Resizing of Buffer Pools in PostgreSQL (V 7.3.2)

Once the DBMS supports multiple buffer pools, the optimal sizes of these buffer pools have to be determined. In Section 4.1, we present the buffer pool sizing algorithm. In Section 4.2, we discuss additions to the PostgreSQL statistics collector to provide the statistics required by the algorithm. Section 4.3 addresses the integration of the sizing algorithm into PostgreSQL.

4.1 Buffer Pool Sizing Algorithm

The objective of Tian's algorithm [30] is to calculate optimal sizes of buffer pools in a DBMS so that the cost of retrieving data is minimized. Tian experiments with two cost measures; namely hit rate and data access time (DAT). He found DAT to be a more

suitable measure, thus in this work, we also focus on minimizing the DAT for retrieval of data objects.

A *buffer pool state* is a configuration of the sizes of all the buffer pools. For example, $<s_1, s_2 \ldots s_k \ldots s_n>$ is a possible state where s_k is the size of the k^{th} buffer pool, $s_k \geq 1$, and there are n buffer pools. For state S, a neighboring state S' differs from S by a shift of delta (Δ) pages from some buffer pool i to another pool j while all other buffer pools remain the same size. For example, if Δ is 1 and the current state is <1, 3, 2> then <1, 4, 1>, <2, 3, 1>, <2, 2, 2> and <1, 2, 3> are its neighboring states.

Tian's algorithm is based on the concept that if an optimal state is not met, there exists at least one neighbor state that is more optimal. The average DAT for a requested data object is estimated for each neighbor state at each step of the algorithm. If the current state does not provide the lowest average DAT then the 'best' neighbor state is selected.

In Figure 4 - 1, for example, there are a total of 6 buffers that are separated into three buffer pools. The current state is <1, 3, 2> which yields an average DAT of 260 milliseconds. The best neighboring state is determined to be <2, 3, 1> because it yields the lowest DAT of all <1, 3, 2>'s neighboring states (206 milliseconds). The best neighboring state of state <2, 3, 1> is <3, 2, 1> which yields an average DAT of 205 milliseconds. The configuration <4, 1, 1> yields the lowest estimated average DAT of 180 milliseconds and because all its neighboring states do not yield a lower average DAT, this is the final state returned by the algorithm.

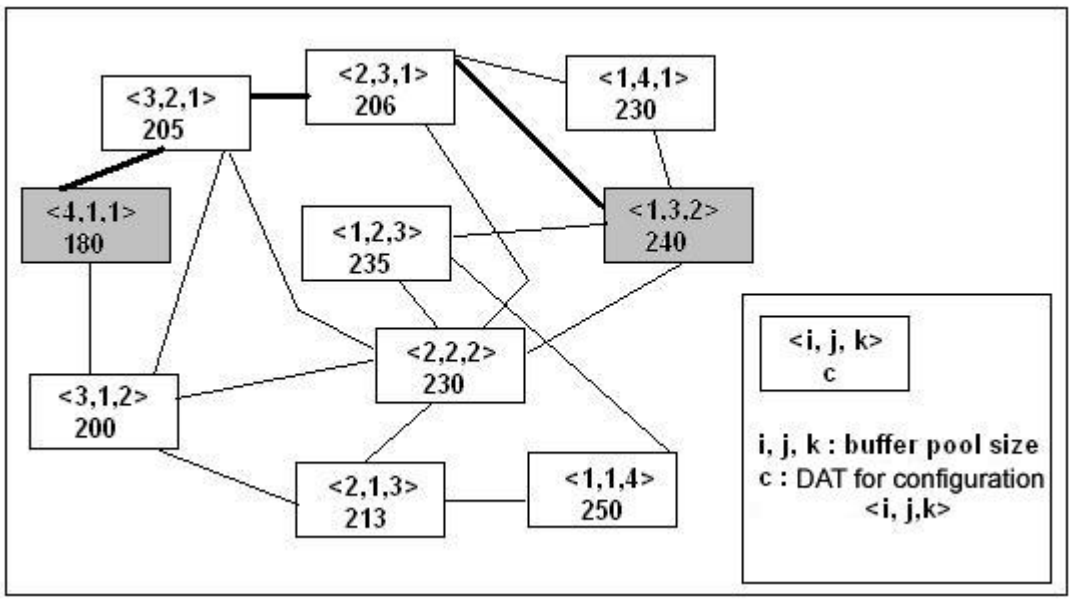

Figure 4 - 1 Example Search Space for Configuration with the Lowest DAT

Tian's sizing algorithm uses a greedy approach to estimate the optimal sizes of the buffer pools. A justification of the results of this algorithm is given in his work [30]. The algorithm requires two sets of statistics collected for each buffer pool at different buffer pool states. Information such as the sizes of the buffer pools, the number of logical reads requested from each buffer pool, the number of physical reads and the average DAT (measured in milliseconds) incurred from a logical read are collected for each buffer pool. The state that acquires the minimum average DAT is returned as the optimal state. Figure 4 - 2 outlines the algorithm.

```
SizingBufferPools (statistics from 2 different states for each buffer pool){
        S = initial state;
        found = false;
        repeat
            get all S's neighbor states (N_S);
           choose S'' from N_S such that cost(S'')< cost(v), for all v Є N_S;
        if (cost(S'') < cost(S))
             S = S'';
       else found= true;
      optimum = S;
      return optimum;
}
```

Figure 4 - 2 Tian's Algorithm [30]

The input to this algorithm is two sets of statistics from each buffer pool. The algorithm iterates through all neighboring states of the buffer pools for each current state. An estimate of the time complexity of the algorithm is worst case Θ (Num_S) where Num_S is the number of states. The number of possible states is given by

$$Num_S = (N/\Delta\text{-}1)! / [(N / \Delta\text{-}N_{bp})! \times (N_{bp}\text{-}1)!] \qquad \textbf{(Equation 4 - 1)}$$

according to Tian [30], where N_{bp} is the number of buffer pools, N is the number of buffers and Δ is the user defined transferable number of buffer pages between two buffers. As we increase the number of buffer pools, the number of states increases. For example if Δ is 5, N is 6 and N_{bp} is 2, the number of states is 4 but if we increase N_{bp} to 3, the number of states increases to 6.

4.1.1 Estimating Performance Measures

Two performance measures used in Tian's sizing algorithm are buffer pool hit rate and DAT for requested data objects. Hit rate is defined as a ratio of the number of physical reads to the number of logical reads. Ideally the hit rate of a buffer pool should be one as this guarantees that all requested data pages are found in the buffer pool. Generally the hit rate is in the range 0-1. For a buffer pool with size s, Tian [30] models buffer pool hit rate function HR(s) as a function of a, b and s where a and b are specific to a particular combination of workload and buffer page replacement policy. The buffer pool hit rate function is stated in Equation 4-2.

$$HR(s) = 1 - a \times s^{b} \qquad \textbf{(Equation 4 - 2)}$$

Two different hit rates must be collected from two different buffer pools states and substituted in Equation 4 - 2 to calculate a and b.

In this work, average DAT is used as the performance measure, so a method of predicting the DAT incurred by each buffer pool state is required. An approach to estimating DAT for requested data objects was presented in the dynamic reconfiguration algorithm (DRF) proposed by Martin et al [18]. DRF uses a least squares approximation, curve fitting technique. This method is simplified by Tian [30] who calculates DAT using Equation 4-3, where costLR is the average time required to perform a logical read and noLR is the number of logical reads.

$$\text{Total DAT} = \text{costLR} \times \text{noLR} \qquad \textbf{(Equation 4 - 3)}$$

Memory access time is negligible compared to disk access time. Tian [30] proposed that DAT can be calculated solely from disk accesses and derived Equation 4 - 4 where costPR is the average time required to perform a physical read and noPR is the number of physical reads.

$$\text{Total DAT} = \text{costLR} \times \text{noLR} = \text{costPR} \times \text{noPR} \qquad \textbf{(Equation 4 - 4)}$$

By converting this equation, we obtain

$$\text{costLR} = (\text{noPR} / \text{noLR}) \times \text{costPR} \qquad \textbf{(Equation 4 - 5)}$$

where (noPR / noLR) is the miss rate of the buffer pool, equivalent to 1- HR. The equation can be further reduced to

$$\text{costLR} = (1 - \text{HR}) \times \text{costPR}. \qquad \textbf{(Equation 4 - 6)}$$

For a specific database container (device), the cost to perform a physical read is assumed to be fixed [18]. Therefore equation 4 - 6 represents a linear function between buffer pool hit rate and average DAT (costLR) of requested data objects. A linear function can be represented in the form $f(x) = kx+c$.

Average DAT is a function of hit rate, which is modeled by Belady's equation [1]. Therefore, average DAT for a requested data object that is mapped to a buffer pool with size s can be estimated by the following equation:

$$\text{Average DAT}(s) = \text{costPR} - (\text{HR}(s) \times \text{costPR}) \qquad \textbf{(Equation 4 - 7)}$$

Figure 4 -3 shows the ideal relationship between hit rate and average DAT. The highest attainable buffer pool hit rate is 1.0 and at this point the average DAT must be

zero. The lowest buffer pool hit rate is 0 where average DAT is the cost to perform a physical read.

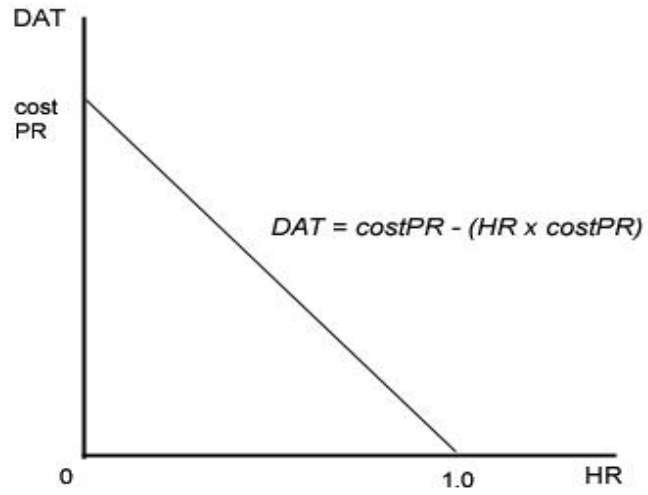

Figure 4 - 3 Ideal Linear Relationship between Average DAT and Hit Rate

4.2 Additions to the PostgreSQL Statistics Collector

The statistics collector is a new feature that comes with the PostgreSQL (V 7.3.2) package. The role of the statistics collector is to store statistics about server activity. Statistics are reported on databases, relations, indices and clients connected to the DBMS.

Some callable SQL functions built into PostgreSQL (V 7.3.2) include functions that report on the number of tuples fetched, deleted, inserted and updated for each relation or index. Statistics such as the number of clients connected to the database or the number of transactions committed or rolled back in the database can also be retrieved. A function is also available that resets all the statistics in the DBMS.

Additional functions are added to the statistics collector so that extra statistics can be reported for data objects, databases and buffer pools. This information includes the number of blocks fetched from a data object (the number of logical reads), the number of

blocks requested that do not reside in the buffer pool (the number of physical reads) and the DAT incurred to fetch a data object (measured in milliseconds). The data objects are identified by their unique object ids. Each database also has a unique database identification, which can be used to retrieve the number of logical reads requested on a database, the total number of physical reads and the total DAT for all the objects belonging to the database.

The number of blocks that do not reside in the buffer pool includes all the blocks that reside in the kernel file cache and the disk. The time taken to retrieve a data object from the buffer pool is negligible but the DAT needed to retrieve a data object from the file cache or the disk is much larger. Most other DBMSs do not have a kernel file cache, so we treat the file cache like a second disk.

Since we split the single buffer area into multiple buffer pools, statistics need to be gathered about each buffer pool. This is attained by summing the statistics of all the data objects that reside in the buffer pools as shown in Figure 4 - 4. To calculate the number of logical reads for a buffer pool, the number of logical reads for all the data objects associated with this buffer pool are summed. Similarly the number of physical reads incurred by these requests and the DAT to retrieve these objects are also calculated.

```
CalculateBufferPoolsStats( ){
    for (BPDesc *bp= BPDescriptors to number of buffer pools)
      // initialize the fields to store the statistics
      bp->LR, bp->PR, bp->DAT = 0;
     // access the object queues of each buffer pool
     for (j = 0 to object queue size){
       nextPtr = next object id in the queue;
       access the object descriptor;
       bp->LR = number of logical reads for object descriptor + bp->LR;
       bp->PR = number of physical reads for object descriptor + bp-> PR;
       bp->DAT = time for a logical read for object descriptor + bp-> DAT;
   }
}
```

Figure 4 - 4 Algorithm to Calculate the Statistics for the Buffer Pools

The time complexity of this algorithm is θ (N) where N is the number of data objects in the buffer area. The space complexity is θ (N_{bp}) since statistics are stored on each buffer pool. If there is one buffer pool, all the data objects will be assigned to this buffer pool. As the number of buffer pools increases, the data objects are divided among the buffer pools since the entire buffer area is fixed. Therefore, if the number of buffer pools is increased, the collection of statistics has the same time complexity; however more space is required to store information about the buffer pools.

4.3 Integration of the Buffer Pool Sizing Algorithm in PostgreSQL

4.3.1 Collecting Sample Statistics and Execution of Sizing Algorithm

To execute Tian's sizing algorithm, statistics for all buffer pools from two different buffer pool states have to be collected. We collect these statistics using the procedure shown in Figure 4 - 5. Note that this procedure is only necessary at initial startup for each workload and calculates the initial sizes of the buffer pools. These sizes may change later on if the workload changes. The DBMS is first given sufficient time to warm up (x transactions are executed) and all the statistics are reset. The system is allowed to execute y transactions and the first set of statistics (S1) is collected under buffer pool state a for each buffer pool. We collect these statistics using the algorithm described in Figure 4 - 4. The sizes of the buffer pools are then changed (to a different buffer pool state - b) and the system is given time to warm up (x transactions are executed) under the new configuration. The statistics are reset, the system is allowed to run for the same period (y transactions are executed) and the second set of statistics (S2) is collected under buffer pool state b. The sizing algorithm is then executed and the optimal sizes of the buffer pools are calculated.

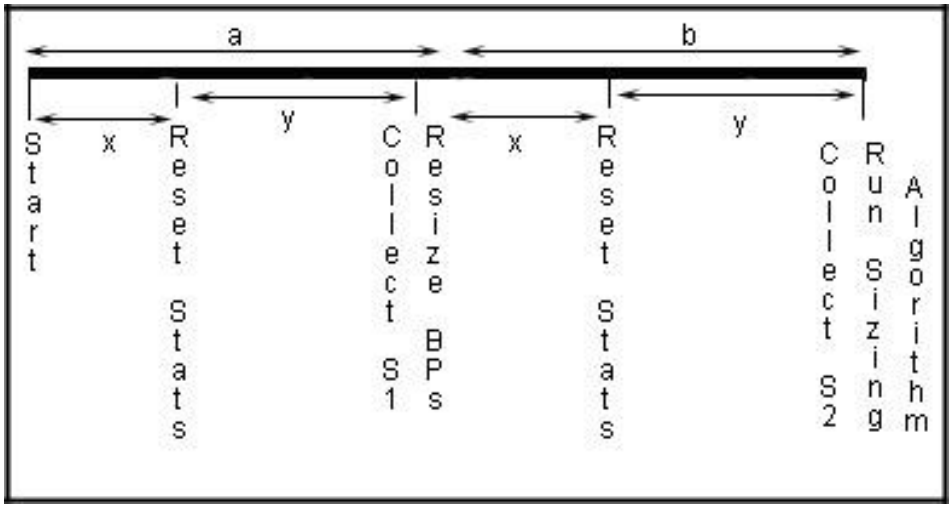

Figure 4 - 5 Steps to Size Buffer Pools at System Startup

4.3.2 Reallocating Buffer Pages

We create a field in each Buffer Pool Descriptor (see Appendix A) to store the new buffer pool sizes returned from the sizing algorithm. In order to reset the buffer pools, pointers to the next and previous buffers in each buffer pool free list need to be reinitialized as shown in Figure 4 - 6. A particular buffer may now belong to another buffer pool so the buffer pool identification needs to be updated.

```
ReinitializeBufferPools (Buffer Pool Descriptors) {
    BufferDesc buffer = BufferDescriptors;
    Integer counter = 0;
    for (j =0 to number of buffer pools){
        for (i = 0 to new number of buffers in the buffer pool){
                buffer->next = (i + counter) + 1;
                  buffer->previous = (i + counter) - 1;
                  // need to reset the buffer pool identification of the buffer
                  buffer->pool_id= j;
                buffer ++ ; counter ++;
         }
         close the circular queue;
    }
}
```

Figure 4 - 6 Algorithm to Reinitialize the Buffer Pools

The time complexity of this algorithm is θ (N_b) where N_b is the number of buffers. Since the total number of buffers is fixed, if we increase the number of buffer pools, the data objects is distributed among the buffer pools therefore, no extra overhead is added.

After reinitializing the buffer pools, buffers that belonged to buffer pool x could now belong to buffer pool y. However, the data objects associated with these shifted buffers should still be associated with buffer pool x (because we assign data objects to buffer pools only when they are created in the DBMS). To ensure that the data objects associated with the shifted buffers are not stuck in the wrong buffer pools, the contents of the pinned buffers of the buffer pools have to be flushed to disk.

Flushing data objects can be done in two ways. The first approach entails flushing the contents of all dirty pinned pages to disk after reinitializing the buffer pool. With the

second approach, upon a subsequent request for a data object, the pinned page associated with this data object is flushed out to disk when it is determined that the requested data object is in the wrong buffer pool. We chose to implement the second approach as it is more efficient. The algorithm for this approach is shown in Figure 4 - 7.

```
FlushPinnedPage (data object) {
  BufferDesc buffer = buffer assigned to data object;
  if (buffer's buffer pool id = =  buffer pool id assigned to data object)
        return buffer descriptor;
  else{
       check if page is dirty;
         if (dirty) {flush data object to disk;}
         read data object into a buffer from the buffer pool assigned to the data object;
         return buffer descriptor;
   }
}
```

Figure 4 - 7 Algorithm to Reallocate Pages Residing in a Pinned Buffer

No serious overhead is added by this algorithm because only dirty pages are written out to disk. These pages would have been written out to disk by the buffer manger at a later time anyways. We are just writing them out sooner. If we increase the number of buffer pools, the same theory applies.

To ensure that each buffer pool has the correct number of buffers and the data objects are assigned to the right buffer pools after the sizing algorithm is executed, both algorithms (Figure 4 - 6 and Figure 4 - 7) have to be executed. Figure 4 - 8 shows an example of the contents of two buffer pools. There are a total of 16 buffers and the buffer

pool configuration is {8, 8} (buffers 0 – 7 are stored in the first buffer pool and buffers 8 – 15 are stored in the second buffer pool). Buffers 1, 2 and 6 in the first buffer pool are all being used to execute transactions while buffers 0, 3, 4, 5 and 7 are free buffers that can be requested by the buffer manager.

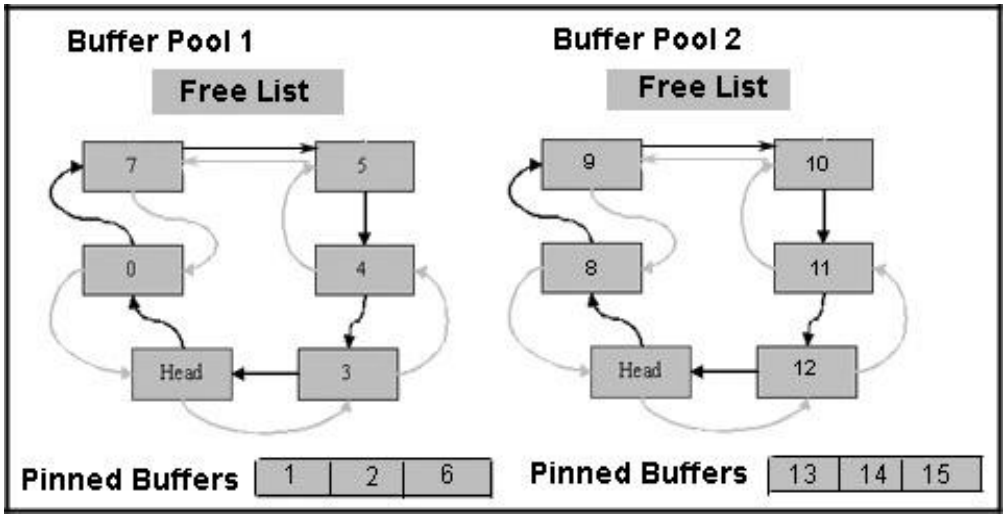

Figure 4 - 8 Contents of 2 Buffer Pools with Configuration {8, 8}

Upon execution of the sizing algorithm, it is determined that the optimal buffer pool configuration should be {6, 10}. This means that buffers 0-5 now belong to the first buffer pool and buffers 6-15 belong to the second buffer pool. Figure 4 - 9 shows the contents of the free lists and list of pinned buffers for both buffer pools after reallocating the buffers. Buffer 7 is moved from the free list of buffer pool 1 to the free list of buffer pool 2 and buffer 6 now belongs to the second buffer pool (the algorithm from Figure 4 - 6 is executed). Buffer 6 was pinned by the buffer manager when the sizing algorithm was executed. The next time a data object from this buffer is accessed, a check is made to determine whether the buffer still belongs to the first buffer pool (the algorithm from Figure 4 - 7 is executed). As this is not the case, the page associated with the buffer is

checked to determine if it has been dirtied. If it is, it is written out to disk. Buffer 6 now belongs to the list of pinned buffers in the second buffer pool because it holds the page associated with the requested data object.

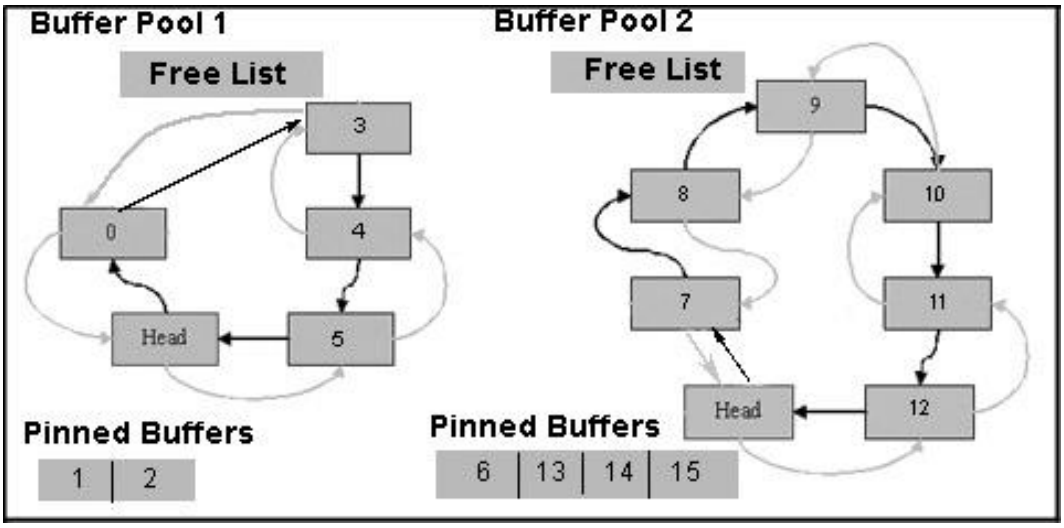

Figure 4 - 9 Contents of Buffer Pools after Reallocating all Buffer Pages

4.3.3 Determining When to Resize the Buffer Pools

The section above described the initial steps taken by the DBMS to size the buffer pools. The DBMS has to determine when to change these buffer pool sizes. Resizing should be done when it is determined that the DBMS is not running efficiently. We use DAT as the basis for this evaluation. An increase in DAT indicates an increase in the number of physical reads [30]. The number of physical reads should be minimized to ensure good performance so the aim is to minimize the average DAT needed to retrieve a data object.

To determine whether a buffer pool state is reasonable, the average DATs for the retrieval of data objects associated with the buffer area is studied under this buffer pool state (the DAT for the buffer area is calculated by summing the DATs of all the buffer pools). We use the sum of the DATs and minimize this value so that there could be an *overall* improvement in the database system. If the DAT is larger than a threshold (best DAT collected for the buffer pool so far times x, where x is a percentage that caters for minor changes in the collected DAT), the buffer pools are resized. The DBMS continues to collect statistics as long as there are transactions to execute. Figure 4 - 10 shows the algorithm to determine whether or not to resize the buffer pools. Statistics 3 and 4 are those collected while the system continues to run. A history is kept of the best DAT collected so far (bestDATsofar) for the buffer pools. This value is specific to the current workload, if the workload changes, the original statistics have to be collected and the best DAT has to be determined once more.

```
ResizeBufferPoolsCheck ( ) {
  S3 = third set of statistics;
  bestDATsofar = DAT from S3;
  while (more transactions to execute){
    bestDATsofar = minimum (DAT from S3, bestDATsofar);
    S4 = fourth set of statistics;
    S3 = S4;
    if (DAT from S3 > bestDATsofar * x)
        resize buffer pools;
  }
}
```

Figure 4 - 10 Checking DATs to Determine if Buffer Pools need to be Resized

Table 4- 1 summarizes the overall time and space complexities required by the algorithm used to check when to resize the buffer pools and other algorithms introduced in this chapter. Num_S is the number of possible states that the buffer pools can have, N_b is the number of buffers in the buffer area and N is the number requested data objects.

Procedure	**Time Complexity**	**Δ in Space Complexity**
Sizing Buffer Pools	θ (Num_S)	θ (Num_S^2)
Calculate Buffer Pools' Statistics	θ (N)	θ (N_{bp})
Reinitialize Buffer Pools	θ (N_b)	0
Flush Pinned Page	θ (1)	0
Resize Buffer Pools Check	θ (1)	θ (N_{bp})

Table 4- 1 Complexities of Algorithms used To Size the Buffer Pools

Chapter 5 Experimental Design and Results

We introduce the experimental design and method in Section 5.1. We present our findings using two buffer pools in Section 5.2. We show why DAT is a good performance measure in Section 5.3. An analysis of the results obtained after monitoring statistics under specific buffer pool states is shown in Section 5.4. Section 5.5 evaluates the effectiveness of the dynamic resizing of the buffer pools by dynamically adjusting the experimental workload.

5.1 Experimental Method

We use PostgreSQL (V 7.3.2) running on a Sun Solaris machine (2.5.1 operating system) in our experiments. PostgreSQL (V 7.3.2) is equipped with a TPC–B benchmark. We modified the TPC– B benchmark because it is not sufficient for our evaluation. The modified database schema for the TPC– B benchmark consists of four relations (the savings account relation is added), which are shown in the Entity Relation diagram (ER

diagram) displayed in Figure 5- 1. Manipulations are made to this banking database in our experiments.

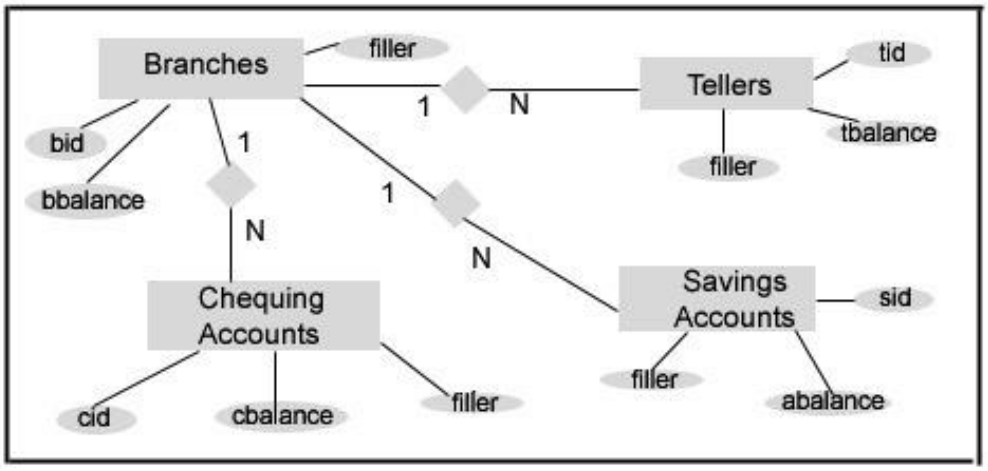

Figure 5- 1ER Diagram for TPC– B Banking Database

Each tuple in each relation occupies 50 bytes and has an integer identification number, an integer balance, possibly a reference to another relation via an identification number, and some other information that is referred to as the filler. The fields of all the relations are shown in Appendix A. There is one branch with 100000 chequing accounts, 50000 savings accounts and 10 tellers working at this branch. No index is associated with the relation in this database.

The experiments in this section compare the DBMS performance while executing transactions using two buffer pools versus a single buffer pool (the transactions used in these experiments are presented shortly). The aim of these experiments is to study the impact of multiple buffer pools on the performance of the DBMS and to quantify the improvements associated with the dynamic execution of the sizing algorithm developed by Tian [30] in PostgreSQL. To determine whether there are benefits to integrating this algorithm, we compare the performance of the DBMS in the suggested buffer pool

configurations to the performance obtained under the configurations prior to executing the algorithm. We also present some experiments that measure the overhead of our modified DBMS. Lastly, we show that our system is dynamic in nature by studying how the DBMS reacts to changes in its environment. Change in environment is simulated by change in the sizes of the relations in the database. We show that the modified DBMS reacts to change and reconfigures the DBMS accordingly.

Each experiment is executed three times and the means are calculated within a confidence interval of 95%. Appendix C outlines the details of the calculations. There are either one or two buffer pools in our experiments. Clustering of data objects into buffer pools is done manually in this thesis. When one buffer pool is used, all data objects are assigned to this buffer pool. When two buffer pools are used, data objects that belong to the branches and savings accounts relations are assigned to the first buffer pool and data objects belonging to the tellers and chequing accounts relation are assigned to the second buffer pool as shown in Figure 5- 2.

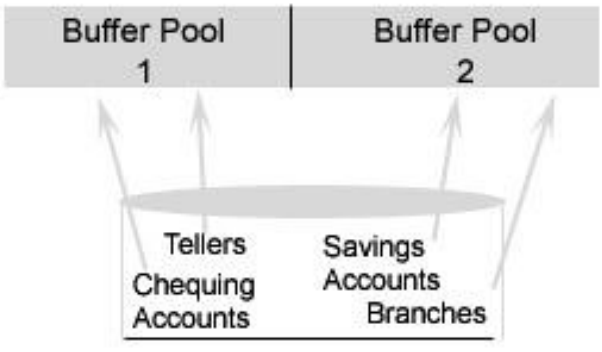

Figure 5- 2 Clustering of data objects into respective buffer pools

5.1.1 TPC –B Workload

The original TPC– B workload consists of multiple executions of a single TPC– B transaction shown in Figure 5- 3 where rsid, rtid and rbid are random, valid savings account identifications, teller identifications and branch identifications respectively and delta is a randomly generated integer. The first two statements deal with the chequing accounts relation and involve updating a random tuple and then selecting the updated tuple. The last two queries involve an update of a random tuple in the tellers relation followed by a selection of a random tuple in the branches relation.

```
Update Chequing Accounts set abalance to abalance + delta where cid = rcid;
Select abalance from Chequing Accounts where cid = rcid;
Update Tellers set tbalance to tbalance + delta where tid = rtid;
Update Branches set bbalance to bbalance + delta where bid = rbid;
```

Figure 5- 3 TPC– B Transaction

The TPC– B workload handles multiple random tuples from all relations, so if many transactions are executed using this workload, it is difficult to track what is occurring in the buffer pools. Additionally, the TPC– B workload is not sufficient for some of our experiments because only one tuple is selected /manipulated in every query. We require a more complex workload. The standard TPC– C benchmark is more complicated because it simulates multiple different transactions within the workload. However, the TPC– C benchmark has not yet been implemented for PostgreSQL [25]. We therefore design simple workloads that select multiple tuples from the larger relations to use in some of our experiments.

5.1.2 Select Only Workload

To test the impact of dividing the data objects between two buffer pools, we designed specific queries that utilize relations from different buffer pools in the TPC– B banking database. The queries outlined in Figure 5- 4 make up a '*select only transaction'* (where rtid and rsid are randomly generated, valid identification numbers). Multiple executions of this single transaction are referred to as a select only workload. Statement A is a random selection from the tellers relation with data objects residing in the first buffer pool and statement B is a random selection from the savings accounts relation with data objects residing in the second buffer pool.

```
A. Select tbalance from Tellers where tid = rtid;
B. Select abalance from Savings Accounts where sid = rsid.
```

Figure 5- 4 Select Only Transaction

If only one buffer pool is used, after multiple executions of statement A it is likely that all the tuples from the tellers relation will reside in the buffer pool due to the small size of this relation. However, once the buffer pool is full, pages will be replaced to accommodate the needs of statement B. If pages from the tellers relation are replaced, the performance of statement A will degrade thus affecting the performance of the DBMS.

In the case where there are two buffer pools, we can ensure that the execution of statement B does not interfere with the execution of statement A by separating data objects that are affiliated with these relations into different buffer pools. Data objects associated with the tellers relation are assigned to the first buffer pool and data objects

associated with the savings accounts relation are assigned to the second buffer pool as shown in Figure 5- 2.

After multiple executions of statement A, tuples from the tellers relation have a greater probability of residing in the first buffer pool due to the small size of this relation. These tuples are referred to as the '*hot set*' because they are constantly being accessed and they should remain in the buffer pool. On the other hand, if multiple executions of statement B are made there is a less guarantee that the data page associated with the requested tuple will be in the buffer pool due to the large size of the relation. Figure 5- 5 clarifies the expected interaction of data pages with disk for multiple executions of the select only workload.

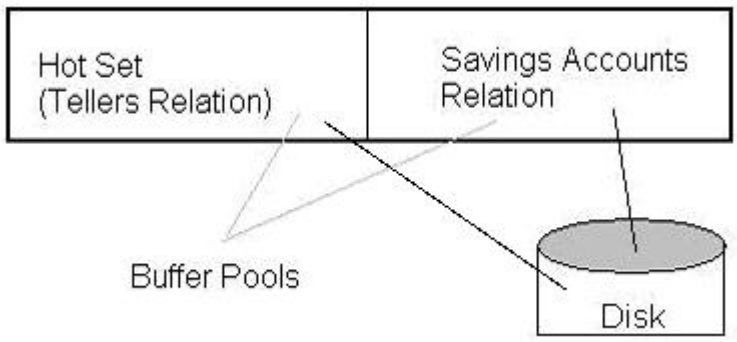

Figure 5- 5 Interaction of Buffer Pools with Disk for Select Only Workload

We expect better performance with the use of two buffer pools for the select only workload and the experiments confirm this fact. We also use the select only workload to examine the overhead involved with our multiple buffer pools.

5.1.3 Extended Select Only Workload

We present a workload called the *'Extended select only workload'* that deals with selecting tuples from the larger relations. Figure 5- 6 shows the queries executed in this workload. The statement set labeled A focuses on the chequing accounts relation and the queries labeled B focus on the savings accounts relation. All the tuples from each relation are selected, followed by requests for specific sets of tuples in each relation.

```
A. Select * from Chequing Accounts;
   Select cid from Chequing Accounts where cid < 50;
   Select cid from Chequing Accounts where cid < 100;
B. Select * from Savings Accounts;
   Select cid from Savings Accounts where sid < 50;
   Select cid from Savings Accounts where sid < 100
```

Figure 5- 6 Extended Select Only Transaction

Recall that the data objects associated with the chequing accounts relation are assigned to a different buffer pool than data objects associated with the savings accounts relation [Figure 5- 2]. Since the number of tuples in all the relations in the TPC– B database is variable we study the expected results when there are 15000 in the chequing accounts relation and 7500 tuples in the savings accounts relation.

The capacity of each buffer is 8 kilobytes (8192 bytes) and the size of each tuple is 50 bytes. Therefore, the maximum number of tuples that can fit in a buffer pool with k buffers can be calculated according to Equation 5 -1.

Maximum number of tuples = floor [(k x 8192) ÷ 50] **Equation 5- 1**

The maximum number of tuples that can fit in a buffer pool of size 64 is 10485 tuples. If all the tuples from the chequing accounts relation are selected, they will not all fit in a buffer pool with 64 buffers. In fact, since PostgreSQL (V 7.3.2) uses a least recently used page replacement policy, selecting all the tuples from the chequing accounts relation will result in multiple page replacements. However because there are only 7500 tuples in the savings accounts relation, if all the tuples from the savings accounts relation are selected, they will fit in a buffer pool that uses 64 buffers and as a result no page replacements are necessary.

The analysis above suggests that upon multiple executions of the extended select only workload, the hit rate of the first buffer pool should be close to 0 while the hit rate of the second buffer pool should be close to 1.0 indicating that all requested tuples reside in the affiliated buffer pool for a buffer pool configuration of {64, 64}. Furthermore, after executing Tian's sizing algorithm [30], the number of buffers in the first buffer pool should be greater than 64 buffers and the number of buffers in the second buffer pool should be less than 64 buffers.

In the experiments that follow, we use the extended select only workload to monitor statistics needed by the sizing algorithm and show that the results obtained are consistent with the expected results. We also manipulate the sizes of the relations and study the impact on the DBMS.

5.2 Change in Throughput Using 2 Buffer Pools

We discussed the expected improvements using two buffer pools as opposed to one while executing the select only workload (Figure 5- 4) and highlighted that we also expect some overhead. To study the impact of multiple buffer pools on the DBMS, the select only workload is executed using a single buffer pool and then executed using two buffer pools. The number of tuples in each relation can be varied as well as the total number of buffers employed. The values of these parameters are shown in Table 5 - 1. In the case where there are two buffer pools the buffers are equally divided so that each buffer pool has 64 buffers.

Number of / Workload	savings accounts	tellers	buffers
Select Only	50000	10	128

Table 5 - 1 Experimental Parameters used to Study Modifications to PostgreSQL

The throughput measured in transactions per second is recorded while varying the number of transactions executed. The total number of transactions executed can be increased by increasing the number of clients; each client executes a fixed number of transactions. The database is recreated each time a set of transactions have completed execution. This ensures that the buffer pools do not contain data objects that may affect the results. The results are summarized in Figure 5- 7.

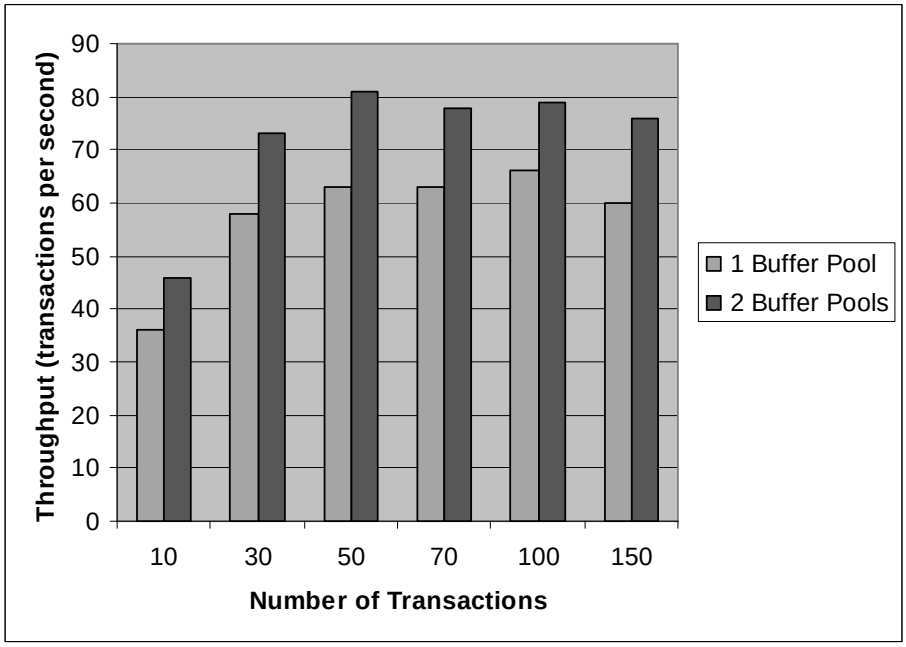

Figure 5- 7 Throughput from Executing Select Only Transactions for 1 and 2 Buffer Pools in the Modified DBMS

The throughput is calculated by dividing the total number of transactions executed by the time taken to execute the transactions. The throughput collected after executing 10 transactions is not as large as the throughput for a greater number of transactions because for 10 transactions, the DBMS has not been given sufficient time to warm up. Pages are loaded into the buffer area during warm-up period. Therefore, the results are more accurate when large numbers of transactions are executed. However, we still see a difference in the throughput for one versus two buffer pools no matter how many transactions are executed. The general increase in performance is approximately 26%. This result verifies that the throughput of the DBMS is significantly larger for the designed select only workload with the use of two buffer pools as opposed to one.

CPU utilization is also collected as a percentage over 5 second intervals while all the transaction sets are being executed. Monitoring of CPU utilization is started right before

executing the transactions and stopped soon after the transactions have completed execution. Table 5 - 2 summarizes the average results obtained. The increase in CPU utilization is found to be 0.64% for switching from 1 to 2 buffer pools.

Number of Buffer Pools	Average CPU Utilization (%)
1	77.71
2	78.35

Table 5 - 2 CPU Utilization Using 1 and 2 Buffer Pools in the Modified DBMS

No extra time is spent creating the second buffer pool, as discussed in Section 3.3, but CPU overhead is incurred when determining the buffer pool descriptor associated with the requested data objects. The increase is so small that we consider the CPU overhead for this experiment negligible.

Additions are made to the original PostgreSQL DBMS to support multiple buffer pools. New data structures and fields that support multiple buffer pools have to be maintained. Every time a data object is created, its object identification number has to be placed in the object queue of its associated buffer pool descriptor. The object descriptor and the buffer also have to store what buffer pool they are associated with.

To study the overhead incurred from the maintenance of this information we compare the modified version of PostgreSQL using a single buffer pool to the original PostgreSQL. The parameters of the experiment are the same as those in Table 5 - 4. The throughput is recorded while varying the number of transactions using the select only workload (Figure 5- 4). The results are summarized in Figure 5- 8.

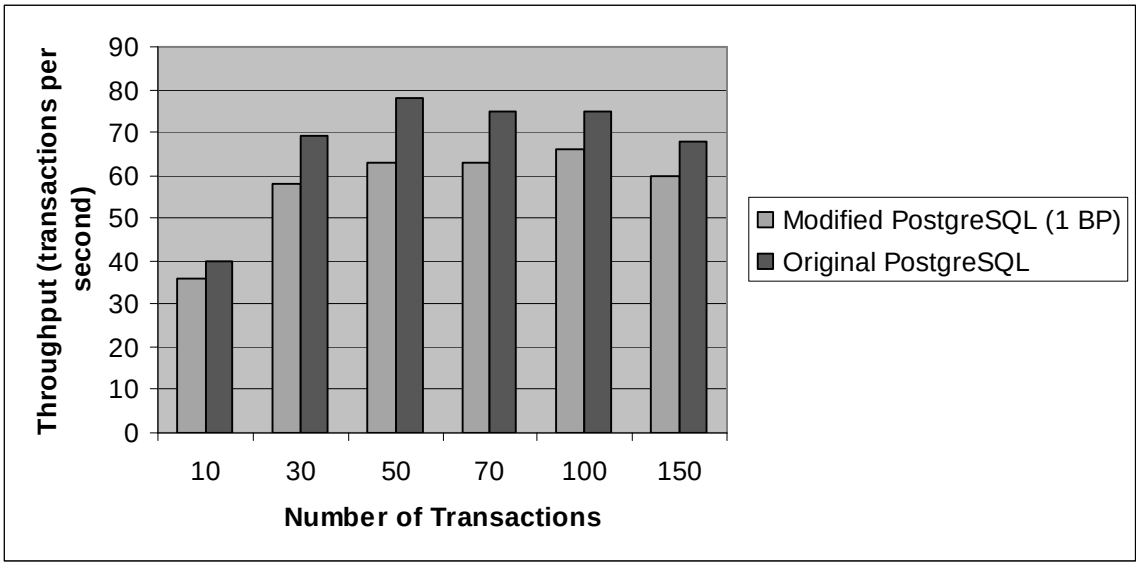

Figure 5- 8 Overhead from Maintaining Buffer Pools

For all the sets of transactions, the throughput is greater using the original PostgreSQL compared to the modified PostgreSQL suggesting that there is overhead created by supporting multiple buffer pools. The general decrease in performance using the modified DBMS is approximately 16%. Figure 5- 9 summarizes the overall increase in performance with the use of 2 buffer pools.

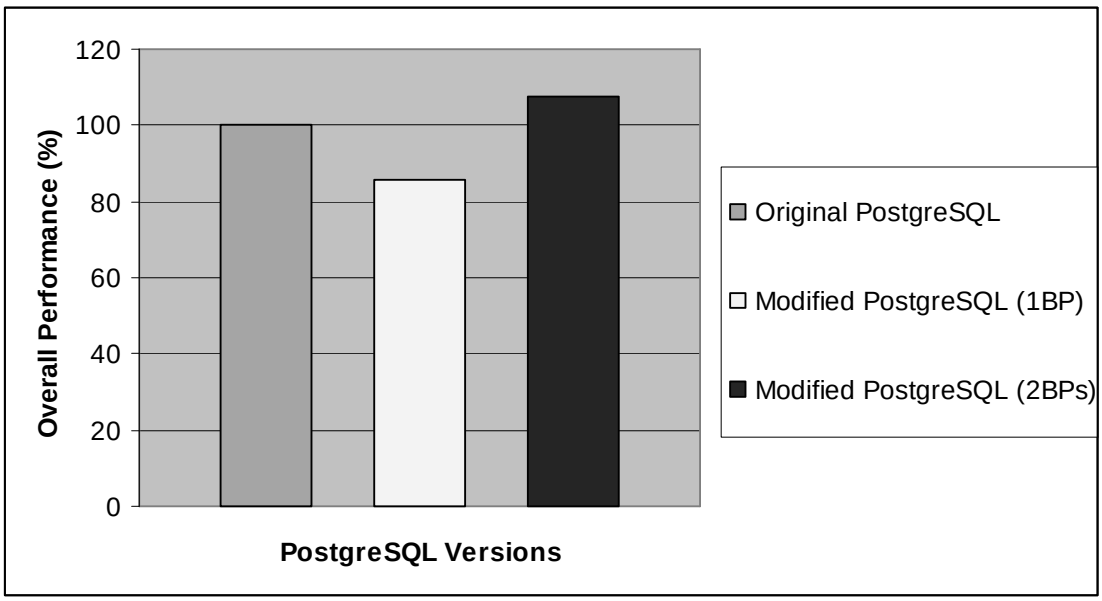

Figure 5- 9 Overall Percentage Performance for Different Versions of PostgreSQL

When we compare the throughputs obtained in the original PostgreSQL with our modified PostgreSQL using 2 buffer pools, we see an improvement of approximately 7.6%. This difference suggests that the use of two buffer pools is beneficial while executing the select only workload. When we only use one buffer pool in our modified version there is an overall decrease in performance of approximately 14.1%.

5.3 Relationship between DAT and Throughput

As discussed in Chapter 4, data access time (DAT) is the performance measure used to determine when to resize the buffer pools. To validate that DAT is a good performance measure, we examine the relationship between DAT and throughput. The DAT for requested data objects in the buffer pool should be minimal so that less time is spent accessing data objects from disk. If this is the case, then more transactions can be executed per second.

In this experiment, we trigger change in throughput by manipulating the workload and changing the size of the single buffer pool. Three workloads are used in this experiment, these included the TPC– B workload, (Figure 5- 3) the select only workload (Figure 5- 4) and the extended select only workload (Figure 5- 6). The same number of transactions are executed (80) for each workload using both 128 and 256 buffers. Total DAT in seconds for the buffer pool and throughput (transactions executed per second (tps)) are measured and recorded in Table 5 - 3.

Number of buffers / **Workload**	128		256	
	tps	DAT(s)	tps	**DAT(s)**
Extended select only workload	**0.17**	**57.0**	**1.0**	35.1
Select only workload	**4.0**	**16.7**	**6.1**	12.1
TPC – B workload	32.1	8.3	53.0	7.2

Table 5 - 3 Statistics for Varying Number of Buffers and Workloads

The values in Table 5 - 3 are also plotted in a graph shown in Figure 5- 10. We observe that as DAT increases, throughput decreases. This indicates that if we want the DBMS to be more efficient and execute more transactions per second, no matter what workload is being executed, the DAT for the retrieval of data objects has to be minimal.

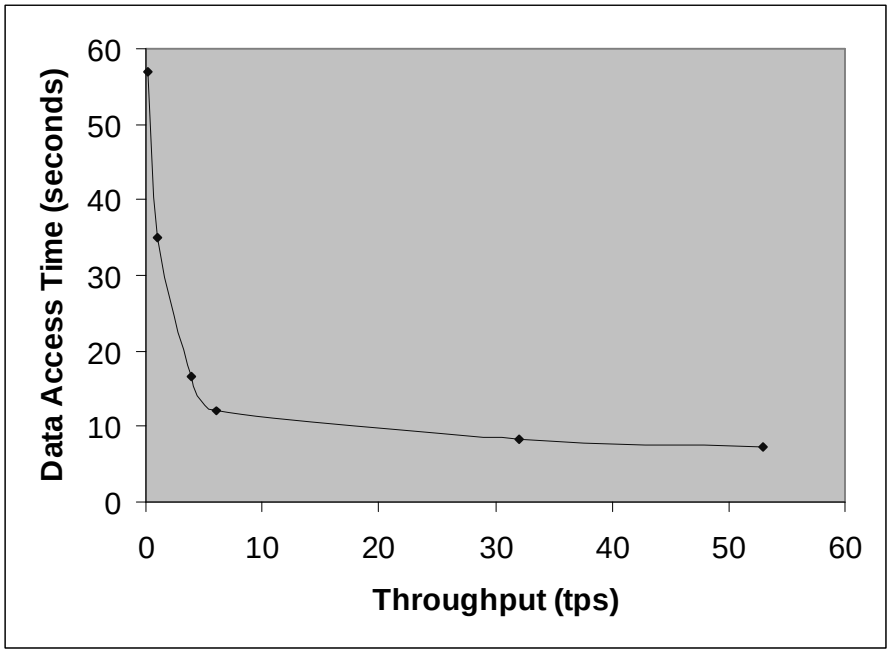

Figure 5- 10 DAT vs. Throughput for Varying Workloads and Buffer Pool Sizes

5.4 Monitoring DAT and Hit Rate

Monitoring the hit rate of each buffer pool and the DAT for the retrieval of requested data objects is essential in determining the optimal sizes of the buffer pools. The modified statistics collector described in Chapter 4 retrieves the number of logical reads (LR), the number of physical reads (PR) and the average data access time (DAT) to retrieve requested data objects associated with each buffer pool. In order to collect these statistics, the modified statistics collector has to be turned on in the DBMS.

5.4.1 Overhead from Switching the Statistics Collector On

To study the effect of the statistics collector on the performance of the DBMS, the throughput obtained while executing a number of transactions in two scenarios is

compared. The first scenario involves the original version of PostgreSQL, with the statistics collector switched on, and the other involves the original version of PostgreSQL, with the statistics collector off. The parameters for the experiment are shown in Table 5 - 1. The number of select only transactions (Figure 5- 4) that are executed is varied and the throughput is recorded. The results are summarized in Figure 5- 11.

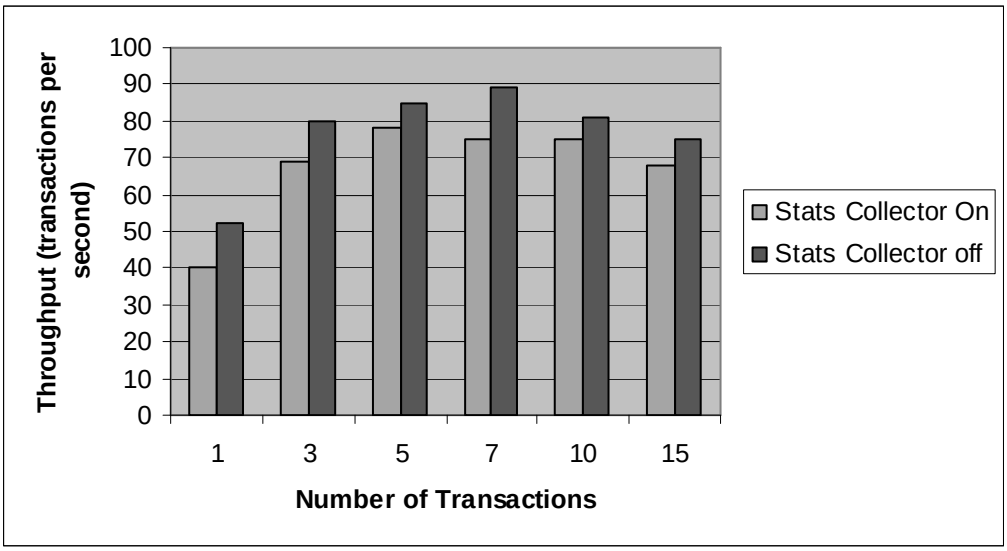

Figure 5- 11 Throughputs Obtained by Switching the Statistics Collector On and Off

It is noted that the overall decrease in performance is approximately 16% when the statistics collector is turned on using the select only workload. This percentage is specific only to the select only workload but it suggests that there is overhead incurred from the collection of statistics required by the modified DBMS. It is up to the database administrator to switch the statistics collector on. It is recommended that this is done because many new features of the DBMS are built into the statistics collector [25].

5.4.2 Analysis of Collected Statistics under Different Configurations

Monitoring the hit rate of each buffer pool and the DAT for the retrieval of requested data objects is required to determine the optimal sizes of the buffer pools. Buffer pool hit rate can be calculated as (LR – PR) / LR where PR is the number of physical reads and LR is the number of logical reads. If the hit rate of the buffer pool is 0, this implies that reads and/or writes are constantly being made to disk. If the hit rate is 1.0, no disk accesses are required as all requested data objects reside in the buffer pool.

We show that the modified PostgreSQL DBMS monitors DAT and hit rate correctly by analyzing the statistics collected under specific configurations of the buffer pools. The number of tuples in all the relations from the TPC– B database is variable. For the experiments that follow, we change the size of the savings and chequing accounts relations and the sizes of the buffer pools and study the collected statistics.

For the first experiment, we alter the number of tuples in the savings and chequing accounts relations as shown in Table 5 - 4. We employ two buffer pools each with 64 buffers and execute 150 extended select only transactions after the DBMS is given sufficient time to warm up (50 transactions are executed).

Number of / Workload	savings accounts	chequing accounts	buffers per BP	transactions
Extended Select Only	7500	15000	64	150

Table 5 - 4 Experimental Parameters for Monitoring DAT and Hit Rate

Table 5 - 5 reports the number of logical reads (LR), number of physical reads (PR), the calculated buffer pool hit rate, and the average DAT to retrieve a requested data object for each buffer pool after the transactions are executed. From our analysis above (Section 5.1), it is expected that the hit rate of the first buffer pool is close to 0 because the chequing accounts relation is too large to fit in the buffer pool. The hit rate of the second buffer pool is expected to be close to 1.0 since all the tuples in the savings accounts relation can fit in the second buffer pool.

	LR	**PR**	**Hit Rate**	**DAT (ms)/ LR**
Buffer Pool 1	36000	36000	0	120.3
Buffer Pool 2	17904	0	1.0	0

Table 5 - 5 Statistics Collected for Extended Select Only Workload with BP Configuration {64, 64}

The hit rate of the first buffer pool is calculated as 0, which validates our claim. The average DAT per logical read is larger than 0 because accesses are made to disk for requested data objects. The hit rate of the second buffer pool is calculated as 1.0 as expected. When a data object resides in the buffer pool, the time taken to access this data object is so small that we consider it negligible. Therefore, as expected the DAT per logical read is 0 milliseconds.

We next alter the sizes of the buffer pools to a configuration of {96, 32}. The experimental parameters are shown in Table 5 - 6. The transactions are executed after the DBMS is given sufficient time to warm up (50 transactions are executed). 96 buffers can hold 15728 tuples and 32 buffers can hold 5242 tuples. Therefore if there are 15000 tuples in the chequing accounts relation, they should all fit in a buffer pool of size 96.

However, the 7500 tuples from the savings accounts relation will not fit in the second buffer pool.

Number of / Workload	savings accounts	chequing accounts	buffers		transactions
			BP1	**BP2**	
Extended Select Only	7500	15000	96	32	150

Table 5 - 6 Experimental Parameters for Monitoring DAT and Hit Rate

Table 5 - 7 shows the number of logical reads (LR), the number of physical reads (PR), the calculated hit rates and average DAT needed to read a requested data object. Observe that, the hit rate of the first buffer pool is 1.0 because the chequing accounts relation fits in the first buffer pool and the hit rate of the second buffer pool is 0 because it is not large enough to hold all the tuples from the savings account relation. We can therefore conclude that the statistics collection operates as expected.

	LR	**PR**	**Hit Rate**	**DAT (ms)/ LR**
Buffer Pool 1	35904	0	1.0	0
Buffer Pool 2	17952	17952	0	117.2

Table 5 - 7 Statistics Collected for Extended Select Only Workload with BP Configuration {96,32}

5.5 Initially Sizing the Buffer Pools

Determining the optimal sizes of the buffer pools is essential to ensure efficiency in the DBMS. Our modified DBMS calculates the *optimal* configuration of the buffer pools

by first automatically collecting statistics under two different buffer pool configurations [30].

The hit rate of each buffer pool in the first set of statistics has to be different from its hit rate in the second set of statistics so that we can accurately calculate a and b for each buffer pool (refer to Equation 4-2). Using these values we can predict the expected hit rate and DATs for different buffer pool states [30]. From the experiments in the previous section, the hit rates calculated with 15000 tuples in the chequing accounts relation and 7500 tuples in the savings accounts relation under a buffer pool configuration of {64, 64} are different than those calculated using a configuration of {96, 32} after executing the extended select only workload. Therefore, these are good configurations to collect statistics required by the sizing algorithm implemented by Tian [30] for this workload.

To observe the actions taken by the modified DBMS in calculating the optimal buffer pool configuration, the extended select only workload is utilized. The parameters of the experiment are given in Table 5 - 8.

Number of / Workload	savings accounts	chequing accounts	buffers per BP	transactions
Extended Select Only	7500	15000	64	600

Table 5 - 8 Experimental Parameters for Sizing Buffer Pools

Figure 5- 12 shows the steps taken by the modified DBMS before calculating the optimal sizes of the buffer pools. Statistics are automatically collected after every 200 transactions. The statistics are reset after the first 50 transactions are executed. The first

set of statistics are collected (Table 5 - 5) after 150 more transactions are executed. Then the buffer pools are resized to {96, 32} and 50 transactions are executed before the statistics are reset. 150 transactions are allowed to run in the {96, 32} configuration and the second set of statistics are collected (Table 5 - 7). The hit rates from these statistics are compared to ensure that each buffer pool has a different hit rate for both sets of statistics. The integrated sizing algorithm is then executed and the optimal buffer pool configuration is calculated.

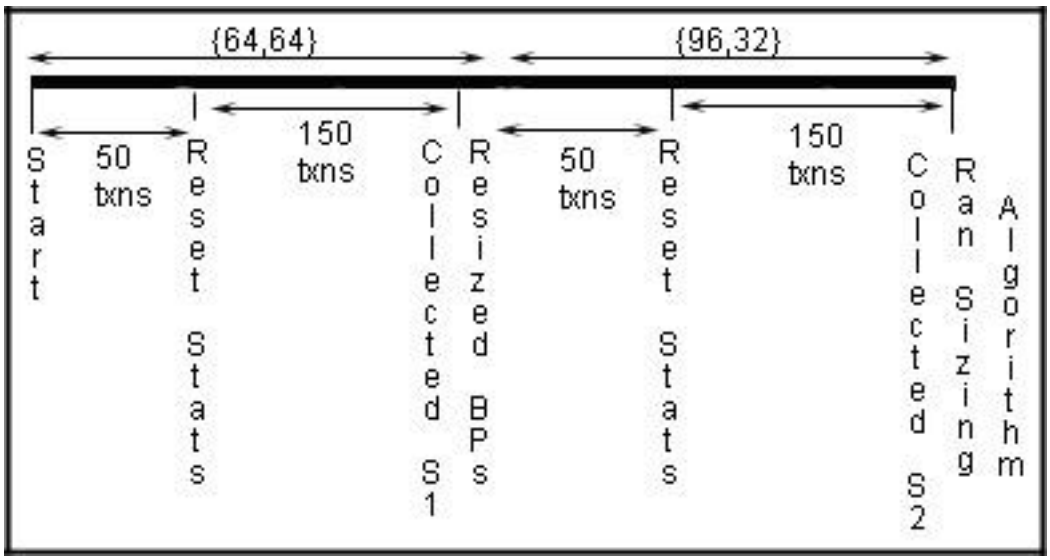

Figure 5- 12 Steps Taken by DBMS to Calculate BP Sizes Using Configurations {64, 64} and {96, 32}

A configuration of {81, 47} is returned by the sizing algorithm. The buffer pools are then automatically sized to this configuration. So far, 400 transactions are executed. After 50 more transactions are executed, the statistics are reset. The remaining transactions (150 transactions) are executed under the optimal configuration of {81, 47}. Recall that the modified DBMS collects statistics at this point. The DAT per logical read is reported

in Table 5 - 9 for both buffer pools and compared to the average DAT required to read data objects from each buffer pool with a configuration {64, 64} running the same 150 transactions from the extended select only workload (Table 5 - 5).

	Size	**DAT (ms) / LR**	Size	**DAT (ms) / LR**
Buffer Pool 1	64	120.3	81	112.8
Buffer Pool 2	64	0	47	0

Table 5 - 9 DAT per Logical Read for BP Configurations {64, 64} and {81, 47}

An analysis of the suggested buffer pool configuration shows that 81 buffers hold 13271 tuples and 47 buffers hold 7700 tuples. The algorithm selects the buffer pool configuration which minimizes the average DAT to retrieve requested data objects associated with all the buffer pools. Most of the tuples from the chequing accounts relation fit in the first buffer pool and all the tuples from the savings accounts relation fit in the second buffer pool after sizing the buffer pools.

We compare the statistics collected under the optimal buffer pool configuration with the original configuration {64, 64} to show whether there is any improvement in our results. The average DAT for the retrieval of a requested data object has decreased by approximately 6.2% suggesting that our approach is worth exploring further.

5.6 Resizing the Buffer Pools

The buffer pools need to be resized if the performance of the DBMS decreases. The algorithm in Figure 5- 13 describes the process taken by the DBMS to determine whether

or not to resize the buffer pools. This algorithm is applied by the system after the buffer pools are sized. The DAT collected under the optimal buffer pool configuration should be the minimal DAT collected so far as guaranteed by Tian's sizing algorithm [30]. The next time the DBMS collects statistics (after another 200 extended select only transactions are executed) the average DAT of a requested data object is compared to the minimum DAT collected so far. If the incoming DAT is greater than a threshold, the buffer pools are resized. The threshold is set as the minimal DAT collected so far times 105%. This percentage can be varied but 105% allows for any minor changes in the collected DAT. The DBMS does not waste time calculating new buffer pool configurations if the change in DAT is not significant. We use 5% in our modified DBMS specifically for this workload. We have monitored the average DAT per logical read and determined that a 5% change is enough to make a significant difference in the calculated optimal sizes of the buffer pools for the extended select only workload.

```
ResizeBufferPoolsCheck ( ) {
  collect S3;
  bestDATsofar = DAT from S3;
  while (more transactions to execute){
        bestDATsofar = minimum (DAT from S3, bestDATsofar);
     collect S4;
        S3= S4;
     if (DAT from S3 > (bestDATsofar * 105%))
           resize buffer pools;
  }
}
```

Figure 5- 13 Algorithm to Determine When to Resize the Buffer Pools

To determine whether the modified DBMS correctly determines when to resize the buffer pools, 800 transactions are executed with the experimental parameters given in Table 5 - 10.

Number of / Workload	savings accounts	chequing accounts	buffers per BP	transactions
Extended Select Only	7500	15000	64	800

Table 5 - 10 Experimental Parameters for Testing Resizing of Buffer Pools

Figure 5- 14 shows the steps taken by the DBMS while these transactions are being executed. The events that occur in the first 600 transactions have already been discussed. The best DAT so far after the initial sizing of the buffer pools is {112.8 ms, 0 ms} (refer to Table 5 - 9) (112.8 ms for retrieval of data objects associated with the first buffer pool and 0 ms for retrieval of data objects associated with the second buffer pool). The fourth statistic set gives average data access times of {113.5 ms, 0 ms}. This statistic set is not larger than {118.4 ms, 0 ms} (the threshold) so the resizing algorithm is not executed. The results of this experiment show that the modified DBMS correctly monitors changes in average DAT.

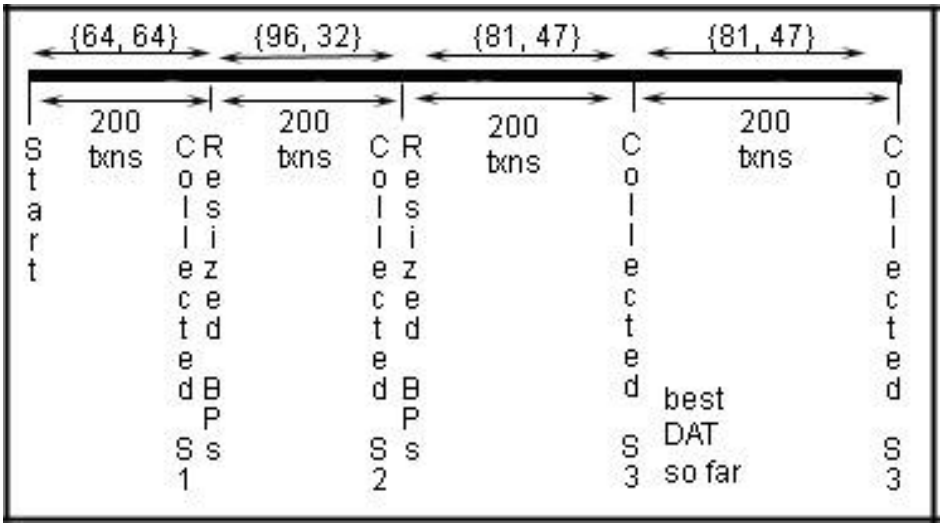

Figure 5- 14 Monitoring changes in average DAT per logical read

A significant increase in the average DAT for retrieval of data objects can trigger a resizing of the buffer pools. The DAT for retrieval of data objects in the second buffer pool is 0 milliseconds in the experiments above because all the request data objects associated with this buffer pool reside in the buffer pool. One way to trigger an increase in DAT is to request more data objects from the second buffer pool.

If 7500 tuples are removed from the chequing accounts relation and 7500 tuples are added to the savings account relation and the same extended select only workload is executed, we expect the average DAT in the first buffer pool to decrease and the average DAT in the second buffer pool to increase. Table 5 - 11 summarizes the parameters for the next experiment.

Number of / Workload	savings accounts	chequing accounts	buffers		transactions
			BP1	**BP2**	
Extended Select Only	15000	7500	81	47	600

Table 5 - 11 Experimental Parameters to Trigger Change in DAT

The optimal sizes of the buffer pools have to be determined since the database has been significantly changed. Two buffer pool configurations need to be chosen that give different buffer pool hit rates. The first is the current configuration of {81, 47} and the second is {32, 96} (since {96, 32} gives the same hit rates as {81, 47}). Figure 5- 15 shows the steps taken by the DBMS while the first 400 extended select only transactions are being executed using the modified TPC –B database.

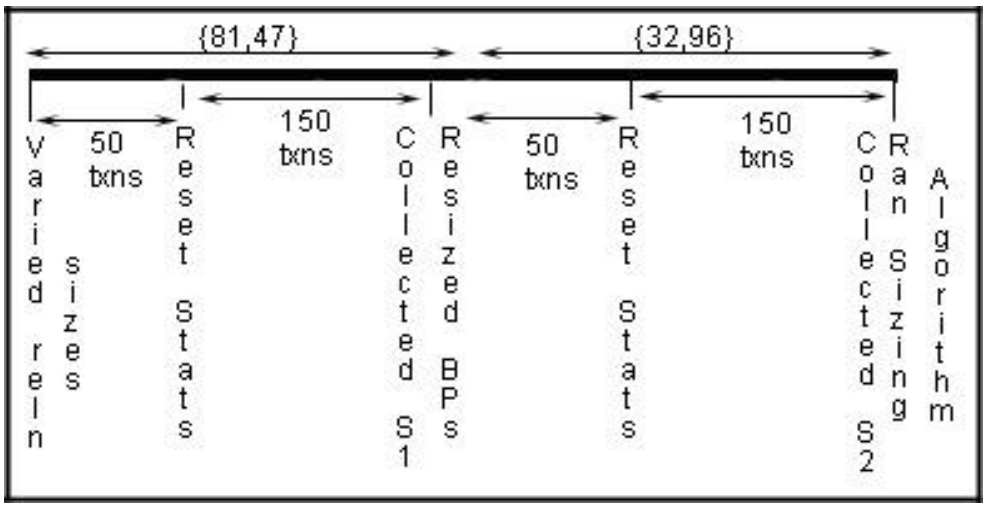

Figure 5- 15 Determining Optimal BP Sizes After Changing the TPC –B Database

The DBMS automatically collects two sets of statistics shown in Table 5 - 12 where LR1 and LR2 are the number of logical reads from the first and second set of statistics respectively, PR1 and PR2 represent the number of physical reads required and DAT1 and DAT2 are the average DATs required to retrieve a requested data object (measured in milliseconds).

	Size	**LR1**	**PR1**	**DAT1**	**Size**	**LR2**	**PR2**	**DAT2**
BP 1	81	18000	0	0	32	18000	18000	124.2
BP 2	47	35712	35712	121.0	96	35904	0	0

Table 5 - 12 Statistics Collected Under BP Configurations {81, 47} and {32, 96}

The configuration {81, 47} gives different DATs than that in Table 5 - 9 because the database has been changed. The DAT for the second buffer pool has significantly increased, which triggers the DBMS to resize the buffer pools. The resizing algorithm returns {55, 73} as the optimal buffer pool configuration. An analysis of the suggested buffer pool configuration of {55, 73} shows that 55 buffers hold 9011 tuples and 73 buffers hold 11960 tuples. The algorithm selects the buffer pool configuration that minimizes the DAT for all the buffer pools. All the tuples from the chequing accounts relation can fit in the first buffer pool and most of the tuples from the savings accounts relation fit in the second buffer pool.

So far 400 transactions have been executed, after the next 50 transactions the modified DBMS resets all its statistics. After the remaining 150 transactions are executed in the configuration {55, 73}, the average DAT for the retrieval of a data object under this configuration is collected by the DBMS. Table 5 - 13 reports the average DAT

collected under this configuration and the average DAT collected under configuration {81, 47} (Table 5 - 12).

	Size (buffers)	**DAT (µs)/LR**	Size (buffers)	**DAT (µs)/LR**
Buffer Pool 1	81	0	55	0
Buffer Pool 2	47	121.0	73	71.5

Table 5 - 13 Average DAT per Logical Read for BP Configurations {81, 47} and {55, 73}

The average DAT for the retrieval of data objects from the second buffer pool has decreased by approximately 41% under the new configuration. This percentage suggests that the modified DBMS is more efficient as we have significantly reduced the average DAT to retrieve a requested data object and as a result increased the throughput of the DBMS.

Our modified DBMS resizes the buffer pools to {81, 47} during execution of extended select only transactions in the first database state (15000 tuples in the chequing accounts relation and 7500 tuples in the savings account relation). When the database state is changed (7500 tuples in the chequing accounts relation and 15000 tuples in the savings account relation), the DBMS resizes the buffer pools to {55, 73}. Both configurations reduced the average DAT required to access a requested data object.

5.6.1 Overhead due to Sizing Buffer Pools

The extra overhead added to the system in this phase is due to the execution of the sizing algorithm and the resizing of the buffer pools. To study this overhead, CPU utilization is monitored before execution of the sizing algorithm, while the algorithm is

running and after executing of the algorithm. CPU utilization is collected for 5 time intervals (each interval is 5 seconds) before execution of the resizing algorithm, for 5 time intervals during execution of the algorithm and for 5 intervals after executing the algorithm.

The average CPU utilization is 92.3% before execution, 93.7% during execution of the algorithm and 92.7% immediately after executing the algorithm. The sizing algorithm increased CPU utilization by 1.4 % and the actual resizing of the buffer pools increased CPU utilization by 0.4%. These percentages indicate that there is little overhead caused by executing the sizing algorithm.

Chapter 6 Conclusions

To obtain the desired performance from a database management system, resources must be used as effectively as possible. In this thesis we focus on the use of multiple buffer pools to optimize performance. Tuning buffer pool sizes is crucial to achieving good performance but this tuning is complex and manual tuning can be time consuming. Manual tuning entails shutting down and restarting the DBMS and it is not responsive to changes in workload. In this thesis we present a method to allow the PostgreSQL database management system to tune its own buffer pool sizes and to adapt to changes in workload.

6.1 Thesis Contributions

We extend PostgreSQL (V. 7.3.2) to support multiple buffer pools and present an approach to verify our modifications. We show that there is an increase in throughput using two buffer pools as opposed to the original PostgreSQL. The overhead created using the modified DBMS is also studied.

We provide an approach to dynamically size the buffer pools based on a previous algorithm. We integrate the sizing algorithm into PostgreSQL and monitor statistics from different buffer pool configurations to use as input into this algorithm. The buffer pools are sized to the algorithms suggested buffer pool configuration. Statistics are incrementally collected and used to determine when the DBMS is not functioning as efficiently as possible and should be resized.

We show that the modified DBMS is dynamic by significantly changing the database. Our system recognizes the change and collects statistics to determine the optimal buffer pool sizes for executing transactions using the new database. The buffer manager can resize its buffer pools without shutting down or restarting the DBMS.

6.2 Conclusions

Based on our experiments we can draw the following conclusions:

- Although there is overhead from using multiple buffer pools, they are beneficial to performance in some cases.
- Configuring the sizes of the buffer pools based on the results obtained from Tian's [30] sizing algorithm can result in an increase in performance of the DBMS.
- Dynamically resizing the buffer pools when it is determined that the DBMS is running poorly can lead to increase in overall performance because the buffer pools are optimally sized and the DBMS does not have to be shut down before the sizes are changed.

6.3 Future Work

The work studied in this thesis presents some interesting topics that can be explored further:

- A method to determine the number of buffer pools that should be used by the DBMS is needed. Xu's clustering algorithm [32] does not consider this.
- The clustering algorithm proposed by Xu [32] to decide what data objects should be grouped together needs to be integrated into PostgreSQL.
- The decision of when to employ multiple buffer pools needs to be made. We studied changes in the DBMS using specific workloads where multiple buffer pools were guaranteed to have an effect. Other cases need to be considered.
- The least recently used (LRU) page replacement policy currently used by PostgreSQL should be changed to an LRU-k page replacement policy [23] and this change should be studied to determine if it is beneficial.

References

1. Belady. “A Study of Replacement Algorithms for Virtual Storage Computer”. *IBM System Journal, Vol. 5, No.2*, July 1996.

2. Brown, K.P, Carey, M.J. and Livny, M. “Managing Memory to Meet Multiclass Workload Response Time Goals”. *Proceedings of the 1993 Very Large Data Base Conference*, Dublin, Ireland, August 1993, pp. 328-341.

3. Brown, K.P, Carey, M.J. and Livny, M. “Goal Oriented Buffer Management Revisited”. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Quebec, Canada, June, 1996, pp. 353-364.

4. Chen, C.M. and Roussopoulos, N. “Adaptive Database Buffer Allocation Using Query Feedback”. *Proceedings of the 1993 Very Large Data Base Conference*, Dublin, Ireland, August 1993, pp. 342-353.

5. Chou, H.T. and DeWitt, D.J. “An Evaluation of Buffer Management Strategies for Relational Database Systems”. *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, Sweden, August, 1985, pp. 127-141.

6. Chung, J.Y., Ferguson, D. and Wang, G. “Goal Oriented Dynamic Buffer Pool Management for Database Systems”. *Proceedings of the1st International Conference on Engineering of Complex Computer Systems (ICECCS)*, Ft. Lauderdale, Florida**,** November, 1995**.**

7. Davison, D.L. and Graefe, G. “Dynamic Resource Brokering for Multi-User Query Execution”. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data***,** San Jose, California, United States, May 1995, pp 281 – 292.

8. Effelsberg, W. and Haerder, T. “Principles of Database Buffer Management". *ACM Transactions on Database Systems, Vol. 9, No. 4*, December 1984, pp 560-595.

9. Elmasri, R. and Shamkant, N.B. “Fundamentals of Database Systems, Second Edition”. The Benjamin/Cummings Publishing Company, Inc. 1994.

10. Elnaffar, S., Powley, W., Benoit, D. and Martin, P. “Today’s DBMSs: How Autonomic Are They?” *Proceedings of the 1st International Workshop on Autonomic Computing Systems*, Prague, September 2003.

11. Faloutsos, C., Ng, R. and Sellis, T. "Predictive Load Control for Flexible Buffer Allocation". *Proceedings of the 17th International Conference on Very Large Databases*, Barcelona, Spain, 1991, pp. 265-274.

12. Faloutsos, C., Ng, R. and Sellis, T. “Flexible and Adaptable Buffer Management Techniques for Database Management Systems”. *IEEE Transactions on Computers, Vol. 44*, April 1995, pages 546-560.

13. Ganek, A.G. and Corbi, T.A “The Dawning of the Autonomic Era”. *IBM Systems Journal on Autonomic Computing. Vol. 42*, November, 2003 pp. 5-17.

14. IBM, DB2 Universal Database. Available: http://www-3.ibm.com/software/data/db2/udb/, November 6th, 2003.

15. Jann, J., Browning, L.M. and Burugula, R.S. “*Dynamic Reconfiguration: Basic Building Blocks for Autonomic Computing on IBM pSeries Servers*”. IBM Systems Journal on Autonomic Computing. Vol. 42 November, 2003, pp. 29-37.

16. Larson, P. and Krishan, M. "Memory Allocation for Long Running Server Applications". *Proceedings of the International Symposium on Memory Management, (ISMM '98)*, Vancouver, British Columbia, Canada, October, 1998, pp. 176-185.

17. “Mammoth PostgreSQL for Win32, MacOSX and Red Hat Linux x86 platforms”. Available: http://www.commandprompt.com, October 1st, 2003.

18. Martin, P., Li., H., Zheng, M., Romanufa, K. and Powley, W. “Dynamic Reconfiguration Algorithm: Dynamically Tuning Buffer Pools”. *Proceedings of 11th International Conference on Database and Expert Systems Applications*, London, United Kingdom, September, 2000.

19. Mehta, M. and DeWitt, D.J. “Dynamic Memory Allocation for Multiple-Query Workloads”. *Proceedings of the 1993 Very Large Data Base Conference*, Dublin, Ireland, August 1993, pp. 354-367.

20. Microsoft SQL Server. Available: http://www.microsoft.com/sql/, November, 8th 2003.

21. Momjian, B. “PostgreSQL Performance Tuning, Tweak your Hardware to get the most from this Open-Source Database”. *Linux Journal Issue 88*, August, 2001.

22. Ng, R., Faloutsos, C. and Sellis,T. “Flexible Buffer Allocation Based on Marginal Gains”. *Proceedings of the 1991 ACM SIGMOD Conference on Management of Data*, Denver, Colorado, May, 1991, pp. 387-396.

23. O'Neil, E. O'Neil, P. and Weikum, G. "The lru-k Page Replacement Algorithm for Database Disk Buffering". *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May, 1993, pp. 297-306.

24. Oracle, Oracle9i. Available: http://www.oracle.com/ip/deploy/database/, November 6th, 2003.

25. PostgreSQL Database Management System. Available: http://www.postgresql.org, October 1st, 2003.

26. Rennhackkamp, M. “DBMS Online, Server Side”. Available: http://www.dbmsmag.com/9610d17.html, August 3rd, 2003.

27. Sacco, G. and Schkolnick, M. “A Mechanism for Managing the Buffer Pool in a Relational Database System using the Hot Set Model”. *Proceedings of the 8th International Conference on Very Large Data Bases*, Mexico City, Mexico, September 1982, pp. 257-262.

28. Soloviev, V. “An Incremental Memory Allocation Method for Mixed Workload”. *Information Systems Journal*, Vol. 21, No. 4, 1996, pp. 369-386.

29. Stillger, M., Lohman, G.M., Markl, V. and Kandil. “LEO – DB2’s Learning Optmimizer”. *Proceedings of the 19th International Conference on Very Large Data Bases*, Rome, Italy, 2001, pp. 257-262.

30. Tian, W., Martin, P. and Powley, W. “Techniques for Automatically Sizing Multiple Buffer Pools in DB2”. *Proceedings of Center of Advanced Studies Conference (CASCON)*, Toronto, Canada, October 2003.

31. Want, R., Pering, T. and Tennenhouse, D. “Comparing Autonomic and Proactive Computing”. *IBM Systems Journal on Autonomic Computing. Vol. 42*, November, 2003, pp. 129-135.

32. Xu. X., Martin, P. and Powley, W. “Clustering Buffer Pools in DB2/UDB”. *Center of Advanced Studies Conference (CASCON)*, Toronto, Canada, October 2002.

Appendix A Relations and Data Descriptors

Relations of the TPC –B Benchmark

Branches (**bid** [primary key], bbalance, filler);

Tellers (**tid** [primary key], bid [foreign key], tbalance, filler);

Savings Accounts (**sid** [primary key], bid [foreign key], abalance, filler);

Chequing Accounts (**cid** [primary key], bid [foreign key], cbalance, filler)

Buffer Descriptor

```
/* Relevant fields of each buffer descriptor. Unique for each buffer in the buffer pool/s,
new fields highlighted in bold */
// Flags for buffer descriptors
#define BM_VALID                (1 << 0)
#define BM_FREE         (1 << 1)
// some other flags here
typedef bits16 BufFlags;
typedef struct BufferDesc{
    // stores the buffer descriptor's pointers to next and previous buffers in the free list
    Integer   freeNext;
    Integer   freePrevious;
    BufferTag   tag;            // identifies which disk block the buffer contains
    SHMEM_OFFSET  data;         // pointer to data stored in buffer
    Integer   buf_id;           // buffer's identification number (from 0)
    Integer  bp_id;             // buffer pool associated with this buffer
    BufFlags   flags;           // see bit definitions above
    Integer   refcount;         // number of backends holding pins on buffer
} BufferDesc;
```

Object Descriptor

```
/* Relevant fields for each data object in the DBMS. Unique for each data object, new
fields highlighted in bold */
typedef struct ObjDesc{
        Integer o_id;                   // object identification
        Stats_Info stats;               // statistics collected on the object
        Integer bp_id;                  // buffer pool associated with this data object
      Integer  ref_cnt;                 // number of times the object has been referenced
} ObjDesc;
```

Object Queue

```
/* New data structure - a shared memory queue that stores all data objects associated with
a buffer pool */
typedef struct OBJ_QUEUE{
        list of object id's
        Integer    size;
} OBJ_QUEUE;
```

Buffer Pool Descriptor

```
/* New data structure – stores information for each buffer pool */
typedef struct BPDesc{
        Integer bp_id;
        Integer Num_Descriptors;          // number of buffer descriptors in the buffer pool
        Integer New_Num_Descriptors;      // size after executing the sizing algorithm
        OBJ_QUEUE *objqueue;              // stores the head of the object queue
        Integer Free_List_Descriptor;     // for easy lookup of the buffer pool
        Integer LR;                       // number of logical reads performed
        Integer PR;                       // number of physical reads
        double DAT;                       // total data access time for the buffer pool
} BPDesc;
```

Appendix B Sample Results

BP(0): LR1 = 36000, LR2= 36000, PR1= 36000, PR2= 0, TotalDAT1= 4236180, TotalDAT2=0, size1= 64, size2=96

BP(1): LR1 = 17904, LR2= 17952, PR1= 0, PR2= 17952, TotalDAT1= 0, TotalDAT2=2131531, size1= 64, size2=32

---- Greedy Algorithm (DAT) -------

66 62 35.449345

72 56

60 68

72 56 3.745369

78 50

66 62

78 50 0.542377

84 44

72 56

84 44 0.507680

90 38

78 50

87 41

81 47

81 47 0.349868

84 44

78 50

82 46

80 48

New Buffer Pool Sizes 81 47

BP(0): LR1 = 18000, LR2= 18000, PR1= 0, PR2= 18000, TotalDAT1= 0, TotalDAT2=2235709, size1= 81, size2=32

BP(1): LR1 = 35712, LR2= 35904, PR1= 35712, PR2= 0, TotalDAT1= 4321176, TotalDAT2=0, size1= 47, size2=96

----- Greedy Algorithm (DAT) --------

---- Greedy TLR-------

59 69 0.351578

65 63

53 75

53 75 0.287025

59 69

47 81

56 72

50 78

56 72 0.266651

59 69

53 75

57 71

55 73

55 73 0.262339

56 72

54 74

New Buffer Pool Sizes 55 73

Appendix C Confidence Intervals

The PostgreSQL TPC- B workload consists of a singe transaction that is executed multiple times. The results obtained are fairly consistent. The same is true for the select only and extended select only workloads that execute the same select statements in every transaction. However, to be more scientifically accurate, boundaries are specified to express the confidence in the data collected.

Consider N runs for the same experiment: X_1, X_2, …, X_N. The sample mean is given by

$$M = \frac{1}{N}\sum_{i=1}^{N} X_i$$

and the variance of the sample values, S_x^2 is calculated as

$$S_x^2 = \frac{1}{N-1}\sum_{i=1}^{N} (X_i - \overline{X}_N)^2$$

The standard deviation (S_M) of the sample mean can be estimated as $S_x / \sqrt{N}$. Since an estimation of the sample mean of the experiment runs is required, the t rather than the z-distribution is used. The values of t are larger than the values of z so confidence intervals when S_M is estimated are wider than confidence intervals when S_M is known.

The t-distribution is dependent on the degrees of freedom and the level of confidence. The degrees of freedom is calculated as N-1. Since we repeat all the experiments 3 times, the degrees of freedom is 2. We use a 95% confidence interval. The

lower limit on the confidence interval is $M - t\ s_M$ and the upper limit is $M + t\ s_M$. All our collected data falls within this range.

Glossary of Acronyms

ADBMS	Autonomic Database Management System
DBA	Database Administrator
DBMS	Database Management System
BP	Buffer Pool
LRU	Least Recently Used Page Replacement Policy
TPC- B	Transaction Processing Performance Council Benchmark B
TPC- C	Transaction Processing Performance Council Benchmark C