UNIVERSITY OF OSLO
Department of informatics

Porting and Performance Analysis of InTraBase from PostgreSQL to Oracle

Master thesis

60 credits

Marius Frøisland

30 Jul 2007

Abstract

Network monitoring is an important task in the Internet. It is also a very complex task. Especially managing the measurement data, since the amount of data involved makes it difficult to organise and store. InTraBase was introduced to solve the data management problem experienced with other network monitoring tools. InTraBase stores all the information inside a database and perform the analysis directly in the database.

Our hypothesis is that a system like InTraBase will perform better if implemented in Oracle. We base this on the claims of the database vendors, along with the fact that Oracle has a collection of tuning tools and advisors. These tools are easy to use because they can be accessed through the GUI in Enterprise Manager. ;

pgInTraBase is a PostgreSQL-based implementation of InTraBase. During the porting of pgInTraBase we identified a small set of differences in PostgreSQL and Oracle's syntax. For instance Oracle does not have '`LIMIT`' or '`OFFSET`' clause for SQL. We have found simple ways to work most of these clauses. In most other cases we found equivalent functions in Oracle. In pgInTraBase the queries were concatenated as a string to include tablename and various variables and then the string was given as a parameter to be executed. In oraInTraBase we used a synonym to include tablename, and bind variables for other variables. This was done in order for the parser to be able to reuse the execution plans.

We see the process of analysing data as a two phases process:

1) Upload – upload packet headers into the database from a flat file.
2) Analysis – analyse the data to find the root cause for throughput.

Systematically tuning of the procedure that populates the database with packet headers, reduced the running time of the Oracle prototype to approximately 20% of the original time when using 1GB trace. Compared to PostgreSQL the execution time of the tuned process is almost cut to half. The Oracle version performs significantly better than the PostgreSQL version in the upload phase.

The process of analysing the packet headers was also tuned in a systematic way. After tuning, the analysis of a 50MB trace takes 2% of the time it used before tuning. Compared to PostgreSQL a 50MB trace analysed in Oracle takes only about 42% of the time. Comparing the running time of traces of 10MB, 50MB, 100MB and 1GB show the same results, while using a 10GB trace Oracle needs 182% of the time PostgreSQL does. However, Upload is much faster in Oracle than in PostgreSQL. So much that if we add the time Oracle use for both Upload and Analysis this takes less time than the Upload alone in PostgreSQL. This highlights the very different performances of the two DBMS depending on the task.

Acknowledgements

I would like to take this opportunity to thank Vera Hermine Goebel and Matti M. Siekkinen for being excellent supervisors. Without their expertise and motivating meetings, the process of writing this thesis would not have been as educational and productive. A student could not ask for better supervisors. I feel privileged to have had such excellent supervisors.

I would like to thank my family and friends for their support during this period.

Table of Contents

List of Tables

List of Figures

1 INTRODUCTION

1.1 Motivation

InTraBase is a database system designed for packet-level analysis of network traffic. The prototype of InTraBase is designed to perform root cause analysis of such traffic. The root cause analysis can help explain why we see different throughput compared to what we expected.

In this thesis, we address two hypotheses:

1. Whit reasonable effort possible to port a complex system like pgInTraBase from PostgreSQL to Oracle.
2. Implementing InTraBase with Oracle instead of PostgreSQL will speed up analysis and improve extensibility.

Modern computer systems can be used to perform very complex tasks. Still, it is not always enough to be able to complete the task, we also have to complete it within a certain time frame. This is true in network monitoring too. It is of less value to know that we were under attack last year as opposed to knowing that we are under attack now or one minute ago. This is one example of a network monitoring task where it will be beneficial to obtain the information without any delay, which allows us to employ more efficient countermeasures.

Monitoring the Internet becomes a more complex process with every new protocol or feature added to the network. As the complexity increases so does the execution time. From a user's perspective it would be preferable to have the result as close to real-time as possible.

Another element that makes monitoring the Internet difficult, is the amount of data. In our study we are looking at packet headers. To perform the root cause analysis it is imperative that every packet is taken into account. We do not need to look at the body of the packet, as the packet headers contain all the necessary information. The packet header is at average about 68Bytes of the entire packet's 1500Bytes[1], which means that in about three minutes a gigabit link fully utilised will produce 1 GB of packet headers. Managing this amount of data in a structured and efficient manner, is not a trivial task. InTraBase uses a database management system (DBMS), and has successfully implemented such an approach.

In an ideal situation, the system is able to perform the analysis at the same rate as the input. This is called on-line analysis. There are systems today that can do this given sufficient resources (memory and CPU), for instance *Tcptrace*. They do this by reducing the amount of work and complexity of the analysis process. However, such an approach reduces the accuracy of the result and is therefore no option for us. We will instead try to optimize performance for the process in order to keep the complexity, but reduce the

1 Assuming a MSS of 1500 bytes (common case) for TCP, and capture size of 68 Bytes of the header.

runtime.

Two of the main advantages of PostgreSQL are that it is a free and open source that allows you to look at the source code to see what is going on [1], as well as modifying and/or copying it. A major motivation the pgInTraBase developers had for choosing PostgreSQL in the first place, was that it is an object relational database that can be extended through procedural languages (e.g. PL/pgSQL). PostgreSQL's PL/pgSQL has less features and capabilities compared to Oracles PL/SQL [2]. Since the main functionality of pgInTraBase is written in this procedural language, we see it of utmost importance that these languages have maximum capability. One often overlooked fact, is the skills of the technicians who implement the system. This is often the deciding factor when it comes to the end result [3].

We decided to port the prototype of InTraBase to Oracle because we believe an Oracle version will perform better than the PostgreSQL version. The reason for this is that Oracle is a commercial and very widely used system. Oracle has more tuning features and an easier interface. In addition, it has a richer procedural language, which will allow us to easily extend the system with new features.

1.2 Problem Description

The task can be divided into two main parts. The first part is porting the prototype from PostgreSQL 7.4.2 to Oracle 10g.

Every part of the system has to be modified to work with Oracle. This includes the upload sequence, where the packet headers are copied into the database. Numerous functions have to be translated from PostgreSQL's PL/pgSQL to Oracle's PL/SQL. Functions in C++ and PL/R have to be remodelled for Oracle or worked around. Even though both systems support the SQL standard, they are not identical in their syntax, and PostgreSQL has some features and capabilities that Oracle 10g does not have. We have to work around these features and find new solutions for the Oracle version. We need to make modifications to take advantage of features that are specific to Oracle. Finally, the system has to be validated. The ported function should precisely duplicate the functionality of the original system, even if different methods are used to achieve the result.

When we have a running prototype, we wish to compare the performance of the new oraInTraBase with the old pgInTraBase. In order to have comparable results, they both have to run on the same server, so that they share the same environment. The first part of this task will be to install and prepare the server. When this is completed, we can start the measurements. We are mostly concerned with the overall runtime of the system. How much time is required to analyse traces sized 10MB, 100MB and 1GB? To get a better understanding of where the two systems have their strong and weak, points we will also measure individual parts of the system. We will focus on three parts:

- Upload: Before we can do the analysis of the packet header we need to import the headers to the database. This is performed by a Java program named Upload.

- C-Query: Previous studies of InTraBase identified a query that was responsible for most of the load on the database. This query was called common query (C-Query). The assumption is that if this query performs well the analysis process will do so, too.
- Analysis process: The entire process that performs the root cause analysis. It consists of executing multiple functions that each performs a particular piece of the root cause analysis. The process will in turn populate various tables with the results of the analysis.

1.3 Outline

In Chapter 2 we introduce the problems and challenges associated with network monitoring by showing the complexity of network monitoring and motivating for the use of database systems to manage monitoring data. Before we can understand how and what InTraBase does, we need to understand the most common performance optimisation techniques used in database systems. A overview of these are given in Chapter 3. Chapter 2 and Chapter 3 establishe a vocabulary used throughout this thesis.

In this thesis we port the prototype of InTraBase (pgInTraBase) from PostgreSQL to Oracle. Chapter 4 explains the concept of InTraBase and pgInTraBase. This chapter also establishes a base line of the thesis, describing the implementation available to us when we started this thesis. This lays the foundation for discussions in Chapter 5 and Chapter 6.

In Chapter 5 we look at the features used to implement pgInTraBase and discuss similar features in Oracle.

Chapter 6 compares the features of pgInTraBase and oraInTraBase, and explains how we ported pgInTraBase. The purpose of this comparison is to evaluate which of the systems is most likely to give the fastest implementation of InTraBase. Then we give a detailed explanation of how we ported pgInTraBase to Oracle (oraInTraBase).

After successfully having ported pgInTraBase into oraInTraBase we wanted to optimise and evaluate the new implementation. However, before we can do this we need to look at the tools and techniques available in Oracle for this purpose. In Chapter 7 we study these tools and how they relate to the optimisation techniques presented in Chapter 3.

In Chapter 8, we systematically optimise the execution time of oraInTraBase. We perform measurements of pgInTraBase and oraInTraBase in order to compare the performance of the two systems, using tools presented in Chapter 7. After having completed the optimisation and evaluation of oraInTraBase, we summarise and evaluate the thesis in Chapter 9. In the end we present possibilities for future work in this field.

The Appendices contain the installation guide for oraInTraBase, content list for the DVD distribution of the source code, and more details about trace files.

2 NETWORK MONITORING

In this thesis we discuss porting of Integrated Traffic Analysis Based on Object Relational Data Base Management System (InTraBase) [5], [6], [7], [8]. In order to understand the complexities and challenges in network monitoring we now give a general introduction to network monitoring.

Today the Internet has 1 billion users, the annual growth in the user population for 2006 was 202% [4]. The amount of traffic on the Internet makes it difficult to measure because of its volume. There are many reasons why we want to monitor a network or certain attribute of a network. Here are a few of the many examples:

- Internet Service Providers (ISPs) need to know how their network is performing. An ISP sells a certain uplink and downlink capacity to their customers. Because of this it is important for the company to know how many of these subscriptions they can sell before the network is uable to provide the promised service level. Another metric of interest to an ISP is when the current performance is below a certain threshold. For the customer, it is important to know that they are receiving what they are paying for. In cases that they experience lower performance than expected, it is important to know the reason for this lower performance. InTraBase and root cause of throughput can be used to answer many of these questions. For more information about root cause analysis see Section 2.5.

- By monitoring the traffic on a network it is possible to predict which applications and services are being used. Even if the application port is changed from default we can still predict which application is running by looking at the traffic pattern and type. Users playing an on-line game would want low round trip time (RTT), usually measured as ping time. This same user might not need much throughput. Users downloading files on the other hand, would want a high throughput but would not necessarily care about the RTT. By knowing which applications are being used, we can tune the network to the user's needs. We would also like to discover which, if any, users are running unwanted applications or engaging in illegal activities.

Network Monitoring is a cyclic activity as shown in Figure 1. There are three phases that have to come together to make a network monitoring system successful. First we have to gather meaningful data in a format that we can use, this is the collection phase. Then we need to perform the analysis on this data. Finally we have to react upon our findings. Through

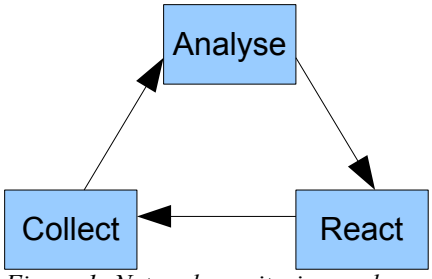

Figure 1: Network monitoring cycle

these phases we have to manage the data in such a way that we can reproduce the result and reuse computed data.

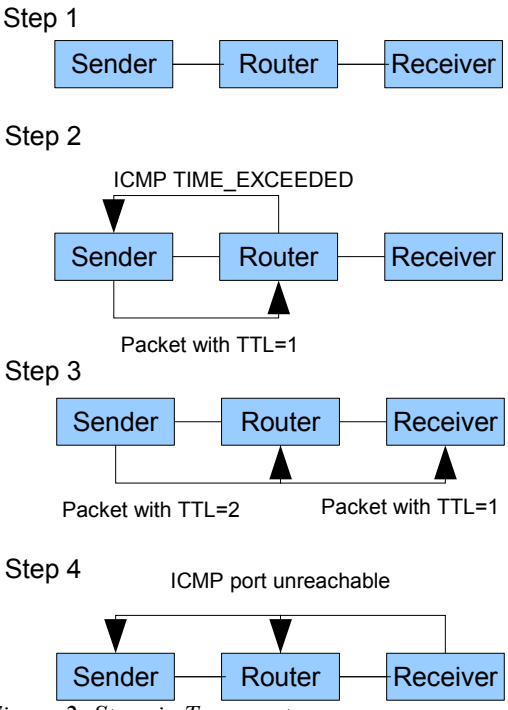

Figure 2: Steps in Traceroute

Measuring the Internet is not a straightforward process. Querying the Internet for performance information is only possible in a limited way with the Simple Network Management Protocol (SNMP), which requires privileged access to each router. Instead we take advantage of certain protocols to derive a result. We will make an illustration with an example not directly related to InTraBase, but which shows how difficult it is to analyse the Internet.

Traceroute is a much used tool to find the routes between two locations on the Internet. It does not query the Internet about this information, as the network in itself has no intelligence. Traceroute sends messages that will trigger the ICMP (Internet Control Message Protocol) message from other servers/routers along the route. Each IP packet has a TTL (time-to-live). When a router receives a packet, it decreases this number by one before sending it towards its destination. If a router discovers a packet with TTL=0, it will send an ICMP TIME_EXCEEDED message back to the sender indicating that the packet expired at this router [9]. (Step 2 in Figure 2.), This allows Traceroute to find the route to the destination by sending subsequent packet with an increasing TTL, starting from 1. Each sent packet will generate an ICMP TIME_EXCEEDED message until one of the packets reaches its destination. (Step 3 in Figure 2). By collecting the locations from these ICMP messages Traceroute produces a list of all the hops between itself and the destination. The final message is ICMP port unreachable either if a packet has done the maximum number of hops 30[2] or the destination is reached. This is just one example of how we must go about measuring the

2 The -m flag allows you to alter this number.

Internet.

When we say measuring the Internet, it is a bit unclear what we actually are measuring. It is possible to measure the infrastructure, e.g. capacity of links or end to end paths, topology of the network, the traffic in the network or the applications responsible for the traffic [9]. In this thesis we will measure the traffic. Note that applications are often measured through the traffic it produces. We now discuss the collection phase and analysis phase in detail. Related to collecting measurement data, we also discuss management of this data. As for the analysis part, we explain in detail a particular type of analysis; root cause analysis of TCP throughput, because this is the type of analysis that InTraBase is tailored for.

2.1 Collection of Measurement Data

Gathering information from the network can be done in several ways. In a smaller network, one organization may control all the devices, or at least all the traffic routing devices. In such cases, we will have access to traffic and monitoring data directly from the routers. In many cases this can be very helpful, but there is a limitation to what information a router can provide. Normally, they provide aggregate per interface.

In the Internet no single organization has control over all the routing devices, so this type of monitoring is not available here. It can be used inside one single or several cooperating organization's domain since they will have the necessary access to the routers. Different measurement techniques have been, and are being, developed for the Internet.

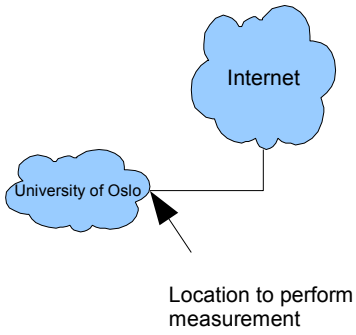

Figure 3: Where to perform network monitoring.

One can gather monitoring information by looking at all the packet headers, from both sent and received packets, on a single device. The post gathering analysis would be easiest if we could do this at both the sending and the receiving side [8] simultaneously, especially for TCP because of difficulties calculating round trip time. It is nearly impossible to measure both sender and receiver in a production system, as we would have to have a measuring point at every possible location the user might send or receive packets from. In a controlled environment, it is possible to measure both sender and receiver. For an ISP, having a measuring point at every customer is likely to be too difficult. Therefore, an ISP or a University for instance, would

probably prefer to gather information only at the edges of their network.

By looking at these packet headers we can learn many things about traffic going to and from the network. Looking at them as they pass a location in the middle of the path between the sender and the receiver complicates the analysis. The reason for this is that we do not see the packets as the TCP layer of the sender or receiver does. Why this cause problems can be best illustrated by an example.

A sends a packet to B. As a monitoring point between A and B we can observe one of two things. Either we can observe the packet, or we do not observe it. If we do not observe it then it was probably lost between A and us. If we observe it at the measurement point, but do not observe an ACK from B to A, there are several possibilities: The packet never arrived at B. The packet arrived at B and B sent an ACK, but the ACK was lost between B and the measuring point. Even if no packets are lost we still are unabel to tell when it arrived at B.

This is just one example illustrating some of the possible events complicating the analysis. Because of this we have to employ complex algorithm to understand what is actually happening. In this thesis we will look at a system designed to monitor packets at a point in between the sender and receiver.

2.2 Managing Measurement Data

Analysing non-sampled packet headers is a difficult task as the amount of data can become very large. Therefore, it is of utmost importance how the data is stored and handled during the analysis process.

Traditionally, packet headers, or other measurement data, are stored in flat files (e.g. In raw Tcpdump files). The same is true for intermediate data. The effectiveness of this approach is largely dependent on the analysts sense of order and structure. If an analyst structured his files in an not easily understandable way it can be difficult, if not impossible, for another analyst to understand the data. The sheer amount of files that such analysis produces complicates the process further, especially when considering temporary results and more than one pass over the data or intermediate data. Furthermore, flat files become very cumbersome to work with as they become larger [1]. The reason for this is that finding any given entry in a flat file takes a long time as they are not organized in a fashion to provide easy lookup, but are instead organized in a sequential manner.

There are some tools for analysing trace files, but making ad-hoc script is a very popular way to analyse trace files. A major disadvantage with ad-hoc scripts is the difficulty to reuse them and to reproduce the result you achieved through them. This is mainly due to the unstructured process of both the analysis and storing of intermediate results.

DBMS have since the 70's been used to store large amounts of data. It is commonly accepted that a DBMS is a good way to store data in an organized fashion [10]. Companies all over the world entrust their data to databases with good results. A natural question is: Why do we not use a

database for storing data and procedures used in network monitoring? The common answer has so far been: “We tried them but they were too slow” [11]. InTraBase showed them to be wrong [5], in the sense that it is a feasible task to store data in a database and use it for network monitoring.

Storing this type of data in a database will give several benefits compared to flat files. To find a packet header in a flat file you will at average need to read half the file before you find it. This is not a problem in small files, but when the file becomes large, it can cause significant performance issue. In a database it is possible to employ an index and find it in a few disk reads. The index will however consume additional disk space. The cost of disks is very low these days, so this is not a significant disadvantage.

Using a database for data storage will create natural structure in the data. In this way it is easier to reuse intermediate results than with flat files. Furthermore, it can be easier for other researchers to reuse the data.

There exist systems and methods for doing traffic analysis today. Some of them are: ad-hoc scripts, specialized tools (tcptrace)[12], toolkits (CoralReef)[13]. These tools do not scale linearly. Therefore, doubling the amount of data more than doubles the amount of time required. Which leads to fairly low upper limit in the size of trace files that can be analysed. For example even with a heuristic to determine the end of a connection Tcptrace was unable to process files with size larger than 6 GB [8]. Because of the large amount of connections, Tcptrace ran out of memory and was unable to complete. There are also some systems using a database, these systems are all being developed/used by ISP's and are unfortunately not publicly available (e.g Gigascope [14]).

InTraBase was developed as a pure database alternative to these other systems for traffic analysis, and in contrast to the ISP developed systems InTraBase is publicly available. A prototype has been developed [5]. This prototype was developed using Linux and PostgreSQL. InTraBase proposes to use a DBMS to provide the infrastructure for the analysis and management of data from the measurements, related meta-data, and obtained results as well as for doing the analysis.

As earlier mentioned the flat file and ad-hoc script method has its management issues, but that is not the only problem with that method. Many analysis processes require several iterations and it can be cumbersome for the researchers to manage such an iterative analysis [10]. It would be preferable if the system had enough “comprehension” of the data to make the right choices of which process to start next.

Another benefit of the database approach is that it becomes relatively easy to combine data from more than one source. The reason for this is that the analysis process only see the packet headers as they are presented in the packet table. It does not matter if one populates this table from many different sources, as long as the connection identifier is unique for each connection. We would need a separate upload procedure for each source. Or we could combine data sources after they have been uploaded by joining them on a common attribute (e.g. Timestamp).

2.3 Analysis

We focus our discussion around TCP measurements. Many of the same principles and techniques we discuss in this thesis could be used for UDP. UDP is less complex, as it does not send acknowledgement on packets. This reduces complexity of the analysis in some area, but complicates it in other. InTraBase was designed with TCP in mind.

As mentioned in the Section 2.1, we want to collect the data at an intermediate point, not at the end of each transmission. This complicates the analysis process because we do not see the packet at the same time the receiver does. The receiver might never even see that particular packet, even though the monitoring point saw it. Because it may be lost after passing the monitoring point. This makes it more difficult to understand why certain things happen. For instance: In the trace we see two packets with identical sequence numbers, which is an obvious retransmission. As mentioned earlier, in this chapter, there can be many reasons for this retransmission. It is a complex task to determine which explanation is the correct for the observed behaviour. For more on this topic look at [10].

2.3.1 Off-line and On-line Traffic Analysis

We can group traffic analysis into two groups, off-line analysis and on-line analysis. Both have advantages and disadvantages. By on-line, we mean that the analysis is performed on the data as soon as it is generated. Off-line means that we gather data for a period, and then, when all the data is collected we start analysing it.

The on-line analysis has the advantage that it can give response immediately and one do not have to store the data. For instance for quality-of-service or intrusion detection this can be essential. The main disadvantage is that on-line analysis usually is only able to do one pass over the data. It also often has problems keeping up with the rate of data being produced and will have to use different data reduction techniques to cope with the large amount of data. Some of these techniques are presented in Table 1. It shows the advantage, disadvantage and the estimated data reduction for each technique.

Measured data	**Data reduction**	**Advantage**	**Drawbacks**
Full packets	None	Have it all	A lot of data, privacy concerns
Packet headers	Around 1/20 (70 B hdr vs 1.5 KB pkt)	Have most of knowledge in summarized format	Still a lot of data
Flows	a/avg(flow size in pkts) x 1/20	Data reduction, feasible on-line (Cisco's Netflow)	Loose packet details, connections needed to be reconstructed
Sampled headers/flows	Depends completely on the scheme	Improved data reduction	Not usable for al types of analysis e.g. Loss estimation.

Table 1: Different measurement approaches to achieve data reduction. Data reduction values are only indicative [8].

When doing on-line analysis the two phases of the analysis process become intertwined; the process of gathering data is also to some extent the process of analysing it. Traceroute that was explained in the introduction of this chapter is an example of this.

Off-line analysis has the "luxury" of time and can do several passes and complicated analysis, but it will give response with some (hours, days, weeks) delay. In this project, we are mainly concerned with off-line analysis with database approach.

2.3.2 Active and Passive Traffic Analysis

Another way of grouping measurements is passive and active measurements. It is fully possible to have passive off-line, passive on-line, active off-line and active on-line measurements as these two features of the analysis process are independent.

Active measurements is done by sending probe packets in to the network and observe their behaviour. This is especially suitable for finding end to end properties [8] or infrastructure (link capacities, available bandwidth and topology for instance). One way of doing this is to send packets from location A with interval Δt and to observe the same packets at location B and see how the interval has changed. By analysing the packet dispersion we can infer what is the available bandwith in the path. Another way could be to send a large amount of data and just measure the time it takes before it arrives at location B. Ping and Traceroute are two examples of active measurement tools. Both of these tools only require a single measuring point, but many of the active measurement techniques require two or more measuring points.

Passive measurements is a paradigm that is more general than just traffic analysis but is mostly used for analysing traffic. It is done by observing users traffic, analysing patterns and query supported information about state of network devices. We can use passive measurements to discover throughput, bottlenecks and so forth. Passive measurements can be further divided into three different types; SNMP/RMON based measurements, packets monitoring, and flow measurement. In order to perform SNMP/RMON based measurements one needs access to all the devices that one want to measure. Statistical data from the devices are usually not made available to anyone outside the ISP. The routers usually only produce highly aggregated values per interface [8]. Packet monitoring can be done by copying packet headers, for both received and sent packets, and then by analysing them. Flow measurement are done by aggregating packet into groups of packets. In TCP connections could be the basis for such groups.

Active measurements are more intrusive on the network than passive measurements. InTraBase uses passively measured (non-sampled) TCP/IP packet headers.

2.4 TCP

2.4.1 Flow Control

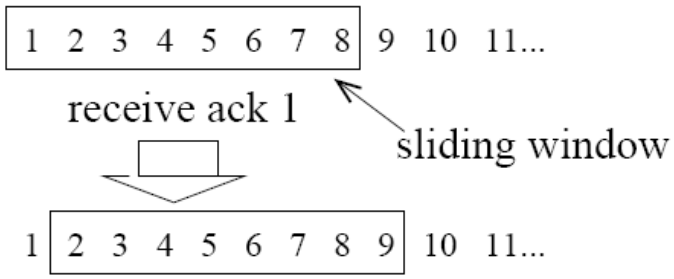

Figure 4: Sender's window slides [8]

TCP uses a technique called Sliding Window to enable flow control. The receiver announces the size of its available buffer as 'receiver advertised window' (rwnd).

This technique allows the receiver to adjust the rate of new packets. Figure 4 shows how this is perceived by the sender. It has a window size of eight[3] in this case. The sender cannot send packet nine before it has received acknowledgement for packet one. When the acknowledgement is received by the sender, its slides its window forward. Note that TCP does not acknowledge every packet. It uses “cumulative acknowledgements,” but normally sends one ack per packet unless “deleayed” acknowledgement-strategy is used (one acknowledgement per two packets). Acknowledgement includes the sequence number of the next packet it expects to receive. For instance, if it receives packet 1, 2, 3, 5, 6 it would acknowledge that it needs packet 4 which implicitly informs the sender that it has received packet 1, 2, and 3, but it would not send acknowledgement with packet nr 6 until it has received packet number 4, because it only acknowledge the last received packet in a complete sequence of packets.

2.4.2 Congestion Control

Congestion control is achieved through a Congestion window (cwnd). For the sender, it works under similar manner as rwnd. The transmission rate of the sender is the minimum of cwnd and rwnd. There is, however, a difference in how the window size is defined. As TCP has evolved, more sophisticated congestion control techniques have been implemented. We will look at only a few variations.

2.4.2.1 Slow Start and Congestion Avoidance

Slow start means that the cwnd is set to one, and each time an acknowledgement is received the cwnd is increased. When a packet is lost, cwnd is reset to 1 again, but a threshold - slow start threshold (ssthresh) - is set to half the size of the cwnd at the time the packet was lost. It starts over with a cwnd of one, and continue to increase for every acknowledgement

3 Window size is measured in number of outstanding bytes. In this example it is simplified to number of packet for ease of understanding.

until cwnd reaches ssthresh. After this TCP enters congestion avoidance mode, the cwnd is increased only when a full cwnd worth of packets has been acknowledged. Packet loss is handled as before; setting ssthresh to half the current cwnd and setting cwnd to one.

2.4.2.2 Fast Retransmit

The only reason for the receiver to continue to send the same acknowledgement, called duplicate acknowledgements, is that it has already received a later packet, but is missing one in between. For instance it has received packets 1, 2, 4, 5. Then it knows that there should be a packet 3 that it is missing. If, on the other hand, it receives packets 1, and 2 it might be the case that 2 was the last packet, and it will send only one acknowledgement.

In the Tahoe version of TCP, if the sender receives multiple copies of the same acknowledgement, it is interpreted as an indication of the fact that this packet was lost, especially if it receives three or more copies. It will retransmit the packets immediately, ignoring designated time out values. As described above in Section 2.4.2.1 the sender enters slow start mode, setting ssthresh equal to half of cwnd and resetting cwnd to one.

2.4.2.3 Fast Retransmit & Fast Recovery

When the senders receive more then one acknowledgement, it knows that the path to the receiver is not blocked, since some of the packets have arrived. This might call for less backing off than if all the packets sent after a certain point were lost. The Reno version of TCP addresses this situation. After three duplicate acknowledgements have been received, Reno sets ssthresh to half of cwnd, as normal. In fast recovery instead of setting cwnd to one and entering slow start mode Reno sets cwnd to ssthresh plus three. Each time it receives another duplicate acknowledgement, it is increased by one. When REno receives the first new acknowledgement, it sets cwnd to ssthresh and resume normal operation. In other word Reno avoids entering slow start mode.

This technique was improved in NewReno. The motivation for improvement was that when more than one packet is lost, the fast recovery mode is ended and one has to start over with the time out for the next lost packet. For instance, if the cwnd covers packets 2, 3, 4, 5, 6 and packet 2 and 5 was lost, then the receiver would send duplicate acknowledgement for packet 1 to indicate the loss. Sender would enter fast retransmit and fast recovery mode, but this mode would end when it receives the acknowledgement for packet 3. Then the receiver would have to send three duplicate acknowledgements for packet 3 before fast retransmit and fast recovery mode would be resumed. What NewReno proposes is that the fast retransmit and fast recovery mode is not ended until the last packet that was in the cwnd when the mode started, is acknowledged.

2.5 Root Cause of Throughput in Long Lived TCP Connections

In order to understand how we can analyse the root cause of TCP throughput

we have to understand how TCP works. Details about the root cause of throughput in long lived tcp connections can be found in [8]. Here only a overview will be given.

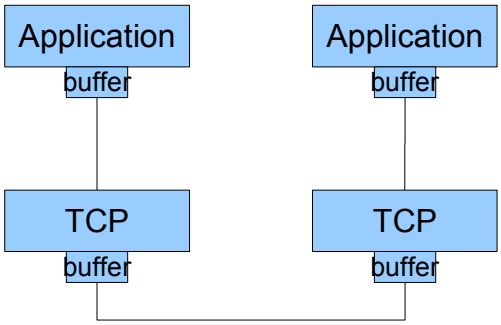

Figure 5: Packets path from sender to receiver

Between the two TCP boxes in Figure 5 there are several layers, including the hardware. We will not go into details about this, but we need to understand how it works on a general principle. The layers between TCP transports packets from one TCP buffer to another through the network. It is possible that the packet never arrives, or that it arrives and finds the buffer full. Both cases can be summarised as one; the packet does not end up in the receiving TCP buffer, but is lost. To avoid loss of data, TCP demands that the receiver acknowledges all correctly received packets in sequence in its buffer. Only when this acknowledgement is received by the sender, does it removes the packet from its sending buffer. Using flow control, as described in Section 2.4.1, the receiver can limit the sending rate.

A root cause analysis for throughput tries to discover what the true reason for the experienced throughput is.

InTraBase implements the algorithm to find the root cause for the given throughput in the network. It will identify the limiting factor as either:

- Application
- TCP limiting factors
  - TCP receiver window
  - TCP protocol mechanisms (e.g. slow start, congestion avoidance)
- Network layer
  - Shared bottleneck
  - Unshared bottleneck

First, traffic is divided into two types of groups bulk transfer periods (BTP) and application limited period (ALP). BTP occurs when the application supplies a steady stream of data, at least at the rate which TCP is able to send. In periods of ALP the application does not have enough data to send, and therefore do not fill the capacity of the path. In many peer-to-peer applications, like BitTorrent, the user can define constraints of how much

capacity the application is allowed to consume. This type of connections would result in ALPs. It is not uncommon for applications to send large amounts of data in bursts and then have long time periods where only keep alive messages are sent.

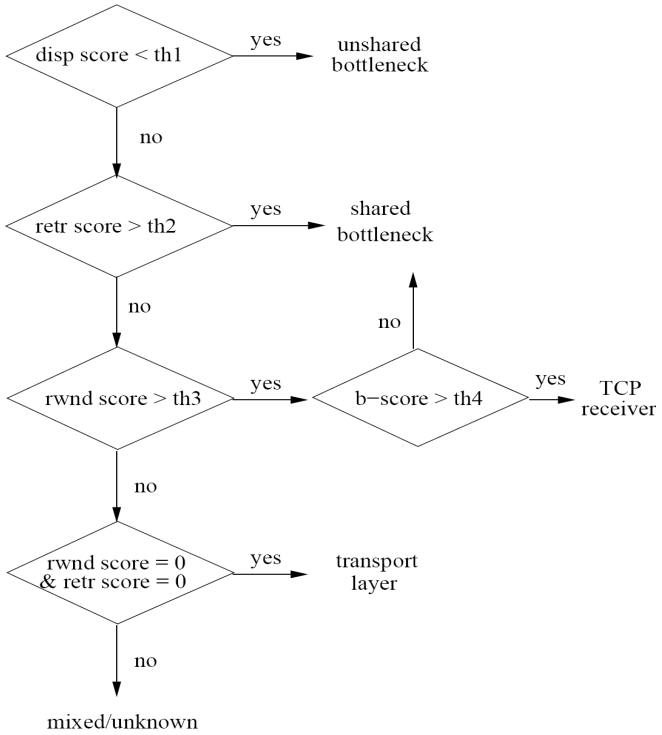

Figure 6: How BTP are classified in to limiting causes by thresholds [8].

Identifying which group traffic belongs to, can be deducted by looking at the size of the packet body[4]. TCP will try to send packets that have size equal to Maximum segment size (MSS), which is the largest packet size the path is able to send. If TCP sends packets of smaller sizes then MSS it indicates that the application is not supplying enough data. Otherwise it would send packets of size equal MSS. Therefore, this is considered an indication of an ALP. The second sign of being in a ALP, is that there is a long period with no packets being sent.

In the first part of the analysis process, the ALP is filtered out. This is an important step because in an ALP, the traffic is limited by the application rather than by TCP or the net. So a study of this period will at best result in information pertaining to the application. It would be a mistake to include this in a study that focuses on TCP and IP level root cause [10].

The next step after filtering out ALPs is to study the BTP and classify the period into its limitations

- Unshared bottleneck

4 The packet body's size is an attribute in the TCP header.

- Shared bottleneck
- TCP receiver window
- Transport layer
- Mixed or unknown

Certain aspect of the traffic can be expressed in a series of scores. We will not go into the details of the scores. They are described in more details in [8]. By looking at these scores, we can see what was the limiting factor during that period. This largely depends on threshold values that have to be tuned and configured. This is done by doing controlled experiments and seeing at which value the score represent the actual cause that the experiment has created. Once you have these thresholds (th1-th4) calibrated, the scheme in Figure 6 can be used to find the limiting cause.

3 Concepts for Database Performance Optimisation

In this chapter we will give a basic introduction to some of the most common techniques used in database to achieve the efficiency that we require and expect of a modern database. These techniques are in use in most databases to day.

3.1 Logical Optimisation

In this section we will look at how the DBMS internally tries to optimise how it execute a query. SQL is a query language where the user defines which set of tuples they want returned. This is very different from, for instance, programming languages where the user tells the computer exactly what to do. Because of this difference the DBMS has to transform the query into such instructions on its own. These instructions are expressed as relation algebra, defining the operations the DBMS will perform on the data. A set of such operations are usually called an execution plan, or just plan.

In most cases there is more than one plan that will generate the correct result for a query. For instance if the user requests all persons that has a first name starting with an A. We can imagine that the DBMS will fetch all the tuples from disk, then discard all those starting with other characters. Or the DBMS can try to figure out which people have a name starting with A and fetch only those from disk.

Therefore, databases employ different techniques to select the most appropriate plan. In the past the plans were created by following a set of rules, but this method is being replaced by cost based approach. In the cost based approach the system generates several execution plans and estimates the cost of performing each of them. Then it selects the one with the lowest estimate. To generate good estimate the database needs statistical information about the performance of the hardware as well as about the tables, indexes and such that are involved in the query. This task is performed by the query optimiser.

The query optimiser might not be able to produce the perfect execution plan. In such cases the user can utilize techniques to make the DBMS create better execution plans. This will be explained more in the next section SQL tuning.

3.2 SQL Tuning

The query optimiser is far from perfect. It is often unable to produce good plans for a given query. In such cases it can benefit from some help from the user. In this section we look at how one can improve performance by changing the query one passes to the DBMS. This is usually done in four forms:

1. Statistics
2. Hints
3. Using different SQL clauses

4. Hard coding an execution plan

The first one is easy to implement in most databases. You can tell the database to perform the analysis on one or more tables or indexes. Depending on the vendor on may or may not have to manually request the database to gather statistics about the hardware performance and the OS performance. These are also needed to perform an accurate cost estimation.

Number two is more difficult, because the user must know what is the correct choice before he/she can give the database a hint. The database will in most cases be able to tell you which choice it is making. This tool is often referred to as an 'explain plan'. In some cases it is obvious for the user what might be the problem.

For instance, if we have a table where the values of attribute a are distributed as follows: a=x is 80% of the tuples, a=y is 10% and a=z is 10%. If the query requires all tuples with a=y, it is beneficial to do an index lookup, since the query touches on only a small portion of the data. While if the query requires the tuples with a=x, a full table scan is appropriate since it touches a large portion of the tuple. The DBMS might know, from statistics, that there are three distinct values for attribute a. Therefore, it might assume that each of the values has approximately 33% of the attributes. This would lead the DBMS to do a full table scan for queries requesting a=y. A hint embedded in the query letting the database know about the distribution of values might make it choose an index scan instead. As the optimisers are improved many of these problems are handled by the database. To keep to our previous example Oracle today evaluates the value of y before deciding on a plan. In that way it can detect that it only needs to retrieve 10% of the tuples.

The use of SQL clauses is also largely dependent on the user. With SQL there is often more than one way to return the same set of tuples. Not all of the ways perform equally well. In many cases you can gain much performance improvement from rewriting the query to the form that performs better. The method for this is vendor and product specific. We will look more at this in the optimisation chapter later. Suffice to say that there is a large difference in most cases between EXIST and IN clause. It is trivial to show how both of these functions can be used to retrieve the same result.

The query optimiser has limited time to perform the optimisation tasks. Otherwise, the gain of finding the optimal plan is spent finding it. In most cases we are not running a query only once, so we can allow a separate process to spend a significant amount time to gather data and compute execution plans for a query. In Oracle, this information is stored in SQL Profiles. The profiles are updated when the query is executed to keep it up-to-date.

3.3 Indexing

Modern DBMSs employ indexes to reduce the number of disk accesses needed to retrieve a certain set of tuples. A large table that is stored on disk can be thought of as a flat file. If you want to find all the tuples that have attribute a=x you would have to read through the entire file to find the tuples

that satisfied your condition. If one knows which attributes that will be use to select tuples, it is possible to make an index on that attribute. The index

would store the attribute and the address of the entire tuple. Creating the index can be a time consuming operation, but when you have the index it can significantly reduce the time needed to find a tuple. Creating indexes meant to be used by a person is significantly different from creating one to be used by a machine. A person would probably prefer an alphabetic ordered list, one that he could easily find the page he was looking for by flipping through the pages. While for a computer it would be preferable to have an index that reduced the number of disk reads that is necessary. This is mostly done with B-trees.

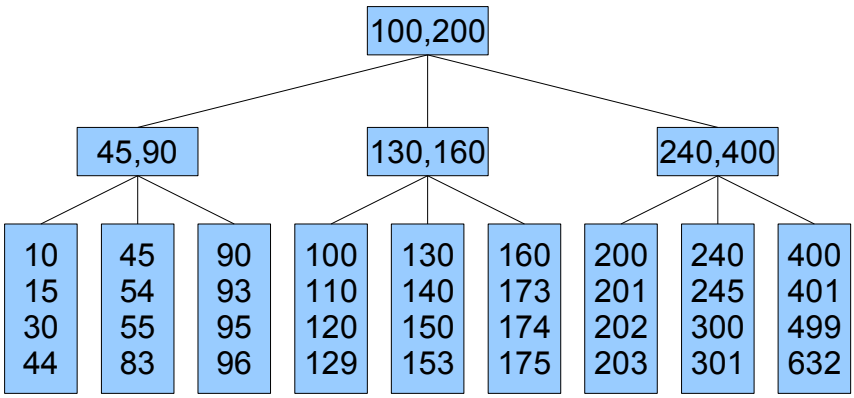

Figure 7: B-tree

Figure 7 shows a very small tree of this type. In this example each of the nodes has three children. The values in each of the nodes says where to find a certain group of values. All values smaller than 100 can be found under the left child of the top node (also called root node). All children between 100 and 200 are in the middle and all larger than 200 in the right one. In the following example we want to find the tuples with value 110. Starting in the root node (100,200) we see that it should be in the middle child (130,160) because 110 >= 100 and 110 < 200. For the next step we inspect the middle child (130,160). We can then see that it is in the left leaf because 110 < 130. Finally, we would look through the left leaf and find the exact address.

In a practice the tree is much larger. The nodes would also have more children. Generally, a node has X children such that X * (attribute size + child nodes address size) + overhead = block size. This allows the system to read one node with one disk read. In our example we would need three (root, child, leaf) reads. If X = 128 we could find one attribute address with three disk reads among more than two million tuples. Reading only the index sequentially would require about 16.000 blocks to be read.

Indexes have some drawbacks. First of all storing indexes requires extra disk space. Some studies [1]have shown that it requires about 15% of the tables disk space. With disks being priced as they are today this is not a significant problem. Furthermore creating indexes on an existing table is an expensive operation, but this only has to be performed once which reduces the impact it has. A more significant problem is the added overhead required

to update the index when one execute INSERT, UPDATE or DELETE statements.

If all the columns needed in the query are included in the index it is enough for the DBMS to look in the index, it does not need to check the table. This idea has been extended upon in Oracle. One can create index organised tables, where the entire table is stored as an index to take advantage of this.

3.4 Clustering

Clustering generally means that the data is stored together in sequential disk block, and in as few disk blocks as possible [15]. Clustering data together on the disk can result in much lower read/write time of the data. Furthermore it minimises the number of disk blocks the DBMS need to read. A disk block will in many cases contain several tuples. If the tuples are randomly distributed on the disk we might only need one of all the tuples in the disk block read. However, if the data is clustered, the tuples needed will be in sequence, and only the first and the last block in a large sequence will contain tuples that aren't need. Looking at the data sets in Figure 8 and9 we see that if we want to retrieve the values from 5 to 9 we need to read two disk block from the clustered data set or four disk blocks from the data set with arbitrary placed values.

Figure 8: Clustered data

Figure 9: Arbitrary placed data

To understand the advantage of clustering one must have some understanding of how a disk works. When we need to read from or write to a disk, the disk head has to move to the correct location. It has to move out or in to the right track on the disk. When the right location is found it can read all the data on this track without moving again. Moving the disk head takes considerable time compared to reading the data on a track when the head is above it. The time it takes to move the head to the correct place depends how far the head needs to move. Because of this we want to minimise the distance the disk head has to move between each read.

With this in mind, it would be convenient if the data we wanted always was on one track, or at least on adjacent tracks to minimise the time spent on moving the disk head. That is what clustering attempts to achieve. The system itself is not able to tell which data we are likely to need together. Therefore, this is something the user has to specify. In a normal file system one usually needs the entire file together and they are therefore stored together on the disk[5]. In a database we could and often do store the entire table together on the disk. However, it is common that only parts of a table

5 This is true if the disk is not fragmented.

is accessed at a time. So, further improvements can be implemented by also sorting the clustered data. This is done when we cluster by an index. Then the data is not only stored together, but also kept in the order of the index. As long as the data is clustered by the index we are using, it increase preformance. Having the data clustered by indexes not only minimises the number of disk blocks we need to fetch, but also returns the data sorted.

From the above, it might seem that clustering by index is the answer to all problems with disk reads. Unfortunately, this is not the whole truth. Clustering a table takes time, as all the data has to be rearranged to the prescribed order. Every time a new tuple is added or changed, it is possible that the DBMS needs to move large amounts of data to make room for the new or changed tuple at the correct place. Figure 10 shows the initial state of our cluster. There are three disk blocks of data. Then we insert a new value (3) to the table. As we can see, all the tuples after 2 needs to be moved one place to the right to make space for the new value. Figure 11 shows the end result. We could reduce this problem by leaving some open space in between the tuples. This would require more disk space and eventually, with enough inserts, we would still have to move the data. The final problem is that a given table can only be clustered by one index, so we have to chose which one to use.

Figure 10: Cluster example: Before insert

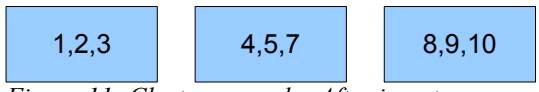

Figure 11: Cluster example: After insert

Some DBMSs store the tuples by default on disk together in the order they were inserted into the database. This is the case for PostgreSQL and Oracle 10g. Therefore, we can achieve much the same as with clustering by inserting the tuples to the table in an ordered fashion. This is what pgInTraBase does, because this solution is much faster then using the clustering function of the database. This only works because there is no further inserts in the table after it is initially created.

3.5 Caching

Reading from disk is one of the most time consuming operations a database does. Therefore much effort is put in to reducing the number of disk reads that is needed. Caching can be an efficient way of doing and can be done at many levels. The CPU has one or two levels of caches, the operation system has a cache, and the database has a cache.

Cache keeps data that it believes to be useful in the future as long as possible in memory. There are many different algorithms for deciding which data should be kept in memory and which data should be discarded. Generally when a new piece of data is read from disk and there is no more free space in memory for the new data we have to overwrite some other

data. There are several ways of deciding which data to overwrite. First in first out (FIFO), overwrites the oldest data in the cache. While least recently used (LRU) overwrites the data that has stayed the longest time unused in the cache.

When one reads a large amount of data like in a full table scan the cache can be fully replaced with the new data because there is such a large amount of new data being read. In many cases this is a undesirable behaviour and some algorithms trying to prevent this.

To successfully tune the caches, one must evaluate how the OS caching works in relation to the databases own caching. It is largely dependent on the application which approach will be most beneficial.

3.6 Striping and Redundancy

Striping and redundancy are technologies that can give increased performance for read and write operations to disk by utilizing more than one disk as well as prevent data loss. In theory twice as much data can be read or written to two disks compared to one disk in a given time. This is true if the disk is the bottleneck for the read operation and not the CPU or memory. Striping is handled by the operation system or the hardware so that the application only sees one large disk while it in reality might be *n* disks.

As with all optimisation techniques there is a price associated with this technology. If all the data stored on one of the *n* disks in the striping set fails then the data for all the *n* disks are useless. Because every n^{th} disk block was on the disk that is lost. So there is *n* times as large risk of losing all your data when using a striping set of *n* disk compared to using only 1 disk.

RAID and Striping is often used in conjunction. We will not discuss the details of all the different RAID versions here. RAID 5 can be used to reduce the risks associated with striping. To understand RAID 5 it is easier if one understand RAID 4, so we will begin there. In this example, we will use n=4 disks. What RAID 4 will allow for one disk to fail without losing any data. This is done by calculating a parity bit for each bit on the other disks. Among the four disks we want to have an even number of 1's for each position. The parity bit is set in such a way to make this reality. As in the example bellow:

```
Disk 1:   10101010
Disk 2:   11111111
Disk 3:   00011111
Parity:   01001010
```

If one of the disks is lost, then we can look at the remaining three disks and the lost bit will be 1 if there is an uneven number of 1's in the remaining bits or 0 if there is a even one. The lost disk can be reproduced. This prevents disaster (loss of data) from happening if one disk fails, but all the data is still lost if two disks fails. There are other more advanced algorithms that allow you to have more than one parity disk and by that be able to reconstruct data even if more than one disk is lost.

It should be apparent that each write to any of the three “data” disks also requires the parity disk to be updated. This results in the parity disk being a bottleneck and the benefit of striping is lost for write, because the set of disks is limited by the parity disks ability to keep up. For read the benefit of striping will still be apparent as one does not need to read the parity bit's in a normal read. They only need to be read when one is trying to recover lost data or when writing new data.

RAID 5 is the solution to the write problem. In RAID 5 there is not any one parity disk the parity bit's have been distributed between the four disks. This gives the same protection against lost disks and we regain some of the benefits of striping. Only some of the benefit is regained for writing since we will still have to write on two disks to update a single byte. For reading the full benefits of striping is achieved as with RAID4.

3.7 Write-Ahead Logging

Write-Ahead Logging is an important part of the recovery strategy in databases. Ensuring that all committed data can be recovered in case of a crash is time consuming. Therefore, it must be taken into account when one wishes to optimise performance of the database. Write Ahead Logging a.k.a redo logging, is used to ensure the atomicity of operations so that no committed data is lost during a crash. The DBMS will in many cases not want to write a disk block changed by a query to disk immediately. It will instead be written when it is evicted from the cache. This is done to reduce the number of write operations. Write Ahead Logging is a mechanism designed to prevent loss of data in the period between the user is informed that the transaction is committed and the data is actually written to disk. It does this by making a record of which changes were done. This record is kept in a file on disk. Only after this log has been written to disk can the user be told that the transaction is committed, because that is when one can guarantee that the change is permanent. The log must be written at least before every commit.

When there is no more free memory the DBMS can use for caching it has to evict one or more data blocks from memory. If this data block has been changed since it was read from disk the DBMS has to write the new version to disk before evicting it from memory.

The amount of transactions that can be committed, and have their data blocks only in memory, depends on how many different data blocks the system is using at the time and available memory. It is possible that many transactions are in this state. If the DBMS crashes the system would need to reconstruct the data blocks in memory from the information in the Write-Ahead Log. It would have to redo all those transactions. This is a time consuming process. The more transactions that have to be redone, the more time is consumed by the recovery process. In many applications the MTTR (mean time to recovery) is very important, because it is required to minimise the downtime in case of a crash. The only way to avoid having a long recovery time is to write the data blocks often to disk. The DBMS does this by performing a checkpoint. When performing a checkpoint the DBMS writes every data block that was changed prior to a certain time to disk. That

way only transactions that committed after the latest checkpoint have to be recovered in case of a crash. Many databases allow the user to configure how often the DBMS should perform checkpoints. In our case, this can be relaxed as much as possible to avoid writing to disk more often then necessary. This is because we can afford to have an extended recovery time.

4 pgInTraBase

In this chapter we discuss InTraBase and the prototype pgInTraBase as it was before this thesis was begun. We explore both the general concepts and ideas behind InTraBase as well as the features used to implement pgInTraBase.

InTraBase is a publicly available tool for measuring traffic and doing root cause analysis. Some articles [5], [6], [7], [8] explore the feasibility and benefits of using a Database Management System (DBMS) to perform the analysis. All data pertaining to the measurements are stored in the database. The functions used to arrive at the results are also stored in the database. These functions can be implemented using procedural languages. pgInTraBase is a prototype of InTraBase that was designed for PostgreSQL. Its implementation is explained in further details in Section 4.3.

PostgreSQL was originally chosen for the prototype because:

- It is object relational.
- It can be extended through PL/pgSQL and external functions and libraries.
- It has a large and active user community from which one can get help.
- It is free and publicly available.

4.1 Conceptual Overview of InTraBase

InTraBase is a concept and a method for performing network monitoring. There is one prototype of InTraBase called pgInTraBase, which will be discussed in the next section of this chapter. Table 2 shows the goals of InTraBase.

I. conserve the semantics of data during the analysis process;
II. enable the user to manage his own set of analysis tools and methods;
III. enable the user to share his tools and methods with colleagues;
IV. allows the user to quickly retrieve pieces of information from analysis data and simultaneously develop tools for more advanced processing;
V. includes a portable graphical user interface for facilitating the exploratory analysis;

Table 2: Goals of InTraBase

InTraBase advocates the use of a DBMS as the analysis environment, because of the many advantages to this approach. Packet headers are well-structured data, which allows for them to be easily modelled in a DBMS. In InTraBase, packet data are stored in the database and remains unchanged. Intermediate and final results are stored in separate tables. This design choice conserves the packet data and allows for a trace to be used in more

than one analysis. Furthermore, a DBMS is designed to handle a large amount of data. InTraBase is aimed towards trace files of moderate sizes (< 50GB) [8]. It can only be an advantage that DBMS are able to handle much larger amounts of data.

The developer of InTraBase wants to perform complex traffic analysis tasks that cannot be performed with a single pass over the input data. The development team identified three major challenges in this type of analysis: management of data, optimisation of the analysis process cycle, and scalability.

The process of analysing a trace require more complex logic than can be expressed in SQL alone. Therefore, it is essential that the DBMS selected to implement InTraBase has bindings to programming languages (e.g. PL/Perl). Functions written in these languages can easily be distributed and managed by the users themselves.

4.2 Design Overview of pgInTraBase

InTraBase successfully uses a DBMS to store and analyse packet data. It is open source software. When considering which DBMS to use for the prototype, it was therefore important for the development team that the DBMS is a public domain system.

The analysis process is more complex than what can be expressed in SQL alone. Therefore, an extensible DBMS was required. This functionality was achieved by taking advantage of the bindings PostgreSQL has with various programming languages. Functions are written in these programming languages and integrated with the SQL statements to express the complex algorithms.

Object-Oriented DBMS were considered, but object-relational DBMS were found to be better suited. The two main reasons were:

- The performance of Object-Oriented DBMS is not good enough.
- Packet headers are relational in nature and therefore well-suited for a relational database system.

Step 1: Copy packets into the packets table in the database.
Step 2: Build an index for the packets table based on the connection identifier.
Step 3: Create connection level statistics from the packets table into the connections table.
Step 4: Insert unique 4-tuple to cnxid mapping data from packets table into cid2tuple table.
Step 5: Separate bulk transfer periods from application limited periods in TCP connections and store them into bulk_transfer and app_period table, respectively.
(Step 6: Analyse bulk transfer periods for other types of limitations and store results in rwnd_test, retr_test and bnbw_test tables.)

Table 3: Steps in the analysis [5]

Before the analysis can be started the packet data has to be uploaded into the database system. This is done by Upload, a short java program using odbc. This program does steps 1-4 as defined by Table 3. It is currently able to read tcpdump and Dagpos format. When the Upload process is completed, the user has to start the function test_app() to run the analysis, which conforms to Step 5. Due to time and resource constraints Step 6 was not fully ported and tested. Therefore, its tables are also omitted from the Figure 12, showing the layout of the tables.

Figure 12 shows the layout of the tables used by pgInTraBase. Upload creates a new copy of the "*packet*" table when it uploads a new trace. The “paket” table is named according to the parameter given to Upload. The name of the new packet table and the tid[6] are inserted into the traces table.

Each packet header from the trace is inserted into the packet table for this trace. Cid2tuple matches source IP, source port, destination IP and destination port to a connection id. cid2tuple has a one-to-one relationship with the cnxs table, which contains the statistics for each TCP connection and a one-to-many relationship to the packet table.

6 Unique integer, that identifies the trace.

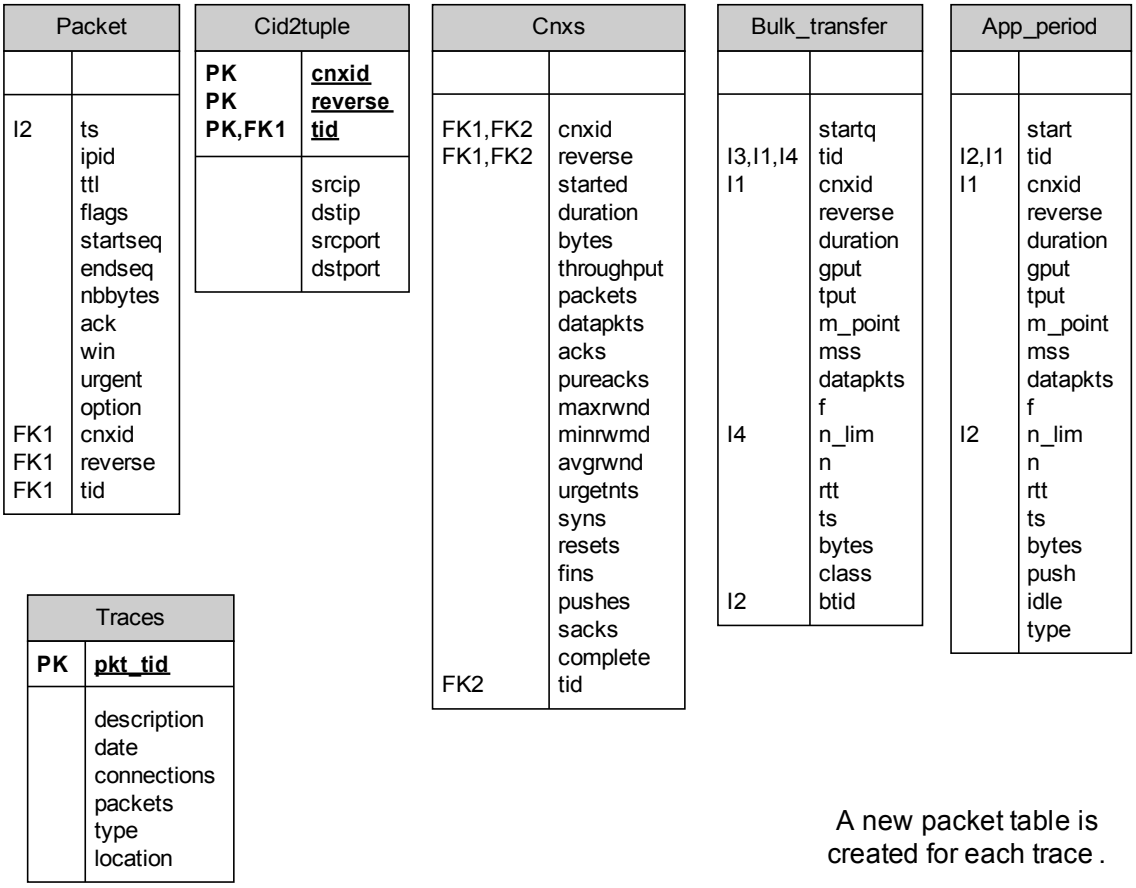

Figure 12: Layout of core tables in pgInTraBase after the five processing steps. Underlined attributes form a key that is unique for each row [8].

During the analysis, pgInTraBase populates the app_period and bulk_transfer tables. Each of the tables contains periods in which the traffic was limited by application (app_period) and in which it was not (bulk_transfer). The most important thing to notice is that there can be many instances of the Packet table, one for each trace, while other tables contains data from all of the traces.

4.3 Implementation

In the following sections, we show the main features of PostgreSQL and the tuning efforts that are used in pgInTraBase. Later in Chapter 5, we will do the same for Oracle, before comparing in Chapter 6 the features of PostgreSQL to Oracle in order to assess the possible benefits of porting the prototype to Oracle 10g.

When doing traffic analysis of this type, it is reasonable to assume that the usage of the database will follow certain patterns. First, it is unlikely that many users are simultaneously accessing the system. Furthermore, it follows that any given query is likely to touch a large amount of data. In the PostgreSQL implementation this has led to certain tuning efforts, described below. More details about most of what is discussed in this section can be found in [6].

4.3.1 SQL COPY

The packet table is populated by using a modified version of tcpdump. Output from this program is piped directly to SQL COPY that populates the table without the normal overhead of individual inserts for each packet. The data is written directly to the data structure of the database without transactional protection.

The above procedure is possible because tcpdump produces well-structured data that only needs minor alterations before it can be inserted. The major modification to tcpdump's output is to ensure that all packets have the same number of attributes. This is achieved by padding the information where needed, so that all tuples contain all the fields defined by the packet table. This is done because a table has a fixed number of columns and expect to have a value for each of the columns for every tuple, even if the value is "null".

4.3.2 PL/pgSQL, PL/R

Through a procedural language PL/pgSQL new functions can be added to PostgreSQL. These functions can be used with SQL. PL/pgSQL is a programming language inside PostgreSQL. It is used to implement the algorithms that was too complex to be implemented using SQL alone. It has a tight integration with SQL, and SQL like syntax.

Through PL/pgSQL, external library functions in C++ can be called. In addition to PL/pgSQL and the external C++ library, the prototype uses PL/R [6]. PL/R is a loadable procedural language that enables to write PostgreSQL functions and triggers in the R programming language [16]. PL/R has mainly been used in pgInTraBase to make statistical calculations and draw graphs in xplot format [5]. It is also used in one of the central analysis functions called r_pprate to manipulate matrices.

4.3.3 Write Ahead Logging

Usually, the ability to recover from a crash efficiently and correctly is crucial. With pgInTraBase this is less the case. We can recreate any data in the database from the trace files. Most of the time, we are only concerned with the results of a completed analysis and that the analysis process take a minimum of time. Since the analysis time is measured in hours, it is of less importance that the mean time to recovery is short. Therefore, we can allow data blocks to stay in memory for a longer time before they have to be stored on disk, which is achieved by setting the commit delay to the maximum value. In this way the system is allowed the flexibility to write the log to disk as few times as possible, thereby saving resources.

4.3.4 C-query

pgInTraBase focuses on analysing TCP connections. During the analysis it was found that the following query was central to performance of the analysis proces:

```
SELECT * FROM trace1_packets
```

```
WHERE connection_id=x
ORDER BY timestamp;
```

The above query was named C-Query, (c for connection and/or common). This query returns in chronological order all the attributes for all packets in connection x.

A typical analysis task will run as follows: [6]

1. The task is started by getting all the data for the connection, that is C-Query is executed.
2. The result is computed through various operations implemented as an algorithm.
3. The result is stored.

4.3.5 Caching and Memory

In the prototype, caching has been manipulated with respect to the size of the available memory and the algorithm used. The optimal way of using the cache in this instance would be if we could execute C-Query once and have the result cached while doing all the analysis on that connection. In that way C-Query (our most used query) would only have to read data from disk once per connection and not once per combination of analysis task and connection.

PostgreSQL uses an Adaptive Replacement Cache (ARC) algorithm. Before evicting a page, the ARC algorithm takes into account how often and how recently the page has been accessed. This is very beneficial in most cases, as it avoids flushing the entire cache if one user does a large table scan. But in the case of InTraBase we actually want the cache to be flushed and replaced as soon as a new C-Query is executed.

To achieve this, InTraBase minimizes the use of PostgreSQL's own cache and attempts to force the DBMS to use the Linux file system caching as much as possible [6]. Linux can only allocate 3GB of memory to one process on a 32-bit system, even if you have more available on the server. Furthermore, PostgreSQL cannot take advantage of more than 2GB of memory in certain critical operations, such as sorting. In InTraBase PostgreSQL was limited to using 1.5 GB as maximum memory allowed for sorting. This was done in order to avoid running out of memory when two sorting operations were executing concurrently. It was set to this value because the of the 3 GB limit imposed by the OS.

4.3.6 Indexing

Indexes in PostgreSQL induce a 15% overhead in disk space consumption [8]. This is an acceptable cost for the benefits indexes give. Disk space is relatively cheap these days, and when working with traces smaller than 50 GB, the cost of 15% extra disk space is insignificant. The InTraBase team found that the total overhead of storing the data in a database was 50% and stated that this would be acceptable.

In InTraBase the largest table is the one containing the packet data. This

table is not changed after the initial population. Therefore, indexes only have to be created once, and never updated. Since it is preformed only once for each trace, it is beneficial to create the index because the speed afterwards is greatly increased [6]. In InTraBase this feature is used to create the index after the packets table has been populated, since creating the index before populating the table is less efficient.

4.3.7 Clustering

In Section 3.4 we discuss clustering. In PostgreSQL 7.4 clustering is normally achieved by running the following statement:

```
CLUSTER indexname on tablename;
```

When new entries are added to the table, PostgreSQL places the new data blocks at a "random" place. Thus, if new entries are added one will have to run the clustering again on the table. If a table is defined as a cluster, the DBMS will handle this automatically.

In pgInTraBase this does not cause any problems as all the data in the packets table are inserted before the index is created and the clustering is performed. Furthermore, there are never any updates or inserts in the packets table after the initial phase.

During the clustering process, PostgreSQL requires free space equal to the size of the table that is being clustered. InTraBase is at the moment used with trace files that are about 10GB large. Which is an insignificant amount of disk space for current servers.

The CLUSTER operation in PostgreSQL is very slow [8]. Another way of achieving the benefit of clustering is what we call implicit clustering. The data is first loaded into a temporary table and then inserted into the final table using:

```
INSERT INTO <table> AS SELECT ... ORDER BY <cluster key>
```

Since PostgreSQL is storing the entries on disk in the order they are inserted, this will lead to the data being clustered. This is much faster than using the clustering feature in PostgreSQL.

Still one can say that this is no guarantee that it will be 100% clustered and sequential since PostgreSQL might not get one continuous area on disk. The OS might give PostgreSQL a fragmented file. Hence, even if PostgreSQL has the tuples clustered one after another in the file, they can still be fragmented on the disk.

4.3.8 RAID and Striping

Reading and writing to disk is among the most time consuming operation a computer can perform. pgInTraBase is an I/O-bound application, meaning that the other parts of the application have to wait for I/O. After enabling RAID and Striping for the disks pgInTraBase becomes CPU-bound.

4.3.9 Data Storage

Modern DBMS allows table to contain a large amount of tuples. In

PostgreSQL for instance there is no limit on the number of tuples in a table, but a table can be maximum 32 TB. They can spread the table over many disks to avoid limitations on disk space. When considering performance the size makes a difference: An excessively large table will impose performance problems both for full table scan and index lookups. Therefore, the developers of InTraBase chose to create a new packet table for each trace file. The other tables are shared between the different traces as they do not consist of a significant amount of tuples per trace compared to the packet table.

5 ORACLE 10G

In this chapter, we will first discuss why we chose to port pgInTraBase to Oracle 10g. Then we discuss the different concepts and features we use when porting pgInTraBase to Oracle 10g. The structure of this chapter is similar to Section 4.3 to allow easy comparison.

We chose to port pgInTraBase to Oracle 10g because we wanted to compare the performance of an open source DBMS vs a commercial DBMS. Oracle was a natural choice since we already had the necessary licence. In addition Oracle have the features that we need to port pgInTraBase. One may write functions in other third generation programming language and integrate them with SQL. Oracle is object relational. It is widely used and has a large user community in addition to professional support. The syntax of PL/SQL and SQL in Oracle is similar to that in PostgreSQL, which allows us to leverage much of the work put into pgInTraBase.

5.1 SQL*Loader

SQL*Loader is used to efficiently copy data into the database. It is much more efficient than a series of insert statements. SQL*Loader is similar to PostgreSQL copy, but with more features. It consists of two parts: the data file or pipe and the control file. The control file tells Oracle how to read the data file and which modification should be done to the data. SQL*Loader can do complex selection operations and transformations while loading data. Data that for some reason can not be inserted into the database is stored in a .bad file where it may be inspected later or passed through SQL*Loader with a different control file. This is a big advantage compared to copy in PostgreSQL, because PostgreSQL copy terminates if one bad row is found, which can lead to significant loss of time in certain cases.

The performance can be improved by doing the insertion while the database is in NOARCHIVELOG mode (see Section 5.10). If it is impossible to run the entire database in NOARCHIVELOG mode, it is possible to set this mode for only the SQL*Loader with the unrecoverable option. Which means that the operations preformed by SQL*Loader will not generate any log, and can not be rolled back.

SQL*Loader can also take advantage of multiple CPUs by running in parallel mode. Direct path loading is a very fast way of loading data into a table. The data is passed through the API directly into Oracle data blocks in memory and then flushed to disk in large jobs. It rebuilds the index after all the data has been loaded. It neither fires triggers nor works on clustered tables. It can be used to load the package information into a temporary table and from that table populate a clustered table and at the same time fire any triggers. Direct Path loading has the following limitation: if this option to SQL*Loader is used, SQL*Loader can not execute any functions on the data.

5.2 Programming Languages

PL/SQL is a Turing-complete procedural language that strongly resembles

the Ada programming language [17]. It can be compiled to native code for more efficient execution. Compiled library can be imported from C++ for instance. After the database imports the shared library, the function of the library can be used through PL/SQL as any other function. That gives us a wide variety of languages to write extensions (triggers, functions and procedures) for the database.

5.3 Write Ahead Log

In Oracle you can use LOG_CHECKPOINT_TIMEOUT to determine the number of seconds that any modified data block can exist in memory before it has to be written to disk. FAST_START_MTTR_TARGET is a parameter that one can set for Oracle. The DBMS will attempt to keep the database in such a state that the mean time to perform the recovery process always will be less than this parameter. One can disable that function by setting it to 0, to allow for long Mean Time To Recovery (MTTR). In most production system one wish to have MTTR as low as possible. If the server goes down this is the time it takes to perform recovery after the server is back on-line. Before a transaction can be committed in Oracle it is written to the redo log (on disk). It might take some time before the block is written to the actual file of the database (on disk). The purpose of the redo log is to keep record of the transaction. If the server crashes after the redo log has been written to file, Oracle can reconstruct the blocks that where kept in memory and lost during the crash from the blocks on disk and the redo log.[7] This process may take a lot of time depending on the amount of transactions that have to be reconstructed. MTTR is a value that states how much time you will allow for this process. By reducing this value the recovery time is reduced, but Oracle would then have to write blocks to the disk more often so that it has less “unstored” data in case of a crash. In our case, we are not concerned with the MTTR and can allow it to become arbitrary long. The benefit of this is that you avoid the performance penalty of writing blocks to disk often.

The default isolation level in Oracle is rather relaxed (READ COMMITTED) which means that not to much resources should be wasted trying to make the transactions serializable. Ensuring that one statement has a consistent view of the data, as with READ COMMITTED, is beneficial as it might cause unforeseen problems if the data set is changed during the execution of the statement. It is rare that many users work simultaneously on a given set of data at any time. Hence, we do not need to tighten the isolation level beyond the default level.

5.4 Indexing

Oracle provides two different types of indexes: bitmaps and b-trees. Bitmaps are used when there is a limited and unchanging set of keys, for instance gender. B-trees can be used with a large set of keys, or to enforce unique keys. In our case B-trees is what we need. Indexes can be created at any time for any table(s) that one has access to.

7 Undolog is also used to remove any trace of transactions that did not commit.

5.5 Clustering

When creating a cluster or an index, Oracle writes the key value only once followed by the rest of the values. A simple example illustrates this: The example table have two attributes a and b. We will have 4 tuples on format (a,b): (1,2)(1,3),(1,4)(2,3). If this was clustered on the disk we could store this as (1,2)(3)(4),(2,3). Since (3) and (4) followed (1,2) we would know that the value of the clustered key for each of them is 1. This technique can save significant amounts of data if each clustering key has many tuples with the same value. Saving disk space will in the end lead to faster execution as less data have to be read. Since the clustering key is only stored once in the cluster and once in the index disk space is saved. Clustering, as we discussed earlier, can be very useful. If more than one table is clustered on the same key, the time for executing join operations are reduced since the DBMS will find data for both the tables at the same place on the disk[18]. The reason for this is that when you find the disk block containing the information needed from one of the table the matching information from the other table would be in the same or adjacent disk blocks.This is very useful when doing joins. It is not that beneficial if one of the two tables is read.

Oracle provides several cluster types:

- Index clusters, use a traditional Btree index. This is a good choice when the tables in the cluster mostly are queried together. If the table has few insert, update and delete and the data is evenly distributed it is well suited for clustering [18].
- Hash clusters which use a hash function. This is a good choice when the data is evenly distributed on the index attribute. Few inserts, updates or deletes operations performed on it. The table has predictable number of values in the indexed column. Equality operator is used to retrieve rows [18].
- Sorted Hash clusters are similar to Hash cluster with the difference that values on the same hash key are kept sorted. This does not give much overhead, but can be beneficial. When creating a sorted hash cluster you have a additional attribute that defines the sorting order for the tuples with identical key values. If a query has the same sorting attribute as the sorted hash cluster it could retrieve the tuples in an already sorted order and avoid having to sort the result leading to a performance benefit.

The process of creating a clustered table is:

1. Create the cluster.
2. Create the index.
3. Create the table.
4. Populate the table.

It is worth noting that you cannot cluster an existing table in Oracle. Instead, clustering has to be configured before the table is defined. One must instead make a new clustered table and populate it from the old one. Then the old

table can be dropped and the new one renamed. As noted in section 3.4, clustering is an expensive operation in the form of performance reduction on insert and update statements because tuples may have to be moved to make room for the new or changed tuple. Section 3.4 explains this in more details.

5.6 Caching and Memory

Oracle uses the least-recently-used (LRU) algorithm to determine which database page is to be flushed from memory [18]. It also has an algorithm to protect the cache from a full table scan by not letting it flush the cache [10]. It would be beneficial if we could turn this last feature off, as we would like to flush the cache between each iteration of the C-Query. The ideal situation would be if we could tune the cache in such a manner that no matter which connection_id was requested for C-Query, the result would always fit in the cache.

Oracle allows one to decide to which part of the database one want to allocate memory. This will allow you to tune the database to your needs. Oracle does this through various pools of memory. Each pool is used for a specific purpose. There are several separate pools for caching in Oracle.

- System Global Area (SGA)
  - Shared Pool
  - Database Buffer Cache
  - Redo Log Buffer
  - Java Pool
  - Streams Pool
  - Large Pool
- Program Global Area

When a SQL statement is parsed and ready to execute, it is stored in the Shared Pool so that the next time the same query is given to the database, it does not need to reparse and optimise it before execution.

When data is used by the database, it is stored in the Database Buffer Cache. It will remain in that buffer in case it is needed. The LRU algorithm is used to decide which execution plan gets overwritten when the system needs space for a new query.

Large Pool and Redo Log Buffer is part of the recovery system for Oracle. They cache information needed by various recovery managers. In addition Large Pool holds some Shared Server components.

Blocks containing data from tables or indexes are held in the database buffer cache. We would want this to be large enough to hold at least all the data blocks returned by C-Query. Parsed SQL statements, stored procedures and data dictionary are held in the shared pool [10]. Ideally we would like this to be large enough to hold parsing data for every query in our system and all procedures to minimize the overhead.

To help us tune, Enterprise Manager keeps information about caching and cache hit ratio. Enterprise Manager is a tool that comes with Oracle 10g. It can be used for many of the most important administration tasks and provide performance information in an easily accessible way. (See section 7.4 for more information on Enterprise Manager)

5.7 RAID and Striping vs ASM

Oracles Automated Storage Management (ASM) is a volume manager included with Oracle 10g. It can provide striping and mirroring data to increase performance and availability [10]. Oracle is able to take into account the database objects type when storing them [19], which RAID is unable to. Another possible benefit is that the underlying OS and its file system are circumvented by giving Oracle control over the disk. Oracle stores disk information in a DB, which needs 60-100MB of memory [18], but this should provide fast lookup to find where on the disk a certain data block is placed. In ASM Oracle does the work of the volume manager and the file system. Because of this, it is not possible to share a ASM disk with other applications, but more Oracle instances and databases can share an ASM disk.

5.8 Dedicated Server vs Shared Server

Dedicated server means that one process serves only one connection in the database, while Shared server means that one server process handles several connections.

InTraBase should be on a dedicated server, as you only get benefits from shared server if there are many short connections [19]. The only reason for using shared server would be if we changed the system in such a way that it did several things in parallel.

Another issue with shared servers is that you cannot do index or table rebuilds over a shared server connection [19].

5.9 Real Application Cluster (RAC) and Grid

Running more than one database server in parallel seems attractive in most cases, especially if there is a large amount of read operations compared to write operations. One thing that might prove to be a problem for this system is that the analysis is done within the DB and not from an external application. This is an issue because load balancing is done at connection level or by a middle ware application.

Load balancing can be done by the clients, if they have a set of servers to select from and select one at random, this will lead to load balancing. This is supported by the Oracle client. Another more sophisticated way is to have an application between the client and the database that keeps track of which of the servers has the least load and direct the client to that server. This kind of connection hand off is also supported.

It would require major redesigning of the analysis process to be able to benefit from more than one server. Thus, we do not consider it in this thesis.

5.10 No Archive Logging

Oracle 10g fills redo log in a cyclic fashion. While in archivelog mode these redo log files will be archived to a different destination so that they will not be overwritten. This archiving is performed on-line, that is, while the DB is running and operational. When this mode is first started, a full backup is needed. After that the DB can be recovered to any point in time after that backup with the archive log files. Being able to recover the DB to any point in time is not one of the features that InTraBase requires. Since it will impose some performance penalty and prohibit use of direct path loading, we will run the system in NOARCHIVELOG mode, and instead base backup on regular off-line backup.

6 COMPARISON AND PORTING

In this chapter we first compare the features used in the implementation of pgInTraBase with the features available in Oracle. Then we explain in details how we ported pgInTraBase to Oracle. In Section 6.2 the major part of the results from our porting is presented. The information in this section can be used to port PostgreSQL applications to Oracle.

6.1 Comparison of Features

In this section we will summarize the differences between PostgreSQL 7.4.2 and Oracle 10g, focusing on the elements used in the implementation of InTraBase.

6.1.1 PL/SQL, PL/pgSQL, PL/R

According to [17] PL/pgSQL tries to emulate PL/SQL to an extent. It implies that PL/SQL is more comprehensive than PL/pgSQL, this is further supported by [2].

[2] list it as the main difference between PL/SQL and PL/pgSQL. Keep in mind the list was written as a guide to port from PL/SQL to PL/pgSQL not the other way around.

- No default parameters in PostgreSQL.
- You can overload functions in PostgreSQL. This is often used to work around the lack of default parameters.
- Assignments, loops and conditionals are similar.
- No need for cursors in PostgreSQL, just put the query in the FOR statement
- In PostgreSQL you *need* to escape single quotes.[8]

From these sources, we understand that it should be possible to port PL/pgSQL to PL/SQL without too much trouble as we are porting it to the language with more functionality.

Oracle 10g do not have anything equivalent to PL/R. A work around to this problem exists though. Oracle can import shared library from C++ and through it call C++ code. Rserve is a program that sets up a server to communicate with other programming languages over a socket. It allows for other programming language to execute R functions. Rserve can be used in conjunction with C++. In PostgreSQL you could write R function directly in the database.

6.1.2 Write-Ahead Logging

Both PostgreSQL and Oracle support features to relax attempts to make the transaction serializable and to postpone the write ahead log to a point in time where it can be done more efficiently.

8 Different quotation may be used in latest versions.

6.1.3 Caching

Both PostgreSQL and Oracle lack a perfect caching algorithm for our use, both use variations of the same algorithm. Oracle seems to have more options to adjust caching for individual parts. Having different pools in the cache allows the user to better decide which part of the caching algorithm they want to use. The buffer cache could be reduced to a minimum and thereby make the file systems algorithm[9] the only caching algorithm for disk blocks, but still keep the internal DBMS algorithms for other parts like query optimizing.

6.1.4 Indexing and Clustering

To support the C-Query, Sorted Hash Cluster with Cluster Key = connection_id and ORDER BY = timestamp would be beneficial, as it would minimize the seeking on disk and would save an 'ORDER BY' for all queries that would sort by timestamps (which is done by every C-Query).

In PostgresSQL the cluster was created by populating a table and then executing a query which select all tuple from the original table sorted and then insert them in to the new table. In Oracle the clustering has to be defined before the table is populated. Oracle saves data sequentially so implicit clustering possible. As in PostgreSQL, it requires that we do not perform delete operations followed by more inserts.

Since InTraBase has data that is uploaded to the database before the querying starts it would be best to create the cluster after the table is populated. If it is defined before populating the table, it will cause disk overhead in loading the date into the database. The reason for this is that the tuples have to be moved around to make room for new entries in the cluster.

The same arguments are true for indexes. It would be beneficial to create them after populating the tables.

6.1.5 RAID and Striping

Only Oracle supports internally managed striping, in the form of ASM. Documentation claims that the internally managed striping is superior to hardware striping in the form of RAID. We chose to use hardware managed RAID and not ASM.

6.1.6 Conclusion

Porting IntraBase from its current form in PL/pgSQL, PL/R and C++ to Oracle 10g and PL/SQL should be possible. We expect that integrating or porting the functions written in PL/R and C++ will be the largest challange.

9 If we want to use the file systems caching, we actually have to use the file system. This means that ASM is out of the question. So we have to compare them up against each other.

6.2 Porting Techniques

In this section we show and explain the techniques used to port pgInTraBase to Oracle 10g. The main differences are described in detail while the minor are only mentioned.

6.2.1 PL/SQL

The initial assumption that porting pgIntraBase from it's current form in PL/pgSQL to Oracle 10g's PL/SQL would be a relatively easy task is correct. Only at a very few occasions does it need to be rewritten.

The two most significant changes are that in PL/pgSQL the table name is sent as a parameter in a text string to most of the functions, and queries is executed by assembling a query as text string and then passed to the EXECUTE statement. These things have to be changed because of the way Oracle 10g work and to optimize performance, as explained below.

If you assemble a query in Oracle 10g as a text string the parser will be unable to recognize that two statements have the same structure, unless they are 100% identical. If one instead use bind variables the parser will recognize that they have the same structure and will not be parsed again as long as the execution plan is still in the cache

Sending the table name as a parameter in a string would force us to use the same procedure to execute the query. Assembling the query in a text string and passing it to the EXECUTE statement. To avoid this a private synonym was created. Then all function could refer to the table in question as tbl instead of through a text parameter. A private synonym can be different from user to user so more than one analysis can be run at the same time on different tables as long as they are run from different users, but a user can only analyse one trace at the time.

Query 1

```
SELECT *
FROM tbl
WHERE cnxid = 11
```

Query 2

```
SELECT *
FROM tbl
WHERE cnxid = 12
```

Query 3

```
SELECT *
FROM tbl
WHERE cnxid = id_val
```

Query 1 and 2 are two completely different queries, at least as far as the Oracle 10g parser sees it. If you first issue query 1, the parser will generate a execution three. If you moments later issue query 2 the parser will again do

the same process of generating a execution plan. If you on the other hand used query 3 with the variable id_val and just changed the value of the variable the parser will understand that the only different between executing the query with id_val = 11 and id_val = 12 is the value of the bind variable, and that therefore the execution plan is the same. This will prevent the parser from unnecessary parsing statements that it already have parsed, and improve performance. Oracle 10g keep parsed queries in the buffer. It removes execution plans by a least recently used policy. There are other ways of using bind variables, but in PL/SQL a variable inside a statement will always be taken as a bind variable. Using a variable in a statement we have to be careful not to name it the same as the column, therefore we adapted a policy of prefixing all input variables with in_ and a postfix for all variables that could cause conflict as _val.

Furthermore, in PL/pgSQL you had to have the statement in a for loop to get it executed even if there was only one tuple to be returned. Some examples of what the statements in PL/pgSQL and the equivalent in PL/SQL are shown below. With a short explanation for why it was done this way.

PL/SQL:

```
FOR row IN EXECUTE ''
SELECT count(*)
FROM bulk_transfer
     join
     rwnd_test
     using(btid)
WHERE tid=''||tid||'' and n_lim=''||n_lim
LOOP
  IF row.count > 0 THEN
    RETURN ''Trace already processed.'';
  END IF;
END LOOP;
```

PL/SQL:

```
SELECT count(*)
INTO count_val
FROM bulk_transfer
     join
     rwnd_test
     using(btid)
WHERE tid = tid_val and n_lim = n_lim_val;

IF count_val > 0 THEN
  RETURN 'Trace already processed.';
END IF;
```

In PL/SQL you can write queries directly in to the code if you choose to. The INTO clausal tells PL/SQL where to put the returned values. In the

ported code you will very seldom find statements using the INTO functionality. Mostly cursors are used in stead. Then we can separate the query from the code. The ideal was to have all SQL queries in the package header, but this was unfortunately not practical in all cases. The query above would look like this using a cursor,

Package header:

```
CURSOR count_bulk_transfer_cur(in_tid IN NUMBER,
in_n_lim IN NUMBER) IS
SELECT count(*)
FROM bulk_transfer
     join
     rwnd_test
     using(btid)
WHERE tid = in_tid and n_lim = in_n_lim;
```

Package body:

```
OPEN  count_bulk_transfer_cur(tid_val, n_lim_val);
FETCH  count_bulk_transfer_cur INTO count_val;
CLOSE  count_bulk_transfer_cur;

IF count_val > 0 THEN
      RETURN 'Trace already processed.';
END IF;
```

Notice that a cursor is like a function that can take parameters as input and use it in the statement.

The benefit of using a text string to hold the query is that if you have a long list of if, else if, else if, else then you can build the query in the if statements, and execute it and run the code that needs to be run on the result only one place. This is more difficult with the cursor approach as each query has its own name. This can be solved by using reference cursors. First a variable is declared as a sys_refcursor in the declaration part of the function.

```
rc sys_refcursor
```

Then a reference cursor can be opened in the body of the function.

```
IF a = 0 THEN
  OPEN rc FOR
    SELECT cnxid
    FROM tbl
    WHERE ack IS NULL;
ELSE
  OPEN rc FOR
    SELECT cnxid
    FROM tbl
    WHERE ack IS NOT NULL;
```

```
END IF;

FETCH rc INTO id_val;
CLOSE rc;
```

This solution was selected when there where many if selection and complicated changes between each of the statements. In this case, and in many others, it can simply be rewritten in SQL to the statement below and used in a normal cursor.

```
SELECT cnxid
FROM tbl
WHERE (a=0 and ack IS NULL) or (a<>0 and ack IS NOT
NULL);
```

Another interesting thing to note is that in PL/pqSQL you have to write calls to other PL/pgSQL functions as SQL queries.

PL/pgSQL

```
FOR row IN EXECUTE ''SELECT mss(''||row2.cid||'',''||
abs(row2.r-1)||'',''''''||tbl||''''''')''
LOOP
  mss := row.mss;
END LOOP;
```

PL/SQL

```
mss := mss(row2.cid,abs(row2.r-1));
```

Note that the tbl part is omitted from the PL/SQL version this is because the table is “hard coded” as the synonym tbl and does not have to be included as a parameter.

On a very few occasions the use of tbl as a synonym caused challenges. In PL/pgSQL the text string containing the table name was used as a “key” to store information about the trace in other tables. A query of this type (omitting everything but the query)

```
''SELECT pkt_tid FROM traces where packets=''''''||
tbl||'''''' ''
```

In this case tbl is used as a text string to be compared to the column packets. To achieve this in Oracle we have to look up what the synonym is pointing to, then use this value for the comparison.

```
SELECT t.pkt_tid
FROM user_synonyms s, traces t
WHERE s.synonym_name=v'TBL' and UPPER(s.table_name) like
UPPER(t.packets);
```

In PL/pgSQL the input variable does not need to have names they are used as $1, $2.... while in PL/SQL they always have name. In most cases they had been given names through aliases in PL/pgSQL.

PL/pgSQL

```
CREATE OR REPLACE FUNCTION
combine_periods(INTEGER,NUMERIC) RETURNS text AS '
DECLARE
  tid ALIAS FOR $1;
  n_lim ALIAS FOR $2;
```

PL/SQL

```
CREATE OR REPLACE FUNCTION combine_periods(in_tid IN
NUMBER,in_n_lim IN NUMBER) RETURN VARCHAR2
AS
```

Also note that the final s in returns and declare has been removed, this is another of the minor differences between PL/pgSQL and PL/SQL.

6.2.2 SQL

What we had expected was the amount of porting nneded for the SQL embedded in the PL/SQL. Though PostgreSQL and Oracle have a very similar syntax there are small differences and to our surprise there are some functions that PostgreSQL have and Oracle do not. In this section we will look at the various porting issues that did arise and explain how they where solved.

6.2.2.1 Variable Types

Variable types where converted as follows:

Bigint, signed 8byte integer	NUMBER(19,0)
Integer, signed 4byte integer	NUMBER(10,0)
double precision, 8 byte float?	BINARY_DOUBLE
Bit(n), fixed-length bit string	NUMBER(1,0)
interval, time span	INTERVAL DAY TO SECOND
numeric	NUMBER
smallint	NUMBER(5,0)
date	DATE

In addition to the above mentioned data types pgInTraBase used inet type to hold IP addresses. Oracle does not have any equivalent data type, but this could be developed with PL/SQL. In the core functions that we are looking at in this thesis, inet is not used. In many cases it would be sufficient to replace inet with VARCHAR2(15) as the only purpose is to store the IP information. Though in other functions variables of inet data type is used in calculations, and for that a more complex substitute would have to be developed.

6.2.2.2 Tables and Reserved Names

Oracle 10g has a long list of names that has been reserved for use internally,

names that you are not allowed to use any other place. At a few occasions such names had been used for column names in PostgreSQL version of IntraBase. Start and Date both been used as column names. The simplest solution for this problem was just to rename the columns to a similarly appropriate name. In our case we added a post fix 'q' to each of them.

6.2.2.3 Distinct On

The query below show how distinct on is used.

```
SELECT DISTINCT ON(a,b) a,b,c
FROM distinct_on_test
ORDER BY a asc,b desc,c ASC ;
```

In a table with the following valued:

```
 a | b | c
---+---+---
 1 | 2 | 2
 1 | 2 | 3
 1 | 1 | 3
 1 | 1 | 4
 2 | 3 | 3
 2 | 3 | 4
 2 | 2 | 2
(7 rows)
```

Our query would return

```
 a | b | c
---+---+---
 1 | 2 | 2
 1 | 1 | 3
 2 | 3 | 3
 2 | 2 | 2
(4 rows)
```

What DISTINCT ON does in this case is to limit the out put to prevent that two rows have the same value in a and b column. It returns the tuple in the order it normally would, and if any tuple arrives that is not unique on a and b they are discarded. Without ORDER BY you get a "arbitrary" tuple. In principle you could with DISTINCT ON (a,b) get 1,2,2 one time and 1,2,3 the next. But in practice you will get the same one each time as there is a order, even if you did not decide on one. Using ORDER BY you can control it so that you always get the largest or smallest c.

The example result could be achieved with this query in Oracle 10g:

```
SELECT a,b,min(c)
FROM distinct_on_test
```

```
GROUP BY a,b
ORDER BY a,b,min(c);
```

There is a problem though, if there is more than one value outside the GROUP BY clausule, as in:

```
SELECT DISTINCT ON(a) a,b,c
FROM distinct_on_test
ORDER BY a asc,b ASC,c ASC ;
```

In this case we would want to have the following result:

Will return:

```
  a | b | c
----+---+---
  1 | 1 | 3
  2 | 2 | 2
(2 rows)
```

The equivalent given our solution in Oracle would be:

```
SELECT a, min(b), min(c)
FROM distinct_on_test
GROUP BY a
ORDER BY a, min(b), min(c);
```

```
         A     MIN(B)     MIN(C)
---------- ---------- ----------
         1          1          2
         2          2          2
```

This is not the same result. The reason is that the Oracle version is not returning any existing tuple from the table, it is creating a new one with the two smallest b and c's it could find. What we wanted was to have the tuple with the smalest b for each a. If there were more than one tuple with the smallest b value, we wanted the one tuple with the smallest c value too.

Some research revealed that Oracle could to this, but in a more verbose statement:

```
SELECT a, MIN(b) KEEP (DENSE_RANK FIRST ORDER BY a  ASC)
b, MIN(c) KEEP (DENSE_RANK FIRST ORDER BY a ASC, b ASC)
c
FROM distinct_on_test
GROUP BY a
ORDER BY a, b, c;
```

This returns the exact same thing as DISTINCT ON.

6.2.2.4 Limit and Offset

Limit and offset are postfixes that can be with any query in PostgreSQL

either alone or both together.

SELECT a,b FROM some_table limit 1 offset 2.

limit x, limits the number of tuples that the query will return to x, the first x tuple of the tuple set is returned. Offset x, discard the first x tuples and returns the rest.

There is no equivalent for this in Oracle, we will instead have to mimic the functionality. Oracle numbers all rows in any returned set. This column is called rownum. Initially it seems that this query is equivalent to the one above.

```
SELECT a,b
FROM some_table
WHERE rownum > 1 and rownum < 3;
```

But this does not work if the query includes ORDER BY. Because rownumber is calculated before the ORDER BY is applied. Therefore, it needs to be in a sub query for it to work like:

```
SELECT a,b
FROM (SELECT a,b, rownum rn
          FROM some_table
          WHERE rownum > 1 and rownum < 3)
WHERE rn > 1 and rn < 3;
```

The column of the subquery has to be renamed so that it will not be confused with the rownum column of the outer query.

6.2.2.5 RETURNS SET OF RECORDS

In PostgresSQL a set of records can be returned. Record being a valid variable type. For each record you want to return you do a

```
return next <variable of type record>
```

The fields of the records is unnamed, and have to be named when they are used in the receiving function. below is an example of a statement that uses the return values from a function that returns set of records.

```
SELECT *
FROM avg_rwnd_shift_(''||cid||'',''||abs(direction-
1)||'',''''''||window||''''''',''''''||tbl||''''''',''||
f||'',interval''''''||rtt||''''''',''||ts||'') AS t(c2
integer, rwnd integer)) as t2 ON(c1=c2)'';
```

In Oracle one has to define a type that can contain each record, and another type that is a table of the first type. Using the same as in the example above.

```
CREATE TYPE avg_rwnd_shift_t as OBJECT (c2 NUMBER(10,0)
,rwnd    NUMBER(10,0));
CREATE TYPE avg_rwnd_shift_set_t IS TABLE of
```

```
avg_rwnd_shift_t;
```

When this is done the function can be set to return avg_rwnd_shift_t. At the end of the declaration you also have to add PIPELINED. For each tuple you want to return you have the following statemetnt

```
PIPE ROW(<variable of type avg_rwnd_shift_t);
```

On the receiving end you can do:

```
SELECT *
FROM TABLE(avg_rwnd_shift(in_cid, abs(in_direction-1),
in_window, in_f, in_interval, in_rtt, in_ts);
```

You can use the function as any other table in the database. Since the types are declared you do not need to specify names in the receiving end as in PostgreSQL. In oracle the two function can run in parallel allowing faster computation.

6.2.2.6 Substring

During a select one does not always want the entire column value. This is especially true for he flags and options. Both of these columns can contain more than one value at the time. In many cases it is neccesary to get hold of only one of them or just a part of the option. In pgInTraBase this is done with for example the following statement, this extract the digits that follows the string 'wscale_' and cast them to number:

```
to_number(substring(options from
''''%wscale_#"[[:digit:]]+#"%'''' for
''''#'''')','''9999''''
```

Oracle has a powerful ANSI regexp engine that allow the same functionality. In this case I had to use nested regexp to ensure that it is only the digit that follows wscale_ that is returned. Unnested I could not get only the digit and guarantee that they originally had followed wscale_ :

```
REGEXP_SUBSTR(REGEXP_SUBSTR(options,'wscale_[[:digit:]]+
'), '[[:digit:]]+');
```

6.2.2.7 Case In Where Statements

In PostgreSQL you can easily add a case statement to a where clausal and it will use different logic to select the return set depending on the case statement.

```
SELECT *
FROM mf_j_10
WHERE CASE WHEN direction=1 THEN reverse NOTNULL ELSE
reverse ISNULL END
```

Oracle does not allow for case statements in the where clausal in this way. But it can be simply rewritten to a and or statement like this:

```
SELECT *
FROM mf_j_10
WHERE ((direction=1 and REVERSE IS NOT NULL) or
(direction=1 and REVERSE IS NULL));
```

6.2.2.8 (t1, t2) Overlaps (t3,t4)

PostgreSQL have this feature where t1 and t3 is timestamps and t2 and t4 may be timestamps or interval. t1 and t2 together for a interval either from t1 to t2 (in the case t2 is a timestamp) or from t1 to t1+t2 (in the case t2 is a interval). The same is true for t3 and t4. This feature returns true if the two intervals overlaps. This could have been implemented as

```
FUNCTION overlap(in_start1 IN TIMESTAMP, in_end1 IN
INTERVAL DAY TO SECOND, in_start2 IN TIMESTAMP, in_end2
IN TIMESTAMP) RETURN NUMBER
IS

BEGIN
  IF (in_start1 BETWEEN in_start2 AND in_end2 OR in_end1
BETWEEN in_start2 AND in_end2 OR in_start2 BETWEEN
in_start1 AND in_end1) THEN
    RETURN 0;
  ELSE
    RETURN 1;
  END IF;
END;
```

It is generally better to do what can be done in SQL in SQL instead of doing it in a seperate function when considering performance due to the context switching. Therefor this was changed to a simple where statement:

```
t1 < t4 and t2 > t3
```

This does the same, it only include the tuples where the intervals overlap. The only difference is that one had to manually fix t2 and t4 if they where interval and do the addition in the query. If t2 and t4 is interval it would be as follow:

```
t1 < t3+t4 and t1+t2 > t3
```

6.2.2.9 Array

In PL/pgSQL using array is strait forward, as in many other programming languages. Arrays in pgInTraBase is used only to implement a queue, many programming language have this as part of their standard function set. Neither PL/pgSQL or PL/SQL does.

In PL/pgSQL A variable is declared to be an array and then it can be used as one. No limit on the length of the array. Note that in PL/pgSQL array index start on 1 not 0 as in most other languages. The array allow you to manipulate set of values by specifying the start position and end position. This code shows PL/pgSQL declaration and the manipulation of the a set. In

this code position 2 to 10 from the pkt_array is made in to a “new” array and the variable new_val is added to the end, that is the position 10 of the new array. The the new array is assigned to the same variable as the old one. In effect putting one more value at the end of the array and removing the first one. This implements a queue.

```
DECLARE
pkt_arr INTEGER[];
BEGIN
  LOOP
    pkt_arr := array_append(pkt_arr[2:10],new_val);
  END LOOP;
END;
```

In Oracles PL/SQL this is done very differently. First of all array has do be declared as one of three types of array, this section will show VARRAY. Before using a position in the array it has to be initialized. This can be done implicitly while declaring the variable like we do it in this code or it can be done in the body. In this case we are using 10 spaces in the array so there is ten null's in the declaration. If there is many more it is cumbersome to do it this way and it can instead be done in a for loop. PL/SQL can not as easily take a set of values from an array and make a new array, therefore we have to create this ring buffer.

```
DECLARE
pkt_arr NUMBER_VARRAY :=
(null,null,null,null,null,null,null,null,null,null);
array_index NUMBER := 1;
BEGIN
  LOOP
    pkt_arr(array_index) := new_val;
    array_index := array_index +1;
    IF (array_index = 11) THEN
      array_index := 1;
    END IF;
  END LOOP;
END;
```

In addition you would have to handle the special case when there is less then 10 values in the array.

Prior to the above code the following declaration has to be made. This can either be made as a global type, like shown here or it can be declared inside the function.

```
CREATE TYPE number_varray IS VARRAY(10) OF NUMBER(10,0);
```

6.2.2.10 Minor Syntax Differences.

In PostgreSQL ISNULL and NOTNULL is used for comparison. In Oracle

they are respectively IS NULL and IS NOT NULL. In the statement below it will return all tuples that has reverse is not null when direction is 1 and the ones with null when direction is different from 1.

6.2.2.11 Full Outer Join

In PostgreSQL it looks like this:

```
SELECT a
FROM (query1) as t1
FULL OUTER JOIN (query2) as t2
USING(value);
```

In Oracle the as can be dropped for the table names. USING(t1.value = t2.value) becomes ON q1.value = q2.value; So the final result looks like this:

```
SELECT a
FROM (query1)  q1
FULL OUTER JOIN (query2) q2
ON q1.value = q2.value;
```

6.2.2.12 Number to Interval

If you want a interval of 1 second in PL/pgSQL you can write

```
INTERVAL DAY TO SECOND''0.0000001 sec'';
```

The two examples below show two different ways of doing it in Oracle. One is a function call with the overhead that comes with it and the other is written on the format for intervals and is therefore considered an interval.

```
numtodsinterval(0.0000001,'second');
'+000000000 00:00:01.000000000'
```

6.2.3 PL/R

A few functions had been written in PL/R and adapt ion of R to PostgreSQL. The reason for this is that R have certain statistical function that PL/pgSQL do not. We attempted to avoid using R at all, by rewriting them in PL/SQL or SQL.

r_quantile_mean returns the mean, or 50th quantile in a percentile division for a set of numbers. Oracle has a SQL statement that can do the same. The function below can be used to return a number that represent the quantile number the value in number_column should be in.

```
NTILE(100) OVER (ORDER BY number_column)
```

Returning the 50th quantile can be done by selecting only the tuples that has this value = 50 and then selecting the min() of those. In IntraBase the r_quantile_mean function is used like this:

```
''SELECT r_quantile_mean(''''SELECT * from iat_ack(''||
```

```
row.cid||'',''||1-row.r||'',''''''''''||
tbl||''''''''''',''||offset||'',''||limit||'')'''',0.99)
as b'';
```

This can be replaced by:

```
SELECT min(quantile)
FROM TABLE(iat_ack(row.cid, 1-row.r, offset, limit),
0.99) quantile
WHERE NTILE(100) OVER (ORDER BY quantile) = 50;
```

6.2.4 C++

Oracle can integrate C++ in a similar fashion to PostgreSQL. We have not included any thecnical example, because we did not adapt the calc_rtt. Calc_rtt is the only C++ function in pgInTraBase. Calc_rtt does a very complicated estimation of round trip time (rtt). It partly mimic the TCP protocol to give an as accurate estimate as possible. It would be a major task to port this function to PL/SQL. Since it is one of the more computation intensive part of the InTraBase it is best to keep it in C++ and adapt the link to Oracle 10g.

Calc_rtt is a pipelined function returning a set of tuples that is used as a table. This is possible to do in C++ for Oracel 10g, but it is time consuming task. Also calc_rtt is uncommented and the integration with PostgreSQL is not modularised, because of this we did not have the time to implement it in oraInTraBase.

7 PERFORMANCE OPTIMISATION IN ORACLE

In this chapter, we discuss the various tools and techniques available in Oracle for performance tuning. We show how these tools can be used to tune and configure the various optimisation concepts presented in Chapter 3. Before looking at the tools we discuss optimisation strategy.

7.1 Optimisation Strategy

Before the year 2000 most of the optimisation of Oracle followed the Method C [20].

"Method C: The Trial-and-Error Method That Dominates the Oracle Performance Tuning Culture Today:

1. *Hypothesize that some performance metric x has an unacceptable value.*
2. *Try things with the intent of improving x. Undo any changes that make performance noticeably worse.*
3. *If users do not perceive a satisfactory response time improvement, then go back to step 1.*
4. *If the performance improvement is satisfactory, then go to step 1 anyway because it may be possible to produce other performance improvement if you just keep searching." [20]*

In this thesis we will try to avoid Method C in favour of the more scientific approach called Method R.

"Method R: A Response Time – Based Performance Improvement Method That yields Maximum Economic Value to Your Business:

1. *Select the user actions for which the business needs improved performance.*
2. *Collect properly scoped diagnostic data that will allow you to identify the cause of response time consumption for each selected user action while it is performing sub-optimally.*
3. *Execute the candidate optimisation activity that will have the greatest net pay-off to the business. If even the best net pay-off activity produces insufficient net pay-off, then suspend your performance improvement activities until something changes.*
4. *Go to step 1." [20]*

The first step is already decided, we want to improve the performance of oraInTraBase and test_app.

The second step of Method R is far more complex. Through fixed views Oracle provides information about many different aspect, such as for instance runtime and wait events. These views are kept in memory and constantly updated with statistics and other current information concerning the database.

There are many hundred fixed views. Through them one can find information about nearly everything Oracle has done since start-up. For instance, the number of active sessions and what they are doing at the moment can be seen in v$session. The names of fixed views all start with v$, and are therefore, often called v$-views.

In the description of Method C performance, metric x almost always refers to data from one or more of the fixed views. The exact number of fixed view will depend on the release and version of Oracle.

In many cases the information from these fixed views can give a good indication of the problem, but they represent far from the perfect tool for analysing performance problems. Most of the fixed views can only give you aggregated information, either on session or by instance. Using aggregate statistics introduces more complexity to the analysis because of the hidden details [20].

Even with the weakness mentioned above, fixed views have been a popular tool for optimisation Oracle databases. Fixed views are often used with Method C [20]. They show the current status of the database. This is convenient in many cases, but, unfortunately, not for us. We need more detailed information over a certain time period. To achieve this, we would have to poll the fixed views as many as 50 or 100 times a second. Polling so often is impossible, because of the performance overhead. However, Oracle can make this detailed information available through trace files, which are discussed in the next section.

7.2 Explain Plan

Explain plan can be used to display the execution plan the DBMS is using for a given query. Below is the command to use explain plan and some of the output it would produce[10].

```
explain plan for select * from mf1 where cnxid = 10
SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());


PLAN_TABLE_OUTPUT
------------------------------------------------------------------------
Plan hash value: 3970876707
------------------------------------------------------------------------
|Id| Operation         |Name            |Rows|Bytes|Cost(%CPU| Time   |
------------------------------------------------------------------------
| 0| SELECT STATEMENT  |                | 4751 | 273K| 62(0)|00:00:01 |
| 1| TABLE ACCESS BY INDEX ROWID | MF1 | 4751 | 273K| 62(0)|00:00:01 |
|*2|  INDEX RANGE SCAN | MF1_CID_IDX    | 4821 |     | 13(0)|00:00:01 |
------------------------------------------------------------------------
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("CNXID"=10)
```

10 The output has been slightly reformatted to make it easier to read.

```
14 rows selected.
```

Explain plans are best read from the bottom up. In this case we can see from the last row that an index range scan would be performed using the index called MF1_CID_IDX. Then the table MF1 would be be accessed using the rowid found in the index.

The cost column can be very interesting. The units of the costs have no relevance outside the DBMS, they can only be used to compare the cost against other statements. The first line gives the cost of the entire statement, 62. The second line shows the cost of doing the table access, but since the index range scan was necessary in order to do the table access, the cost of that operation is included.

Information from explain plan can be found in both trace files and in Enterprise Manager. In this thesis we do not run explain plan from the command interface instead we access it through the Enterprise Manager.

7.3 Trace Files

In trace files, Oracle emits details about most of the Oracle kernel's actions along with summary information. The command to start the trace for a given session is:

```
alter session set max_dump_file_size=unlimited;
alter session set timed_statistics=true;
alter session set events '10046 trace name context
forever, level 12';
```

This method can be used if one has access to the code of the application you want to measure or the application can be started from sqlplus. If it is a third party product that one cannot modify, one has two options for starting the trace. We will not discuss these as they have no relevance for this thesis. Details about these two methods and their pitfalls can be found in [11].

Because of the detail level of the trace files it can be very time consuming and difficult to use them in practise. There are tools to ease this processing, Tkproof is one such tool. It produces a summary that is easy to read and understand. Tkproof provides a nice overview of the performance of the query, and what events Oracle has spent time on.

In this thesis, we chose not to use either trace files directly or with Tkproof. Instead, we rely on Oracle Enterprise Manager, described in the next section. For more details about how to read and interpret trace files see Appendix C.

7.4 Oracle Enterprise Manager

In this thesis, we make extensive use of Oracle Enterprise Manager (OEM) and the tuning features accessible from it. In this section, we explain how OEM can be used to improve performance through logical tuning, create the correct indexes and tuning SQL. The task of tuning queries can be divided into three parts:

1. Identifying time consuming queries that are candidates for tuning

2. Run automatic tuning advisors, implement their advice and evaluate the effect.

3. Attempt manual tuning by rewriting the queries.

Below we describe how we use OEM for Step 1 and Step 2, but first a general introduction to OEM.

OEM is a management software that is installed with the DBMS. OEM collects information about the database through an agent that runs on the server. This agent stores collected data in the database, where the user can access it through a GUI. The GUI is web-based and is by default accessible at http://domain:1158/em. OEM is Oracle's recommended way of administrating an Oracle 10g database system.

The performance page of OEM gives a graphical representation of the performance of the system. Figure 13 shows the performance tab during an Upload of a 1GB trace. From the graph one can see that Upload is a very CPU-intensive operation.

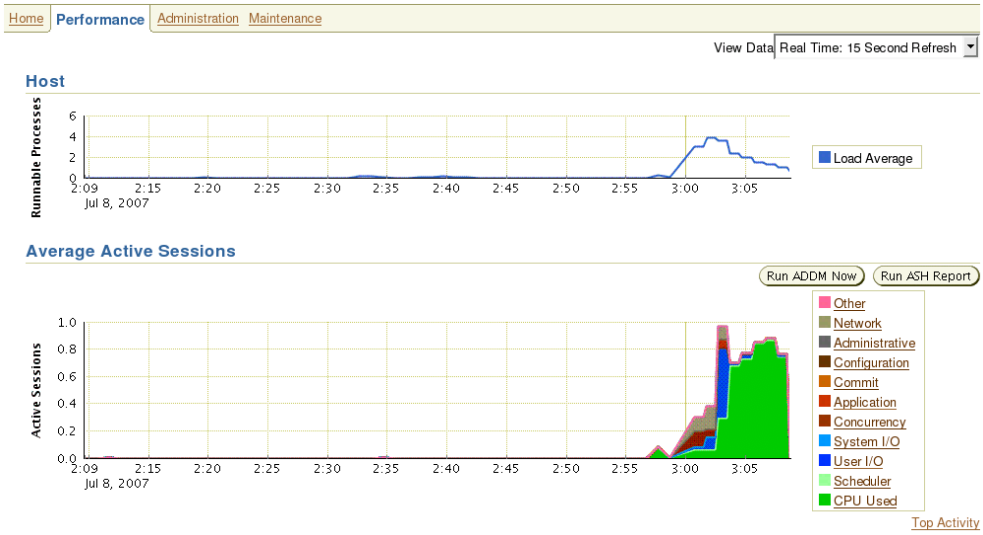

Figure 13: Oracle Enterprise Manager, Performance tab.

We can further go into the various groups to get more details. In this case, clicking on User I/O would get us another graph detailing how the I/O was divided among db_file_scattered_read, direct_path_read_temp and db_file_sequential_read. Almost all the graphs related to performance in OEM have the same drill down functionality as this one.

One of the most useful pages in OEM for performance tuning is the top activity page. From this page we can easily identify which query and sessions are the most resource consuming in a given interval. Figures 14 and 15 show the top activity page.

The list in Figure 15 shows both the most time-consuming SQL queries and sessions during the specified interval. Clicking on either SQL ID or Session ID will display the detail page for either SQL or session. This functionality is very important. It allows us to run the analysis process and in a few clicks identify on which queries we should concentrate our tuning efforts, and thus

completing Step 1.

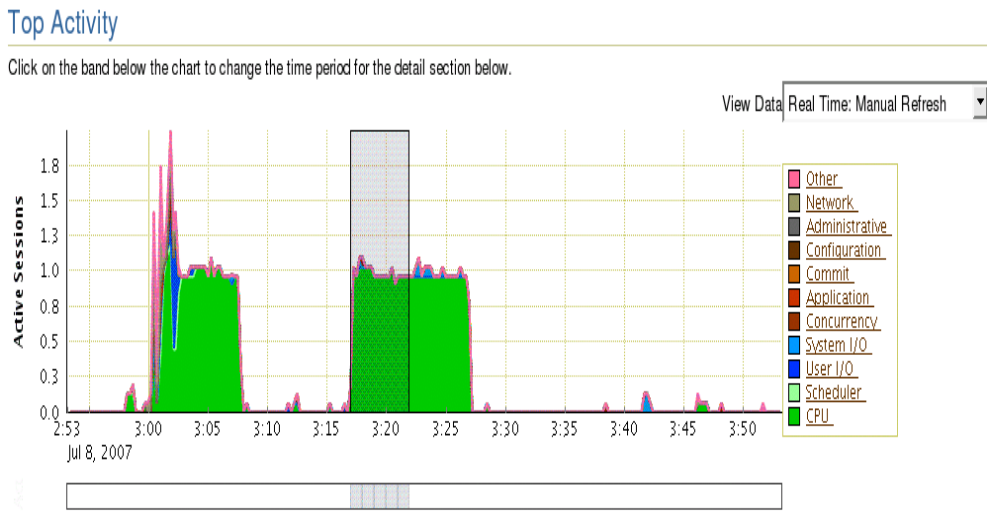

Figure 14: Oracle Enterprise Manager, Top Activity (Part 1)

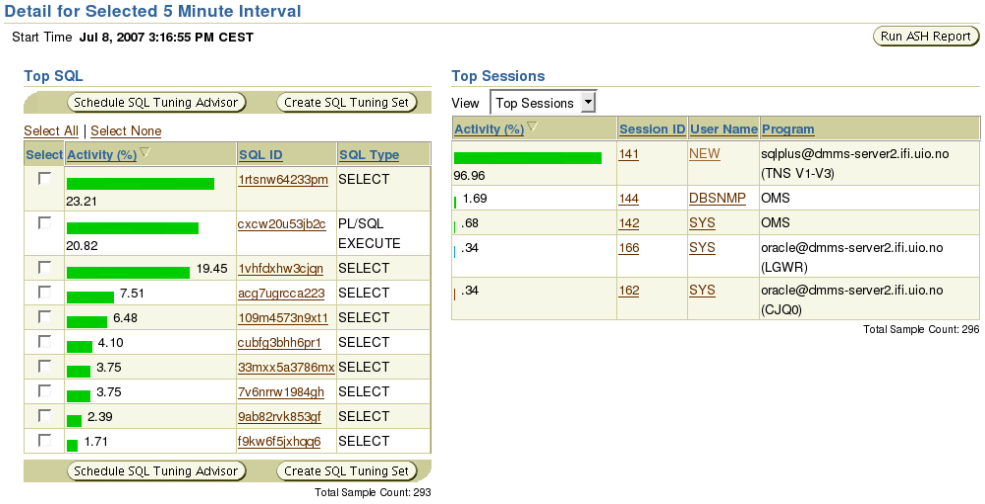

Figure 15: Oracle Enterprise Manager, Top Activity (Part 2)

The automatic tuning advisors can be accessed by several paths. We choose to go through the SQL details page. This page can be accessed by clicking the SQL ID in the top activity page. The tuning advisor can either be scheduled, or run immediately. The setup of this task is intuitive. If it was run immediately, it would display a result page after it had completed. The advisor might, suggest gathering new statistics, creating a specific index, or implementing a SQL Profile. It conveniently lets you implement the suggestion directly from the advisor. This would conclude Step 2.

A SQL Profile is a collection of extra statistics specific to one query. These statistics are used by the DBMS when the optimiser generates an execution plan. Having this detailed level of information allows the optimiser to generate better execution plans. The main concept behind the profiles is that it allows for a lengthy analysis to find the information needed. The advisor consumes much more time than one normally would allow the optimiser.

This is possible since the profile only has to be created once, and not at every execution of the query. SQL Profiles should not be confused with stored outlines, which are static execution plans. Outlines are not used in this thesis, and are therefore not discussed further.

By the use of SQL profile a query can be tuned without altering the application or query. This is invaluable when working with third party applications with no access to the source code. Or in our case to make fine adjustments to the execution plans. In the past this had to be done through hints, and rewriting of the queries. More details regarding SQL profiles can be found in [23].

7.5 Using the optimisation tools

In this section we give concrete examples of how we take advantage of the concepts described in Chapter 3.

Logical Optimisations

Logical optimisation is done by the DBMS without any user interaction. The user can however influence how the execution plans are created through hints and SQL profiles. Hints and SQL profiles are discussed below.

SQL Tuning

Tuning the SQL queries is a task that requires more of the user than many other tuning tasks. In Chapter 3 we identify three forms of SQL tuning.

1. Statistics
2. Hints
3. Using different SQL clauses

The SQL tuning advisor can help us with the two first forms. It will tell whether the statistics for the tables and indexes are out date or non-existing. The advisor will also help you create the statistics if needed.

The tuning advisor will not directly suggest hints, but it will instead suggest SQL profiles. SQL profiles and hints are both used for the purpose of changing the execution plan to a more suitable one. SQL profiles can in most cases replace hints. This is very convenient since SQL tuning advisor can create SQL profiles.

The final form of SQL tuning is the most complex one and requires most of the user. The best tool available here is explain plan. It shows the cost of different operations and queries. We can use this by rewriting a query and see how the cost changes compared to the original version.

Indexing

If we have a query that we want to tune we should start by identifying which, if any, indexes it is using. We do this by accessing the explain plan information as explained in Section 7.2. Next step is either to run the SQL Tuning Advisor and the SQL access advisor from OEM. The advisors can recommend what indexes we should create. Otherwise we have to make an educated guess.

No matter how we decide which index to create, we wish to evaluate it afterwards. We can either measure the time of the entire query, or we can run explain plan again and compare the costs. To continue the example from Section 7.2, if we made the mf1_cid_idx index unavailable and ran explain plan again we would see that the cost of the query increased by a factor of 30 to 1888. We would also see that the DBMS would use a different index. The conclusion of this experiment would be that dropping that particular index would increase the runtime of the example query by a factor of 30.

Clustering

There is no direct tool that helps us make decision about clustering, but we can use explain plan to see how the cost change after implementing a cluster. The main thing to keep in mind is that a clustering affects all queries for the involved tables. Therefore, it is important to test a representative set of queries when measuring the benefits and costs of clustering.

Caching

Using fixed views we can investigate the hit ratio in the various cache pools. Especially buffer cache hit ratio and library cache hit ratio are important. Buffer cache hit ratio tells us how many percent of the requested blocks were found in memory as opposed to on disk. Library cache hit ratio tells us how many percent of the SQL queries were found in the cache as opposed to how many the optimiser needed to generate. Generally they should both be above 95%.

If the library cache hit ratio is low it is likely that the application does not use bind variables in the queries. We can investigate this by looking in the Duplicate SQL tab in OEM. From this page we can see how many times a query has been parsed and generated the exact same execution plan as a different query. This usually only happens when bind variables have not been used. In this thesis we do not focus on this since we know that bind variables have been used in all queries.

Striping and Redundancy

Besides the cost of disks there is no disadvantage of striping and redundancy, only advantages. When evaluating whether the cost will be worth the benefit we can look at the performance page in OEM. There we find the dominating resource. If I/O dominates there is a good chance that striping might be beneficial.

8 Performance Analysis and Optimisation of pgInTraBase and oraInTraBase

In this chapter we will measure, improve and evaluate oraInTraBase and compare it to pgInTraBase. We use a program called Upload to populate the database with packet headers. We need to populate the database before we an start the analysis. Therefore, we find it natural to measure and evaluate the Upload procedure first. This will be done in Section 8.4 after having explained the measurement set-up and strategy in Section 8.1 and 8.2. In pgInTraBase c-query was responsible for a major part of the load on the system. We believed that the performance of c-query would be directly proportional to the performance of the system. Therefore, we devoted Section 8.5 to c-query. In Section 8.6 we measure and improve the analysis.

8.1 Measurement Set-Up and Parameters

Throughout this chapter, we use a script to measure the time spent on each operation/program. Unless otherwise noted, each test is performed 5 times. We have chosen to measure five times because we feel that it gives us a reasonable confidence in the results. With fewer measurements the confidence interval could become too large. For measurements we are using the following annotation: measured mean +/- 90%-confidence seconds. An example will illustrate this. Given the following five measured times in seconds: X_1=16,220959 X_2=21,612327 X_3=18,131913 X_4=17,806869 X_5=20,757124. This gives us a mean of =18,905838 seconds, a standard deviation s=2,22 seconds, a 90% confidence of 1,635270 seconds. The 90% confidence interval would be 17,270569 seconds to 20,541108 seconds. In this thesis, we will summarise this as: 18.905838 ± 1,635270 seconds. In graphs confidence interval is expressed as two graphs with the same label, one being the lower bound and one the higher bound of the confidence interval.

In the initial stage of measuring we discovered that there was a certain performance degradation when more and more traces were added to the same database. Some of the tables in the system are shared: traces, retransmissions, cid2tuple and cnxs. Because these tables become larger with each new trace we decided to remove all previous tuples in the database between each iteration of the measurements[11]. It is also very difficult to eliminate all of the caching mechanism. We can flush Oracle's internal cache:

```
alter system flush shared_pool;
alter system flush buffer_cache;
```

This is not that easy with the OS, the file system and hardware. Therefore, we chose to do all tests in a “warm” system. That is, we will run each test once to populate the cache in a realistic way before we do the experiments.

11 The PostgreSQL version performs better on the second iteration without removing the tuples from the first iteration.

This should ensure that each iteration of the experiments are started under the same circumstances.

The experiments are performed on an AMD Opteron(tm) Processor 254 2.8GHz processor with 8GB RAM, running Red Hat Enterprise Linux AS release 4 (Nahant Update 4). The server is 64-bit, but the OS and all applications are 32-bit. The PostgreSQL experiments are done on PostgreSQL 8.2.3, using the source code that was designed for pgInTraBase when running on PostgreSQL 7.4. We did this because installing version 7.4 on the new server proved difficult. Because of this the measurements are not directly comparable to the original papers about InTraBase. The Oracle experiments are done on Oracle Database 10g Enterprise Edition Release 10.2.0.1.0.

8.2 Measurement Strategy

From the research on pgInTraBase we learned that c-query is very important for the analysis process performance. Therefore, when we plan our research we devoted a significant amount of time to analysis of the c-query. As we will see, the study of c-query proved not to be of the same value for the Oracle version as it had been for the PostgreSQL version.

In this chapter we will use trace files that were prepared for the pgInTraBase research [5][6][7]. They where created by capturing approximately 10GB trace of TCP/IP packet headers. These trace files were then used to create different sized trace files. In other words the 10MB trace file is the first 10MB of the original 10GB trace file, for instance. In Sections 8.4 and 8.5, we will run all experiments on a mixed Internet traffic (MT) trace. After optimising we will validate our results with a trace file that has BitTorrent traffic (BT). This is shown in Figure 33, Section 8.6.2. The two types of trace files have different characteristics. Mixed Internet traffic usually consists of many small/short connections and a few long connections, while traffic generated with BitTorrent will have many more large/long connections. Different trace files will be used as we expect the number of connections and connection size to have an impact on performance. We expect the analysis to take longer time with the mixed traffic than with the BitTorrent traffic, because of the larger amount of connections and the fact that most analysis tasks are performed for each connection. Therefore, we will focus on mixed traffic in our study and only when we achieve a satisfactory result with mixed traffic will we compare it with BitTorrent traffic.

8.3 PgInTraBase performance analysis

pgInTraBase was implemented and evaluated with the following configuration [7]:

- Mandrakelinux release 10.1 (kernel version 2.6)
- PostgreSQL 7.4.2
- tcpdump 3.8, libpcap 0.8
- awka 0.7.5 (not in the current version of InTraBase)

- PL/R 0.6.0b-alpha
- Java 1.4.2_03
- Intel Xeon Biprocessor 2.2GHz with SCSI RAID disk and 6 GB RAM.

The pgInTraBase team measured BitTorrent (BT) and Mixed Internet Traffic (MT) different type of traffic. BitTorrent traffic consists mostly of long lasting connections while Mixed Internet traffic consists mostly of short connections. The difference in the two trace types are explained in further detail in Section 8.5.5. BT took 0.6 hours per GB while MT traffic was 0.9 hours per GB [5]. The MT trace file contains more packets per MB then the BT trace. The run time of the analysis is shown to be a factor of both the number of packets and the number of connection. Further details about the trace files shown in Figure 23 and 24. We will perform our one measurements for pgInTraBase to ensure that it is done in the same environment as oraInTraBase.

8.4 Upload

Upload perform analysis step 1-4 as presented in Table 3, Section 2.3.1. In this section we will optimise and evaluate it against the upload procedure in pgInTraBase.

When we ported Upload.java we did this on a one for one basis, i.e we did not modify the queries or steps in general. In this section, we investigate how we can improve it. We will now go through a series of performance optimisation steps and measure the benefit of each of them. Some of them have already been implemented in the PostgreSQL version, while others are Oracle specific, or cases where Oracle behaves differently than PostgreSQL.

We can't perform any analysis without loading the packet data into the database system, which is done through Upload in both pgIntraBase and oraInTraBase. Upload is a relatively small Java program, which performs the steps 1 to 4 of the analysis process as described earlier in Section 4.2.

Tests for upload are started with the following commands

pgInTraBase (PostgreSQL):

```
./plotsar.pl java Upload /disks/fc/trace/jussieu_10m
intrabase a b c mf16 t
```

oraInTraBase (Oracle):

```
./plotsar.pl java -classpath
.:/disks/fc/oracle/oracle/product/10.2.0/jdbc/lib/ojdbc1
4.jar Upload /disks/fc/trace/jussieu_10m orcl a b c mf1
t
```

Plotsar returns the time spent on the operation given to it as a parameter. In this case we are using 10MB trace, for the sake of efficiency. We will later

do it with larger trace to ensure that we get similar results with larger traces too. Uploading a 10MB trace in pgInTraBase takes 9,356874 ± 0,49696 seconds while it takes 18,905838 ± 1,635270 seconds in oraInTraBase. The Oracle version does not perform well at this stage, as expected because we ported Upload from PostgreSQL to Oracle in a one to one manner. The two systems have different ways of default behaviour. Therefore, it would have been surprising if the Oracle version had performed better before we had optimised it.

SQL*Loader used by the Oracle version to populate the temporary table need a pipe from which it receives data. This pipe is created using

```
mkfifo fifo1
```

To guarantee that this pipe exists, before the upload procedure is started, we delete anything with the name fifo1 and create a new pipe. These commands ensure that fifo1 is not just a regular file or directory, but the pipe we need:

```
rm -rf fifo1 && mkfifo fifo1
```

As a first step we removed this statement from the Upload process. Instead this will now be done as part of the install process. It only needs to be done once for each system/user. Removing this part from Upload does not change the result significantly; the average result decreased 0,784434 seconds. If we compare each measurement of the two tests we see that the difference between them have a 90%-confidence interval -2,436363 seconds to 0,867494 seconds. Since the confidence interval includes 0 we can, according to normal statistical practise, not say conclusive that it was beneficial to performance to remove the statement [22]. It is, however, beneficial to our system because it does less work, even if the performance gain is negligible.

Next we want to investigate which parts of the Upload process are the reason for the long running time. We do this by identifying the processing task (pt) in Upload that we want to measure. Before and after each of the processing tasks we insert a getTime statement. For pt_x, t_{x-1} is just before pt_x while t_x is just after. That way we can calculate $\Delta t = t_1 - t_0$ = time to process pt_1. We define pt_1 to pt_{10}, from the original Upload, later we will add a pt_{11} for the Oracle version. Each of the pt relate to one or more Analysis Steps (AS) as they are defined in [5]. Notice that Upload does not follow the AS in chronological order, neither does each pt correlate to exactly one AS. In the description of the pt we will explain which AS it belongs to.

AS1: Copy packets into the packets table in the database.
AS2: Build an index for the packets table based on the connection identifier.
AS3: Create connection level statistics from the packets table into the connections table.
AS4: Insert unique 4-tuple to cnxid mapping data from packets table into cid2tuple table.
AS5: Separate bulk transfer periods from application limited periods in TCP connections and store them into bulk_transfer and app_period table, respectively.
AS6: Analyse bulk transfer periods for other types of limitations and store results in rwnd_test, retr_test and bnbw_test tables.

Table 4: Analysis Steps (AS) as defined in [5]

pt_1: Copy packets into the packets table in the database.

By running a modified version of tcpdump a stream of packet headers is generated. This stream is sent through fifo1 to SQL*Loader. SQL*Loader copied the headers in to temp_table. This is part of AS1 in the analysis.

```
t0
pc.copy();
t1
```

pt_2: Indexing temp_table on cnxid

This is similar to AS2 in the analysis process but it is not on the final table, therefore we will consider this part of AS1

```
t1
CREATE INDEX temp_table_cid_idx ON temp_table (cnxid)
t2
```

pt_3: Mapping cid to tuple

To avoid storing source and destination ip addresses and ports numbers for each tuple a CID to tuple mapping is created. This is AS4.

```
t2
INSERT INTO cid2tuple SELECT DISTINCT cnxid, reverse,
srcip, dstip, srcport, dstport, tid FROM temp_table
t3
```

pt_4: Dropping columns that are no longer needed

To reduce the amount of data we drop columns that we no longer need. These columns where used in Step 4, but are no longer needed.

```
t3
ALTER TABLE temp_table DROP COLUMN srcip
ALTER TABLE temp_table DROP COLUMN dstip
```

```
ALTER TABLE temp_table DROP COLUMN srcport
ALTER TABLE temp_table DROP COLUMN dstport
t4
```

pt5: Create a new table and populate it with implicit clustering

The clustering operation in PostgreSQL is very slow, the same is true for Oracle. Thus, in order to avoid clustering the table is recreated and tuples are inserted into the table in the wanted order for clustering. This will lead to a structure similar to clustering as the DB store the data in the order it is received. We will consider this pt part of step 1.

```
t4
CREATE TABLE tab AS SELECT DISTINCT * from temp_table
ORDER BY cnxid
t5
```

pt6: Remove no longer needed objects.

Cleaning up after the upload process can be considered part of AS1.

```
t5
DROP INDEX temp_table_cid_idx;
DROP TABLE temp_table;
t6
```

pt7: Create index on tab for cnxid,ts

This is pt is equivalent to AS2.

```
t6
CREATE INDEX tab_cid_ts_idx ON tab (cnxid)
CREATE INDEX tab_cid_ts_idx ON tab (cnxid,ts)
t7
```

pt8: Gather statistics

This is part of AS1.

```
t7
begin
dbms_stats.gather_table_stats(ownname=>
'MARIUS',tabname=> '"+tab+"' , estimate_percent=>
DBMS_STATS.AUTO_SAMPLE_SIZE, cascade=>
DBMS_STATS.AUTO_CASCADE, degree=> null, no_invalidate=>
DBMS_STATS.AUTO_INVALIDATE, granularity=>
'AUTO',method_opt=> 'FOR ALL COLUMNS SIZE AUTO');
end;
t8
```

pt9: Calculate connection level aggregate statistics

This is part of AS3.

```
t8
INSERT INTO cnxs (tid, cnxid, reverse, started,
duration, bytes, throughput, packets, datapkts, acks,
pureacks, maxrwnd, minrwnd, avgrwnd, urgents, syns,
resets, fins, pushes, sacks )
SELECT  tid, cnxid, reverse, min(ts) AS started,max(ts)-
min(ts) AS duration, sum(nbbytes) AS bytes, case max(ts)
when  min(ts) then 0 else round(sum(nbbytes) /
(extract(hour from max(ts)-min(ts))*3600  +
extract(minute from max(ts)-min(ts))*60  +
extract(second from max(ts)-min(ts)) )  ) end AS
throughput, count(ts) AS packets, count(nbbytes) -
SUM(DECODE(nbbytes,''0'',1,0))as dataPkts, count(ack) AS
acks, count(ack) - REGR_count(ack,nbbytes) AS pureAcks,
max(win) AS maxRwnd, min(win) AS minRwnd, avg(win) AS
avgRwnd, count(urgent) AS urgents,
SUM(SIGN(INSTR(flags,''S''))) AS syns,
SUM(SIGN(INSTR(flags,''R''))) AS resets,
SUM(SIGN(INSTR(flags,''F''))) AS fins,
SUM(SIGN(INSTR(flags,''P''))) AS pushes,
SUM(SIGN(INSTR(options,''sack''))) AS sacks
FROM tab
GROUP BY cnxid, tid, reverse
t9
```

pt10: calculating retransmissions per connection

This can be considered part of AS5, because it gather statistics needed for the analysis. One might even say that this should not be part of the Upload process.

```
t9

INSERT INTO retransmissions
(tid,cnxid,reverse,unique_bytes,retr_bytes)
SELECT tab1.tid, tab1.cnxid, tab1.reverse, tab2.nbbytes
AS unique_bytes, tab1.bytes - tab2.nbbytes AS retr_bytes
FROM (SELECT tid,cnxid,reverse,bytes
      FROM cnxs
      WHERE tid = "+tid+") tab1
      JOIN
     (SELECT t1.cnxid cnxid, t1.reverse reverse,
             sum(t1.nbbytes) nbbytes
      FROM (SELECT distinct cnxid, reverse, nbbytes,
                   startseq, endseq
            FROM tab) t1
      GROUP BY t1.cnxid, t1.reverse) tab2
```

```
        ON  tab1.cnxid = tab2.cnxid and tab1.reverse =
tab2.reverse
t10
```

pt11: analysing other tables

This pt will not be included in the first measurements since it was not a part of the first versions of oraInTraBase.

```
t10
ANALYZE TABLE retransmissions ESTIMATE STATISTICS SAMPLE
5 PERCENT FOR ALL INDEXED COLUMNS
ANALYZE TABLE cnxs ESTIMATE STATISTICS SAMPLE 5 PERCENT
FOR ALL INDEXED COLUMNS
ANALYZE TABLE cid2tuple ESTIMATE STATISTICS SAMPLE 5
PERCENT FOR ALL INDEXED COLUMNS
t11
```

First of all measuring using the time like this is susceptible to many errors. Still we believe that by doing it 5 times that we can get a good indication of which part of Upload is responsible for the long response time. Measuring this way showed us that the time spent is distributed as follows between the 10 monitoring points. (Table 5)

Processing task	% of the time	Percent of total time
pt_1	$\Delta t = t_1 - t_0 = 5164 \pm 35{,}40$ seconds	28,32%
pt_2	$\Delta t = t_2 - t_1 = 488 \pm 329{,}96$ seconds	2,68%
pt_3	$\Delta t = t_3 - t_2 = 103 \pm 0{,}62$ seconds	0,56%
pt_4	$\Delta t = t_4 - t_3 = 8814 \pm 1561{,}52$ seconds	48,34%
pt_5	$\Delta t = t_5 - t_4 = 626 \pm 114{,}16$ seconds	3,34%
pt_6	$\Delta t = t_6 - t_5 = 293 \pm 97{,}18$ seconds	1,60%
pt_7	$\Delta t = t_7 - t_6 = 514 \pm 19{,}09$ seconds	2,82%
pt_8	$\Delta t = t_8 - t_7 = 1504 \pm 39{,}64$ seconds	8,25%
pt_9	$\Delta t = t_9 - t_8 = 519 \pm 3{,}69$ seconds	2,85%
pt_{10}	$\Delta t = t_{10} - t_9 = 209 \pm 6{,}09$ seconds	1,15%
Total	$\Delta t = t_{10} - t_0 = 18233 \pm 1421{,}06$ seconds.	100%

Table 5: Percent of time consumed by each pt before any improvement/optimisations.

The first thing we should notice from these result is how large the confidence interval for some of the pt's are. Because we are looking for a indication of which part of Upload is responsible for the largest time consumption we accept the large confidence interval in some of the pt's. As we can see from Table 5, most of the time is spent on pt_1 and pt_4. Since these tasks consume most time, we have the largest potential for

improvement by optimising them. pt_4 has an obvious solution: pt_4 consists of dropping columns no longer needed. Looking at the rest of the system we discover that in pt_6 we drop the table that pt_4 spent so much time altering.

We rewrite oraInTraBase upload so that temp_table is created by the installation procedure. When Upload starts, temp_table is empty, and Upload will return it to this state before it ends. This will prevent one user from running more than one instance of the Upload script concurrently. Since Upload is such a resource intensive operation, it is unlikely that one would run more than one Upload simultaneously. In order to implement this idea we move the creation of temp_table from upload to the install part of the system. In this way it is executed only once at the installation time. In pt_6 the drop table statement is transformed into a truncate statement:

Original:

```
DROP TABLE temp_table;
```

Modified:

```
TRUNCATE TABLE temp_table;
```

Truncate will remove all the tuples from the table, but keep the table, its indexes and its statistical data. In pt_5 all the columns of temp_table were previously selected. This was done after *srcip*, *dstip*, *srcport*, and *dstport* have been dropped. Since we are no longer dropping these columns, we have to list the columns that we want to select. In our case that is all the columns of temp_table except the four previously mentioned. The new query in pt_5 will be:

```
CREATE TABLE tab AS
SELECT DISTINCT TS, TTL, IPID, LENGTH, TID, CNXID,
REVERSE, FLAGS, STARTSEQ, ENDSEQ, NBBYTES, ACK, WIN,
URGENT, OPTIONS
FROM temp_table
ORDER BY cnxid;
```

After doing these optimisation steps we expect 48% improvement on the runtime. Previously we had 18,905838 ± 1,635270 seconds now we get 9,268238 ± 0,060897 seconds, which implies 51% reduction in total execution time. After these modifications the only single pt that consumes more than 10% of the time is pt_1 with 57,48% and pt_8 with 15,94 % of the total execution time.

We continue by investigating pt_1. There is a difference between the default way that SQL*Loader and PostgreSQL's works. SQL*Loader keeps the transaction and recoverability by default. The SQL*Loader also has more functionality (see Section 5.1 for more details). We can enable direct path loading in SQL*Loader. This mode of operation is very close to the default of PostgreSQL's copy. All the technical documents about the topic, promise a performance boost with direct path loading. When using direct path loading you can not use any function on the values read from the input file. This is the reason why one might not want to use direct path loading. We have no need to perform any functions on the input data as all the necessary

pre-computations are handled by the modified version of TCPdump.

Enabling direct path loading delivers the promised performance boost. Total runtime is now reduced from 9,268238 ± 0,060897 seconds to 5,590859 ± 0,125682 seconds. For a 10MB trace the runtime of pt_1 was reduced from 5164 ± 35,40 milliseconds to 1501 ± 162,46 milliseconds.

Still pt_1 is the largest contributor to the runtime, contributing 29,45% of the runtime, closely followed by pt_8 with 27,77%.

In an attempt to reduce the runtime we will try a different approach to gathering statistics.

```
begin dbms_stats.gather_table_stats(ownname=>
'MARIUS',tabname=> '"+tab+"' , estimate_percent=>
DBMS_STATS.AUTO_SAMP LE_SIZE, cascade=>
DBMS_STATS.AUTO_CASCADE, degree=> null, no_invalidate=>
DBMS_STATS.AUTO_INVALIDATE, granularity=>
'AUTO',method_opt=> 'FOR ALL COLUMNS SIZE AUTO'); end;
```

We changed this to:

```
ANALYZE TABLE "+tab+" ESTIMATE STATISTICS SAMPLE 5
PERCENT FOR ALL INDEXED COLUMNS;
```

The new analysis process only estimates the statistics used by sampling 5% of all the tuples in the table. Generating statistics by sampling in this way will in most cases produce the same execution plan as a more expensive 100% computation of the statistics [21].

Now after these improvements the runtime of oraInTraBase Upload on a 10MB trace is 4,160645 ± 0,020595 seconds. pt_8 was reduced in time from 1415 ± 36,28 milliseconds in the last measurements to 76 ± 4,90 milliseconds. It is now longer a major contributor to the response time, only 2%.

PgInTraBase Upload creates an index for the temporary table on the cnxid attribute. If a query accesses more than about 12% of the tuples in a table the database system will perform a full table scan instead of using the index [21]. We remove measuring point 2. Since we do not create the index there is no longer a need for dropping it in pt_6 either.

pt_5 touches all tuples in the temp_table, and therefore, according to the statement above it does not use an index. Removing the index and measuring again shows that the time for pt_5 is unchanged. pt_3's runtime is increased by 41%, which shows that in this query the index is helpful. The total runtime is reduced to 3,852644 ± 0,015282 seconds. Compared to the pgInTraBase upload this is only 41% of the runtime. The final question is how the performance of these systems when we run them on a 1GB trace instead of 10MB? PostgreSQL spent 830 ± 4,31 seconds, the unmodified Oracle version spent 1850 ± 12,70 seconds, and the optimised Oracle version spent 382 ± 5,38 seconds.

There is one final improvement we want to do to Upload before we

conclude this section. In pgInTraBase upload the tab table is implicitly clustered because the database system saves the tuple in order that they are inserted in to the table. Because of this the table can implicitly be clustered by inserting the tuples in an ordered fashion, which is done in pt5. This procedure does not guarantee that the table becomes clustered or ordered by index. Instead it depend on the implementation. We feel this is not satisfactory because this behaviour might change in later releases of the database system. Creating a clustered table in Oracle has to follow a certain order as described in Section 3.4. Above we showed that it was faster to populate the table than create the index as opposed to having the index created when the table is populated. Because of the procedure of creating a clustered table we have to do it the least beneficial way. Therefore, we expect to observe reduced performance when we cluster the table. We were surprised at the actual cost of clustering. The runtime with clustering increased from 3,852644 ± 0,015282 seconds to 6,732734 ± 0,39 seconds for a 10MB trace and from 382 ± 5,38 seconds to 822 ± 26,06 seconds for a 1GB trace. For a 10MB trace the increase is 173% while it is 215% for a 1GB trace. Seeing that the difference increases with a larger trace file is an indication that clustering scale worse than the rest of Upload does.

Because of the cost we associated with the cluster clause we will revert to implicit clustering.

In this section, we have seen that we are able to reduce the runtime of Upload.java to a fraction of the original time with some relatively simple modifications. In the end, the runtime of oraInTraBase upload is about half of pgInTraBase upload. This difference is only seen in larger trace files, while the difference in smaller files is insignificant.

8.5 C-Query

C-query as mentioned earlier the most used query in InTraBase. Therefore, we want to measure and optimise its performance. We also want to measure its performance with implicit clustering and clustering using the Oracle functionality.

8.5.1 C-Query for All Connections

The first measurement we want to do is to perform the c-query once for all of connections. Below is the code for the function that performs this test for Oracle.

```
CREATE OR REPLACE FUNCTION test RETURN NUMBER AS
CURSOR all_cid IS SELECT distinct cnxid FROM
unclustred_1g;
all_cid_row all_cid%ROWTYPE;
sum_val NUMBER(19,0);
CURSOR c-query(in_cid IN NUMBER) IS SELECT * FROM
unclustred_1g WHERE cnxid = in_cid ORDER BY ts;
c-query_row c-query%ROWTYPE;
BEGIN
```

```
    sum_val := 0;
    FOR all_cid_row IN all_cid LOOP
      FOR c-query_row IN c-query(all_cid_row.cnxid) LOOP
        sum_val := sum_val+1 ;
      END LOOP;
    END LOOP;
    RETURN sum_val;
  END;
```

The PostgreSQL version is very similar.

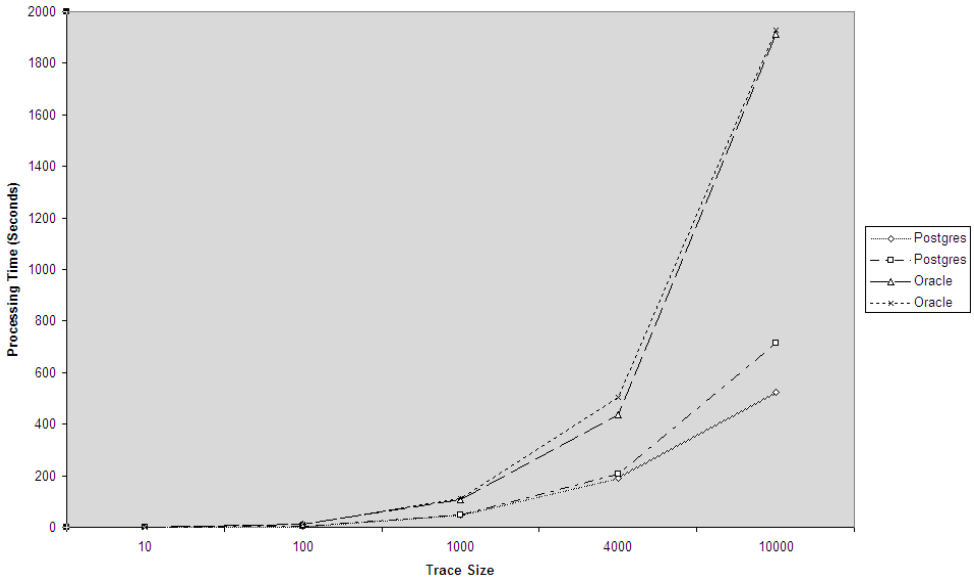

Figure 16: 90% confidence interval for completion time of c-query with different sized trace files.

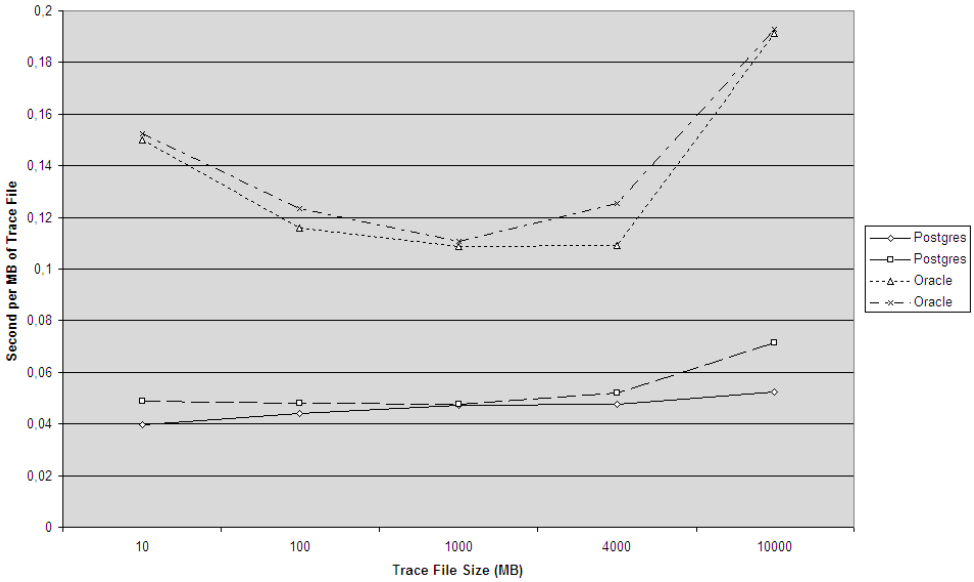

Figure 17: 90% confidence interval for the runtime of c-query per MB of trace file.

Again we did initial tests of this function on a 1GB trace. PostgreSQL needs 46,66 ± 0,33 seconds to perform this test, while Oracle needs 107,59 ± 0,35 seconds. again Oracle can not keep up with PostgreSQL. We did further

testing with 10MB, 100MB, 1GB, 4GB and 10GB traces to whether pggInTraBase or oraInTraBase scale better than the other. Figure 16 shows the 90% confidence interval for the time to completion with different size of trace files. Figure 17 has the same data as Figure 16 but it is now divided by the number of MB in the trace. This will allow us to see how it scales with larger trace files. Linear scaling would be a horizontal line in the Figure 17.

From Figure 17 we can see that PostgreSQL scales close to linearly until 4GB, while Oracle have improvements in performance as the trace file grows towards 4GB. This is an indication that Oracle has a high overhead independent of the table size. Looking at 10GB trace files we can see that both the PostgreSQL and the Oracle version scale worse than linearly. Oracle has a much larger performance degradation with 10GB trace files than PostgreSQL. Because the trace files are larger than the memory allocated to the two DMBS' they need to perform operations on disk to a larger extent than when the trace file could fit in memory. This might be a indication that PostgreSQL handles data better on disk than Oracle.

8.5.2 Analysing the Packet Table

During the previous measurement we used enterprise managers SQL tuning feature to analyse what we could do to improve the performance of our query. It recommends installing another CPU and to run the following:

```
begin
dbms_stats.gather_table_stats(ownname => 'MARIUS',
tabname => 'MF3', estimate_percent =>
DBMS_STATS.AUTO_SAMPLE_SIZE, method_opt=>'FOR ALL
COLUMNS SIZE AUTO');
end;
```

We assume that the same result as above is achieved with:

```
ANALYZE TABLE mf3 ESTIMATE STATISTICS SAMPLE 5 PERCENT
FOR ALL INDEXED COLUMNS
```

Since Enterprise Manager gives us this recommendation, we are no longer sure about our previous assumption that the methods are equivalent. Therefore, we will test c-query on three different tables. One that is not analysed, one that is analysed using ANALYZE, and one that is analysed using dbms_stats.

When optimising Upload we changed from using dbms_stats to ANALYZE, the opposite of what we are investigating now. At that time Uploads runtime was reduced from 1415 ± 36,28 milliseconds in the last measurements to 76 ± 4,90 milliseconds on a 10MB trace. As we can see from Figure 18, there is virtually no difference between having analysed the table or not for both 10MB and 100MB. There is a small difference for a 1GB trace.

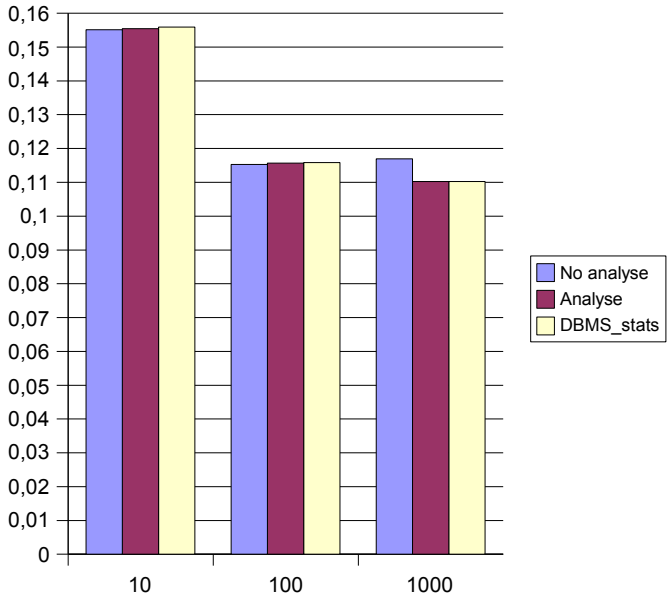

Figure 18: Seconds per MB with using different analysis tools

8.5.3 Which Indexes to Use?

During the previous tests we had indexes for both cnxid and cnxid,ts. We would expect Oracle to use the second one as it could retrieve the data in ordered sequence as requested by the c-query. However, looking at the plan used for the queries, as shown in Enterprise Manager, we see that Oracle prefers to use cnxid and not the cnxid, ts index. Adding a hint in the query will ensure that Oracle selects the index we want. The new query looks like this:

```
SELECT /*+ INDEX(mf3 mf3_cid_ts_idx)*/ *
FROM mf3
WHERE cnxid = in_cid
ORDER BY ts;
```

Running the c-query test on a 1GB trace with this hint reduces the runtime from 107,59 ± 0,35 seconds to 100,91 ± 0,16 seconds. This means that throughout the application we have to insert hints at the appropriate places.

8.5.4 Bulk Collection

Bulk collection is a special way of fetching data inside a PL/SQL function. Normally there is a context switch for each tuple that is returned to PL/SQL when PL/SQL requests the tuple. With bulk collect several tuples are returned for each context switch and kept in a memory structure inside your PL/SQL procedure. Because of the reduction in context switching we expect increased performance. We modify the test for c-query to use bulk collect, and also included the hint to use the “correct” index. The new function looks like this:

```
CREATE OR REPLACE FUNCTION test1 RETURN NUMBER AS
  TYPE number_table is TABLE OF mf2.cnxid%TYPE INDEX BY
BINARY_INTEGER;
  cid_list NUMBER_TABLE;
  TYPE tab_table IS TABLE OF mf2%ROWTYPE INDEX BY
BINARY_INTEGER;
  tuple_list TAB_TABLE;
  sum_val NUMBER(19,0);
  i NUMBER;
  ii NUMBER;
  CURSOR c-query(in_cid IN NUMBER) IS
              SELECT /*+ INDEX(mf2 mf2_cid_ts_idx)*/ *
              FROM    mf2
              WHERE   cnxid = in_cid
              ORDER BY ts;
BEGIN
  sum_val := 0;
  SELECT DISTINCT cnxid
  BULK COLLECT
  INTO    cid_list
  FROM    mf2
  ORDER BY 1;
  FOR i IN cid_list.FIRST .. cid_list.LAST LOOP
    open c-query(cid_list(i));
    LOOP
      FETCH c-query BULK COLLECT INTO tuple_list LIMIT
      1000;
      EXIT WHEN tuple_list.COUNT = 0;
      FOR ii IN 1..tuple_list.COUNT LOOP
         sum_val := sum_val+1 ;
      END LOOP;
    END LOOP;
    close c-query;
  END LOOP;
  RETURN sum_val;
END;
```

Note the LIMIT statement. This parameter defines how many tuples are being fetched into the array in one operation. Changing this number has a large performance impact. If we try to get the entire 1GB table at once it is likely that we will run out of memory. Figure 19 shows the difference in runtime for a 1GB trace depending on the LIMIT value. As we can see it has major impact on the performance. In the best case it reduces the runtime from 100,91 ± 0,16 second to 99,00 ± 0,19 seconds with LIMIT 100.

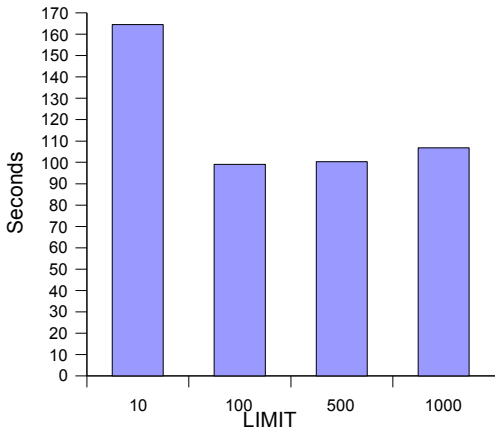

Figure 19: Average runtime of c-query with different LIMIT

8.5.5 Time Per Connection

To better understand why Oracle did not perform to our expectations compared to PostgreSQL on the c-query test, we looked at how the processing time per connection increased with the size of the connection. From earlier research we know that PostgreSQL increases nearly linearly in relation to the size of the connections: retrieving all packets of a connection takes on average twice as long for a connections that contain twice as many packets.

We will define groups of connections based on the number of packets in each connections as shown in Table 6.

Name	**Abrivation**	**Number of packets**
Extra Small	XS	1 – 99
Small	S	100 – 499
Medium	M	500 – 999
Large	L	1.000 – 9.999
Extra Large	XL	10.000 –

Table 6: Definition of connection groups by the number of packets they contain.

For this experiment we use a 10GB trace. Figure 20, 21, and 22 show how the connections and the transferred data is divided among the groups. As we can see from the figures the division of packets in percent between each of the groups is almost identical between the four trace files. The XS group has more connections than the rest added together, while the XL group carry more data than the other groups together. Looking at Figure 23 and 24 we see that there is more packets in mixed traffic (MT) because a different sampling size was used for the two trace files. Figure 23 shows the difference between the two trace files. Mixed traffic has more than 32 times as many connections. We will see how this effect the performance later in this chapter.

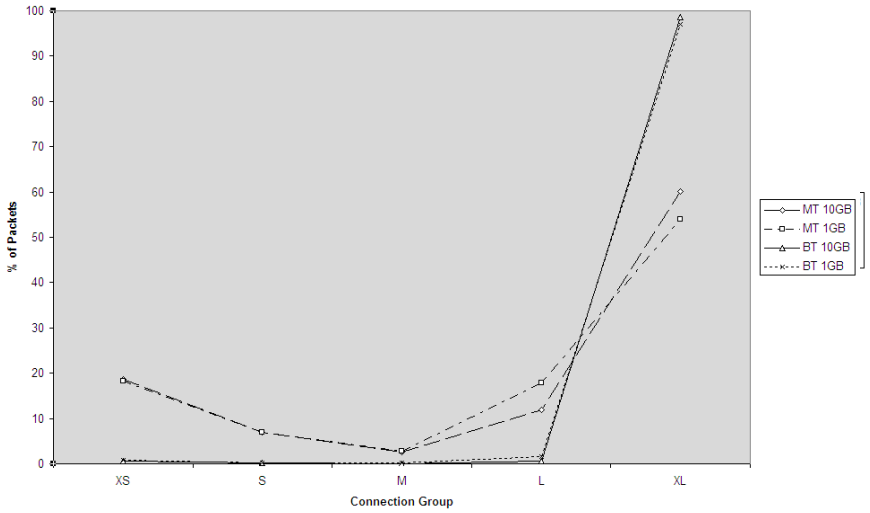

Figure 21: Percent of packets divided by connection groups

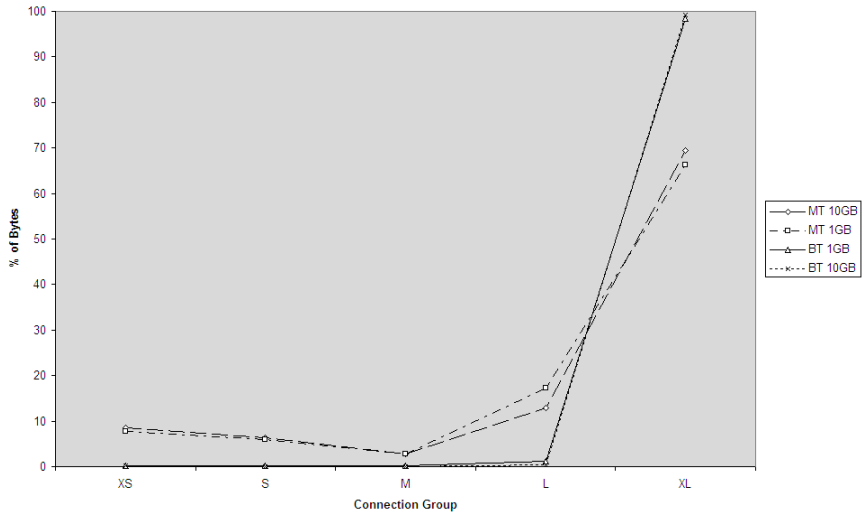

Figure 22: Percent of transferred bytes divided by connection groups

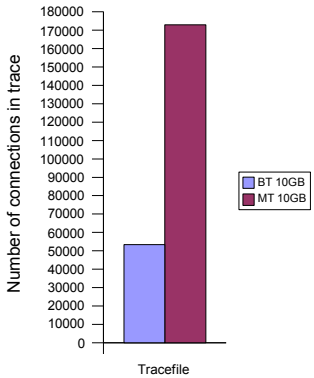

Figure 23: Amount of connections in trace file.

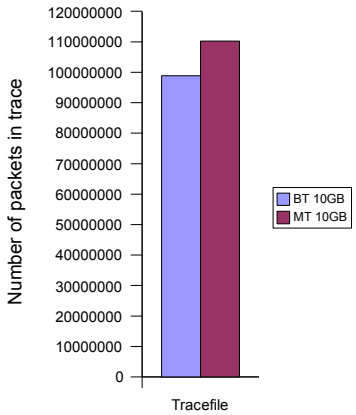

Figure 24: Amount of packets in trace file

Using this knowledge we will see how the c-query test performs on each of

the groups. We also tried various sizes for bulk collection LIMIT. We ran our measurements on the mixed traffic trace file unless otherwise indicated. We separated each of the groups in their own table, indexed them for (cnxid) and (cnxid,ts), and ran c-query test on each of the table.

We can see from Figure 25 that the bulk collect has little impact on retrieval time except that LIMIT 10 is less efficient that the other LIMIT valuer. We note that the performance is the worst for XS and XL, the two groups that also contain the majority of packets. On the PostgreSQL version previous research found that a c-query test scaled close to linearly. For S, M and L this is true for Oracle too. From this experiment we can conclude that bulk collect does not give us the expected performance boost in the c-query test.

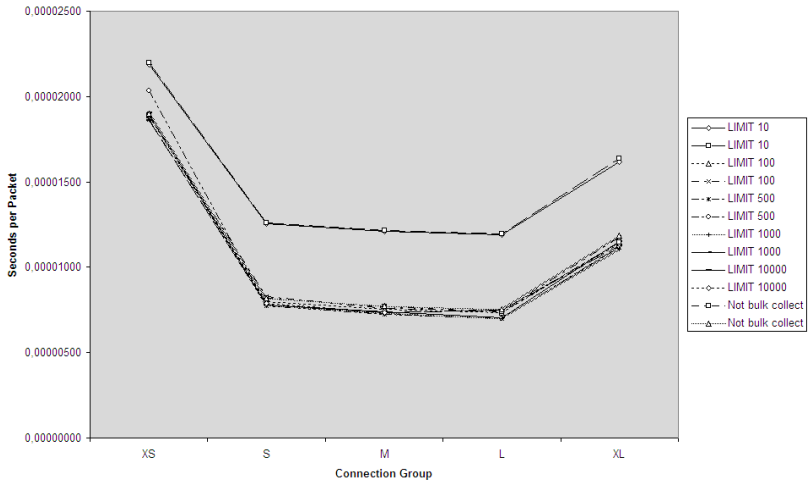

Figure 25: 90% confidence interval for the time to complete c-query

As for the test with BT files we divide the XL group in to two separate groups:

- XL 10.000 – 100.0000
- XXL 100.000 -

Referring to Figure *21,* we see that for a BT trace the XS group consist of a very small amount of the total number of the packets (<1%), but it has the majority of connections (>97%). The XXL group is opposite it has more than 97% of the packets and less then 1% of the connections. As we can see from Figure 26 the XS and XXL group take about the same amount of time to complete. This show that even a very low number of packets can take a long time to return if they are divided among many connections.

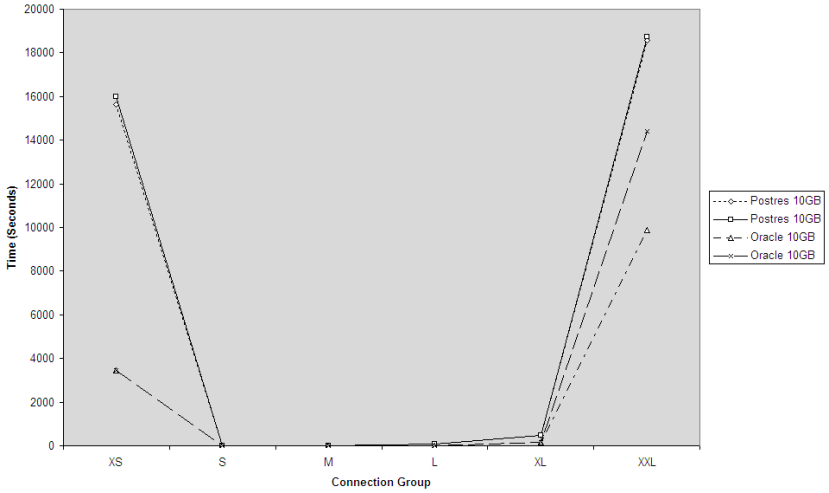

Figure 26: C-Query test on BT trace

8.6 Analysis

In this section we will look at AS5 (See Table 3 in Section 4.2). Due to time constraint it was not possible for us to fully port and validate the entire analysis as it was implemented in pgInTraBase. At first, we did not use bulk collect. Also note that due to time constraint calc_rtt, a C++ function, have not been ported. Therefore, both systems have this function replaced with one that returns a static value. The initial test shows that oraInTraBase completes the analysis of a 10MB trace in 47,61 ± 0,12 seconds, while pgInTraBase only needs 23,56 ± 0,36 seconds. We will first focus on traces of mixed traffic up to 1GB. In the end we will include measurements for BT traffic and traces of up to 10GB.

During this section we make improved versions of oraInTraBase. In order to distinguish each improvement step, we name each version Oracle*n*, where n is the improvement version. As a general rule Oracle*n+1* will be Oracle*n* with some additional improvements.

8.6.1 New Indexes

Enterprise Manager Oracle 10g shows an ordered list of queries and sessions based on which consumes the most resources. The query responsible for most of the resource consumption during the analysis process is:

```
SELECT MIN(CASE WHEN TS_ACK>TS_SENT THEN TS_ACK-TS_SENT
ELSE NULL END) AS RTT
FROM (SELECT ENDSEQ, MAX(TS) KEEP (DENSE_RANK FIRST
ORDER BY ENDSEQ ASC) AS TS_SENT
FROM TBL
WHERE CNXID = :B1 AND REVERSE IS NULL AND ENDSEQ IS NOT
NULL AND FLAGS NOT LIKE '%R%' AND FLAGS NOT LIKE '%F%'
GROUP BY ENDSEQ ORDER BY ENDSEQ ASC,TS_SENT DESC) TS1
FULL OUTER JOIN
     (SELECT ACK, MIN(TS) KEEP(DENSE_RANK FIRST ORDER BY
```

```
TS ASC) AS TS_ACK
FROM TBL
WHERE CNXID = :B1 AND REVERSE IS NOT NULL AND ACK IS NOT
NULL AND FLAGS NOT LIKE '%R%' AND FLAGS NOT LIKE '%F%'
GROUP BY ACK ORDER BY ACK,TS_ACK) TS2 ON TS1.ENDSEQ =
TS2.ACK
```

SQL Tuning Advisor, another tool provided by Enterprise Manager, suggest making a new index. We add the following statement to pt7:

```
CREATE INDEX tab_endseq_cid_idx ON tab (endseq,cnxid)
NOLOGGING
```

Creating the index does not change anything. Oracle query planner/optimiser still chooses to use the same execution plan as before. This fact illustrates that the advisor tool in Enterprise Manager is able to find better execution plans than the query optimiser. We add a hint to the query in question:

```
SELECT /*+ tbl_endseq_cid_idx */ ENDSEQ, MAX(TS) KEEP
(DENSE_RANK FIRST ORDER BY ENDSEQ ASC) AS TS_SENT
FROM TBL
WHERE CNXID = :B1 AND REVERSE IS NULL AND ENDSEQ IS NOT
NULL AND FLAGS NOT LIKE '%R%' AND FLAGS NOT LIKE '%F%'
GROUP BY ENDSEQ ORDER BY ENDSEQ ASC,TS_SENT DESC) TS1
```

With the hint the query optimiser is able to chose the better execution plan, and use the new index:

```
idx1: packet (endseq,cid);
```

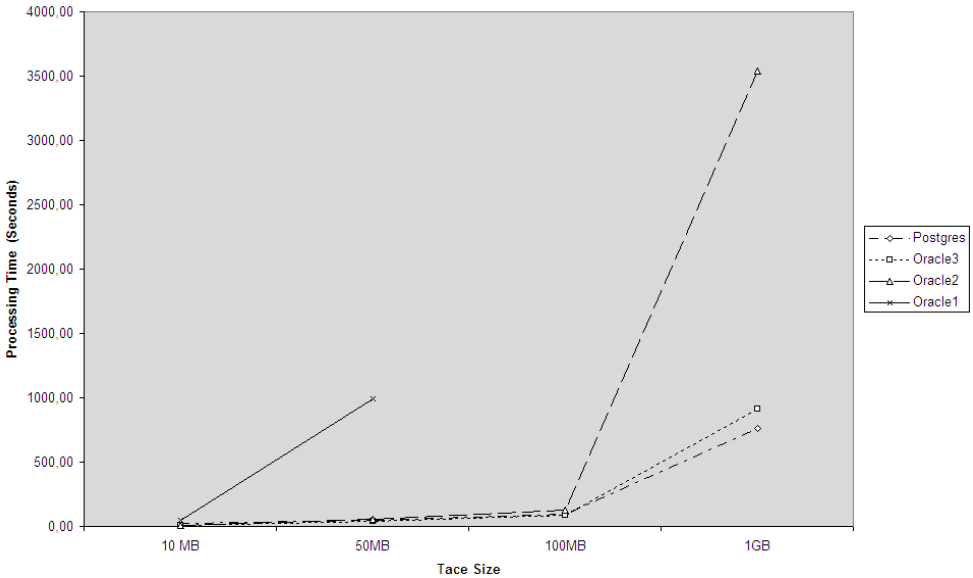

Figure 27: The response time for analysis of various trace files.

This gives us a significant improvement in performance. With the new index we run the advisory tool again, now it suggest to implement a SQL Profile. Implementing it give us further improvements. The new performance is

plotted in Figure 27 as Oracle2. With the new index in place we ran the analysis again. While doing this, we identified the largest resource consumers with Enterprise Manager and ran the advisory tool for them. The end result is that we add the following three indexes to our system.

```
idx2: bulk_transfer (cnxid,n_lim,tid);
idx3: app_period(startq,rtt,ts);
idx4: app_period(n_lim,cnxid,tid,reverse);
```

The indexes don't need to be added to the upload process as the involved tables are never removed from the system. Instead, they are added to the installation process of oraInTraBase. Also, Enterprise Manager suggests implementing a few SQL profiles to get better execution plan for some of the heavily resource consuming queries.

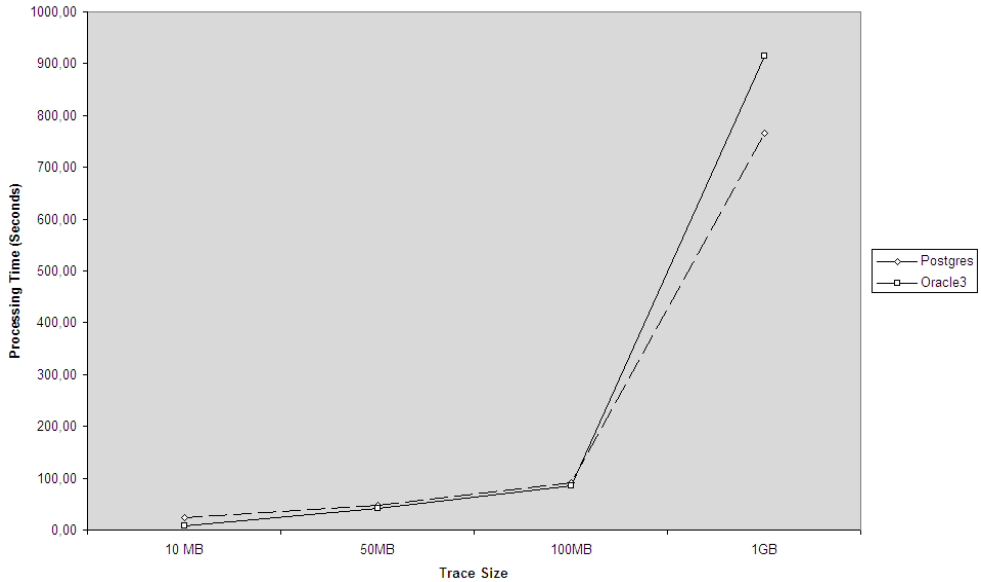

Figure 28: The response time after adding 4 indexes compared to pgInTraBase.

Figure 28. shows how performance changed when we implement the indexes and profiles discusses above. The unmodified version is marked: Oracle1. We stopped the measurements at 50MB for Oracle1 as we saw that it was already then having a response time equal PostgreSQL at a 1GB trace. Oracle2 is the system with idx1 and the hint so that the index will be used. As we can see it has good performance with the smaller traces, but it does not scale very well. Oracle3 is further improved from Oracle2. It include

	Postgres	Oracle3
10MB	23,56 ± 0,36	7,88 ± 0,05
50MB	46,82 ± 1,254	1,26 ± 0,13
100MB	91,56 ± 5,178	4,57 ± 0,35
1GB	765,30 ± 27,06	14,52 ± 6,91

Table 7: Average response time of the analysis for various sized trace files.

idx2, idx3, and idx4. This solution also has problems with scaling, but to a less degree than the Oracle2 version. Oracle3 and PostgreSQL scale similarly. Figure 28 shows the difference between PostgreSQL and Oracle3 version in a scale that makes it easier to see the difference. Table 7 show the values of the data used in Figure 28.

In order to say that one is better than the other the confidence interval over the difference of the two measurement sets should not include zero. Therefore, we take oraInTraBase response time – pgInTraBase response time and compute the 90% confidence interval. Doing this we see that with 90% confidence oraInTraBase has shorter response time for traces of 10MB, 50MB and 100MB. While pgInTraBase has shorter response time for 1GB trace.

8.6.2 Revisiting Bulk Collection

Even though we could only see small improvements when using bulk transfer in the c-query test, we implemented it around some of the key queries. Using Enterprise Managers top activity tool we identified the four queries that are responsible for most of the resource consumption: Query 1 to 4.

Query 1:

```
SELECT CASE SIGN(nbbytes-(mss-12)) WHEN -1 THEN 1 ELSE 0
END as p, CASE WHEN
reverse IS NULL THEN 0 ELSE 1 END as r, flags, ack,
endseq, nbbytes, ts
FROM tbl
WHERE cnxid=row2.cid and ((reverse IS NOT NULL and
nbbytes>0) or (reverse IS
NULL and ack Is NOT NULL)) ORDER BY ts;
```

Query 2:

```
SELECT min(CASE WHEN ts_ack>ts_sent THEN ts_ack-ts_sent
ELSE NULL END) as rtt
FROM (SELECT /*+ tbl tbl_endseq_cid_idx*/ endseq,
MAX(ts) KEEP (DENSE_RANK FIRST
ORDER BY endseq ASC) as ts_sent
FROM tbl
WHERE cnxid = in_cid AND reverse IS NULL AND endseq IS
NOT NULL AND
flags NOT LIKE '%R%' AND FLAGS NOT LIKE '%F%'
GROUP BY endseq
ORDER BY endseq ASC,ts_sent DESC)  ts1
FULL OUTER JOIN
SELECT ack, MIN(ts) KEEP(DENSE_RANK FIRST ORDER BY ts
ASC) as ts_ack
```

```
FROM tbl
WHERE cnxid = in_cid and reverse IS NOT NULL AND ack IS
NOT NULL AND FLAGS
NOT LIKE '%R%' AND FLAGS NOT LIKE '%F%'
GROUP BY ack
ORDER BY ack,ts_ack) ts2
ON ts1.endseq = ts2.ack;
```

Query 3:

```
SELECT CASE WHEN reverse IS NULL THEN 0 ELSE 1 END as r,
ack, startseq, ts
FROM tbl
WHERE cnxid = in_cid AND ( (reverse IS NULL AND nbbytes
> 0) or (reverse IS NOT
NULL and ack IS NOT NULL and (nbbytes=0 or nbbytes IS
NULL))) and flags not like '%R%' and flags not like
'%S%' and flags not like '%F%' ORDER BY ts;
```

Query 4:

```
SELECT *
FROM  (SELECT rtt, ts, n as n_b, f as f_b, m_point as
m_point_b, mss as mss_b,
startq, duration as b_duration, tput as b_tput, bytes as
b_bytes, datapkts as b_pkts
FROM bulk_transfer WHERE tid = in_tid and cnxid=
in_row2_cid and reverse IS NULL and n_lim = 1)
full outer join

(SELECT rtt, ts, n as n_a, f as f_a, m_point as
m_point_a, mss as mss_a, type, startq, duration as
a_duration, tput as a_tput, bytes as n_bytes, datapkts
as n_pkts, push, idle
FROM app_period WHERE tid = in_tid and cnxid =
in_row2_cid and reverse IS
NULL and n_lim=1)
USING (startq, rtt, ts)
ORDER BY startq;
```

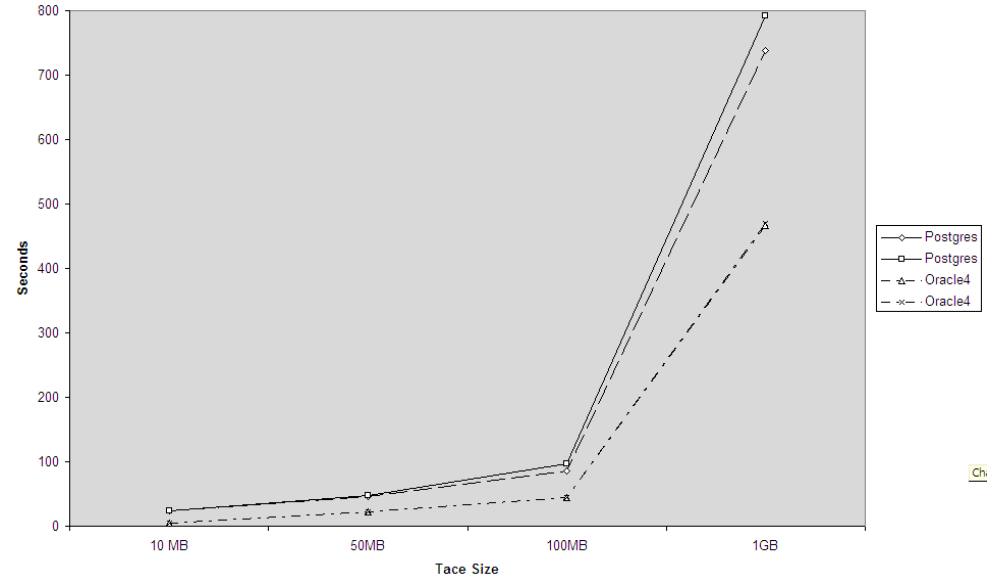

Figure 29: Response time for the analysis process

We implement bulk collect for query 1, which leads to improved version Oracle4. In Oracle5 we also do bulk collect for query 4. Finally, Oracle 6 includes bulk collect for query 3 too. It is not meaningful to add a bulk collect to query 2, because it only returns one row. Note that Query 2 is the same that was identified as the most resource consuming query prior to adding the four additional indexes.

After adding bulk collect to the queries that consumed most resources during an analysis, the performance improved much more than we expected. We had low expectations, because bulk collect gave only minor improvements in the c-query test. We use LIMIT 100 for the bulk collect statements. In our c-query test it was impossible to say that one LIMIT value was better than another among 100, 500, 1000, and 10000. Figure 29 shows that oraInTraBase improvement version 4 with bulk collect for Query 1 performs better than pgInTraBase with all trace sizes. Looking at Figure 31

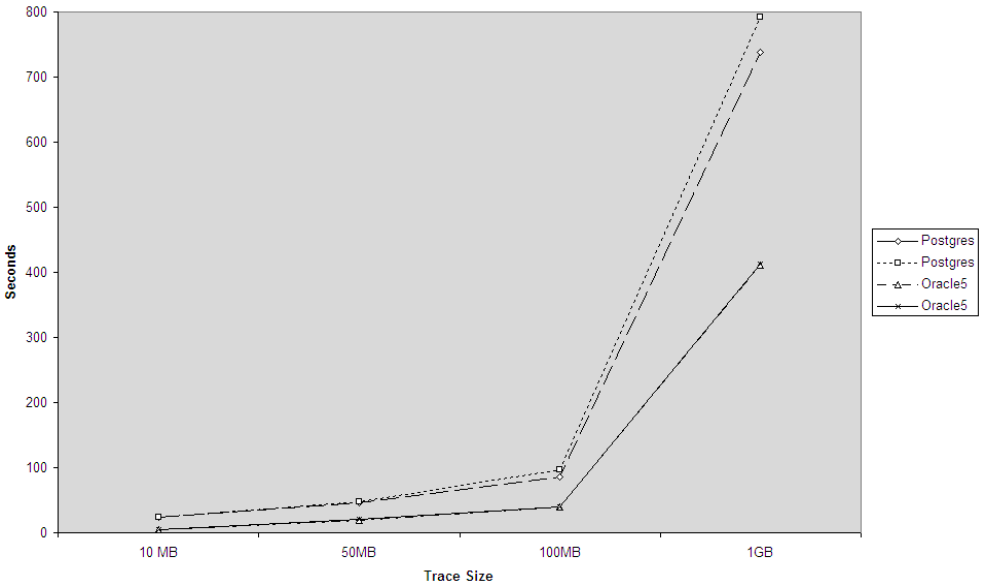

Figure 30: 90% confidence interval of the runtime of the analysis process

we see that oraInTraBase scales approximately linearly over the 4 trace file sizes, while pgInTraBase improves with larger trace files.

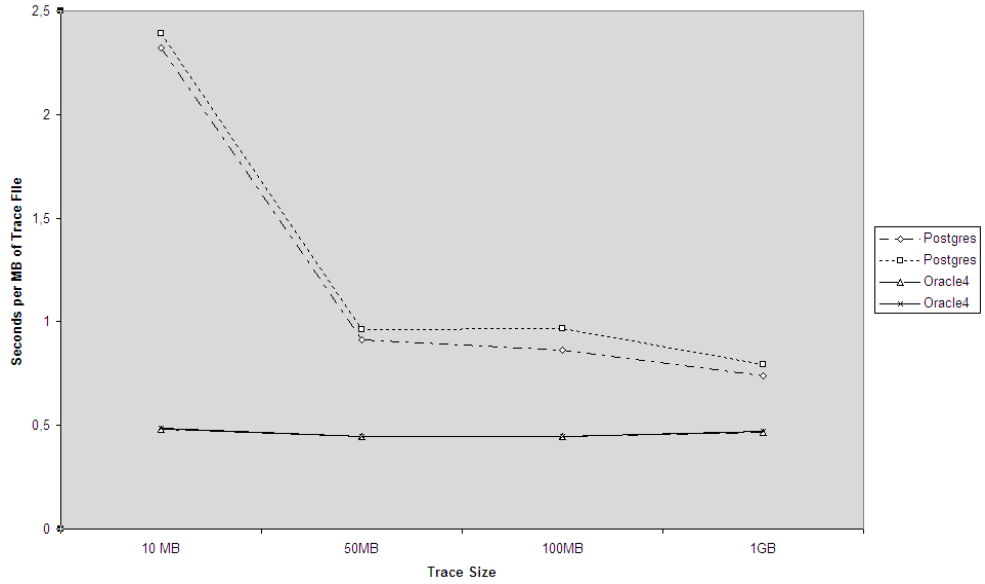

Figure 31: 90% confidence interval of response time per MB of trace file.

For Oracle5 the runtime is reduced to 412,05 ± 0,62 seconds using a 1GB trace file. Our tests left us unable to say with 90% confidence that Oracle6 is better or worse than Oracle5. Because of the extra complexity that the bulk collect adds, we reverted to the Oracle5 version since we could not prove that Oracle6 was an improvement for a 10MB, 50MB, 100MB or 1GB trace.

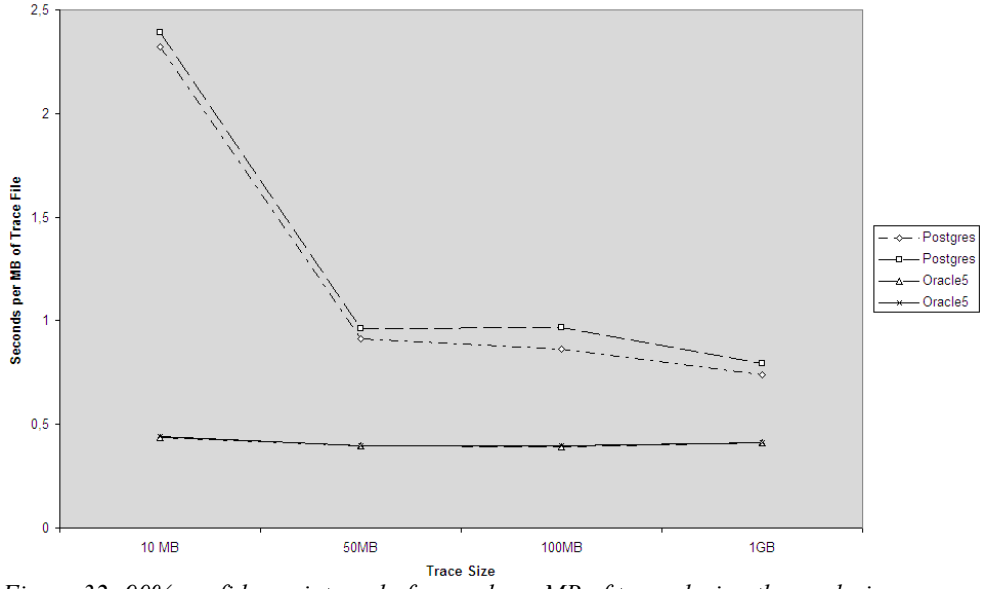

Figure 32: 90% confidence interval of second per MB of trace during the analysis process

Figure 30 and 32 shows the performance statistics of pgInTraBase and Oracle5. Oracle5 has significantly lower run time for trace files of 10MB, 50MB, 100MB and 1GB. From Figure 32 we see that oraInTraBase is still

scaling close to linearly with the bulk collect.

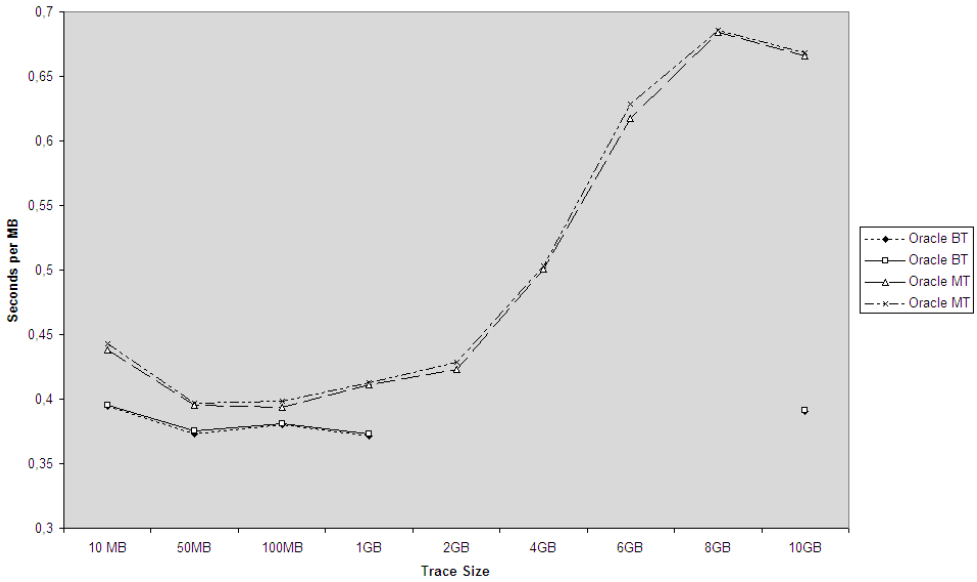

Figure 33: Performance of Analysis with different traffic types.

Now, we investigate the scaling behaviour with trace file size up to 10GB. Figure 33[12] shows how the Oracle5 version scale with BitTorrent (BT) traffic and Mixed Traffic (MT). As we can see it scales linearly with BT trace file, but it scale worse than linearly with the Mixed Traffic trace file. We believe the reason for this is the amount of connections. Our first hypothesis is that the scaling observed with larger than 1GB trace files for mixed traffic is caused by memory constraint per process in a 32-bit OS. A OS is usually able to address individual bytes of memory. In a 32-bit system 32-bit is used as the length of the memory address. Therefore, we can address 2^{32} = 4.294.967.296 individual bytes, or 4GB of memory. Because each process in a modern OS gets its own virtual memory space each process can address the full 4GB, but no more. In a 64-bit system the memory address is 64 bit long and it follows that each process can address 2^{64} = 18.446.744.073.709.551.616 individual bytes. Since this behaviour is apparent only when using Mixed Internet Traffic and not with BitTorrent, as seen in Figure 33, our hypothesis is likely wrong. It will require more investigation to explain why oraInTraBase scales much better when using BT than when using MT.

8.6.3 Without SQL Profile

During this chapter we have run literally hundreds of tuning operations in Enterprise Manager. We have also implemented many SQL Profiles. Maybe too many to duplicate the results. Therefore, we will as a final part of this chapter try to see how much impact the SQL Profiles had. We will remove all objects belonging to oraInTraBase and measure it as if it was a newly

12 Tests were not performed from 2-8GB trace files for BT because we lost the SQL profiles and it would be too time consuming to reconstruct it. We expect it would show linear scaling since both 1GB and 10GB is in line with linear scaling.

installed system. Figure 34 shows the average time for both upload and analysis in a fresh system. In this case fresh system means that we installed oraInTraBase and ran one 10GB trace and implemented all SQL profiles suggested by OEM tuning advisor. As we can see from Figure 34, oraInTraBase is faster in both Upload and analysis for 10MB, 100MB, and 1GB. It is how ever surprising to see that oraInTraBase have a large increase in Seconds per MB when the trace file is increased to 10GB, but pgInTraBase does not. However, oraInTraBase completes both analysis and Upload in the time pgInTraBase use for Upload. We did not have at our disposal larger trace files than 10GB so we are unable to see if this trend remains with increasing size of trace files.

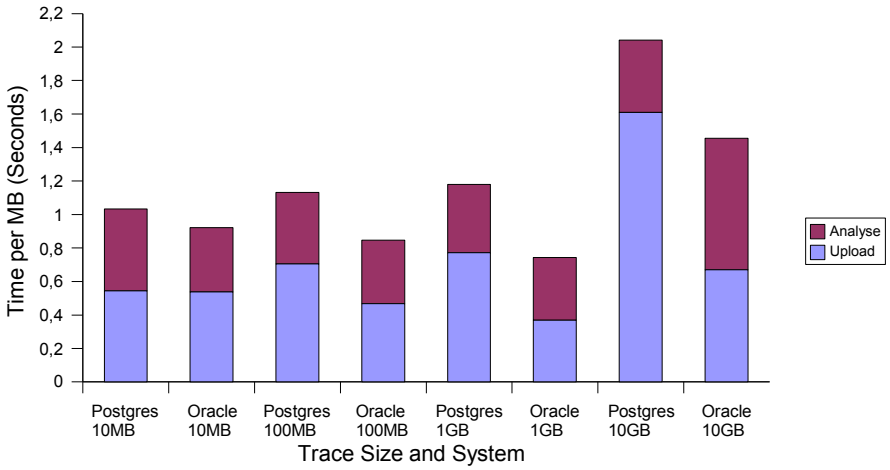

Figure 34: Average time for the upload and the analysis using BT tracses

9 CONCLUSION

In this section we will draw conclusions from our work. In Section 9.1 we will make a summary of the contributions made through this thesis. In Section 9.2 we will do a critical evaluation of our work and finally in Section 9.3 we will present future work in this area.

9.1 Summary

The summary is divided into two separate parts, one for each of the major tasks. In Section 9.1.1 we look at the porting, and in Section 9.1.2 we look at the performance measurements and evaluation.

9.1.1 Porting

InTraBase is implemented using a wide variety of features in PostgreSQL. It is a complex system that is designed to handle large amount of data. We have shown that the major tasks in porting a system from PostgreSQL to Oracle 10g lie the SQL and external functions and not in the PL/SQL. Most of the PL/SQL code can be used as it is. PostgreSQL has some nice capabilities in SQL that Oracle lacks, particularly DISTINCT ON, LIMIT, and OFFSET. All of which can be duplicated in Oracle, but not in an equally neat way.

The source code for oraInTraBase is much easier to understand than the code for pgInTraBase, because the syntax in PostgreSQL 7.4 required a large amount of text string quoting. It has also been more neatly formatted and commented, therefore we believe that it will be easier to extend oraInTraBase than pgInTraBase in the future.

We have found solutions to the lack of PL/R in Oracle. Using Rserver and a C++ function will in theory be a fully good replacement for PL/R.

9.1.2 Performance Measurements and Evaluation

One of the main motivators behind porting pgInTraBase to Oracle was the expected performance gain. If the upload process is taken into account, we have achieved a significant performance gains for all measured trace sizes (10MB, 100MB, 1GB, 10GB). Taking only the analysis into consideration, just a small amount of performance was gained compared to pgInTraBase using the three smaller traces. At 10GB, pgInTraBase performs significantly better than the new Oracle version.

During the course of our measurement, it became clear that the weakest point of pgInTraBase is the upload process. This part of the system was not included in all of the original research papers about InTraBase, but in our opinion it should be taken into account. This is especially true with our stated goal of making a system capable of performing a on-line analysis.

We have shown how to apply the various tools provided by Oracle 10g to perform the tuning process. Also, we have seen their effectiveness in reducing time needed to tune an application this complex.

9.2 Critical Evaluation

If we were to do the entire project again, a few things would have been done differently. At an earlier stage in the project we should have decided which functions to measure. We did port a significant amount of code that we did not have time to either validate or measure. Also we should have spent less time on the calc_rtt and pprate problems, as they could be seen as external to the database application.

9.3 Future Work

There is much work to be done before oraInTraBase can be used in production. The most significant work is to adapt pprate and calc_rtt, which are written in PL/R and C++, respectively. For the consistency of the idea of performing the entire process inside the database and making it easier to port it might be better to port these functions to PL/SQL. The main concern with this idea is performance. It would be an interesting task to evaluate performance of calc_rtt as a C++ function and as a native compiled PL/SQL function.

An interesting research topic would be to see how an optimised version of pgInTraBase would perform against oraInTraBase.

It might be possible to restructure the analysis process to complete all computation for a given connection and only do the C-Query once. Then the analysis process must be redesigned so that it retrieves all the packets in one connection and then complete the analysis for the connection before flushing the packets from memory. If a redesign like this was done it would also allow packets to be inserted into the system in streams of completed connections.

Systems with 64-bit memory address pointers are becoming more and more the norm, therefore it would be interesting to see how the system would perform on such a system. We expect that the large performance loss between 1GB and 10GB would be moved as we added more memory to the system. Now the system is running at the maximum memory the OS can handle per process.

One could also explore the possibility of making the entire Upload program unnecessary by mounting the trace file as an external table. This is likely to further improve the performance of populating the database. Seen from a design point of view it would be beneficial as this essential part of the system would no longer be performed by a separate program.

BIBLIOGRAPHY

[1] , PostgreSQL vs. SQL Server, Oracle: Enterprise-ready and able to compete, http://searchenterpriselinux.techtarget.com/tip/0,289483,sid39_gci1222466,00.html
[2] Roberto Mello, Porting from Oracle PL/SQL, http://pgsqld.active-venture.com/plpgsql-porting.html
[3] Alexander Chigrik, Oracle 9i Database vs DB2 v8.1, http://www.mssqlcity.com/Articles/Compare/oracle_vs_db2.htm
[5] M. Siekkinen, E.W. Biersack, G. Urvoy-Keller, V. Goebel, T. Plagemann , InTraBase: Integrated Traffic Analysis Based on a Database Management System,In Proceedings of IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services, 2005
[6] M. Siekkinen, E.W. Biersack and V. Goebel , Efficient Packet- Level Traffic Analysis Using an Object-Relational DBMS: A Case Study,IEEE E2E Mon, 2006
[7] Matti Siekkinen, InTraBase - Database Tool, Internal documentation, Eurocom,
[8] Matti Seikkinen, Root Cause Analysis of TCP Throughput: Methodology, Techniques, and Application, 2006
[4] , INTERNET USAGE STATISTICS - The Big Picture, http://www.internetworldstats.com/stats.htm
[9] Mark Corvella, Balachander Krishanmurthy, Internet Measurement: Infrastructure, Traffic and Applications.,Wiley , 2006
[10] Rick Greenwald, Robert Stackowiak & Jonathan Stern, Oracle Essentials, Oracle Database 10g,O'Reilly Media, Inc., 2004
[11] J.Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Herber, Scientific data management in the coming decade.,SIGMOD Ree, 2005
[12] , Tcptrace, http://jarok.cs.ohiou.edu/software/tcptrace/
[13] , CoralReef, http://www.caida.org/tools/measurement/coralreef/
[14] Chuck Creanor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, Oliver Spatscheck, Gigascope: High Performance Network Monitoring with an SQL Interface,, 2002
[15] Hector Garcia–Molina, Jeffrey D. Ullman, and Jennifer Widom , Database Systems, The Complete Book,Perntice-Hall Inc , 2002
[16] , PL/R - R Procedural Language for PostgreSQL, http://www.joeconway.com/plr
[17] , PL/SQL, http://en.wikipedia.org/wiki/PL/SQL
[18] Doug Stuns, Tim Buterbaugh, Bob Bryla., OCP Oracle 10g Administration II,Symbex, 2005
[19] Chip Dawes, Bob Bryla, Joseph C Johnson, Matthew Weishan., OCA Oracle 10g Administration I,Symbex, 2005
[20] Cary Millsap with Jeff Holt, Optimizing Oracle Perfomance,O'Reilly, 2003
[23] , Automatic SQL Tuning - SQL Profiles,Metalink, Note:271196.1, 2004
[22] Raj Jain, The art of comupter systems performance analysis,Wiley-Interscience, 1991

[21] Mark Gurry, Oracle SQL Tuning, Pocket Reference,O'Reilly, 2001

APPENDIX A - USER GUIDE

This is a guide to installing the necessary components of oraInTraBase. First Install Oracle 10g. The scripts used in this chapter assume that all passwords are “passord”.

Installs components:

```
sqlplus system/passord @install.sql
```

Compiles Upload:

```
cd intrabase
javac -classpath
.:/disks/fc/oracle/oracle/product/10.2.0/jdbc/lib/ojdbc1
4.jar PacketCopier.java
javac -classpath
.:/disks/fc/oracle/oracle/product/10.2.0/jdbc/lib/ojdbc1
4.jar TcpdumpCopier.java
javac -classpath
.:/disks/fc/oracle/oracle/product/10.2.0/jdbc/lib/ojdbc1
4.jar Upload.java
cd ..
```

Upload a 10MB BT trace, and display elapsed time[13]:

```
cd intrabase
./plotsar.pl java -classpath
.:/disks/fc/oracle/oracle/product/10.2.0/jdbc/lib/ojdbc1
4.jar Upload <PATH>/bt_10 orcl a b c mf1 t
cd ..
```

To create the functions used in the analysis and synonymes run:

```
sqlplus new/passord @ready.sql
```

Run analysis on the uploaded trace, and display elapsed time:

```
./plotsar.pl sqlplus new/passord @test_app
```

13 <PATH> has to be subsittuted for the exact path to the trace file, for instance: /disks/fc/oradata/dvd/tracefile

APPENDIX B - CONTENT ON DVD

/clean.sql – Script that removes data created by Upload and Analysis.

/plotsar.pl – Script used to measure the time a application use.

/ready.sql – Script that should be run between Upload and Analysis to ensure that the code is compiled and that synonyms are correct.

/run – Samples of commands to compile and run Upload.

/test_app.sql – Script that start the test_app function.

/intrabase/ – contains the Upload program and the components needed by it. The content of this folder was inherited from the pgInTraBase project. The files that has been modified are mentioned mentioned below. There is a lot of files in this directory, the are included as they were received.

/intrabase/TcpdumpCopier.java – Small modifications was made to this file to start the SQL*Loader.

/intrabase/test*n*.java - Various iterations of the Upload program as presented in Chapter 8.

/intrabase/Upload.java – adapted version of Upload to work with Oracle.

/src/oraInTraBase/install.sql – Script that set up a user called new with the privileges, tables, and objects needed to run oraInTraBase.

/src/oraInTraBase/intrabase1.* - oraInTraBase, the tested and evaluated functions.

/src/oraInTraBase/intrabase.* - Ported, but untested and validated functions.

/src/oraInTraBase/Oracle*n*.pkb – Files representing the changes needed to the source coode to make improvement version 4, 5, and 6 in Chapter 8.

/src/pgInTraBase/ - contains the source code for pgInTraBase that has been used in this Thesis most of it has been ported.

/tracefile/ - contains 10MB, 50MB, and 100MB tracefiles in both BT and MT version. These are the same trace files that was used for testing in the thesis.

APPENDIX C– TRACE FILE

In this Appendix we explain in details how to read and interpret the trace Oracle 10g can create. It assumes that the reader have read Section 7.3, and follows naturally from that section.

Header

In this section we go through an example trace file.

```
/u01/app/oracle/admin/orcl/udump/orcl_ora_17509.trc
Oracle Database 10g Enterprise Edition Release
10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options
ORACLE_HOME = /u01/app/oracle/product/10.1.0/db_1
System name:    Linux
Node name:      akurei.froisland.no
Release:        2.6.9-22.0.2.ELsmp
Version:        #1 SMP Thu Jan 5 17:13:01 EST 2006
Machine:        i686
Instance name: orcl
Redo thread mounted by this instance: 1
Oracle process number: 15
Unix process pid: 17509, image:
oracle@akurei.froisland.no (TNS V1-V3)

*** ACTION NAME:() 2007-02-07 10:57:44.463
*** MODULE NAME:(SQL*Plus) 2007-02-07 10:57:44.463
*** SERVICE NAME:(SYS$USERS) 2007-02-07 10:57:44.463
*** SESSION ID:(159.10) 2007-02-07 10:57:44.463
Memory Notification: Library Cache Object loaded into
SGA
Heap size 2237K exceeds notification threshold (2048K)
LIBRARY OBJECT HANDLE: handle=35f28e0c
mutex=0x35f28ec0(0)
name=XDB.XDbD/PLZ01TcHgNAgAIIegtw==
hash=e0f82b0c545a707da230a62675d34c80 timestamp=06-30-
2005 19:29:14
namespace=XDBS flags=KGHP/TIM/SML/[02000000]
kkkk-dddd-llll=0000-0000-0000 lock=S pin=S latch#=2
hpc=0002 hlc=0002
lwt=0x35f28e68[0x35f28e68,0x35f28e68]
ltm=0x35f28e70[0x35f28e70,0x35f28e70]
pwt=0x35f28e4c[0x35f28e4c,0x35f28e4c]
ptm=0x35f28e54[0x35f28e54,0x35f28e54]
```

```
ref=0x35f28e88[0x35f28e88,0x35f28e88]
lnd=0x35f28e94[0x35f28e94,0x35f28e94]
  LIBRARY OBJECT: object=35f28994
  type=XDBS flags=EXS/LOC[0005] pflags=[0000]
status=VALD load=X
  DATA BLOCKS:
  data#     heap  pointer    status pins change whr
alloc(K)  size(K)
  ----- -------- -------- --------- ---- ------ ---
-------- --------
      0 35f28d9c 35f28a50 I/P/A/-/-    0 NONE   00
0.34     0.00
      1 35f28a8c 35eda540 I/P/A/-/-    1 NONE   00
```

The first thing that is included in a trace file is a header like the one shown above. It includes various useful information about the system that the trace was performed on. This is done with SET_MODULE and SET_ACTION. Setting these values can be helpful when one needs to identifying the right trace and action.

PARSING IN CURSOR

Some lines have been omitted between the header and the the following statement for clarity.

```
PARSING IN CURSOR #19 len=17 dep=0 uid=63 oct=3 lid=63
tim=1143400682278143 hv=706604910 ad='36bce1ec'
SELECT * FROM tbl
END OF STMT
```

The number following # is the cursor identification, that will be used later to identify which statement motivated a certain action. In this case the cursor is identified as #19. The query is 17 characters long, as expressed by len=17. This query is not a sub query motivated by a parent query. It has dep=0. If dep was >0 it would have been a subquery. Uid is the schema user ID of the user who parsed the statement. oct is the oracle command type. Lid is the privileged user id. In this case the same as the uid. This is not always the case. If user 60 creates a package and give user 63 execution privilege on that package, statements inside this package will have uid=63 and lid=60 when user 63 use the package. tim is time identification, on some systems time = gettimeofday, on other system this is not the case [20]. hv is the statements id. This is not unique, but it is seldom repeated. ad is the library cache address of the cursor.

Database Calls

```
PARSE
#19:c=49993,e=93903,p=7,cr=78,cu=0,mis=1,r=0,dep=0,og=1,
tim=1143400682278130
BINDS #19:
```

```
EXEC
#19:c=0,e=84,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1143
400682278355
<3 WAIT events omitted, see next section>
FETCH
#19:c=0,e=11867,p=6,cr=4,cu=0,mis=0,r=1,dep=0,og=1,tim=1
143400682290345
```

The database calls have been grouped together here because they have the same attributes. This section follows directly after the PARSING IN CURSOR section in the trace file.

The first part is the type of operation. Parse and exec is only done once for each query, while there can be many wait and fetch. Next is the identification #19, which means these operations were all motivated by the query that was parsed in as cursor number 19 in the previous section. Next follows the CPU time spent on the operation, followed by the time spent on this operation. The e value includes the c value. p is the number of Oracle database blocks that where read for the operation. This is not all physical reads[14] as they might have been obtained from a cache. cr is blocks read in consistent mode, while cu is blocks read in current mode. Together they constitute LIO which is memory reads. mis is the number of library cache misses. Each miss motivates a hard parse operation [20]. A hard parse means that Oracle creates a new execution plan as opposed to soft parse were Oracle reuse a execution plan from memory. The number of rows returned by the operation is expressed in r. In our example the fetch operation returns 1 row. The value of og refers to the table bellow:

Oracle query optimizer goal by og value (source Oracle MetaLink note 39817.1)[20]

1	ALL_ROWS
2	FIRST_ROWS
3	RULE
4	CHOSE

Finally tim is the same as for the PARSE IN CURSOR, namely the time reference.

Wait Events

The DBMS will in many situations have to wait for other operations. For instance reading from disk. These periods are called wait events. Their name give an indication of what the DBMS had to wait. Looking at wait events one can tell what the DBMS waited for. They can be a good indication of which part of the system needs tuning.

```
WAIT #19: nam='SQL*Net message to client' ela= 4 driver
id=1650815232 #bytes=1 p3=0 obj#=335
```

14 A read is considered logical if it was read from memory, that means the value was cached.

```
tim=1143400682278416
WAIT #19: nam='db file sequential read' ela= 10341
file#=4 block#=963 blocks=1 obj#=72085
tim=1143400682289019
WAIT #19: nam='db file scattered read' ela= 950 file#=4
block#=964 blocks=5 obj#=72085 tim=1143400682290203
```

These are the wait events that was removed from the precious section to make it easier to read. Wait event is only shown in the trace file if the level was set to 8 or 12 [20]. As in the previously explained parts, #19 refers to the cursor number. nam is the name assigned to the operation to reveal what the kernel is doing in this time. ela is the time spent waiting for this operation to complete. The three next attributes vary depending on the wait event. They can be listed with this query:

```
SELECT name, parameter1, parameter2, parameter3
FROM V$EVENT_NAME
```

Bind Variables

If level 4 or 12 trace is enabled these events are added to the trace. Our example did not contain any bind variables. This example is from a different query using bind variables:

```
BINDS #18:
kkscoacd
 Bind#0
  oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
  oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
  kxsbbbfp=b726e230  bln=22  avl=04  flg=05
  value=72085
 Bind#1
  oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
  oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
  kxsbbbfp=b726e20c  bln=24  avl=02  flg=05
  value=1
```

This example illustrate that there is more than one variable to be bound into this statement. The first parameter oacdty is the data type of the parameter.

Row Source Operations

When trace is activated you get summary data when the cursor is closed. For our example the summary line looks like this:

```
STAT #19 id=1 cnt=109879 pid=0 pos=1 obj=72085 op='TABLE
ACCESS FULL MF (cr=8214 pr=955 pw=0 time=561423 us)'
```

Between this line and the fetch statement in the section about Database calls there is more than 22.000 lines with wait and fetch events, they have for

obvious reasons been omitted.

The summary line identify which cursor it is valid for, #19. cnt is the number of rows that this statement returned, and is the same value the sqlplus prints at the end of the query

```
select * from tbl;
<output omitted>
109879 rows selected.
SQL>
```

pid is the operations parent id. If our query had been a query with dep > 0 pid would have shown which operation motivated this. 'obj: Object ID of the row source operation, if the operation executes upon a “base object.”' [20]. op names the row source operation. pr and pw are the number of physical reads and physical writes respectively.

Order of Trace File and Final Notes on Trace Files

Note that for most operations the line is included in the trace file after that operation is completed. So, for instance, wait events are listed before the fetch operation that motivated them as these operation has to complete before the fetch can be completed. Also note that an operation includes the time of all its children. So a fetch that motivates 3 wait events will include the sum of all the wait events. This allows one to compute the time that is otherwise unaccounted for. There are operations in the kernel that does not print a line in the trace file for. By summing the wait events and comparing it to the motivating statements time one can compute unaccounted time. If you look at the three wait events that was motivated by the fetch in the example above you will notice that the time for the wait events are 4+10341+950=11295, while the fetch statement reports 11867. This gives us 572 that is unaccounted for. Some of this unaccounted time may be from rounding mistakes, some from events that are unaccounted for. [20] claims that they seldom experienced more than about 15% of unaccounted time. In our case it was about 5%. Note that the values are expressed in microseconds from version 9i.

APPENDIX D - RULE BASED OPTIMIZER AND COST BASED OPTIMIZER

The Rule Based Optimizer (RBO) uses a set of rules to decide how to execute a query. The user is given considerable freedom in deciding how a query will be executed. He can, for example manipulate the query in order to gain a specific result or use hints for the optimiser. For instance the order of the tables in the *from* statement has significance. [21] has an example where the runtime was reduced from more than 19 seconds to less than 2 seconds for a certain query only by changing the order of the two tables in the *from* statement.

The Cost Based Optimiser (CBO) uses statistical data to decide how to execute a query. In Oracle, one can gather statistics either through ANALYZE or through the DBMS_STATS package. If there is no statistical data available, the CBO optimiser will generate on the fly statistics. This is a very slow operation.

There has been made significant advancements in the CBO through the last releases of Oracle, the same is not true for RBO. A major feature that was made available with Oracle 9i is that *bind variables* were processed before the optimizer. An example illustrates it:

```
SELECT *
FROM some_table
WHERE some_column = :var1
```

Some_table contained 1 million tuples with some_column = Q and only a few tuples with some_column = P. With statistics, the optimiser would chose to perform a full table scan if var1 = Q and use an index if var1 = P's. When using *bind variables,* the optimiser first decides on an execution plan and then evaluates the bind variable. In our example, the optimiser does not know the value of var1 and can therefore not tell whether var1 = Q or var1 = P. In the releases after 9i, the bind variables value is taken into account by the optimiser before selecting an execution plan and it would now chose correctly for the var1 = P.

The main disadvantage with CBO is that it needs statistical data to work properly. It will have severe performance impact if CBO is used on tables or indexes where no statistics have been gathered. There are also several considerations about how and when to gather statistics. For instance, temporary tables used by batch jobs can cause problems in that they are always empty when the job is not running, they might never be committed containing any data. In such cases, one should populate the table with the data that is representative for the batch job, then commit, analyse and truncate the table. The statistics will not be removed when a table is truncated. If you create statistics while the table is populated and then truncate you can make statistics for a temporary table.

APPENDIX E - GATHERING OPTIMISER STATISTICS

There are two different ways of analysing a table to gather statistics. The first one visits all the tuples of the table to gather accurate statistical data. This is the command to start the collection:

```
ANALYZE some_table COMPUTE STATISTICS FOR ALL INDEXED
COLUMNS;
```

Estimating the statistics is recommended for tables with more than 1.000.000 tuples [21]. The reason is that gathering complete and accurate statistics can consume a significant amount of resources. While looking at merely 5% of the tuples will in most cases force the optimizer into making the same plans as it does with a plane made using 100 computed statistics.

```
ANALYZE some_table ESTIMATE STATISTICS SAMPLE 5 PERCENT
FOR ALL INDEXED COLUMNS;
```

As important as having statistics is having the correct statistics. To get god statistics they must be gathered when the table has both a representable amount and type of data. Otherwise, the statistics is of no use, as it may lead to inefficient execution plans. The following query tells when a table was last analysed and how many rows it had at that time.

```
SELECT table_name, num_rows, last_analyzed
FROM user_tables;
```

If one have queries that was tuned, gathering new statistics can potentially reduce performance. This may happen if the new statistics is not equally representative or if the optimising is based on some “feature” of the old statistics. Therefore, one should always make a backup of the old statistics before analysing again. This can be done with the DBMS_STATS.EXPORT_SCHEMA_STATS function.