In Quest of the Bottleneck — Monitoring Parallel Database Systems*

Stefan Manegold[1] Florian Waas[2] Daniel Gudlat[1]

[1]Humboldt-Universität zu Berlin
Institut für Informatik
10099 Berlin, Germany
⟨lastname⟩@dbis.informatik.hu-berlin.de

[2]CWI, P.O.Box 94079
1090 GB Amsterdam
The Netherlands
flw@cwi.nl

Abstract. Monitoring query processing has proven to be an effective technique to detect bottlenecks in sequential query execution systems' components. Monitoring distributed execution in parallel systems, however, is a difficult task. The monitoring data of all nodes must be collected at the same time to value load-balancing and scheduling. Furthermore, we want to extract not only information about resource allocation of single processes but also allow for monitoring of single tuples and attribute value distribution. The monitoring system we present in this paper is a client-server architecture where each process of the parallel database system is a client of the monitoring server. This architecture enables the user to monitor even load-balancing or scheduling effects. The monitoring extensions, each client is attached with, provide inspection of the processes at a fine granularity. Furthermore user-defined interpretation of monitoring data on the client side enables emulation of special parallel hardware by low costly components.

Keywords: parallel and distributed database systems, performance analysis

1 Introduction

Performance evaluation of *database management systems (DBMS)* is an important but difficult task [5, 1]. Benchmark suites as a straightforward solution became not only the standard for performance analysis but also a striking argument of vendors [3]. However, benchmarks try to map an entire DBMS to a few figures. For sure, those results reflect the behavior of the DBMS—i.e. the collaboration of all components—with regard to particular series of composite operations, such as queries including complex aggregations. On the other hand, single building blocks of the DBMS are not valued on their own and therefore, bottlenecks are not—or not definitely—to identify. Thus, DBMS-developers need tools to inspect single components separately.

Because of the strong coherence between the DBMS and the underlying operating system, it is impossible to apply mathematical models for analyzing a

* supported by the German Research Council under contract DFG Fr 1142/1-1.

DBMS. In contrast to that, monitoring the DBMS by collecting data about resource consumption periodically during running time is a very powerful technique to detect performance lacks effectively.

Monitors are specializations of profiling tools that are used in software development. They produce dump logs for all performance relevant operations—usually on a higher degree of abstraction than in pure profiling. These logs are analyzed afterwards. Hereby, graphical user interfaces facilitate this process, in particular when analyzing concurrent processes like multi-threaded DBMS components. Due to their effectiveness, monitors for sequential DBMSs have already setup in business.

To meet performance requirements of modern database-applications DBMS development has to go in direction of parallel hardware environment. In contrast to sequential DBMS, all resources may exist in parallel in *Parallel DBMS (PDBMS)* [2, 4]. According to the combination of the particular hardware configuration, systems can be classified by the degree of sharing the resources that are attached to the single CPUs. There is no clear separation and most systems combine characteristics of different classes hierarchically [6]. Obviously, with increasing complexity and distribution of system components, the overhead for controlling increases, quickly. Hence, development of PDBMS is not only a matter of parallelizing algorithms but rather of load-balancing and scheduling—including static data distribution—, as the processes implement simple but very data intensive jobs.

In this paper we propose a client-server-monitor based on PVM that serves as a tool for detailed performance analysis of a PDBMS in an arbitrary parallel hardware environment.

The main differences to sequential monitors hereby are described in the following. (1) to value load-balancing all monitoring data is to be seen with regard to a global wall-clock, i.e. monitoring data is collected from all processes at the same time, if possible. The client-server-architecture helps to achieve this: all processes of the PDBMS are clients of the monitor, that plays the server role (practically on a separate host that is not involved in the PDBMS, directly). The inspection of resource allocation and utilization is initiated by the monitor via distributed signals, periodically. Furthermore, this architecture guarantees that all processes of the PDBMS are charged the same way by the monitor independent of their particular equipment, as they do not have to dump data on local resources. Furthermore, this architecture assures that data is only collected on demand and does not have to be stored intermediately. (2) The investigation of new techniques to utilize special hardware environment is a very costly task. Therefore emulations by available components are preferred, e.g. shared disks are often emulated by distributed disks, coupled via NFS. These technique requires a separate interpretation of the monitoring data on the client side, already. (3) Since, execution depends on the underlying data, stored in the database, query execution must always be seen in relation to specific properties of the data set. Thus, we integrated also abilities for *logic monitoring*, i.e. information that is not reflected in resource allocation or usage, directly, can be traced and extracted

by the monitor, also. For instance, number and attribute value distribution of single data tuples can be examined.

Finally, we equipped the monitor server with a Java interface and a graphical interface to display the monitored data on-line if desired. As this monitor is based on PVM it is highly portable to all platforms that are supported by PVM.

Road-map. In Section 2 we briefly outline the principles of query processing and PDBMSs. The design of the monitor system is presented in Section 3. Section 4 contains a demo session of our system. Finally, Section 5 gives a conclusion and an outlook to ongoing and future work.

2 Query Processing in Parallel Database Systems

In this section we briefly outline the principles of query processing in PDBMS. For the remainder of the paper we focus on read-only queries, i.e. queries that perform no update on the database, as they are of particular interest. The software that implements the query processing is referred to as query-engine. The query engine we report on works on top of a relational database, i.e. data is stored in tables consisting of rows—also called tuples—of a certain structure that is given by the columns of the table definition[3]

A query over the database is stated as a term consisting of relational algebraic operators. Typical representatives of operators are for instance the selection and the projection operator, i.e. filter and restructuring of tables. However, the most costly operators—and therefore the most interesting candidates for parallelization—are joins and aggregation operations. Joins combine different tables and produce output tables of a new structure with size $O(N^2)$ where N is the size of the input tables. Aggregation operators perform computations where data of a whole table has to be collected first before any output can be produced, e.g. computation of the median of one column. To achieve efficient processing, the computation of (intermediate) result tables should, of course, be done in physical main memory. Since, even the stored initial tables may be too large to fit into memory, the computation of for example joins often requires sophisticated swapping techniques for both the scanning of the initial/input tables and the generation of the output table—especially in this point, database implementations are very different from other applications.

Thus, important figures to monitor are memory allocation during the processing of space consuming operators on one hand, and I/O behavior for scanning initial tables as well as for storing/swapping intermediate results that do not fit into memory. Of course, depending on the particular implementation of the query engine further information that can be received by system functionality

[3] All further considerations and implementation techniques can be transferred easily to other data-models like the object-oriented or the object-relational data-model. However, for simplicity we only mention the relational approach.

is of interest[4]. Besides this, a finer granularity for inspection is needed, e.g. to trace single tuples during processing or to check attribute value distributions and data skew, etc.

3 The Monitor Architecture

In this section, we describe the architecture of our monitor as well as it's implementation.

3.1 Overview

The idea of our monitor is as follows: Each process of the PDBMS, that is to be monitored, locally collects information needed for monitoring its execution. This information consists of a number of counters, each representing a certain task or event. Thus, collecting the information means incrementing the appropriate counter whenever a task or event occurs. Due to this concept, it is possible to monitor resource utilization and requirements—e.g. I/O request, memory allocation, etc.—as well as function calls or the amount of data, that is processed by a certain process or a single function.

Representing the desired information as counters has several advantages. First, the amount of data needed to represent the information is rather small. Further, the action to collect the data is rather inexpensive. Thus, the additional resource requirements (in terms of memory and CPU time) needed for monitoring are kept low and therefore do not affect the execution of the PDBMS processes significantly.

The PDBMS processes act as clients of the monitor itself, i.e. the centralized server within our architecture. To collect the data from the clients, the monitor periodically requests the data. On each request, the clients send the current values of their counters to the server. In addition to this data, each client requests the information on its resource utilization by calling the corresponding system functions and sends this data to the server, too. The server adds a timestamp to the collected data of all clients and saves it to a local log file that can be analyzed off-line, i.e. after the bench has finished.

Using a centralized monitor-server as described before has several advantages. First of all, the data of the clients is collected on demand. This guarantees, that the server is able to receive and store or analyze the data as soon as the client has sent it. Second, the data of different clients (running on different hosts) is requested with regard to a global wall-clock, i.e. the system clock of the server-host, even if the system clocks of the client-hosts are not synchronized. Of course, the request may arrive at the clients at slightly different times. But, as our experiments showed, the difference is less than 10ms. This accuracy is sufficient within our environment. Further, collecting and analyzing the monitoring information on-line (see below) does not add any additional overhead to

[4] UNIX systems offer a broad variety of information about resource utilization by the `getrusage()` library function.

the clients but getting the resource utilization information and sending the data to the server on each request. Getting the current timestamp, saving the data to secondary storage and probably performing on-line analyzation is done by the server that runs on a separate host. All this means, that our architecture meets the special requirement of monitoring parallel systems.

In addition to a simple commandline interface, we added a JAVA interface to the monitor. This graphical interface provides the possibility to display the monitoring information, either off-line, reading the data from a log file, or on-line while collecting the data.

3.2 Implementation

According to the architecture described above, our monitor consists of four modules: a client library, a server library, a commandline interface and a graphical interface.

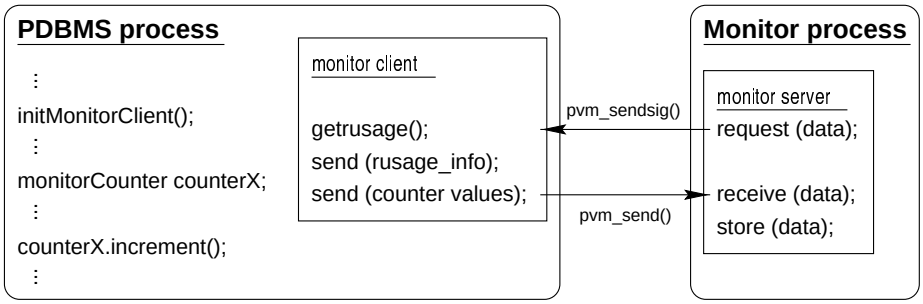

Fig. 1. The Model

The client library provides functions to initialize the client, to create and increment counters and the signal handler to answer a request from the server. To add the client facilities to a given PDBMS process, the statements to initialize the clients as well as those to create and increments the counters must be added to the code of the PDBMS process, and the PDBMS process has to be linked with the client library. The server library provides functions to initialize the server as well as to request the clients and receive their answers.

Communication between the server and the clients is done via PVM. All clients are gathered within one PVM-group. On each request, the server sends a signal to all members of this group via PVM. Then, the client functions, implemented as a signal handler, send the monitoring data to the server, also using PVM. Figure 1 depicts the main components of the monitor. In addition to the functionality needed for client-server-communication, the use of PVM provides the possibility to port the monitor to any platform that PVM is available on.

4 Examples

To demonstrate the functionality of the monitor, we look at the execution of a single join with a complex join predicate. The scenario is as follows: One workstation (*source*) with a local network produces one input relation of the join. Three other workstation (*worker*) evaluate the join in parallel, reading the other input relation from a shared disk, attached to worker 3. Worker 1 and worker 2 can access this disk via NFS. We use a hash function applied to the join attribute, to distribute the tuples that are produced by the source, to the workers. A fifth workstation (*drain*) stores the result of the join to a local disk. The communication and data exchange between the source and the worker as well as between the workers and the drain is done via PVM.

To monitor the evaluation process, each PDBMS process (source, workers, drain) maintains a counter that is incremented whenever the respective process has produced one output tuple. The source maintains three counters, to distinguish to which worker each tuple is sent. The following pictures show plots that we produced from the monitor log file using gnuplot.

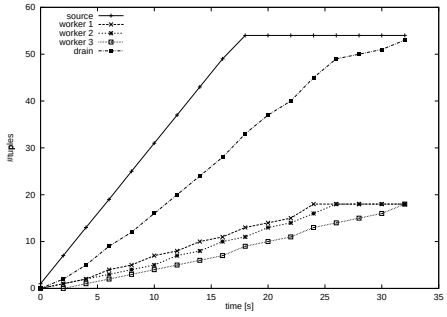

Fig. 2. Number of processed tuples

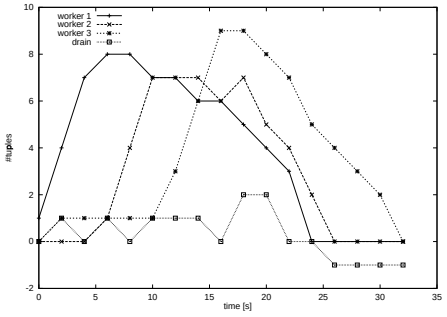

Fig. 3. Number of tuples in receive buffers

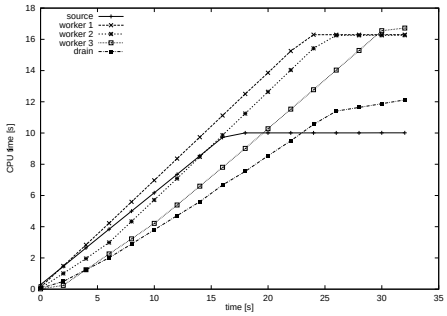

Fig. 4. CPU time

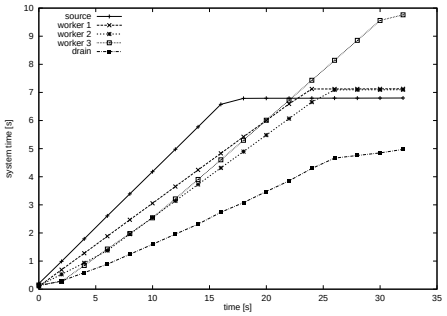

Fig. 5. System time

Figure 2 shows the number of tuples that are processed by each process. As we can see, each worker processes the same number of tuples, i.e. the hash

function that is used to distribute the tuple to the worker seems to be rather good.

But, the next figure shows some shortcomings of the hash function. In Figure 3, for each worker and for the drain, the difference of the respective counter and the counter of the respective producer is plotted. This difference gives the number of tuples that is buffered (by PVM) between each producer-consumer pair. As we can see, at the beginning, the source produces more tuples for worker 1 than for the other workers. Then, worker 2 gets more tuples than the others, and at the end, worker 3 is the one that gets the most input tuples.

Figure 4 shows the CPU time used by each process. It shows, that the workstations of the three workers provide the same CPU performance: they need the same CPU time to process the same number of tuples. This figure also shows, that—despite parallelization—the join is still the bottleneck of this query.

Finally, Figure 5 depicts that worker 3 needs more system time that the other workers.

5 Conclusion

In this paper, we presented an extensible client-server-monitor, based on PVM, for detailed performance analysis of PDBMSs. It supports database-developer by analyzing the query engine and detecting bottlenecks in the parallel execution system.

The client-server-architecture provides the possibility to get detailed (logical and physical) information on the execution of each PDBMS process with out disturbing the execution of the PDBMS processes significantly. In our experience, a combination of physical (i.e. resource usage) and logical (e.g. number of data items that are processed) monitoring is necessary to understand the evaluation and to locate performance bottlenecks. Although, the system was originally designed to analyze our query engine on a Cray T3E SMP, it is easy to port to both arbitrary parallel environments that are supported by PVM and various parallel or distributed query engines.

Related Work. Nearly all UNIX-like operating systems come with either a complete monitoring tool or at least provide library functions that support profiling of arbitrary processes with a detailed granularity. Since, these profiling capabilities are restricted to physical monitoring only, they are not suitable for analyzing and tuning parallel database systems, as we pointed out in this paper.

Future Work. In our current system, the monitor is used as an external tool to support developing and tuning. For the future, however, we plan to integrate the monitor completely into our parallel query evaluation system, in order to use the monitoring data for dynamic optimization, i.e., to use the monitoring data for scheduling and load-balancing of incoming queries.

References

1. D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6), June 1992.
2. J. Gray. A Survey of Parallel Database Techniques and Systems. In *Tutorial Handouts of the 21st Int'l. Conf. on Very Large Data Bases*, Zurich, Switzerland, September 1995.
3. Jim Gray, editor. *The Benchmark Handbook*. Morgan Kaufmann, San Mateo, CA, USA, 2 edition, 1993.
4. H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, and P. Selinger. Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. In *Proc. Int'l. Symp. on Databases in Parallel and Distr. Systems*, Dublin, Ireland, July 1990.
5. D. A. Schneider and D. J DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proc. ACM SIGMOD Int'l. Conf.*, May 1989.
6. P. Valduriez. Parallel Database Systems: The Case for Shared-Something (Keynote Address). In *Proc. IEEE Int'l. Conf. on Data Eng.*, Vienna, Austria, April 1993.

This article was processed using the LaTeX macro package with LLNCS style