PPOST: A Parallel Database in Main Memory

László Böszörményi, Johann Eder, Carsten Weich

Institut für Informatik, Universität Klagenfurt
Universitätsstr. 65, A-9020 Klagenfurt, Austria
e-mail: {laszlo,eder,carsten}@ifi.uni-klu.ac.at

Abstract We present the PPOST-architecture (Persistent Parallel Object Store) for main-memory database systems on parallel computers, that is suited for applications with challenging performance requirements. The architecture takes full advantage of parallelism, large main memories and fast switching networks. An important property of this architecture is its excellent scaling behavior.

Keywords parallel database system, main-memory database system, object oriented database system, object store, database architecture.

1 Introduction

New advances in hardware and systems software demand to revisit design criteria for database management systems. Some well known obstacles against main memory databases (main memory is too small, too expensive and does not scale up) are no longer valid. With (highly) parallel systems made of powerful commodity processors and fast switching networks main memory database systems managing Gigabytes to Terabytes of data can easily be envisioned. While most database vendors take advantage of these developments by extending their (disc-oriented) DBMS with better management of large buffer areas or porting their DBMS to parallel hardware, we take the other approach. Data should reside primarily in main memory (where it can be retrieved and processed very efficiently) and is brought to secondary storage only for the sake of safety and recoverability. Furthermore, the trend to object oriented databases (or extended relational databases) requires that database systems have not only to deal with storing and retrieving data but also with processing user defined data manipulation methods on that data. Of course, this processing is performed much more effective, if data resides in main memory.

In the PPOST architecture parallelism is employed in two ways. With vertical parallelism we delegate processes for logging, checkpointing and archiving to own processors such they do not influence the performance of user operations. With horizontal parallelism we can spread the objects managed by the database across several (maybe many) processors for speeding up the processing of queries or methods and for increasing the size of the databases.

2 The Architecture of PPOST

PPOST's main components are (figure 1): *object store* (consisting of a number of object storage machines), *log machine*, *checkpoint machine*, *archive machine*

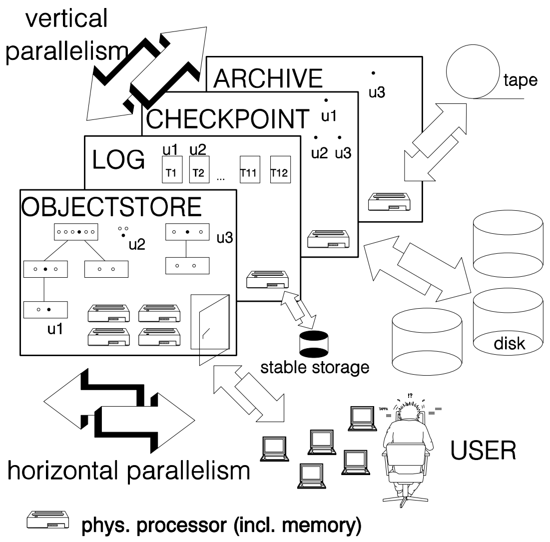

Fig. 1. The components of PPOST

and *users* (consisting of a number of user machines). All the data of the stored objects (i. e. their attributes and methods) lie in the memory of the storage machines. Every transaction that reads or changes the data is executed on those machines. PPOST is transaction-oriented. Transactions are initiated by the user machines and processed by the object store. Changes of the data in the object store are reported to the log machine which saves the information onto a logfile in nonvolatile memory.

The checkpoint machine reads the log produced by the log machine and saves all commited changes to the disc-based database. Only the checkpoint and the log machine is involved in producing the disc image. The user transaction can go on as soon as the information about the changes is transmitted to the log machine.

The archive machine saves the disc-database to a secondary storage, like a magnetic tape. This is considered as a normal activity of the data-store and again is done in background without interrupting the user-transactions.

We call this pipeline-like way to decouple user-transactions from issues of persistence *vertical parallelism*.

3 Transactions

When data is changed in the object store, log information is produced and sent to the log machine. The log machine would ideally store the log-tail in stable main memory. In this case, transactions whose log information arrived in the log machine can be committed immediately. We do not insist, however, on the existence of a stable main memory. In the lack of this, we precommit [5, 7] the corresponding transactions and let run other transactions (locks are released). In the meantime, the log information is stored on disc in the form of simple sequential files (this can be done at full disc-speed). After that, precommitted transactions may be committed. In case of a system crash precommitted transactions are handled as not-yet-committed.

3.1 Parallel checkpointing and archiving

The task of the checkpoint machine is to apply the logs on the last valid disc image [6]. After processing a certain amount of log information, a new image is created, and the corresponding log files are deleted. Checkpointing is done by a separate machine, therefore its speed has no influence on the response time of the transactions. If the database is more or less quiescent, the disc image may come very close to the primary copy. During heavy load the backup might become relatively “old” and the log files become long. This is unlikely, however, because a database rarely has a constant heavy load over a long period of time (i. e. days). The newest image generated by the checkpointer can be archived on additional nonvolatile storage (such as tapes).

3.2 Transaction undo

For transaction undo we use before-images or shadow copies in volatile storage. In the case of a system crash, the primary copy of the database in memory is lost anyway. All not-yet-committed transactions are trivially “undone”. Therefore, transaction undo is in accordance with the concept of a memory-resident database.

3.3 Recovery

In the case of a system crash, a recovery must be executed. The image of the disc based database is loaded in main memory and the log is applied on it. Note that in this case the actual memory image is generated with "memory speed" (instead of "disc-speed", as in the case of checkpointing).

4 Scalability

If we have more than one physical storage machine, we can use *horizontal parallelism* either to speed up operations or to scale up the size of the database without loosing performance: When we spread the objects that are processed by an operation over several nodes of the object store (we call this *data distribution*) then each node can process its part of the set independently of the others. Operations like selecting certain objects or starting a method of a certain set of objects can be done in parallel: The only condition is that the set of objects to be distributed has to be large enough, such that the enhanced speed gained by parallelism can make up for the time needed for communication – otherwise we would lose performance.

When the databases increases in size, we add nodes to the object store. This means, we not only add storage capacity but also computational power. We can show that in many cases it is possible to scale up the size of an object set without degrading the performance of a certain operation on that set by adding nodes [1].

On the other hand we can add nodes to a data distribution to enhance performance. It is possible to calculate the optimal number of nodes with which an operation runs fastest. It is not possible though to keep this optimized speed when the size of the distributed object set grows. Then a new optimal distribution has to be calculated which will be slower in most cases.

Example Our prototype installation consists of very fast processors but comparatively long network latency times (12 DEC/ALPHA OSF1 workstations connected by a FDDI net). For a simple selection-operation distributed among 2 nodes we need 2,000 objects in the set to equal the performance of the same operation with all objects on one node. If the set is smaller than 2,000 objects the parallel operation will become slower than the sequential. If the set is much larger than that we can achieve nearly linear speedup when we add nodes to the distribution. With a setsize of 500,000 objects we reach a speedup of factor 7 with 8 nodes for the selection operation. If we add even more nodes, the additional speedup gets poorer: 10 with 14 nodes for instance. It reaches a maximum of 11.2 with 22 nodes – adding more than 22 nodes will lead to a less than optimal performance [1].

5 Conclusions

We have presented the architecture of PPOST and demonstrated that parallelism can overcome the limitations of memory resident database systems. The horizontal extendability together with little performance penalties are very desired features. PPOST will be the implementation platform for an object oriented database system supporting views [2]. In particular object oriented databases can take advantage of the proposed architecture because it facilitates the integration of databases and programming languages. Major design issues like pointer swizzling strategies become less crucial since the disc is accessed only in the background and all conversions between internal and external format do not slow down user processes. This promises a great performance gain.

The application areas of PPOST are those with high performance requirements. Currently we analyze how PPOST can be integrated with disc-based DBMS such that PPOST will be responsible for the *hot* data while the disc based DBMS manages *cold* data. One of the approaches we investigate is to use a standard DBMS as backup database which contains a (probably through the logging process delayed) image of the main memory database in form of a replication.

References

1. L. Böszörményi, K. H. Eder, C. Weich, *PPost – A Persistent Parallel Object Store*, to appear in the Proceedings of the International Conference Massively Parallel Processing Applications and Development, Delft 1994.
2. M. Dobrovnik, J. Eder, *A Concept of Type Derivation for Object-Oriented Database Systems*, Proceedings of the Eight International Symposium on Computer and Information Sciences (ISCIS VIII), Istanbul 1993.
3. P. Apers, C. van den Berg et. al., *PRISMA/DB: A Parallel, Main Memory Relational DBMS*, IEEE Transactions On Knowledge And Data Engineering, Vol. 4, No. 6, December 1992.
4. H. Garcia-Molina, K. Salem, *Main Memory Database Systems: An Overview*, IEEE Transactions On Knowledge And Data Engineering, Vol. 4, No. 6, December 1992.
5. H. Garcia-Molina, K. Salem, *System M: A Transaction Processing Testbed for Memory Resident Databases*, IEEE Transactions On Knowledge And Data Engineering, Vol. 1, No. 2, March 1990.
6. H. Garcia-Molina, K. Salem, *Checkpointing Memory Resident Databases*, International Conference On Data Engineering, Los Angeles 1989.
7. J. Gray, A. Reuter, *Transaction Processing - Concepts and Techniques*, Morgan Kaufmann Publishers Inc, 1993

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