A Simple Approach to Shared Storage Database Servers*

Luís Soares
Computer Science and Technology Center
University of Minho
los@di.uminho.pt

José Pereira
Computer Science and Technology Center
University of Minho
jop@di.uminho.pt

ABSTRACT

This paper introduces a generic technique to obtain a shared-storage database cluster from an off-the-shelf database management system, without needing to heavily refactor server software to deal with distributed locking, buffer invalidation, and recovery from partial cluster failure. Instead, the core of the proposal is the combination of a replication protocol and a surprisingly simple modification to the common copy-on-write logical volume management technique: One of the servers is allowed to skip copy-on-write and directly update the original backing store. This makes it possible to use any shared-nothing database server software in a shared or partially shared storage configuration, thus allowing large cluster configurations with a small number of copies of data.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases

General Terms

Reliability, Performance

Keywords

Shared-storage clusters; database replication.

1. INTRODUCTION

Provisioning a clustered server for a database management system must consider both performance and dependability. On one hand, one must ensure enough nodes to provide the required CPU bandwidth, while at the same time configuring disks in parallel, often in a stripping configuration, to handle the required storage bandwidth. For instance, the backend for popular web sites requires hundreds of servers to deliver the needed processing bandwidth. On the other hand, one must ensure that several physical copies of data exist to cope with disk failure. This ranges from a couple copies in a mirroring RAID configuration to a few copies in data centeres scattered in different continents for disaster recovery.

A shared-storage cluster such as Oracle RAC [7] allows maximum configuration flexibility: One uses as many nodes as required for processing the workload and to ensure the desired availability, while the storage is configured solely according to the desired storage bandwidth and disk resilience. Unfortunately, a shared-storage approach based on distributed shared memory and distributed locking raises a number of problems, which make such solutions costly to develop and deploy. Anedoctal evidence for this is that none of the mainstream open source database servers provide this option. Most commercial database servers also lack a shared-storage configuration.

In contrast, there have been a number of proposals for shared-nothing database server clusters based on consistent replication [11, 6, 2, 4, 5]. All these share the same basic approach: Updates are ordered and propagated before replying back to the client, thus ensuring that no conflicts arise after the transaction commits. The resulting performance and scalability is very good, especially, with currently common mostly read-only workloads.

The main problem is that in a shared-nothing cluster a separate physical copy of data is required for each node. Therefore, even if a only few copies are required for dependability, a large cluster with hundreds of nodes must be configured also with sufficient storage capacity for hundreds of copies of data. The goal of the approach outlined in this paper is this to combine the configuration flexibility provided by shared-storage clusters with the scalability and ease of implementation of shared-nothing clusters.

It might look simple at first sight to extend the shared-nothing protocol to cope with shared storage: If all replicas perform exactly the same write operations, database state would be identical and thus could be shared. A designated writer could then be elected to update the shared copy. Unfortunately, this simple approach rests on an extremely strong assumption: Data layout is deterministically set from commit order. In fact, even if the replication mechanism is logically a replicated state machine, hence deterministic, the mapping from logical to physical layer is not. For instance, even if different replicas deterministically add the same tuple to a table, it is likely that such tuple ends up being stored in different disk blocks due to internal concurrency.

In this paper we solve this problem with the introduc-

*This work was partially supported by project "PASTRAMY: Persistent and highly Available Software TRansactional MemorY" (PTDC/EIA/72405/2006).

WDDDM '09, March 31, 2009, Nuremberg, Germany

tion of a novel approach to shared-storage database server clusters that builds on a shared-nothing replicated database server. In contrast to previous shared-storage techniques, it can be obtained from shared-nothing replicated database server software without a profound refactoring of locking, buffer management, and recovery mechanisms. The required modification reduces to intercepting and redirecting file I/O system calls. The isolation and durability guarantees of the replicated DBMS are preserved. When compared to the baseline shared-nothing DBMS server, only one node writes to persistent storage to commit a transaction. Other nodes don't ever need to invoke sync, unless when taking over as designated writer after a failure.

The rest of the paper is structured as follows. In Section 2, we explain our base assumptions and clearly state the problem. Section 3 outlines the proposed approach, namely, how it deals with node failure. Section 4 concludes the paper, discussing work in progress to evaluate the resulting performance as well as potential applications of the proposed technique.

2. BACKGROUND

In this section we discuss our baseline assumptions on replicated DBMS server software and on shared-storage clusters. We conclude by clearly identifying why combining them is a hard problem.

2.1 Shared-Nothing Software

The first base assumption is the availability of replicated DBMS server software [11, 6, 2, 4, 5]. The usual resulting shared-nothing cluster that can be built with such software is depicted in Fig. 1(a). In detail, each node keeps a full copy of the the data in a separate non-shared volume. Update transactions are propagated to all replicas by a replication protocol. Update propagation can be performed within the DBMS server [11, 6, 5] or at the middleware level [2, 4]. Depending on the desired consistency, read-only queries can often be executed and answered by a single replica to improve performance [10].

The cluster is overseen by a cluster manager that coordinates replica failover and recovery with the replication protocol. Namely, when one replica comes back on-line it will catch up with the rest of the cluster. Such cluster manager can be centralized in one of the nodes [2] or distributed [11, 6, 4, 5]. The resulting consistency criterion is therefore defined by the combination of replication, load-balancing, and cluster management mechanisms. Note that the number of actual physical copies of data is lower bounded by the number of cluster nodes. Disk failure is taken care by leveraging such redundancy.

Although the proposed approach is compatible with such variations, to simplify presentation, the reader might want to assume a centralized replication, load-balancing, and cluster management mechanism such as C-JDBC [2] throughout the rest of the paper.

2.2 Shared-Storage Hardware

The second base assumption is the availability of shared-storage cluster hardware and an appropriate operating system. These would commonly be used to support a shared-storage DBMS such as Oracle RAC as depicted in Fig. 1(b). In this case, all nodes are attached to the storage enclosure using a storage area network (SAN). A shared volume can then be mounted by all cluster nodes, using a shared cluster file-system providing POSIX semantics such as OCFS2. A shared-storage DBMS will take advantage of this setup by using a distributed shared memory (DSM) mechanism for buffering and a distributed lock manager (DLM) for concurrency control [7], thus providing a single system image.

Again, the cluster is overseen by a cluster manager that coordinates replica failover and recovery with the buffer management and distributed locking mechanism. Since there is only a single logical copy of data, an appropriate RAID configuration must be used for redundancy. Note however that the number of physical copies is independent of the number of cluster nodes.

2.3 Problem Statement

The challenge is thus to make the DBMS software of Fig. 1(a) take advantage of the hardware in Fig. 1(b), thus sharing data and decoupling the number of physical copies from the number of nodes.

Unfortunately, if all replicas are naively configured with the same storage volume, data corruption ensues. Consider the following example: Concurrent transactions insert tuples a and b in the same table in different replicas. Assume also that the commit order, in both replicas, is the same. After updates are propagated, databases are logically consistent containing both a and b. At a physical level, one replica allocates space for a, before being propagated, and then space for b. The other replica does the same in the reverse order, ending up with b and then a in storage.

Avoiding this would be quite expensive, as it prevents concurrent processing of updates, as well as a number of other useful optimizations, such as background compaction of storage pages.

3. APPROACH

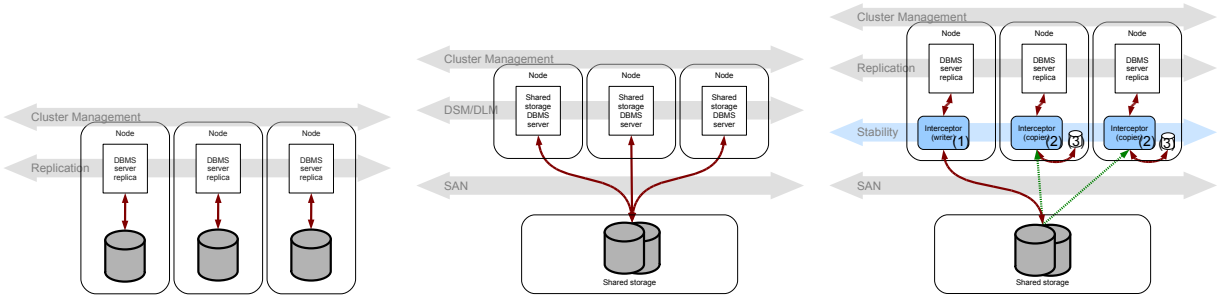
At the core of our approach is an additional layer underneath a replicated DBMS when running in a shared-storage cluster, as shown in Fig. 1(c). This layer intercepts all file I/O operations issued by the DBMS server and provides it the abstraction of multiple non-shared physical copies below. The net result is a shared-storage cluster with minimal changes to existing DBMS software, where a single copy of the data, indistinguishable from what a single DBMS instance would create, is shared.

In detail, one of the nodes (1), the *writer*, is allowed to write back to shared-storage. Other nodes (2), the *copiers*, perform copy-on-write locally to volatile storage (3), *e.g.*, a RAM disk, when necessary to ensure that future read operations will not observe pages modified elsewhere. This technique is reminiscent of the usual snapshot management with copy-on-write that exists in most logical volume managers.

In the remainder of this section we outline how our proposal tackles several key issues: How do we ensure that each node gets a consistent view of data? How are local copies prevented from growing unbounded? Finally, we describe how to deal with nodes entering and leaving the cluster.

3.1 Normal Operation

During normal operation, all nodes try to update the shared volume as they commit replicated updates. One makes sure that updates by each copier are not visible elsewhere by storing a local copy of the page in volatile storage, and using it for subsequent reads. Updates by the writer



(a) Shared-nothing DBMS cluster. A separate volume is used for each node.

(b) Shared-storage DBMS on a typical shared-storage cluster. The same volume is shared by all nodes.

(c) Proposed architecture: (1) Writer I/O interceptor; (2) copier I/O interceptor; (3) small volatile local storage.

Figure 1: DBMS server cluster architectures.

can only proceed after ensuring that the page is stable, *i.e.*, that all other nodes, the copiers, have a local copy of that page.

Stability is ensured by the following protocol: When the writer node issues a write request, it is blocked and all nodes requested to fetch the previous value of the data from shared storage and store it as a local copy. Upon receiving replies from all nodes, the write request can proceed. Read and sync requests are not intercepted at the writer. This needs to be done only once for each block, thus reducing the number of distributed interactions required.

Write requests from copier nodes are directly routed to the copy without any distributed interaction whatsoever. Read requests are first served from the local copy and if unavailable, because yet unwritten, from the shared storage. Sync requests are ignored, since there is no persistent storage involved.

In short, as only the writer replica performs actual write and sync operations, the aggregate disk bandwidth required should be approximately the same as that of a single replica. This is in sharp contrast to the shared-nothing scenario in Fig. 1(a), in which the same disk bandwith is required at each of the replicas.

3.2 Garbage Collection

The protocol of the previous section by itself has one unfortunate consequence: Volatile copies become increasingly large, as pages are updated by different nodes and copies need to be created. If nothing is done to remove pages from local copies, they will grow larger and larger, defeating the goal of having a single shared-storage. In fact, it would eventually default back to replicated shared-nothing scenario of Fig. 1(a).

Unfortunately it is not possible to remove individual pages from local copies, as inconsistencies could be created as explained in Section 2.3. The only way to discard some pages from a local copy is, in fact, to discard them all at once and invalidate all buffers, restarting from the consistent data available currently in the shared-storage. By doing this, a node forgets its own view of physical data layout and adopts the writer's view, which is physically distinct but equally consistent, and logically equivalent due to replication.

The easy way to do this without further changes to the DBMS software is as follows: First, the the server process is stopped, implicitly clearing the local volatile copy. The server process is then restarted, implicitly creating a fresh empty copy. A key issue here is that when a copier starts, it does not need to synchronize on transaction boundaries with the writer. If partially written transactions are found on the storage, the database recovery mechanism will rollback (or forward) as required. All that is needed is that the storage contains a valid prefix of write operations executed by the writer, thus ensuring transactional consistency.

This needs also that the recovery mechanism that is in place for replication, as explained in Section 2.1, is able to efficiently catch up with the rest of the cluster, as some updates might have been lost or rolled back while restarting. In fact, this requirement fits exactly to what recovery mechanisms in replicated DBMS already have to do after brief partitions or node failures and is widely satisfied by existing systems [8].

The last remaining issue is whether the impact of such periodic restarts on performance is acceptable. This depends mainly on the impact of a missing node in the cluster, which should be relatively smaller as clusters grow larger, and how fast local copies grow relatively to database size with typical workloads.

3.3 Node Failure and Recovery

Finally, one has to consider the effect of a failed node. Namely, when a copier fails it will block the stability protocol by not replying and thus prevents the writer from making progress. On the other hand, a failed writer should also block transaction commit, as storage is not updated and durability might be ensured. This depends on the replication protocol used. For instance, C-JDBC uses an additional persistent log at the middleware level to recover without blocking [2]. Protocols on group communication [6, 4] will block until reconfigured when using an uniform or safe multicast primitive [3].

Dealing with a failed copier is straightforward. One needs only to remove it from the cluster configuration, such that the writer does not wait for further replies from it. Recovering from a failed writer is as follows: First, after the previous writer is found crashed, the new writer starts by syncing its copy to disk and updates the shared storage from its local copy, using the normal stability protocol to push all relevant pages into other replicas local copy and finally invokes sync. It then erases the local copy and enters normal operation as a writer.

All operational replicas are active during this procedure, although some operations at the writer will block. Reducing such recovery time boils down to electing as writer the node with a smaller local copy, *i.e.*, the one that is closer to the previous writer.

4. DISCUSSION

This paper addresses the issue of database cluster scalability, specifically, in decoupling the amount of disk space and bandwidth required from the number of processing nodes, without requiring a profound refactoring of the DBMS server to accommodate shared-storage or large restrictions on the workload to cope with partial replication. Our approach leverages a shared-nothing replicated DBMS server, without needing profound changes to server software. The only requirement is that file I/O operations can be intercepted. In short, it works by using a variation of the common copy-on-write logical volume management technique, in widespread use today in many operating systems.

Current work focuses on experimental evaluation to determine the overhead of the stability and garbage collection mechanisms. A proof-of-concept implementation based on MySQL 5.1 is available at `http://holeycow.org`. Preliminary results obtained with the TPC-W benchmark [9] indicate that indeed the overhead of the stability protocol is negligible. By measuring the growth of local copies, one can also conclude that garbage collection by periodical restarts should be feasible [12].

The contribution presented here also opens up a number of interesting future work possibilities. Namely, to explore in detail what are the key performance factors and scalability limits and how it compares to a traditional shared-storage approach. First, one might consider multiple writer nodes with a few copiers each to further improve resilience and scalability, mainly, when using multiple data centers. Also, the combination of the proposed approach with asynchronous or certification-based replication protocols is particularly intriguing. Moreover, one should also be able to use the same technique with a virtual shared storage such as the Amazon Elastic Block Storage (EBS) to obtain an elastic database service from an existing shared nothing replicated database server. This is currently a a challenging research goal [1].

The containment of each replica by its local copy-on-write file should also provide important resilience guarantees in face of faults, such as software bugs in the DBMS server, that lead to memory and disk corruption. Moreover, the limited dependency on shared structures introduced by our approach should also be advantageous to enable on-line upgrade of the DBMS server to a new version.

5. REFERENCES

[1] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 251–264, New York, NY, USA, 2008. ACM.

[2] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Partial replication: Achieving scalability in redundant arrays of inexpensive databases. *Lecture Notes in Computer Science*, 3144:58–70, July 2004.

[3] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: a Comprehensive Study. *ACMCS*, 33(4):427–469, Dec. 2001.

[4] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 477–484, Washington - Brussels - Tokyo, July 2002. IEEE.

[5] A. C. Jr., J. Pereira, and R. Oliveira. AKARA: A flexible clustering protocol for demanding transactional workloads. In *Proc. Intl. Conf. on Distributed Objects, Middleware and Appocations (DOA)*, 2008.

[6] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 134–143, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[7] T. Lahiri, V. Srihari, W. Chan, N. Macnaughton, and S. Chandrasekaran. Cache fusion: Extending shared-disk clusters with shared caches. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 683–686. Morgan Kaufmann, 2001.

[8] W. Liang and B. Kemme. Online recovery in cluster databases. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 121–132, New York, NY, USA, 2008. ACM.

[9] D. A. Menascé. Tpc-w: A benchmark for e-commerce. *IEEE Internet Computing*, 6(3):83–87, 2002.

[10] R. Oliveira, J. Pereira, J. Afrânio Correia, and E. Archibald. Revisiting 1-copy equivalence in clustered databases. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 728–732, New York, NY, USA, 2006. ACM.

[11] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[12] L. Soares and J. Pereira. Implementation and evaluation of the HoleyCoW proof-of-concept. http://holeycow.org, 2009.