Scaling a Highly-Available DBMS beyond a Dozen Nodes

Svein Erik Bratsberg
Department of Computer and Information Science
Norwegian Universitity of Science and Technology
N-7491 Trondheim, Norway
E-mail: svein.erik.bratsberg@idi.ntnu.no

Abstract

Highly available database systems built on clusters of computers show problems when scaling beyond a dozen nodes. This paper describes some of the problems encountered and outlines some directions of solutions. One problem is the distributed algorithms used to manage availability of nodes, where every node in the system knows about the state of every other node. Another problem described here is the high cost of the brute force algorithm used in redistribution of data.

1 Introduction

Clusters of computers are used to build cheap and scalable server solutions. These solutions are often assumed to be scalable by definition. However, experience shows that it is hard to be scalable, because in a complex system every component and their interactions must be designed such that they are scalable. Based on experience from building a highly available database system [HTBH95], we have some thoughts about what does not scale so well.

One of the fundamental algorithms used in distributed, highly available systems is the *availability algorithm*. Our experience is from using the *virtual partition algorithm* [ESC85]. This algorithm lets nodes in the cluster know about the state of every other node in the system, and in real-time inform about changes in the state of nodes. This ensures that all nodes have an equal view of the system. However, the price paid for this is bad scalability. Changes in node state must be detected and coordinated among all nodes. Thus, the actions that are necessary when a node dies are delayed. We outline two different solutions to this problem, the first uses a hierarchical multicast system and the second takes a fundamentally different view by applying techniques from routing in peer-to-peer systems.

The second fundamental problem which we describe in this paper, is the data move that is necessary to have data and load balancing after adding new nodes to the system. We have implemented a straight forward approach to data scaling [BH01], which copies data from every node in the system to every other node in the system during scaling. This creates a tremendous network traffic during scaling, hurting both ordinary transactions processing and the availability messaging traffic.

The organization of this paper is as follows. We start by a brief description of the architecture. Then we treat the availability algorithm problems and possible solution. The next topic to be addressed is the data scalability problem and some solutions. Finally, we conclude the paper.

2 Architecture

We briefly describe the architecture used. The system consists of N "shared nothing" nodes which are peers, i.e. each node has the same role as every other node in the system. While every node runs the same software, the data is declustered onto the system with great care.

A table is fragmented (partitioned) and replicated to the nodes of the system. Every node is typically storing two fragments of the table, where one fragment is primary and the other is hot standby. Every node has a mirror node, which stores replicas of the same fragments, but with opposite roles with respect to primary and hot standby. In this way load and data storage needs are equally spread to all nodes. Hashing of primary keys is used as the partitioning function.

The system is horizontally scalable, by adding new nodes and using redistribution of data. This is done on-line and in parallel with ordinary user transactions. The technique used is fuzzy copy of data, while shipping the log along with the data. In this way, the new copies are kept up-to-date during the copy process.

For high availability everything is designed to be managed while the system is running, e.g. upgrade of hardware, software and operating system. This is all based on shutting one node down at the time, and do the necessary upgrade or repair, and then restart the node.

3 Virtual Partition Protocol

To discover failed nodes, the system applies an I-am-alive protocol between nodes. This protocol is organized as a ring. Each node sends and receives I-am-alive messages regularly from both its neighbors in the ring.

A *virtual partition protocol* [ESC85] is used to maintain a consistent set of available nodes. If consecutive I-am-alive messages are missing from one of the neighbors in the ring, the protocol is activated. It runs in two phases. The node which detects the change takes role as a coordinator, and it sends a *build virtual partition* (BuildVP) message to all known nodes. The participating nodes should respond with a Participate message. If a node has not responded within a certain number of resends, the node is assumed to be down. In case of conflicting builds, i.e. two nodes start to build a new partition simultaneously (with the same vpRound counter), the node with the highest identifier wins.

While the first phase surveys which nodes are up and down, the second phase of the protocol commits the virtual partition by informing the other nodes about the outcome of the survey (the CommitVP message). The original virtual partition protocol article [ESC85] assumes a reliable datagram type communication protocol. When using UDP, you must build the reliability and sequencing of messages on top of UDP. Thus, we have to apply a certain number of resends, both during the build phase and during the commit phase.

The main scalability problem with this protocol is the amount of work put on the coordinator. Especially, the second phase consumes CPU. The coordinator has to send a message which size is proportional with the number of nodes to all nodes of the system. Thus, the CPU usage is $O(N^2)$, where N is the number of nodes. Figure 1 shows the total message sizes for the coordinator for different numbers of nodes.

Nodes	2	4	8	16	32	64	128	256
BuildVP	38	114	266	570	1178	2394	4826	9690
Participate	115	345	805	1725	3565	7245	14605	29325
CommitVP	195	1053	4641	19305	78585	316953	1272921	5101785
CommitVPAck	37	111	259	555	1147	2331	4699	9435

Figure 1: Message volume for different messages and number of nodes.

The main problem with this is the high *takeover time* this leads to. A takeover means that the mirror node takes over the responsibilities for the failed node. To have a consistent database all nodes have to agree about which node has the responsibility for a fragment of the data. Thus, the takeover time is dependent on the time of the virtual partition build and commit. For small numbers of nodes the takeover time may easily be a sub-second number. This is decided by the timeout values used in the I-am-alive and virtual partition protocol. When scaling beyond a dozen nodes, the takeover must be several seconds to make the system able to commit a virtual partition.

If the coordinator of a partition build dies, a participant must start to build a new virtual partition. During a partition build, the I-am-alive protocol is turned off. Thus, there is a *participant timeout* which triggers a new build. When increasing the number of nodes, this timeout must be increased since the time of sending out the CommitVP increases.

There are some other problems as well. At startup, each node goes through a certain number of steps which boot the system. At each step a node announces a certain service by building a new virtual partition with this service. This informs the other nodes about the service. When the number of nodes increases, this startup become more complex. When a new virtual partition builds, another build will kill the first one. Thus, the changes made by the first one is not

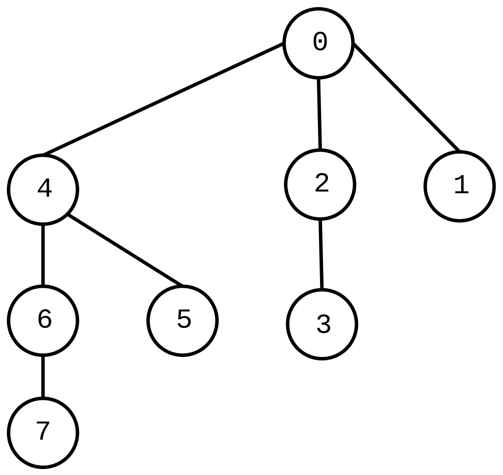

Figure 2: Multicast tree

committed yet. By repeatedly announcing new services, it may take a long time to start the system.

Hierarchical Multicast

As a simple solution to the CPU problem we suggest the following two key points:

- *BuildVP* is sent to all nodes from the virtual partition coordinator, just as in the traditional approach.
- *CommitVP* is multicasted in a hierarchical and parallel fashion.

The BuildVP message is not sent in a hierarchical fashion, because it is not known yet which nodes are not there. When the commit phase starts, we know which nodes are there.

When the coordinator is to commit the new virtual partition, it organizes a vector of nodes in the CommitVP message. It puts all the participating nodes in a vector with itself as element 0. The order of the other nodes in the vector does not matter.

The message is like this: (CommitVP, vpRound, recvIdx, nNodes, node0, node1, ..., nodeN-1)
RecvIdx is the index of the receiver node in the vector. When a node receives the CommitVP message, it forwards the message to its descendants. The forwarding information is static and precalculated, and may be formulated in a table.

Figure 2 illustrates the parallel multicast algorithm for 8 nodes. The numbers in the figure are the index of the node in the vector. Each node in the vector will send to its descendants from left to right in the tree. This makes the sending go on hierarchically and in parallel.

Every forwarding node starts a timeout after they have sent the message. When it has received an ack from all descendant nodes, it replies with an ack to its node above in the tree. If a node does not receive ack within a certain number of timeouts/resends, it can either ignore to send an ack, or it could send a negative ack, i.e. a nack.

The timeout used should be large at the top of the tree, and get shorter down the tree. The timeout for one node should be proportional to the height of the tree below it.

Distributed Hash Tables (DHTs)

The main scalability problem with the virtual partition protocol is that every node should know about every other node. *Distributed hash tables* (DHTs) have appeared as a solution to efficiently route messages in unstructured, dynamic and large peer-to-peer networks. The basic idea is that every node in the network has some information about other nodes, and with the help of each other they provide efficient routing of messages toward an object identified by a key. The DHTs are designed to handle dynamic networks well, the amount of work to do when a node fails or joins is small.

The lookup of an object is typically done using *consistent hashing*, e.g. [KLL$^+$97]. Chord [SMK$^+$01] is regarded as the original DHT. It organizes nodes and data in a ring by using consistent hashing on nodes' IP address and data's keys. Each node has a successor and a predecessor node on the ring. A node has responsibility for the keys from its hash value and down to the hash value of the predecessor. Efficient lookup is done by a finger table at each node. This efficiently makes up a binary search structure on the nodes on the ring. The availability is maintained by a stabilization process running at each node. This process polls for all other nodes it knows about and updates the search structure if necessary.

The cost of routing messages through intermediate nodes is mainly on response time. However, the response time should only increase log-arithmetically with the number of nodes. Gupta et al. [GLR03] proposes a hybrid structure in DHTs, by letting every node know about all nodes, but with a hierarchical structure to maintain availability information. Kelips [GBL$^+$03] organizes nodes into groups, where all nodes in a group know about each other. Each node knows about some contact nodes in each of the other groups. Availability messages are "gossiping" around all the time.

To make the DHTs highly available one must ensure both redundant routing and replicated data. The first is taken care by the existing algorithms. The second needs to be designed into the system. For Chord a simple solution is to let the successor node store a replica of the data of a node. If the node dies, this is the correct node to have the data anyway. In this way, we let the Chord protocol take care of takeover, by using the regular Chord algorithm for node failure. However, replication must be added to the system. This is done by letting the primary node forward the write operations to its successor. These requests must be tagged as a hot standby operations, so that the successor can separate between wrongly addressed primary operations and hot standby operations. This resembles much what is called *chained declustering* [HD90].

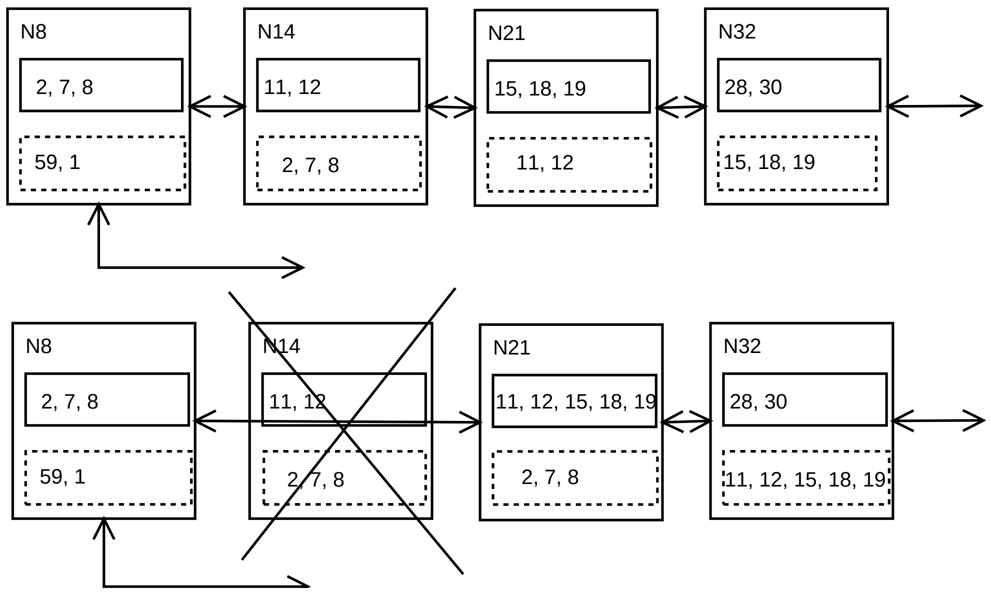

Figure 3: Adding replication to the Chord ring

This is illustrated by a part of the Chord ring in Figure 3. At the top we show four nodes with some hash values they have responsibility for. Primary data is illustrated in a fragment at the top of the node, while hot standby data is at the bottom of a node. E.g., Node 14 (N14) has the data with hash values 11 and 12 as primary data and 2, 7 and 8 as hot standby data. In case N14 dies, N21 will take over and it will make its hot standby data become primary. This new primary data is copied to N32 as hot standby data. Similarly, N8 copies its primary data as hot standby to N21. Fault tolerance of routing is taken care of by the existing Chord algorithms, where each Chord node has successor pointers to several nodes in the ring.

4 On-line Scaling of Data

On-line data scaling is provided by fuzzy copy of data while updates are allowed [BH01]. This is done by copying the log along with the data. The declustering method used makes this a brute force algorithm. All data in one table is copied from *all* nodes to *all* nodes (including the new ones) simultaneously. Each record is copied to two nodes, the new primary and the new hot standby node. Every log record produced by ordinary user transactions will be sent to the regular hot standby node, and it will be replicated to the new primary and new hot standby nodes as well.

The brute force method for on-line scaling of data results in heavy consumption of CPU and network resources, making the system vulnerable to failures. Additionally, this has severe impacts on the user transactions executed in parallel with the scaling transaction. This creates problems for resource allocation and control, and for scheduling of different activities.

By limiting the amount of data copy, we could make this situation much better. Distributed hash tables (DHTs) provide an interesting scalability approach by applying hashing and local splitting of data when nodes are added. In Chord a new node takes place in between two other existing nodes. Some data is copied from the successor node, and the successor ring is updated.

In the replicated solution we described above, a node join results in data to be copied both from the predecessor (hot standby data) and the successor (primary data). To repair a node failure, data needs to be copied from the failed node's successor to the successor's successor. The main problem with this approach is that when adding a new node, data is only balanced between two nodes. Additionally, the extra load imposed by the join or failure is on the few nodes involved. It is not spread onto more nodes.

Some interesting solutions to the data scaling problem is based on linear hashing, which is a hashing method used to scale data in a block-based storage [Lit80]. This method is extended to a distributed system where data is spread over nodes [LNS93]. The main scalability problem with this method is that it is based on a central unit to coordinate the splitting of data. SCADDAR [GSYZ02], which is developed to scale over homogeneous disks, scales data smoothly by only moving data which ends in the new nodes (or disks). Every time the system scales a new re-mapping function is computed. To find the location of data the history of re-mapping functions must be applied. This balances data among nodes, but the re-mapping function must be distributed among the nodes.

5 Conclusions and Further Work

We have shown two different scaling problems in a highly available database management system. One possible way to scalability is to look to recent developments in peer-to-peer systems.

This is clearly a field which needs further development. With the vast amount of computers available on the net, many new solutions utilizing these resources will emerge. Many of these solutions may be build on top of current infrastructure, and in some sense build their own virtual private net. This type of virtual infrastructure may be used for task force type of organizations, which are created ad hoc across other organizations, and without any server type of computers.

When building highly available systems, there is one advice which is more important than any other advice: Keep the system simple. Simple solutions are the easiest to test, maintain, extend and do fixes to. They are also better when there are changes in people to maintain or further develop the system. Some of the DHT proposals may be too complex to be used as a basis for a highly available system. Chord is probably the simplest one. By combining this with the simple chained declustering we think that a highly available solution is provided. The main problem with chained declustering is that node failure and join result in poor data balancing and the extra load to stabilize the system involves a few nodes.

Query processing in a distributed DBMS where you do not know all participating nodes is also a challenge.

References

[BH01] Svein Erik Bratsberg and Rune Humborstad. Online scaling in a highly available database. In *Proceedings of the 27th International Conference on Very Large Databases, Roma, Italy (VLDB 2001)*, pages 451–460, September 2001.

[ESC85] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 215–229. ACM Press, 1985.

[GBL$^+$03] Indranil Gupta, Ken Birman, Prakesh Linga, Al Demers, and Robbert van Renesse. Kelips*: Building and efficient and stable p2p dht thorugh increased memory and background overhead. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[GLR03] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 7–12, May 2003.

[GSYZ02] Ashish Goel, Cyrus Shahabi, Shu-Yuen Didi Yao, and Roger Zimmermann. Scaddar: An efficient randomized technique to reorganize continous media blocks. Technical report, Computer Science Department, University of Southern California, 2002.

[HD90] Hui-I Hsiao and David J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*, pages 456–465. IEEE Computer Society, 1990.

[HTBH95] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21st International Conference on Very Large Databases, Zurich, Switzerland (VLDB '95)*, pages 469–477, September 1995.

[KLL$^+$97] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigraphy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[Lit80] Witold Litwin. Linear hashing: A new tool for file and table adressing. In *Proceedings of the Sixth International Conference on Very Large Databases, Montreal, Canada (VLDB '80)*, October 1980.

[LNS93] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. Lh*: Linear hashing for distributed files. In *Proceedings of ACM/SIGMOD (Management of Data)*, 1993.

[SMK$^+$01] Ian Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalabale peer-to-peer lookup protocol for internet apllications. In *ACM SIGCOMM Conference*, 2001.