Hyrise-R: Scale-out and Hot-Standby through Lazy Master Replication for Enterprise Applications

David Schwalb, Jan Kossmann, Martin Faust, Stefan Klauck, Matthias Uflacker, Hasso Plattner

Hasso-Plattner-Institute, Germany
Contact: david.schwalb@hpi.de

ABSTRACT

In-memory database systems are well-suited for enterprise workloads, consisting of transactional and analytical queries. A growing number of users and an increasing demand for enterprise applications can saturate or even overload single-node database systems at peak times. Better performance can be achieved by improving a single machine's hardware but it is often cheaper and more practicable to follow a scale-out approach and replicate data by using additional machines.

In this paper we present Hyrise-R, a lazy master replication system for the in-memory database Hyrise. By setting up a snapshot-based Hyrise cluster, we increase both performance by distributing queries over multiple instances and availability by utilizing the redundancy of the cluster structure. This paper describes the architecture of Hyrise-R and details of the implemented replication mechanisms. We set up Hyrise-R on instances of Amazon's Elastic Compute Cloud and present a detailed performance evaluation of our system, including a linear query throughput increase for enterprise workloads.

1. INTRODUCTION

In-memory databases, like SAP HANA [20, 21], HyPer [15] or Hyrise [11, 10], are well-suited for mixed workloads as they are issued from enterprise applications. Since the use of in-memory technologies for enterprise databases, new applications have been developed, which attract a growing number of users, who submit increasingly complex queries for interactive applications. The resulting increased workload requires scalability. Instead of scale-up, i.e. exploiting an increasingly large shared-memory server, scale-out, i.e. exploiting an increasing number of interconnected servers (cluster), is recognized to be the cheaper, more flexible, more resilient, and more scalable approach [6, 17]. There are two techniques to distribute data within a cluster: partitioning and replication. Partitioning is the splitting of data items, e.g. tables, rows, columns. Database replication means that data items are duplicated and stored on multiple nodes.

The analysis of modern workload distribution shows that more than 80% of OLTP queries and for OLAP even more than 90% are read queries [16]. Read queries are easily parallelizable as they are not able to violate any consistency or isolation requirements which means that no locking mechanisms are required. The large amount of read queries and the possibility to distribute them among several nodes make the concept of database replication desirable for enterprise applications.

In the following we are going to present how we implemented database replication for the in-memory database *Hyrise*[1], which challenges we faced upon doing that and which results we got from evaluating our solution with different experiments. In the end we will give a conclusion and talk about future work.

IMDM '15, August 31 2015, Kohala Coast, HI, USA

DOI: http://dx.doi.org/10.1145/2803140.2803147

2. RELATED WORK

Quite some work has been done in the area of replication and it mainly differs in the applied replication mechanisms. Gray et al. summarize the two main replication models [9]. They distinguish two ways how to "propagate updates to the replicas", i.e. *eager* and *lazy*, and two ways to "regulate replica updates", i.e. *group* and *master*.

Eager replication propagates updates to all replicas as part of the transaction. When a transaction is committed, it is executed on every node atomically. Following, all data items in the cluster, i.e. on all nodes, are at the same state after the end of a transaction. In contrast, *lazy* replication postpones the updates of replicas. The propagation of changes to other nodes is handled asynchronously. Lazy replication delivers better performance than eager approaches since it does not need locking mechanisms or additional messages to keep the nodes synchronized but has to assure consistency in another way as eager replication.

The second model differentiates where data altering transactions can be issued. The first approach allows them only on the *master*, also called *primary node*, which is responsible to propagate changes to the *replica*, respectively *secondary*, nodes. Contrary to master replication, *group* replication, a so-called *update everywhere* strategy, allows write-queries on every node and propagates the changes from there to the other nodes. Resulting, there is no designated master node within the cluster.

Kemme et al. present a way to implement eager replication for a real-world database, *Postgres-R* [14]. They also

[1]https://github.com/hyrise/hyrise

provide techniques to avoid limitations and disadvantages of eager database replication. For example, their implementation executes data altering operations on so-called shadow copies. The changes of these shadow copies are then propagated shortly before a particular transaction is committed. This enables the database to keep consistency without locking on all nodes or suspending of certain queries.

Breitbart et al. focus in their work on lazy database replication [3]. In contrast to our solution they investigate an *update everywhere* solution and propose ways how to optimize the serialization of queries in order to enhance performance. The authors eliminate data placement requirements with their solution and proof their performance enhancement with an extensive evaluation.

Mühlbauer et al. introduce *ScyPer*, a scale-out version of the in-memory database HyPer, which implements lazy master replication [18]. They propose a row-layout for primary nodes, which is better suited for transaction processing. Replica nodes can store data in a row, column or hybrid format.

In [23] Thomson et al. discuss the softening of traditional ACID transactions in order to provide linear scalability and give examples for those systems. In their work they introduce Calvin a transaction scheduling and data replication layer that reduces contention costs by using a deterministic ordering guarantee.

Not completely related to database replication, but important for availability, is the work of Chen et al. [5]. If availability should be increased, a proper failure detection is indispensable. Chen and his colleagues introduce metrics for failure detection algorithms to measure their quality of service. Furthermore they present a new failure detection algorithm and analyze its quality of service. We are not focusing on failure detection, but it is still one aspect of our solution.

Interesting in terms of node-to-node communication is the work of Hoefler et al. [13]. They present a new algorithm that implements `MPI_BCAST` for the *Open MPI*[2] framework over *InfiniBand*[3] in a practically constant time independent of the communicator size. For our work an efficient way of communicating is obviously of importance, we are going to talk about this in Section 3.2.

3. IMPLEMENTATION

Hyrise-R implements lazy master replication for Hyrise, an in-memory database system focusing on optimization for both transactional (OLTP) and analytical processing (OLAP) by implementing a main delta architecture [16]. It also implements an insert-only approach where delete operations mark a record as invalid and updates do the same, but additionally add a new record. We present our implementation Hyrise-R, which expands Hyrise and makes it possible to run it as a cluster to increase query throughput and availability.

3.1 General approach

Figure 1 illustrates our implementation, consisting of a query dispatcher and the database cluster. Users submit their queries to the dispatcher, which redirects them to the cluster nodes. The dispatcher is responsible for fulfilling *replication transparency*, which means that the cluster should,

[2]http://www.open-mpi.org
[3]http://www.infinibandta.org

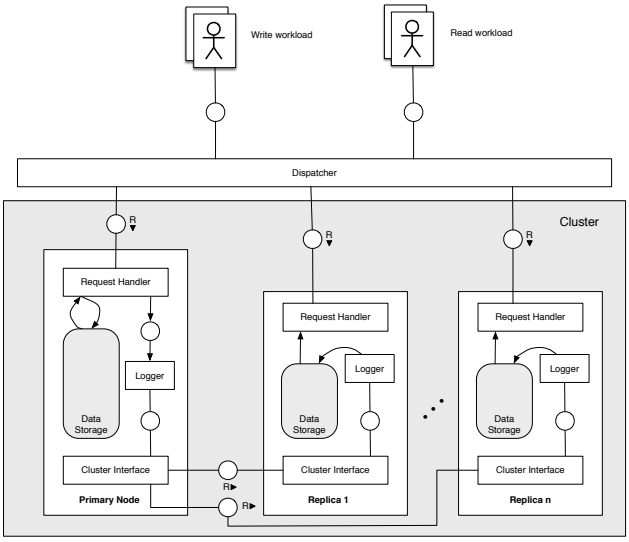

Figure 1: Architecture of Hyrise-R.

from a user's perspective, behave as a single node without clustering functionality. Even more, there should be no way of telling whether a single node or a cluster is accessed [7]. The dispatcher redirects read queries in a round-robin manner to the cluster nodes. There is no logic implemented which recognizes the type of a query. Read and respectively write queries have to be sent to different endpoints of the dispatcher. There is no further metadata needed to dispatch queries to nodes. On the other hand, data altering queries are only sent to the primary node. The current version of the dispatcher is a single point of failure. However, the dispatcher has a lower risk to fail as it is a much less complex software system than a database server. Furthermore, it is easier to harden and extend to increase its availability.

The database cluster consists of one primary node and an arbitrary number of replica nodes. The nodes are by our implementation extended Hyrise instances. We chose to implement *lazy master* replication for Hyrise-R. Using master replication is justifiable by the comparable small amount of write queries [16] and the fact that it has less chances for conflicting updates. The delay caused by lazy replication for read only replica nodes is acceptable for most OLAP applications. OLAP queries as part of writing transactions have to be executed on the master node. The architecture of HYRISE-R is not suitable as basis for applications with only such workload like available-to-promise [24]. Read queries are handled like normal in Hyrise. Data altering queries are always sent to the primary node, which uses the Hyrise `BufferedLogger`. The `BufferedLogger` was initially added to Hyrise with the implementation of a multi-version concurrency control mechanism with in-memory optimized logging [22]. The `BufferedLogger` logs transactions to the file system either when a commit is issued or at regular intervals. The logging information was intended to be used for recovery or persistency use cases. In our case, logging information is not only written to the file system but also sent to the Hyrise `ClusterInterface`. The `ClusterInterface` sends this information every N calls, when the information buffer gets too large or periodically to the replica nodes.

N, the buffer size as well as the time span for the periodic sending are configurable. The `ClusterInterface` of replica nodes replays the transaction information with help of the `BufferedLogger` and applies them to the replicated data set. Replica nodes acknowledge the reception of replication data. Sending only logging information and not complete queries reduces execution overhead. Hyrise implements further in-memory logging optimization techniques, e.g. dictionary encoded logging information by sending valueIds and not the actual values.

We selected suitable values for the number of calls N, the log buffer size and the maximum period until replication information gets sent. The log buffer has the same size as the `BufferedLogger's` buffer used for recovery, 16384 byte, which is reasonable regarding the fact that the data has to be sent over the network. The maximum period works as an upper bound in small load scenarios. As described above, a certain delay is acceptable for most OLAP workloads. We wanted to avoid a delay larger than a second, considering some network delay we chose a maximum period of 500 milliseconds. To send information when the `ClusterInterface` got called a certain number of times is a good way to handle heavy load scenarios and there is obviously a tradeoff between more and bigger messages. By experimenting we found out that sending information with every call of the `ClusterInterface` degrades performance in heavy load scenarios and eight or ten seems to be a good value for N.

3.2 Communication

The theoretical maximum number of replicas is not restricted. However, the cluster management overhead will increase with the number of replicas. The description of the replication in Section 3.1 shows the need for a communication mechanism between the primary node and replicas. This mechanism has to fulfill three requirements:

- **Multicast Communication** The used technology has to support multicast communication since an unicast approach where messages have to be sent to every single node is not acceptable. It would decrease the code quality and increase complexity unnecessarily.
- **Reliability** That messages reach their recipient has to be guaranteed. The loss of messages is not acceptable because even though consistency is not always guaranteed with our approach, at some points in time the nodes are consistent. Non-reliability of message delivery would dissolve this guarantee.
- **Scalability** The amount of data exchanged under heavy workload is large, hence the chosen communication technology must not be the bottleneck especially regarding larger numbers of replicas, otherwise the data sets of primary and replica nodes would converge and lose the possibility to catch up and become consistent again.

At first we tried *OpenPGM*[4], an open source implementation of the Pragmatic General Multicast specification. It implements a multicast protocol and aims to be reliable and scalable. Therefore, it delivers all the requested requirements but it was quite complex to implement it in our scenario and the performance was not completely satisfying.

[4]https://code.google.com/p/openpgm/

We looked into several other opportunities and finally decided to use *nanomsg*[5]. Nanomsg is similar to *ZeroMQ*[6], builds partly on the same concepts and aims for efficiency. It claims to be an improvement regarding performance and architecture by having several changes: it is implemented in C instead of C++, has an improved threading model and a couple of other enhancements. Nanomsg supports several communication mechanisms within processes, between processes and network transport via TCP. Setup and implementation took less effort compared to OpenPGM and simple performance comparisons showed clear advantages for nanomsg.

Nanomsg offers a couple of common communication patterns like bus, publisher-subscriber and survey. The survey pattern matches exactly our needs, where a central entity (the primary node) sends a survey (replication data) to all other entities (the replica nodes) and expects their vote on the survey (acknowledgement of replication data). Nanomsg's implementation of the publisher-subscriber pattern does not allow answers of the subscribers. Nanomsg satisfies all three requirements: multicast communication is achieved by the communication pattern we chose, reliability by using TCP and scalability by the design principles and architecture of the framework.

We implemented two communication channels, both following the survey pattern. The first is used for cluster maintenance, e.g. notifying the primary node when a new replica enters the cluster or exchanging heartbeats. The second one serves as communication channel for replication data. We could have used a single channel for this but using two channels for different tasks offers a cleaner structure and the possibility to handle the channels separately on different threads. We implemented a simple protocol: the first byte of every message declares the message type and the second byte the nodeId of the sender. The message types are: `NEW_NODE`, `HEARTBEAT_REQUEST`, `HEARTBEAT_REPLY`, `LOG_ATTACHED` and `LOG_RECEIVED`. `NEW_NODE` is used by replica nodes to subscribe to the cluster. `LOG_ATTACHED` is the message type used by the primary node for sending replication data and `LOG_RECEIVED` is used as an acknowledgement message by the replica nodes.

3.3 Failover

Besides performance, replication can increase availability by redundancy. The importance of availability becomes clear if we imagine enterprise use cases where seconds of downtime can result in decreasing profit or loss of customers. Cecchet et al. describe a case of a Fortune-500 company, which runs a large travel ticket booking system [4]. It is easy to imagine that even a short outage may force customers to switch to another booking system. This is applicable to other enterprise use cases as well.

If a replica node crashes, the performance of the database cluster will drop. Data altering queries can still be executed on the primary node and the cluster management will be unaffected. This is undesirable but acceptable. If on the other hand the primary node crashes, there is no way to execute data altering queries as we do not allow *update everywhere*. Therefore, the failed primary node needs a successor. When the primary node crashes, the replica nodes are either exactly in the same state or almost. Because of that it is

[5]http://nanomsg.org
[6]http://zeromq.org

efficient to promote a replica node to the new primary node.

In order to be able to do this a failure in the primary node has to be detected. There are three main objectives for algorithms that detect failures: a low message frequency to reduce overhead, a small timespan between the occurrence of a failure and its detection to increase availability and a small probability of premature failure detection to avoid misplaced failover handling [8]. These objectives interfere with each other. Hence, failure detection protocols should be tuned for the specific use case to set focus on those objectives. The probability of premature failure detection is mainly influenced by package loss and delay and not avoidable if the probability of package loss and delay are greater zero.

In our implementation replica nodes detect a failure of the master node. After one of them becomes the new primary node it informs the dispatcher about its new role. The detection is performed by a heartbeat protocol. The primary node sends periodically messages of the type `HEARTBEAT_REQUEST` to all replica nodes. They reply with a message of type `HEARTBEAT_REPLY` and their current commitId. The current commitId is necessary for delivering statistics and to monitor how much the commitIds of all nodes differ. The replica nodes check periodically if they received a `HEARTBEAT_REQUEST` from the primary node. If the amount of time they have not received such a message exceeds a certain threshold (TH), the primary node is considered to have failed and the first replica node which subscribed to the cluster will replace it. The threshold, the heartbeat interval (HI) and the check interval (CI) are configurable. The maximum (1), average if $CI < HI$ (2) and average if $CI >= HI$ (3) time our algorithm takes to detect a failed primary node are:

$$Worst\ Case\ = \lceil TH/CI \rceil * CI \quad (1)$$

$$Avg\ Case = round((TH - HI/2)/CI) * CI + CI/2 \quad (2)$$

$$Avg\ Case = \lfloor (TH - HI/2)/CI \rfloor * CI + CI/2 \quad (3)$$

The reason for letting the primary node request a reply from the replica nodes and not just letting the replica nodes send their commitId and wait for a response of the primary node is due to the fact that all replica nodes should send their status roughly at the same time without a complicated synchronization mechanism. If the information would not be pulled from but pushed by the replica nodes, they would either send it at possibly different times or would have to synchronize between each other.

If transactions are committed on the primary node and it fails before those commits were sent to the replica nodes, they are still saved to a log file on the primary node's persistent storage. After a failover the new primary node will use this information to recover as explained in Section 3.1. The recovering primary node needs access to the failed primary node's log file, which can be quite complicated. Another approach to solve this problem would be to implement eager replication as presented in Section 2.

4. EVALUATION

In order to evaluate Hyrise-R we conducted a series of measurements for different scenarios. The measurements were taken with five instances of Amazon's *EC2 c4.8xLarge* [7]. They ran a specifically optimized Intel Xeon E5-2666 v3 with 36 vCPUs or 132 so-called EC2 Compute Units (ECU). An ECU "provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor" [1]. The main memory measured 60 GiB and their persistent storage devices were SSDs. Of special interest is their *Enhanced Networking* feature, which claims to deliver improved network performance by a new network virtualization stack and a higher guaranteed bandwidth. Amazon offers so-called placement groups for clusters. The instances in a placement group benefit from lower latencies and higher throughput.

In the actual setup up four instances were running Hyrise-R and one instance was running the dispatcher and our benchmark tool. We used the Apache HTTP server benchmarking tool[8] for sending and measuring requests concurrently. For read workloads we executed a *GroupByScan* on a table with one million rows and for write workloads we inserted in a table with twenty thousand records. The following subsections present the conducted measurements and results.

4.1 Scale-Out

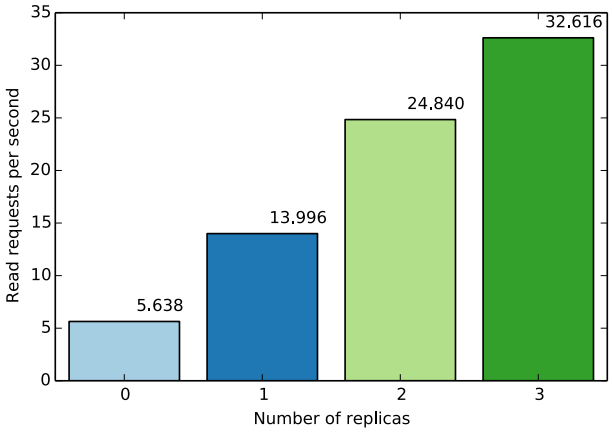

Figure 2: Increasing number of replicas with read-only workload.

One of the main targets of Hyrise-R is to improve the performance of Hyrise by a scale-out approach. In this experiment we send only read requests to the cluster and increased the number of replica nodes from none to three. As we can see in Figure 2 our solution scales really well. The linear performance improvement can result from caching effects [2] and less overhead through context switching. The first target, to improve the performance by scaling out was achieved.

4.2 Availability

The second main target of our implementation was to increase availability, which is achieved by the implementation of a failover and heartbeat protocol as described in Section 3.3. A couple of test cases showed that our failover mechanism works and is completed in the boundaries calculated above.

[7]https://aws.amazon.com/ec2/instance-types/

[8]http://httpd.apache.org/docs/2.2/programs/ab.html

4.3 Mixed Workload Performance

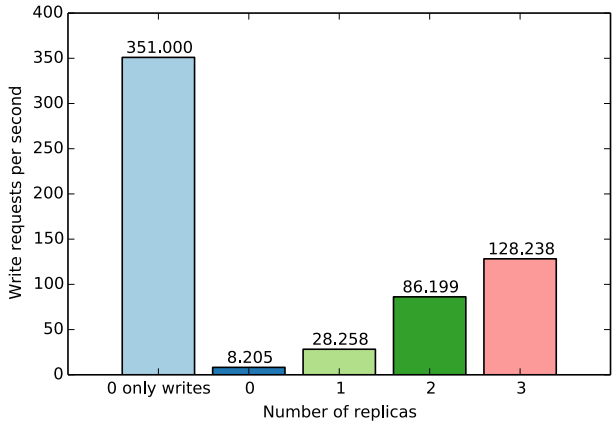

Figure 3: Increasing number of replicas with mixed workload (except first measurement).

The idea behind this experiment was to find out if an increasing number of replica nodes can also enhance the write performance in mixed workload scenarios. This is reasonable because by adding replica nodes, the primary node has fewer read queries to handle. Thus, there is more time to process write queries. Figure 3 verifies this hypothesis and shows that even though our implementation generates overhead for data exchange and cluster maintenance, it still increases the performance significantly. The diagram shows a huge drop in performance for mixed workloads compared to a write-only workload. This has been observed before and can be explained by a queueing delay as described in [25].

4.4 Replication Delay

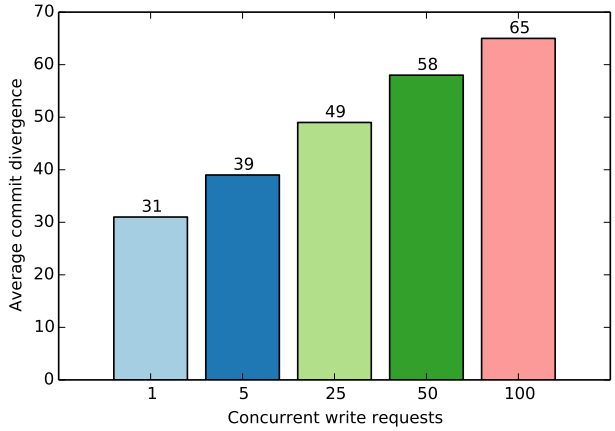

Figure 4: Average commit divergence with increasing number of concurrent write requests.

There is a certain delay between the committing of transactions on the primary node and on replica nodes. This experiment should tackle the question how large this delay is. The commit divergence describes the difference of the last commitId of the primary node and the last commitId of a replica node at a specific point in time. The average commit divergence is the average of the sampling of commit divergences of all nodes. The sampling frequency was 50ms. Figure 4 shows that with an increasing number of concurrent write queries the average commit divergence increases as well. Important to mention is the fact that there was almost no commit divergence between the replica nodes in our test scenario. They diverged by a maximum of two commits.

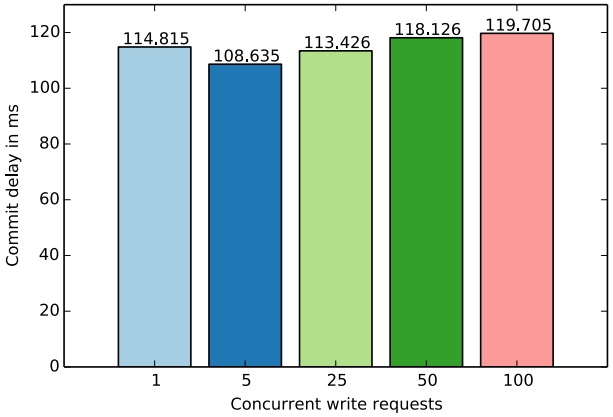

Figure 5: Commit delay in ms with increasing number of concurrent write requests.

More significant than the average commit divergence is the commit delay in relation to time. If we take the number of commits per millisecond and set this in relation to the commit divergence, we will get the commit delay measured in milliseconds, naively said: the time it takes until a commit which was completed on the primary node is completed on a replica node. Figure 5 shows the results of this calculation and in contrast to Figure 4 the values are almost stagnating independent of the concurrency level. Commits take roughly between 108ms and 120ms to get from primary to replica node. The results of these measurements depend a lot on the used hardware and network infrastructure.

If we imagine a burst of write queries and afterwards no requests at all, the cluster will be in a stale state for around 120ms. This is acceptable for our enterprise use case as a replication approach like this would not be used for critical infrastructure like transportation or power supply.

4.5 Impact of Exchanging Replication Data

To propagate the commits throughout the cluster, the exchange of data is necessary as explained in Section 3.2. This data exchange is considered as overhead and it increases with an increasing number of replica nodes. The experiment shown in Figure 6 was supposed to find out the impact of the communication overhead. As we can see, there is no impact of the communication overhead. The small fluctuations could be explainable by network and computation performance variations. At least in our test scenario the overhead of communication is apparently too small to degrade performance. In another environment where nodes would not be that closely connected we would expect overhead caused by network communication.

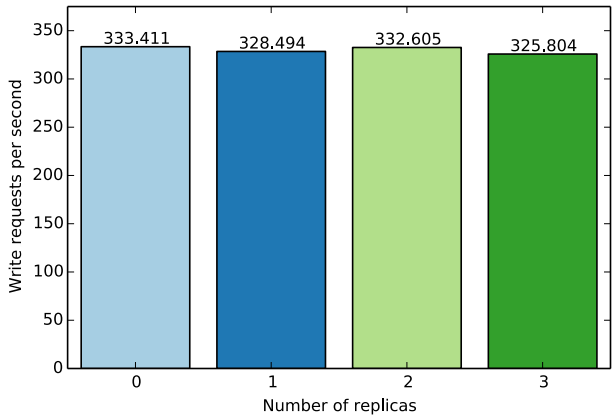

Figure 6: Write-only workload with increasing number of replica nodes.

5. FUTURE WORK

Future work includes a more sophisticated approach to handle the query distribution in the dispatcher. A possible solution could apply cost models [19] in order to estimate the workload a query will cause and then distribute the queries in a way which improves resource utilization. Furthermore, we stated that the dispatcher is a single point of failure. Setting up a second (or more) dispatcher and implementing some kind of heartbeat protocol between these dispatchers can improve the availability. If the primary dispatcher was considered to have failed, the second dispatcher would take over. The switching of dispatchers from a client's perspective could be achieved by DNS manipulation.

Besides improvements regarding the dispatcher, a more complicated failover mechanism could be implemented. Our naive approach has the problem that the message delay can have a huge impact on the worst case detection time: a `HEARTBEAT_REQUEST` of a failed primary node arriving with a delay `d` (where `d` should be smaller as the threshold TH) might delay the failure detection and thus the execution of the failover process by `d`. In our test scenarios message delay and loss were negligible but the implementation of improved algorithms like in [5, 12, 8] could be done to prepare the failure detector for these situations.

The inter node communication as described in Section 3.2 could be realized by other hard- and software technologies. OpenMPI could be considered as software framework to handle the communication. Hoefler et al. show in [13] how InfiniBand, a high-performance networking standard, can be used as the communication medium for OpenMPI.

6. CONCLUSION

In this paper, we presented Hyrise-R as a system to cluster instances of the in-memory database Hyrise using lazy master replication. The communication layer focuses on scalability, reliability and multicast communication to achieve performance enhancements. We implemented a heartbeat protocol and a failover mechanism to increase availability and utilize the redundancy caused by replication and evaluated our approach using an Amazon EC2 cluster.

7. ACKNOWLEDGMENT

Stefan Klauck has received funding from the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866 (SSICLOPS). This document reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

8. REFERENCES

[1] Amazon Web Services, Inc. Amazon elastic compute cloud - user guide for linux (api version 2014-10-01), Feb. 2015.

[2] J. Benzi and M. Damodaran. Parallel three dimensional direct simulation monte carlo for simulating micro flows. In *Parallel Computational Fluid Dynamics 2007*, pages 91–98. Springer, 2009.

[3] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 97–108, New York, NY, USA, 1999. ACM.

[4] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 739–752. ACM, 2008.

[5] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *Computers, IEEE Transactions on*, 51(5):561–580, 2002.

[6] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[7] W. Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the Conference on The future of Software engineering*, pages 117–129. ACM, 2000.

[8] M. G. Gouda and T. M. McGuire. Accelerated heartbeat protocols. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 202–209. IEEE, 1998.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[10] M. Grund, P. Cudre-Mauroux, J. Krüger, S. Madden, and H. Plattner. An overview of hyrise - a main memory hybrid storage engine. *IEEE Data Engineering Bulletin*, 2012.

[11] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, Nov. 2010.

[12] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The φ accrual failure detector. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 66–78. IEEE, 2004.

[13] T. Hoefler, C. Siebert, and W. Rehm. A practically constant-time MPI Broadcast Algorithm for

large-scale InfiniBand Clusters with Multicast. page 232, 03 2007.

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

[15] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[16] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core cpus. *Proc. VLDB Endow.*, 5(1):61–72, Sept. 2011.

[17] M. M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *IPDPS*, pages 1–8. IEEE, 2007.

[18] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. Scyper: Elastic olap throughput on transactional data. In *Proceedings of the Second Workshop on Data Analytics in the Cloud*, DanaC '13, pages 11–15, New York, NY, USA, 2013. ACM.

[19] S. Müller and H. Plattner. An in-depth analysis of data aggregation cost factors in a columnar in-memory database. In *ACM Fifteenth International Workshop On Data Warehousing and OLAP colocated with ACM CIKM, Maui (HI), USA*, 2012.

[20] H. Plattner. A common database approach for oltp and olap using an in-memory column database. *SIGMOD*, 2009.

[21] H. Plattner. The impact of columnar in-memory databases on enterprise systems: Implications of eliminating transaction-maintained aggregates. *Proc. VLDB Endow.*, 7(13):1722–1729, Aug. 2014.

[22] H. Plattner. Efficient transaction processing for hyrise in mixed workload environments. In *In Memory Data Management and Analysis: First and Second International Workshops, IMDM 2013, Riva del Garda, Italy, August 26, 2013, IMDM 2014, Hongzhou, China, September 1, 2014, Revised Selected Papers*, volume 8921, page 112. Springer, 2015.

[23] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[24] C. Tinnefeld, S. Müller, H. Kaltegärtner, S. Hillig, L. Butzmann, D. Eickhoff, S. Klauck, D. Taschik, B. Wagner, O. Xylander, A. Zeier, H. Plattner, and C. Tosun. Available-to-promise on an in-memory column store. In T. Härder, W. Lehner, B. Mitschang, H. Schöning, and H. Schwarz, editors, *BTW*, volume 180 of *LNI*, pages 667–686. GI, 2011.

[25] J. Wust, M. Grund, and H. Plattner. Tamex: A task-based query execution framework for mixed enterprise workloads on in-memory databases. In *IMDM, INFORMATIK 2013*, 2013.