Eliminating the Bandwidth Bottleneck of Central Query Dispatching Through TCP Connection Hand-Over

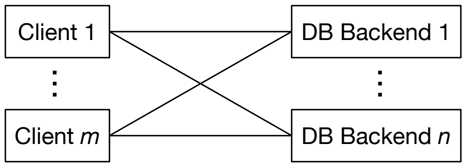
Stefan Klauck[1], Max Plauth[1], Sven Knebel[1], Marius Strobl[2], Douglas Santry[2], Lars Eggert[2]

Abstract: In scale-out database architectures, client queries must be routed to individual backend database servers for processing. In dynamic database systems, where backend servers join and leave a cluster or data partitions move between servers, clients do not know which server to send queries to. Using a central dispatcher, all queries and responses are routed via a single node. In a system with many high-performance backends, such a central node can become the system bottleneck. This paper compares three different approaches for query dispatching in terms of scaling network throughput and processing flexibility. Off-the-shelf TCP/HTTP load-balancers cannot dispatch individual queries arriving over a single connection to different backend servers, unless they are extended to understand the database wire protocol. For small response sizes up to 4 KB, a purpose-built query dispatcher delivers the highest throughput. For larger responses (i.e., BLOBs or data sets for external analysis), a novel approach for network proxying that transparently maps TCP connections between backend servers performs best.We propose hybrid query dispatching that performs a TCP connection hand-over on demand when returning large database results.

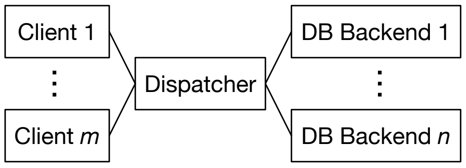
Keywords: Scale-Out Database Systems, Query Dispatching, Load-Balancing

1 Query Dispatching

In scale-out database systems, queries must be routed to individual backend servers. Clients may send queries directly to individual backends (see Figure 1a). In this case, they have to select a suitable server with respect to load-balancing and data distribution. An alternative approach uses one or multiple dedicated controller nodes (“dispatchers”), which act as central load-balancers (see Figure 1b).



(a) Direct communication.



(b) Central dispatcher.

Fig. 1: Query dispatching architectures.

[1] Hasso Plattner Institute, University of Potsdam, Germany
[2] NetApp

doi:10.18420/btw2019-07

TCP/IP is the most common communication protocol to exchange information in shared-nothing architectures. Likewise, database systems use wire protocols over TCP/IP. Without dispatchers, clients communicate directly with backend servers, using one or more TCP connections for each. With central dispatchers, the dispatchers mediate client communication with backend servers, often terminating client TCP connections and maintaining a separate set of (persistent) TCP connections with the backend servers. *Dispatchers introduce overhead, but also provide several advantages.* They allow for simpler clients that remain unaware of individual backend servers, their load levels, and the deployed physical design, i.e., partitioning and replication. This is advantageous for elastic and dynamic systems, because servers can transparently join or leave a cluster and data can be repartitioned on the fly. Such flexible database backends have increased in importance [Cu10, Se16, RJ17].

Dispatchers can be used independently of particular database systems, which may have different communication characteristics. Workloads for transactional systems and key-value stores are characterized by simple read and write queries (or get and put operations). Message sizes comprise usually up to a few kilo bytes, but larger data object comprising mega bytes of data may occur from time to time. The in-memory database Silo can process almost 700K TPC-C transactions per second [Tu13]. Analytical workloads are characterized by more complex queries with varying response sizes for which the query execution time dominates communication costs most of the time. However, Raasveldt et al. claim that transferring large amounts of data from databases to clients is common for complex statistical analyses or machine learning [RM17] in the application.

The existence of a database bandwidth bottleneck depends on the query characteristics, especially the ratio of database processing and result set sizes. Comparing the speed of a single 10 Gb network interface controller (NIC) and 8 CPU cores accessing main memory with 10GB/s, we informally claim that the network is 64 times slower in this scenario. Resulting, a reasonably well-programmed result-set serialization can produce messages that are an order of magnitude larger than those which can be send. (Note that serialization can be parallelized and carried out asynchronously to messages transfer.) In systems with multiple backend servers, a central query dispatcher can further constrain the data throughput. Prism [Ha17] eliminates the dispatcher bottleneck by redirecting client connections to individual backend servers on a query level. Redirecting TCP connections adds overhead for packet transformations but pays off for large data transfers.

Contributions: In this paper, we investigate database query dispatching for large messages. We integrated Prism into the in-memory database Hyrise [Gr10] and compare the performance of Prism against two approaches with the state-of-the-art architecture for central dispatchers, i.e., using two separate TCP connections, one between client and dispatcher, the other between dispatcher and backend. The first approach is the purpose-built Hyrise query dispatcher [Sc15]; the second is the off-the-shelf TCP/HTTP load-balancer HAProxy [Ta]. We investigate for which message sizes the Prism architecture outperforms the classical dispatcher architecture. In case message sizes are known in advance, Prism's connection redirection can be carried out on demand.

Outline: The remainder of this paper proceeds with an introduction of the compared dispatcher implementation in Section 2. In Section 3, we evaluate the performance of implementations. Section 4 discusses the results and most important aspects of query-based load-balancing. Section 5 describes related work, and Section 6 concludes the paper.

2 Dispatcher Implementations

To evaluate the different dispatcher implementations, we use a cluster of Hyrise [Gr10] database instances that implement lazy master replication. Load-balancing can be implemented at connection, transaction, or query level [CCA08]. While query processing at backends is independent of load-balancing, a dispatcher has to understand the message format in order to perform transaction- and query-level load-balancing. Hyrise uses JSON-encoded query plans encapsulated in HTTP request bodies. Other databases use different message-based protocols [RM17]. The Hyrise dispatcher and Prism can be extended to implement these wire protocols, because their underlying architectures are database-agnostic.

2.1 Hyrise Dispatcher

The Hyrise dispatcher is a purpose-built query load-balancer for Hyrise database clusters [Sc15]. Using purpose-build dispatchers for different database systems in common, because they have to understand the database wire protocol. To the best of our knowledge, all central query dispatchers send client queries and database responses over two separate TCP connections successively: the first between the client and the dispatcher; the second between the dispatcher and a database backend.

The Hyrise dispatcher uses the Berkeley socket API with a buffering mechanism that avoids copying data inside the user space. Client requests are first read into a buffer, after which the dispatcher parses the requests and forwards queries to a suitable backend. The Hyrise dispatcher uses the nodejs/http-parser[3] for parsing client requests and extracting queries. Writes go to the master, while reads can go to any node, and tables loads are replicated across all database instances. Query responses are returned to clients also via the Hyrise dispatcher. The current implementation of the Hyrise dispatcher uses one thread per client connection. This implementation was sufficient for the experimental analysis, which uses a small number of persistent client connections. However, we observed some thread interference, especially when not pinning threads to CPU cores.

2.2 HAProxy

HAProxy [Ta] is a popular and general-purpose TCP/HTTP load-balancer. It has a similar design as the Hyrise dispatcher with separate TCP connection between clients and dispatcher

[3] https://github.com/nodejs/http-parser

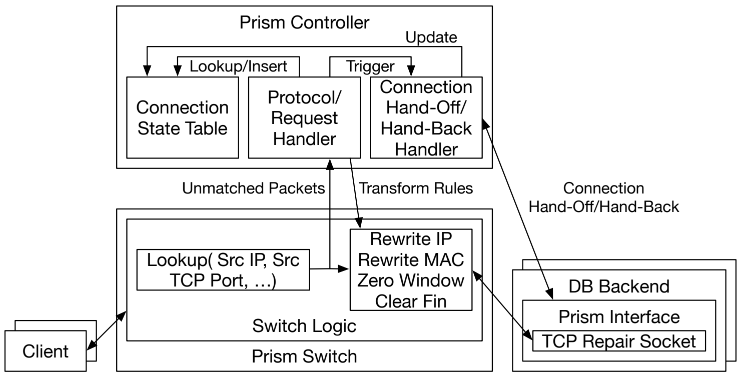


Fig. 2: Prism software architecture, based on [Ha17].

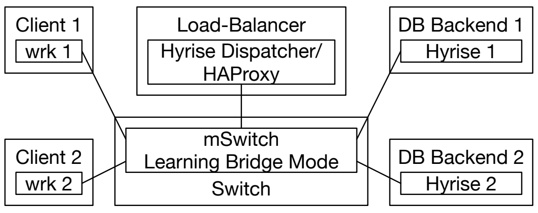
as well as dispatcher and backends. Including an off-the-shelf solution in the evaluation is interesting, because it is consistently updated for performance and support of new features. However, it requires some customization in order to adapt it to the specific scenario.

HAProxy can perform routing and processing decisions based on aspects of an HTTP request, such as the request URI. Following, HAProxy can be used for Hyrise clusters as long as the distinction between reads (to be distributed between replicas) and writes (to be delivered to the master) can be made without a deep inspection of HTTP request bodies. When it comes to load-balancing of application-level protocols other than HTTP (as for other databases), HAProxy is limited to TCP connection load-balancing. Resulting, HAProxy and possibly other off-the-shelf load-balancers cannot be used with database clusters that partition data or replicate only subsets of the data.

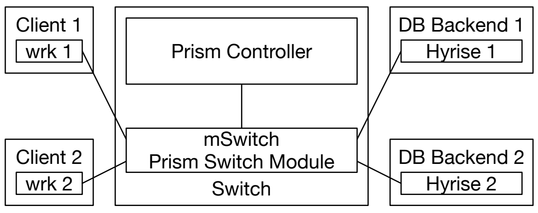
2.3 Prism

In analogy to an optical prism, Prism [Ha17] transparently splits a single TCP connection, breaking out different application-level requests and forwarding them to different backend servers by means of reprogrammable software-defined networking (SDN) switches, such as P4 [Bo14] hardware switches or software switches, such as mSwitch [Ho15]. By default, the mSwitch kernel software switch operates in learning bridge mode, but allows for dynamic packet forwarding decisions by ancillary modules, such as the kernel component of Prism.

Figure 2 shows Prism's software architecture. Prism logically consists of a controller, which handles TCP handshaking and parses client request headers, and a programmable switch, which the controller reprograms to route TCP packets containing request and response payloads directly between a backend server and the client. After a backend server has handled a request, the controller resumes handing off the client's TCP connection. Compared to traditional proxy approaches, Prism significantly reduces load on the controller, because the majority of packets is exchanged directly between clients and backends. Eliminating the controller as a central bottleneck for most of the communication also means that connections can take better advantage of typical multi-connected datacenter fabrics, significantly



(a) Hyrise dispatcher and HAProxy topology.



(b) Prism topology.

Fig. 3: Topologies for the evaluations, based on [Ha17].

improving performance. Dispatching at the switch level does incur some overheads, due to the need to reprogram switches and performing connection hand-offs, making Prism primarily suitable for larger messages. In most scenarios, the Prism connection hand-off only becomes possible after the complete query is received at the controller. Although the bandwidth for queries to the database is not increased in this case, database results can be sent with higher overall bandwidth. Details of Prism, e.g., the connection hand-over and TLS support, are described in [Ha17].

3 Evaluation

We integrated Prism into Hyrise. This integration is transparent to clients, but required changes to the database server, which has to be able to hand-over TCP connections with the Prism controller (see Figure 2). In this section, we compare the query dispatching performance of Prism, the Hyrise dispatcher, and HAProxy. We describe the experimental setup in Section 3.1. The results are presented in Section 3.2.

3.1 Experimental Setup

Our experimental setup is similar to the one employed for evaluating Prism [Ha17], but extends the experiments with 40G Ethernet. It is comprised of six nodes: two clients, two backends and two “dispatcher” machines acting as either software switches or Hyrise/HAProxy dispatchers, depending on whether measurements using 10 or 40G networking are conducted. Figure 3a depicts the logical topology used for the Hyrise dispatcher and HAProxy tests, Figure 3b likewise for Prism. Using only two clients and backend servers is no architectural limitation, but a result of limited own network hardware resources and the impossibility of running the experiments in public clouds due to insufficient hardware control.

The client machines are equipped with one Intel Xeon E5-2680 v2 CPU clocked at 2.8 GHz, 64 GB of 1333 MHz DRAM and Linux 4.11 kernels with Ubuntu 17.04. Backends have two Intel Xeon E5-2650 packages clocked at 2.0 GHz and 128 GB of 1333 MHz memory, running Linux 4.8 kernels with Ubuntu 16.10. The remaining “dispatcher” machines have one Intel Xeon E5-2680 v2 CPU clocked at 2.8 GHz, 64 GB of 1600 MHz DRAM, running Linux 4.8 with Ubuntu 16.10.

For the 10G experiments, all machines use Intel 82599ES dual-port NICs, connected through an Arista 7050QX-32-F switch. For the 40G experiments, the "dispatcher" nodes have Intel XL710-QDA2 dual-port NICs, while backends and clients use Mellanox ConnectX-3 MCX354A-FCB NICs. All 40G cards are directly connected, because of insufficient 40G ports on the Arista switch. The Ethernet MTU is set to the default of 1500 B in all cases.

Depending on which dispatcher approach is being measured, one "dispatcher" node is configured to run mSwitch, either in Prism mode or as a learning bridge. This allows for a fair comparison between the Prism and non-Prism scenarios, by using a software switch in all cases. A baseline netperf[4] TCP network benchmark results in 19.5 Gb/s maximum throughput for 128 MB of bulk data over the 40G setup with the Mellanox adapters, i.e., the bottleneck is not the software switch. Full wire speed is achieved with 10G.

The backends are running Hyrise on all 16 cores, one acts as master with the other as replica. In order to make sure that Hyrise itself is not the bottleneck, a minimal stored procedure returns a database result with the requested number of bytes. The single-threaded Prism controller executes bound to a single core on the same "dispatcher" machine as mSwitch. For experiments not involving Prism, the other "dispatcher" node is either running the Hyrise dispatcher with its client threads bound to four cores or HAProxy 1.7.9 in a multiprocess configuration with both its backend and frontend processes also bound to four cores.

Measurement data is obtained on the clients by running one single-threaded instance of the wrk[5] HTTP benchmark tool. Client transactions result in response payload sizes between 1 B and 128 MB, which are sampled for 20 s using one persistent connection each. The results of the two clients then are aggregated.

3.2 Experimental Results

Figure 4 shows the throughput results for the different dispatcher approaches for the 10 and 40G experiments. A subset of the results is summarized in Table 1. Throughput is provided in terms of HTTP payload in all cases, i.e., from the client/user perspective. The main result is that, for payload sizes over 1 MB, Prism outperforms the other two dispatchers by up to 2× (the number of client/backend nodes). This is because for the Hyrise dispatcher and HAProxy, each payload byte needs to traverse the dispatcher (see Figures 1b and 3a, respectively), which in a typical datacenter fabric is connected via a single 10 or 40G link to the top-of-rack switch. This limits both cases to at most 10 Gb/s and 40 Gb/s, respectively.

In the 10G experiment, the Hyrise dispatcher and HAProxy achieve almost the physical limit of 10 Gb/s for payload sizes over 16 MB. However, the maximum measured throughput in the 40G experiment is only 18.9 Gb/s for the Hyrise dispatcher and 19.4 Gb/s for HAProxy, demonstrating the overheads that limit performance over today's faster network fabrics.

[4] http://www.netperf.org/
[5] https://github.com/wg/wrk

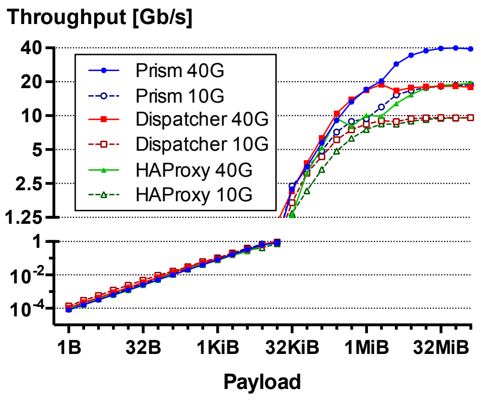


Fig. 4: Dispatcher throughputs for varying payloads.

Payload	Result Unit	Prism 10G	Prism 40G	Dispatcher 10G	Dispatcher 40G	HAProxy 10G	HAProxy 40G
1 B	Mb/s	0.10	0.08	0.14	0.11	0.09	0.11
8 B	Mb/s	0.79	0.64	1.2	0.88	0.74	0.60
64 B	Mb/s	6.3	5.1	9.3	7.0	5.8	4.8
512 B	Mb/s	50	41	63	55	42	44
4 KB	Mb/s	378	321	396	324	303	248
32 KB	Gb/s	2.4	2.2	1.7	2.1	1.4	1.3
256 KB	Gb/s	7.2	9.1	6.1	10.4	4.8	9.5
1 MB	Gb/s	9.39	17.2	8.4	16.9	7.5	10.1
2 MB	Gb/s	11.9	20.3	9.0	18.9	8.4	9.9
16 MB	Gb/s	17.7	37.8	9.5	18.1	9.3	17.7
128 MB	Gb/s	18.8	39.1	9.6	17.9	9.6	19.4

Tab. 1: Selected throughput measurements from Figure 4.

Prism does not share this limitation, because clients and backend servers are communicating directly after a TCP connection has been redirected, i.e., responses are directly sent to the clients, taking advantage of the inherent fan-out in the fabric. Consequently, Prism throughput is only limited by the speed of the network fabric (10G or 40G) or the processing capacity of the backends, whichever is lower. In the 10G case, Prism achieves 18.8 Gb/s, which is close to the physical limit of 2×10 Gb/s and twice as high as for the Hyrise dispatcher and HAProxy. In the 40G case, Prism reaches 39.1 Gb/s, again about twice as high as the other two dispatcher approaches.

For payload sizes below 4 KB, the Hyrise dispatcher outperforms Prism's and HAProxy's throughput by up to 1.5×. For small responses, Prism cannot amortize the connection hand-off overheads. We observed that HAProxy's performance for small payload sizes is reduced by interprocess communication overheads (see Section 4 for more details), which appear to be higher than for the Hyrise dispatcher.

No Dispatcher 10G	No Dispatcher 40G	Prism 10G	Prism 40G	Dispatcher 10G	Dispatcher 40G	HAProxy 10G	HAProxy 40G
59µs	70µs	161µs	214µs	104µs	144 µs	167µs	151µs

Tab. 2: Average user latencies for querying 1B payload.

To evaluate the overhead of query dispatching in comparison to direct client-backend communication, Table 2 shows the user perceived query latencies for 1 B payloads. Latency is measured from sending the request until the reception of the response, including network transmission and database processing times. Without an intermediate dispatcher, the latency is about 59 µs in the 10G and 70 µs in the 40G experiment. Adding a dispatcher, increases the latency by about 45 µs to 144 µs depending on the dispatching approach and NIC type. Among the dispatching approaches, the latency of the Hyrise dispatcher is the lowest.

4 Discussion

In this section, we discuss several aspects of query-based load-balancing. First, we summarize our research findings (see Section 4.1). Based on these, we propose a hybrid query dispatching approach in Section 4.2. In the following, we discuss the scalability of central query dispatching in Section 4.3.

4.1 Research Findings

The experiments show that no single dispatcher approach is best across all payload sizes. The Hyrise dispatcher performs best for small payloads, because the overhead of Prism's connection hand-over does not amortize if the payload fits into a few network packets. Prism outperforms the Hyrise dispatcher for larger payloads, because the majority of payload bytes can be directly handled by the switch.

HAProxy was not able to compete with either of the two, despite a lot of effort on fine-tuning its configuration. Although a single-process HAProxy configuration delivered best throughputs for small payload sizes below 0.5 KB compared to multiprocess setups, it peaked at only 10 Gb/s for payloads above 2 MB in the 40G case. Further, we did not observe any performance benefits from HAProxy taking advantage of socket splicing [MB99] compared to the Hyrise dispatcher, which has to copy all data to and from user space. These findings indicate that although an off-the-shelf load-balancer can be used, it has severe performance limitations, besides its lack of flexibility (see Section 2.2).

4.2 Hybrid Approach

Based on our measurements, we propose a hybrid load-balancing approach, which uses Prism for queries returning large results and the Hyrise dispatcher for all other requests. This requires knowledge about response sizes of individual queries, which can be calculated (or estimated) during the result set serialization. Hence, the connection hand-off has to be postponed until the response size is determined and must be initiated by the backend.

4.3 Scalability

Additional throughput can be achieved with additional and/or faster hardware. First, we can scale-up the dispatcher with additional and/or faster NICs. However, this requires efficient software to actually take advantage of the hardware. Due to its improved usage of hardware and preexisting software, Prism is expected to provide the best price-performance ratio in this regard.

Further, we can hide multiple dispatchers behind a virtual IP, using network-level load-balancing as the first and query-level load-balancing as the second step. Network-level load-balancers can additionally aggregate multiple requests to avoid the network overhead for many connections sending small messages. Especially transactional databases, which are able to process millions of small queries per second, require query batching [Tu13].

5 Related Work

Efficient query dispatching is a research field combining database and network aspects. On the database side, clustered databases with a single entry point require query dispatching. Cecchet et al. summarize different approaches for replicated databases and discuss load-balancing alternatives [CCA08]. Database load-balancing research focuses on equal distribution of load, such as the Tashkent+ load-balancer [EDZ07], rather than efficient load-balancing from a networking perspective. Consequently, there is a lack of focus on throughput and latency measurements. Our work investigates scenarios where query dispatching and the relaying of database responses can become the bottleneck. These scenarios range from thousands of small transactions [Tu13] as one extreme to large data transfers from the database [RM17] as the other extreme. Besides networking as database client interface, high performance networks are a relevant topic for distributed query processing [Bi16, Rö15]. Associated research includes microbenchmarks for single connection RDMA and Ethernet, but no switching measurements that are relevant for query load-balancing.

On the networking side, the focus in recent years has been on large-scale network-level load-balancing, especially on distributing the ingress load of hyperscalar datacenters onto backend servers at connection-open time. Proxy approaches, in which an intermediary remains in the communication path and can therefore provide finer-grained operations than simple load-balancing, have somewhat fallen out of favor, due to their perceived higher overheads. Hayakawa et al. present related work in this space [Ha17].

6 Conclusion

Many scale-out database systems use a central query load-balancer that decides where to send individual client queries. For clusters serving many thousands of requests and returning large objects or data sets, the query dispatcher and the network fabric connecting it to the clients and backend servers can become a bandwidth bottleneck. Prism redirects TCP connections and allows database backends to directly respond to clients without changing the clients. We integrated Prism into an in-memory database and compared its performance against the common dispatching architecture in which each query and result is routed via a single node. The traditional dispatcher architecture provided the highest throughput for small database responses, whereas the Prism approach significantly outperformed it for larger payloads. We propose to use a hybrid load-balancing approach that uses Prism's connection hand-over on demand for large payloads.

References

[Bi16] Binnig, Carsten; Crotty, Andrew; Galakatos, Alex; Kraska, Tim; Zamanian, Erfan: The End of Slow Networks: It's Time for a Redesign. PVLDB, 9(7):528–539, 2016.

[Bo14] Bosshart, Pat et al.: P4: programming protocol-independent packet processors. Computer Communication Review, 44(3):87–95, 2014.

[CCA08] Cecchet, Emmanuel; Candea, George; Ailamaki, Anastasia: Middleware-based database replication: the gaps between theory and practice. In: SIGMOD. pp. 739–752, 2008.

[Cu10] Curino, Carlo; Zhang, Yang; Jones, Evan P. C.; Madden, Samuel: Schism: a Workload-Driven Approach to Database Replication and Partitioning. PVLDB, 3(1):48–57, 2010.

[EDZ07] Elnikety, Sameh; Dropsho, Steven G.; Zwaenepoel, Willy: Tashkent+: memory-aware load balancing and update filtering in replicated databases. In: EuroSys. pp. 399–412, 2007.

[Gr10] Grund, Martin; Krüger, Jens; Plattner, Hasso; Zeier, Alexander; Cudré-Mauroux, Philippe; Madden, Samuel: HYRISE - A Main Memory Hybrid Storage Engine. PVLDB, 4(2):105–116, 2010.

[Ha17] Hayakawa, Yutaro; Eggert, Lars; Honda, Michio; Santry, Douglas: Prism: a proxy architecture for datacenter networks. In: SoCC. pp. 181–188, 2017.

[Ho15] Honda, Michio; Huici, Felipe; Lettieri, Giuseppe; Rizzo, Luigi: mSwitch: a highly-scalable, modular software switch. In: SOSR. pp. 1:1–1:13, 2015.

[MB99] Maltz, David A.; Bhagwat, Pravin: TCP Splice for application layer proxy performance. J. High Speed Networks, 8(3):225–240, 1999.

[RJ17] Rabl, Tilmann; Jacobsen, Hans-Arno: Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In: SIGMOD. pp. 315–330, 2017.

[RM17] Raasveldt, Mark; Mühleisen, Hannes: Don't Hold My Data Hostage - A Case For Client Protocol Redesign. PVLDB, 10(10):1022–1033, 2017.

[Rö15] Rödiger, Wolf; Mühlbauer, Tobias; Kemper, Alfons; Neumann, Thomas: High-Speed Query Processing over High-Speed Networks. PVLDB, 9(4):228–239, 2015.

[Sc15] Schwalb, David; Kossmann, Jan; Faust, Martin; Klauck, Stefan; Uflacker, Matthias; Plattner, Hasso: Hyrise-R: Scale-out and Hot-Standby through Lazy Master Replication for Enterprise Applications. In: IMDM@VLDB. pp. 7:1–7:7, 2015.

[Se16] Serafini, Marco; Taft, Rebecca; Elmore, Aaron J.; Pavlo, Andrew; Aboulnaga, Ashraf; Stonebraker, Michael: Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. PVLDB, 10(4):445–456, 2016.

[Ta] Tarreau, Willy: , The Reliable, High Performance TCP/HTTP Load Balancer. `https://www.haproxy.org`.

[Tu13] Tu, Stephen; Zheng, Wenting; Kohler, Eddie; Liskov, Barbara; Madden, Samuel: Speedy transactions in multicore in-memory databases. In: SOSP. pp. 18–32, 2013.