Accelerating Relational Databases by Leveraging Remote Memory and RDMA

Feng Li Sudipto Das Manoj Syamala Vivek R. Narasayya

Microsoft Research
Redmond, WA 98052, USA
{fenl, sudiptod, manojsy, viveknar}@microsoft.com

ABSTRACT

Memory is a crucial resource in relational databases (RDBMSs). When there is insufficient memory, RDBMSs are forced to use slower media such as SSDs or HDDs, which can significantly degrade workload performance. Cloud database services are deployed in data centers where network adapters supporting remote direct memory access (RDMA) at low latency and high bandwidth are becoming prevalent. We study the novel problem of how a Symmetric Multi-Processing (SMP) RDBMS, whose memory demands exceed locally-available memory, can leverage available remote memory in the cluster accessed via RDMA to improve query performance. We expose available memory on remote servers using a lightweight file API that allows an SMP RDBMS to leverage the benefits of remote memory with modest changes. We identify and implement several novel scenarios to demonstrate these benefits, and address design challenges that are crucial for efficient implementation. We implemented the scenarios in Microsoft SQL Server engine and present the first end-to-end study to demonstrate benefits of remote memory for a variety of micro-benchmarks and industry-standard benchmarks. Compared to using disks when memory is insufficient, we improve the throughput and latency of queries with short reads and writes by $3\times$ to $10\times$, while improving the latency of multiple TPC-H and TPC-DS queries by $2\times$ to $100\times$.

Keywords

Relational databases; RDMA; remote memory; opportunistic caching; buffer pool extension; semantic caching.

1. INTRODUCTION

Cloud database platforms, such as Amazon Relational Database Service, Microsoft Azure SQL Database, and Google Cloud SQL, are deployed in data centers comprising clusters of commodity servers with large memories and fast networks. Each server in the cluster runs an SMP relational database engine (RDBMS), such as MySQL or Microsoft SQL Server, and the service exposes a SQL API. Even parallel DBMSs, such as Actian Matrix MPP Analytics Database (formerly ParAccel) [31] Microsoft Analytics Platform System [26], and relational data warehouse services such as Amazon Redshift, run on similar clusters comprised of a collection of single server RDBMS. Memory is a crucial resource for RDBMS workloads. When memory is insufficient to meet the workload's demands, RDBMSs are forced to use disks (SSDs or HDDs) instead, which can result in significant performance degradation. In a database service, due to the variety and non-uniformity of workloads and resource demands, at any point in time, the memory usage of database servers can vary significantly from one server to another. One database server may experience *memory pressure*, i.e., demand exceeds available memory, while another server might have large amounts of *unutilized* memory which is not committed to any local process on the server.

An emerging infrastructure trend is the commoditization of network adapters that support remote direct memory access (RDMA). With the advent of technologies such as RDMA over Converged Ethernet (RoCE) [19] or iWARP [20], today RDMA is supported within data center environments at competitive prices. RDMA enables efficient, low latency ($\sim 10\mu$sec), and high throughput (~ 56 Gbps) reads and writes of remote memory bypassing the operating system (OS) kernel at both the source and destination. Since the CPU is not involved in a transfer, RDMA avoids context switches and processor cache pollution, and incurs negligible performance impact on both the local and the remote servers.

The above trends open up an interesting opportunity for databases: can an SMP RDBMS with unmet memory demands efficiently leverage unused remote memory using RDMA to significantly improve performance of its workload? Databases have used RDMA primarily as a fast data transfer mechanism from storage to database servers or between database servers [17, 26, 29]. Rödiger et al. [34] present an approach where a parallel database can benefit from RDMA by redesigning the communication mechanism of the exchange operator. We consider an orthogonal problem of leveraging available remote memory to enable an SMP RDBMS to accelerate memory-intensive workloads without a major rewrite. We focus on scenarios where a mature database engine can leverage the benefits without a major rewrite.

Abstraction for remote memory in an RDBMS: The abstraction of distributed shared memory [1, 5] supported by the operating system is not suitable for RDBMSs that require control on which data is memory-resident. RDBMSs already explicitly manage the memory hierarchy and are adept in using disk-based file-oriented structures wherever memory is not adequate. Exposing remote memory via a lightweight file API gives the RDBMS explicit control over what data is placed in remote memory. This abstraction naturally aligns with our goal of integrating remote memory with modest changes to an existing RDBMS. We demonstrate the flexibility of this abstraction through a variety of scenarios that significantly improve the performance of memory-intensive workloads.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882949

Scenarios for remote memory usage in RDBMS: We describe four novel scenarios where an SMP RDBMS can leverage remote memory to significantly improve performance of memory-intensive workloads. These scenarios are: (i) extending the RDBMS caches such as the buffer pool; (ii) spilling temporary data for memory intensive operators such as Sort and Hash; and (iii) supporting a semantic cache [10, 45]—which has traditionally been limited to application or middleware tiers—integrated into an RDBMS and pinned in available remote memory; and (iv) leveraging fast memory-to-memory transfer to prime and warm-up the buffer pool of a newly-elected primary in the event of a planned primary-secondary swap in an RDBMS cluster or a cloud database service. The first three scenarios leverage remote memory as a *a new level in the memory hierarchy of the RDBMS* whose performance lies between local memory and local SSDs or HDDs. While (i) and (ii) are handled by storage engine and lower layers, (iii) introduces interesting challenges of integrating this new form of cache within the RDBMS, query optimizer costing, plan selection, and identifying appropriate structures to cache. All four scenarios leverage the lightweight file API to access remote memory and can dramatically improve the performance of an SMP RDBMS, without affecting correctness and availability of the database server even if the remote server fails, rendering the memory unavailable.

Brokering of unutilized memory in the cluster: Unutilized memory on a subset of servers in a cluster of a cloud database service needs to be brokered to allow sharing among the different servers requesting additional memory. We use a design similar to many standard designs of resource negotiators, such as YARN [49]. Each server reports the unused memory to a memory broker. A server with unmet memory demand can request the broker for a lease to a remote memory region. This lease provides the database server exclusive access to the region. The database server *opportunistically* leverages this remote memory to improve the workload's performance without stealing memory committed to processes executing on the remote server.

Efficient implementation: There are several design decisions to consider to efficiently exploit remote memory. These include: (i) the suitable protocol to access remote memory via RDMA; (ii) whether to treat remote memory accesses as synchronous or asynchronous operations; and (iii) efficiently managing registration of memory regions to NICs which has non-trivial overheads [13]. We present a detailed implementation in Microsoft SQL Server (Section 4). While our implementation is specific to SQL Server, we expect our design to generalize to other RDBMSs particularly due to the choice of a lightweight file API to expose available remote memory in the cluster.

We conduct extensive experiments using a commodity RDMA-enabled cluster of ten servers, and using a variety of configurations, targeted micro-benchmarks and industry-standard TPC benchmarks (Section 5). We compare our implementation against several alternatives: (a) a baseline where the RDBMS uses locally-attached HDDs and SSDs when demand exceeds the memory available on the server; (b) two alternatives that leverage remote memory using off-the-shelf technologies but use different protocols to access remote memory; and (c) when the RDBMS server has sufficient local memory to serve the workload. In all our experiments, we use a high-performance enterprise-grade disk subsystem with a hardware RAID-0 controller and up to 20 disks. For queries with short reads and writes (similar to OLTP workloads), the throughput and latency improvements are $3\times$ to $10\times$. The latency of multiple TPC-H queries can be improved by $2\times$ to $10\times$, and the latency of many TPC-DS queries can be improved by $10\times$ to $100\times$ (Section 6 and Appendix B). For a variety of workloads, the throughput and latency is within 10% of that when the entire workload fits in the server's *local memory*. Moreover, there is no noticeable impact on the performance of workloads running on the server whose unused memory is being accessed remotely via RDMA. Further, by varying the number of remote memory servers a database server accesses, we demonstrate how our solution can scale by pooling memory available on multiple servers within the cluster. Ours is the first in-depth study to quantify the end-to-end benefits of remote memory and RDMA for complex RDBMS workloads. In Section 7, we also identify several potential directions of future work.

Contributions:

- We study the novel problem of how an SMP RDBMS can leverage RDMA and available remote memory in a cluster.
- We abstract remote memory, accessed via RDMA, as a lightweight file API. This abstraction enables easy integration of remote memory into existing RDBMSs and is flexible enough to support four novel scenarios that significantly improve performance of memory-intensive workloads.
- We articulate and address several crucial design challenges to efficiently support this abstraction in a mature RDBMS.
- We implement this abstraction and scenarios in Microsoft SQL Server and conduct extensive experiments to quantify the benefits using targeted micro-benchmarks as well as industry-standard benchmark workloads.

2. RELATED WORK

RDMA has been extensively used for fast data transfers in high performance computing systems using MPI implementations over Infiniband [24]. More closely related to our work are the applications of RDMA in storage and file systems, key-value stores, and relational database systems. We organize related work into the following categories: (a) using RDMA as a faster interconnection protocol; and (b) when multiple servers are accessing and sharing the same database, leveraging RDMA to efficiently exchange data between two servers. In contrast to existing approaches, our focus is on exploring remote memory as a new level in the memory hierarchy for an SMP RDBMS when the workload's memory demands cannot be met by local memory.

A common use of RDMA is in fast transfer of data from compute to storage servers in various application scenarios. For instance, many database appliances, such as Teradata Aster Big Analytics Appliance [43, 44] and Microsoft Analytics Platform System [26], use RDMA to transfer data in the storage servers to the database servers. Windows Server 2012 supports SMB Direct [37] which is a network file sharing protocol over RDMA. Applications, such as the Microsoft SQL Server database, can store data on a remote file server and efficiently transfer data using SMB Direct [39]. In addition, many projects have also explored leveraging RDMA to improve the performance of distributed file systems [21, 50].

The ability to efficiently transfer in-memory pages between two servers have been explored in various systems to improve performance. For instance, scale-out relational databases with a data-sharing architecture, such as IBM's DB2 Purescale [17] and Oracle Real Application Cluster (RAC) [29], uses RDMA to transfer database pages between the servers executing queries that access the shared database pages. The database nodes access the same database for both read and write operations through the global cache of pages through a centralized lock manager. Along similar lines, Barthels et al. [2] leverage RDMA as a fast data shuffling mechanism for distributed in-memory join processing. Rödiger et al. [34] present the architecture of a distributed query engine that re-architects the exchange operator for improved parallelism and proposes an RDMA-based communication multiplexer for fast

intra-node data transfers. These approaches are orthogonal to our scenarios since we focus on scenarios where an SMP RDBMS can leverage RDMA and remote memory without a major architectural rewrite. The buffer pool priming scenario is similar to transferring data pages between servers, though only one server can execute transactions accessing the buffer pool.

Many distributed key-value stores have leveraged RDMA to improve performance of their respective systems. Jose et al. [22] extend Memcached, an in-memory caching infrastructure implementing a key-value API, to leverage RDMA for efficient internode communication. Along similar lines, Huang et al. [15] extend HBase, a key-value store implemented on top of a distributed file system, to leverage RDMA for efficient messaging. FaRM [12] exposes the memory of the machines in a cluster as a partitioned global address space and implements an efficient hashtable with lock free reads over RDMA. FaRM uses RDMA to efficiently and directly access data in a shared address space, and for fast messaging between the nodes. HERD [23] is also a distributed key-value store which uses RDMA for accessing the keys and values in the system. RamCloud, a distributed in-memory key-value store, uses Infiniband send/receive verbs to reduce the replication latencies leveraging kernel bypass [33].

3. NOVEL SCENARIOS

We now discuss four scenarios which provide a good trade-off between the potential performance benefits, complexity of implementing the remote memory abstraction, and ease of integration into an SMP RDBMS.

3.1 Extending the Caches

RDBMSs have several types of caches, such as procedure cache and buffer pool cache, which are critical to improve performance. When the size of a cache nears the available memory, an RDBMS must evict entries from the cache to accommodate newly-accessed entries. This eviction results in degraded performance if the evicted entry is accessed again. Instead of discarding the evicted entry, it can be cached in remote memory. When the evicted entry is accessed again, it can be fetched from remote memory, which is much faster for both sequential and random accesses compared to reading the entry from HDD or SSD. This extension of the cache can result in significant performance improvements when an application's working set does not fit in local memory. Such extension of the cache is similar to buffer pool extensions to SSDs supported in many commercial RDBMSs [3, 4, 28]. Similar to buffer pool, the procedure cache, which stores optimized query plans and partial execution results, can also be extended to remote memory.

3.2 Spilling Temporary Data

An RDBMS can generate a lot of temporary data during the execution of complex queries. Examples include user-generated temporary objects such as tables, table variables, or cursors, as well as system-generated temporary structures and data, such as intermediate results from data-intensive operators such as sorting and hashing. Depending on the workload, the size of such temporary data needed during query execution can be significant. If queries create a lot of temporary data and enough memory is not available, the RDBMS might need to spill the data to a temporary file. For instance, Microsoft SQL Server has a special database, called TempDB, for spilling such temporary data [42] and Oracle RDBMS uses Temporary Tablespaces [30]. Therefore, the I/O performance of the temporary file can significantly affect the performance of queries, specifically data analysis queries, as sort and hash are common operators used in many analysis queries. Storing this temporary data in remote memory and accessing it via RDMA can significantly improve query performance when compared to storing this temporary data in local HDDs or SSDs.

3.3 In-Memory Semantic Caching

Application-level semantic caches for RDBMSs, e.g., Oracle TimesTen Application Tier Database Cache [45], can speed up query execution significantly [10]. Such caches store the result of a SQL expression, support a standard SQL interface, and when possible, answer application queries by using results from the cache. If the query cannot be answered from the cache, the query is routed to the database engine for execution. Traditionally, such caching and opportunistic structures are used in the application-tier to avoid interfering with the memory requirements of the RDBMS. However, if remote memory is available, seamless semantic caching within the RDBMS can become attractive. The basic idea is to create specialized redundant structures, keep them pinned in remote memory, and access them via RDMA when executing queries that benefit from them. Examples include non-clustered indexes, partial indexes [35, 40], and materialized views, which can significantly speed up query execution and can be lazily built and maintained since they are redundant. Such structures are built opportunistically when remote memory is available, and is separate from the buffer pool memory local to the server. RDBMSs support a variety of memory brokers, such as buffer pool and procedure caches, and the semantic cache is yet another broker. In the presence of updates, these redundant structures are updated based on application-specified policies. They can be updated in-sync with transactions, updated asynchronously, maintained as a snapshot, or invalidated with an update. A key advantage of an integrated semantic cache is that it can leverage existing mechanisms within the RDBMS for view matching, and updating secondary indexes or materialized views [8, 14]. Since the semantic cache is entirely in remote memory, a remote node failure can be dealt in various ways. One approach is to invalidate the cache. Another alternative is to leverage the RDBMSs transaction log, which logs every update in the RDBMS, and REDO logic to recover the structures by replaying the log on another remote server with available memory.

Such in-memory semantic caching introduces many interesting challenges. First, the query optimizer's cost model, which is traditionally calibrated for on-disk structures, needs to be re-calibrated since seeks into remote memory is significantly faster compared to scanning HDDs or SSDs. Second, techniques to automate the selection of suitable structures is an interesting physical design problem along with policies to invalidate or update them. While an in-depth study of in-memory semantic caching in the RDBMS is beyond the scope of this paper, we empirically demonstrate the performance gains of integrating such a cache into the RDBMS.

3.4 Priming the Buffer Pool

Cloud database systems run multiple copies of a database for high availability. Typically, the primary copy processes update and read transactions and one or more secondary copies are kept up-to-date via logical or physical replication. If physical replication is used, the databases are kept identical at the page level. A *primary-secondary swap*, i.e., a secondary being promoted to a primary, is a common operation in such clusters due to failures and load balancing. While some failures are unplanned (e.g., machine crashes), there are a large number of swaps that are planned, for instance during load balancing, planned software or hardware upgrades etc. One major challenge with such swaps is that the application workload resumes on the newly-elected primary with a *cold* buffer pool, thus resulting in significant performance degradation

until the buffer pool is warmed up. For physically-identical databases synchronized with physical replication, wire-speed RDMA transfers can be used to warm up the buffer pool of the newly-elected primary (S_2) using contents of the buffer pool of the old primary (S_1). This priming (or warming-up) of the buffer pool at S_2 can be proactive, where S_1 pushes the pages to S_2 or reactive, where S_2 fetches the pages from S_1 on-demand as the workload accesses them. The first scenario is similar to a push technique studied in the context of database and VM migration [9, 11]. The latter scenario is similar to the cache extension scenario described above. In either case, fast access to remote memory using RDMA can significantly reduce the impact of such planned primary-secondary swaps on application workloads.

4. DESIGN AND IMPLEMENTATION

Any RDBMS that aims to exploit remote memory must address several important design challenges. In this section, we first discuss these challenges and rationalize our choices with the goal of efficiency and ease of integration into a mature RDBMS. We then describe an implementation in Microsoft SQL Server, referred to as SQL Server for brevity, that embodies these choices.

4.1 Design Choices and Rationale

4.1.1 Abstraction for Remote Memory

The first design question is the abstraction to use for remote memory, i.e., how is remote memory exposed in the memory hierarchy on an RDBMS. We first discuss four alternatives and explain the abstraction we choose in our implementation.

Byte-addressable memory. Local memory is often accessed using byte-addressing and it is natural to expose remote memory also as byte-addressable memory, similar to that in distributed shared memory (DSM) [1, 5]. This allows existing applications to seamlessly extend their memory from local to remote, similar to the virtual memory abstraction exposed by the OS. However, it requires OS support to intercept memory accesses for address translation, which negates the kernel bypass benefits of RDMA. Moreover, it limits the RDBMS's ability to control which data is stored locally and which ones are moved to remote memory.

In-memory blocks. Many applications that manage their own memory, such as database systems, acquire memory from the OS in blocks. Therefore, remote memory can be exposed as blocks of fixed but configurable sizes. An application requests a block of remote memory which is identified uniquely. The application accesses a block using its unique identifier, an offset into the block, and the size of the data to be read or written. This abstraction allows the application to selectively read/write individual bytes within a block without having to read/write the entire block.

In-memory files. Database systems are designed to operate efficiently with files. Many in-memory structures in a database can be mapped to in-file representations, which makes a file abstraction suitable for databases. An in-memory file is a common abstraction supported by many operating systems [32], e.g., `ramfs` in Linux `Ramdisk` in Windows allow mounting a part of physical memory as a file system. Exposing a familiar file API, remote memory can be abstracted as in-memory files.

In-memory Key-value store. Caching variable-sized objects which are stored and retrieved as whole objects is a common requirement for many applications. A key-value store, where the keys and values are stored in remote memory, is another abstraction to expose remote memory. The key-value store hides the details of memory management and processing the remote reads and writes. The application only interacts through a get/put API which is common with key-value stores.

Our implementation builds on top of an in-memory block abstraction which provides flexible access to memory. To enable easy integration of remote memory into SQL Server, our implementation builds a lightweight file shim over fixed-size in-memory blocks, that exposes a subset of the standard file APIs but are sufficient for the scenarios we consider; the file API is specified in Table 2.

4.1.2 Protocols to Access Remote Memory

The protocol to access remote memory is also critical for performance. One class of protocols can be based on the standard TCP/IP stack in a client-server design using network sockets. TCP/IP uses the remote CPU, involves the OS at both the source and destination of the transfer, copies data from user space to the kernel, and incurs additional protocol overheads. However, the benefit is the standard implementations in both the Windows and the Linux platforms. Due to the common use of TCP-based protocols, we consider one such protocol as a baseline in our study. This baseline uses Server Message Block (SMB) [36], a network file sharing protocol (over TCP/IP or other network protocols), which has been supported in Windows since Windows 95. Though SMB is not directly associated with accessing memory, if remote memory is exposed as an in-memory file system, then SMB can be used to transfer data from remote to local memory over TCP/IP.

RDMA protocols bypass the OS and avoid TCP/IP overheads. However, the protocols for RDMA transfers are specific to the platform. For instance, the Windows ecosystem supports the Network Direct Service Provider Interface (NDSPI) where hardware vendors can implement advanced capabilities of their networking devices, such as RDMA over Infiniband. The Infiniband standard supports a variety of send/receive verbs as interface definitions, which has corresponding mappings to NDSPI. Another alternative is to use SMB 3.0 that supports RDMA transfers using SMB Direct. If the network adapter is RDMA-capable, SMB Direct uses RDMA. Our implementation uses RDMA reads and writes implemented over NDSPI which was the most efficient among the alternatives.

4.1.3 Synchronous vs. Asynchronous Operations

Database systems model I/O operations to the disk or over the network as *asynchronous* operations. This allows the system to hide I/O latencies by interleaving CPU activity on other threads. If a thread does not have any CPU activity after issuing an I/O, it yields the CPU, resulting in a context switch. The thread is switched back in after the OS processes the I/O completion. A remote memory access is an I/O and hence it is natural to treat such accesses as asynchronous. While such asynchronous accesses are efficient to hide disk I/O latencies, which is of the order of milliseconds, remote memory access latencies are in the order of tens of microseconds. A context switch itself consumes a few microseconds. In addition, pollution of the processor's cache lines and the delay between when the I/O is completed to when the thread is switched back in makes the overheads of a context switch comparable to remote memory access latencies. Therefore, even though remote memory access via RDMA is an I/O, we found treating it as *synchronous* accesses, similar to accesses to local memory, is important to enable the RDBMS leverage remote memory more effectively. This synchronous access model is efficient since it prevents unnecessary context switches and allows the database to process RDMA completions faster, thus resulting in significantly lower access latencies and higher throughput for remote memory accesses. While the synchronous model suits the small transfers in RDBMSs, an adaptive approach that switches to asynchronous when transfer

Table 1: Summary of the design choices and alternatives.

Design Choice	Alternative Chosen	Rationale
Remote memory abstraction	In-memory blocks with a file API shim	Ease of integration into existing DBMS engines
Protocol	NDSPI	Least overheads
Sync vs. async remote accesses	Synchronous	Avoid context switches, improve latency
Registering memory regions	Pre-register memory regions	Registration incurs fixed initialization cost
Fault tolerance	Best-effort	Correctness unaffected, simpler implementation

latencies are higher than typical context switches will be more general and is an interesting direction for future work.

4.1.4 Registering Memory Regions

RDMA transfers require that the source and the destination memory regions (MRs) be registered with the NIC. This registration of MRs can be done on-demand before every transfer, or large blocks of MRs can be pre-registered upfront at both the source and the destination. Both approaches have trade-offs, and present an important design choice. Considering a typical page in an RDBMS is $8K$, registering an $8K$ page has a latency of 50μsec on our hardware, which is of the same order as the cost to transfer the page itself. In addition, registering the page on-demand at the destination implies the destination server's CPU also needs to be involved to initiate the transfer, which partly defeats the benefits of RDMA transfers. Hence, efficient management of MR registration is critical to leverage the benefits of RDMA [13,34]. Our implementation preregisters in-memory blocks during initialization. Preregistration is straightforward if the memory region used for transfers is a contiguous block in physical memory. However, caches in an RDBMS, such as the buffer pool, are often not contiguous in physical memory, which introduces an implementation challenge for preregistration; we describe our solution in Section 4.2.

4.1.5 Fault Tolerance and Availability

In a cluster of commodity servers, failures and server restarts are common. When one server S_1 accesses memory of another server S_2, a failure of S_2 can impact S_1, depending on the fault-tolerance and availability guarantees provided by the remote memory abstraction. Exposing remote memory as *best-effort*, where failure of a server can make the memory unavailable, is the simplest abstraction to support, and is sufficient if an application uses remote memory only as a performance optimization. That is, if remote memory becomes unavailable, the RDBMS can continue to operate correctly, though likely with degraded performance. While it is possible to expose remote memory as a highly-available and durable abstraction, it increases the complexity of the implementation and also results in less performance gains due to the need to replicate data or write to non-volatile storage [27]. Since our scenarios can tolerate remote memory failures without affecting correctness, our implementation uses remote memory with best-effort fault tolerance and availability guarantees.

4.2 Implementation in SQL Server

We now present a detailed description of our implementation of the proposed remote memory abstraction in SQL Server deployed in a cluster environment. Our implementation embodies the design choices we discussed earlier which are also summarized in Table 1. Figure 1 presents a simplified illustration of the architecture we implement. The figure shows two sets of servers: the servers running the database process which are experiencing high memory

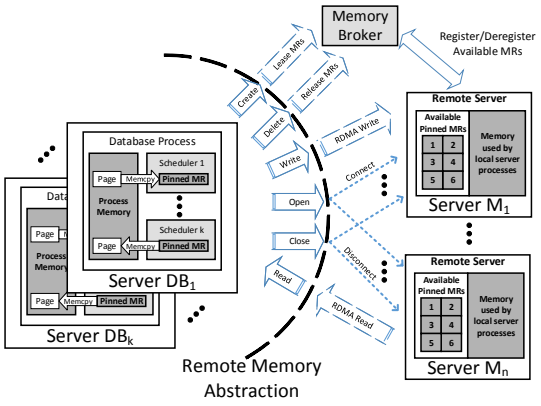


Figure 1: Integrating remote memory into an RDBMS.

pressure ($DB_1, \ldots, DB_k$), and the servers with available memory which can be accessed via RDMA ($M_1, \ldots, M_n$). For simplicity, the figure abstracts out the other processes running on the servers ($M_1, \ldots, M_n$). These servers can also run a database process, which will typically be the case in a cloud database service. The only difference between ($M_1, \ldots, M_n$) and ($DB_1, \ldots, DB_k$) is there is available memory in the former set and unmet memory demand in the latter set.

Brokering available memory. Each server in the cluster reports the available and unused memory to a memory broker that tracks memory availability throughout the cluster. A server M_j with available memory divides its memory into configurable fixed-sized memory regions (MRs) depicted as numbered blocks in Figure 1. A memory brokering proxy process on M_j determines available memory not committed to any local process, pins the available MRs, registers them to the local NIC, marks them unavailable from the OS's perspective, and registers the MRs with the memory broker. Conceptually, the broker now controls the privilege to read and write to those MRs, which it can assign to any other server with unmet memory demands. Note that we do not steal memory committed to processes locally on a server, hence the memory on M_j registered with the broker cannot be committed to any local process on M_j. The memory brokering proxy at each M_j registers for OS memory pressure notifications such that if a local process on M_j requests additional memory from the OS, the proxy will detect such memory pressure. To prevent the OS from paging local applications, the proxy on M_j requests deregistration of one or more MRs from the memory broker. Once the memory broker frees the lease on the MR, they are freed to the OS at M_j which can allocate the pages to local processes. These memory notifications and adaptive memory brokering is a standard API supported by many OSs [25] and is transparent to applications running on M_j.

A database server with unmet memory demand (DB_i) can request the broker for a lease to a remote memory region. This lease provides DB_i exclusive access to the region. The memory broker's design is similar to many standard leasing mechanisms based on Zookeeper [16] which provides a fault-tolerant and highly-available mechanism to obtain timed leases. Many systems, such as the YARN scheduler [49] or the master in HBase which manages the mapping of regions to servers similar to that of Bigtable [6], have a similar use of Zookeeper. When DB_i requests a lease on an MR, the broker determines which server M_j has an unleased and available MR, registers this mapping in a lookup table, and creates the metadata for the lease. Zookeeper guarantees fault tolerance and high availability of this metadata for memory brokering. On obtaining the lease, DB_i directly communicates with M_j

Table 2: Simplified file API and RDMA operations.

File Operation	RDMA Implementation
Create (ServerEndpoint, Size)	Obtain lease on MRs
Open	Connect to server(s)
Read/Write (Offset, Size)	RDMA read/write
Close	Disconnect from server(s)
Delete	Relinquish lease on MRs

using the RDMA read/write commands. Since the memory broker is not in the data transfer path from M_j to DB_i, such a design can scale to thousands of servers [6, 49]. In addition, since the broker's state is all stored in Zookeeper, a broker's failure is easy to tolerate by electing a new broker node orchestrated using Zookeeper. Before the lease for an MR expires, DB_i has to renew the lease. If the lease is successfully renewed, DB_i continues to use the MR. Otherwise, DB_i is forced to release the remote MR. DB_i can request a lease on another MR or continue with local disks. DB_i can also voluntarily release the lease on a remote MR if sufficient memory is available locally. There are many more details, such as how to deal with failures of both DB_i or M_j and the policies for lease management. These aspects are common of many distributed resource and lease negotiating systems, such as YARN, HBase, and Bigtable, and are omitted for brevity. Note that if a lease expiration is forced due to a failure or other policies, the correct operation of DB_i is not compromised and the DBMS can continue query execution, albeit with degraded performance.

Scenarios. SQL Server already has support to extend the buffer pool to an SSD to leverage the higher random I/Os available in SSDs. SQL Server implements the extension as a file by serializing the buffer pool contents. This logic to serialize buffer pool contents to a file is also leveraged to prime the buffer pool on the secondary. SQL Server spills temporary data into a file. Finally, structures in the semantic cache can also be stored and serialized as in-memory files. Therefore, exposing remote memory as a lightweight file API shim significantly reduces the changes needed in SQL Server to leverage benefits of remote memory. For all our scenarios, the database engine uses higher-level locks and latches to synchronize all I/Os. Hence, the in-memory files do not need many properties of files in a classic file system. A lightweight file API supporting reads/writes of a certain size from/to a specific offset is sufficient.

Implementing a lightweight file API. Table 2 summarizes the file API and the corresponding operations, which are also illustrated in Figure 1. To create a file of the specified size, a server DB_i requests a lease on MRs corresponding to the file's size. The broker provides the mapping of which MR is on which server M_j. On file open, DB_i initiates a connection and sets up an RDMA flow to all M_js that it will write to or read from. When a read or write operation is issued, we first translate the file offset into an MR backing the file and an offset within the MR. We then issue an RDMA read or write operation to the MR. A file close action terminates the connections. When a file is deleted, we free the lease on the corresponding MRs.

Our implementation of the RDMA read/write actions utilizes the C++ NDSPI API which is natively supported by the Mellanox NICs used in our implementation. The NDSPI library exposes RDMA accesses as an asynchronous operation which the database server can treat as asynchronous I/Os. However, as discussed in Section 4.1.3, treating RDMA requests as asynchronous I/Os is not efficient. Therefore, we use synchronous RDMA requests where if a thread does not have any additional CPU activity after issuing an RDMA request, it spins for a few microseconds until the RDMA request completes. If the request does not complete within a few tens of microseconds, we can switch to asynchronous mode to avoid spinning for long durations if the RDMA transfer latency increases. Such an adaptive strategy is left for future work.

Table 3: System configurations

CPU	Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20 GHz (20 cores and 40 logical processors.)
Memory	384 GB DDR3, 1866 MHz
HDD	1 TB 7.2K RPM Near-line SAS 6 Gbps
SSD	400 GB SAS SLC 6 Gbps
RAID	Dell Perc H710P SCSI
Network Adapter	Mellanox ConnectX-3 VPI IPoIB (56 Gbps)
Operating System	Windows Server 2012 R2
Database System	SQL Server

Preregistration of MRs. As discussed in Section 4.1.4, another important consideration is preregistering the MRs with the NIC. Preregistration is straightforward when the memory region is a contiguous block of memory, which is the case with server (M_j) that has unused memory. We register large contiguous blocks of memory when the available MRs are registered with the broker. Preregistering MRs on the RDBMS server (DB_i) introduces a new challenge, especially for the buffer pool which in SQL Server is not a contiguous block of memory. First, the buffer pool might grow (or shrink) dynamically. Second, a single memory allocator services memory requests from the buffer pool as well as other consumers of memory within SQL Server. Hence, buffer pool pages can be interspersed with memory consumed by other engine components. Therefore, statically-registering the entire buffer pool could amount to registering the entire address space of the DBMS process, which comprises large chunks of memory which will never be used for RDMA transfers.

To avoid the cost of dynamic registration for every RDMA transfer, we use a preregistered staging buffer in database server's address space. The database server has multiple CPU schedulers and to maximize data parallelism each scheduler issues I/Os in parallel. We create a staging buffer for each scheduler, depicted as "Pinned MR" in Figure 1. When a scheduler evicts a buffer page and transfers the page to the extension on the remote server, it first copies the page into its local staging buffer using a `memcpy` operation. Registering an 8K page has a latency of 50μsec while `memcpy` has a latency of 2μsec. Therefore, introducing the additional `memcpy` actually results in a design that is far more efficient than dynamic registration of the buffer pool pages. The page in the buffer pool becomes available to hold another database page immediately after the `memcpy`. The scheduler initiates an RDMA write from the staging buffer to the appropriate location on the remote server. The staging area's buffer can be reused after the RDMA write completes. The pinned MR's size is a trade-off between memory overhead of the staging area and the number of pending RDMA transfers. We used 1MB for up to 128 pending RDMA transfers per scheduler which was tuned for our setup allowing us to sustain sufficient concurrent RDMA transfers to saturate the network. The "Pinned MR" is also used to fetch buffer pool pages from remote memory using RDMA reads.

5. EXPERIMENTAL SETTINGS

In this section, we present the system configurations, the benchmark workloads and the alternative designs evaluated in our experiments. For brevity, we use buffer pool extension as BPExt, and storage with spilled temporary data as TempDB.

5.1 System Specifications

We use a cluster of 10 servers each of whose hardware configuration is shown in Table 3; the servers are connected using a top-

Table 4: Summary of workloads

Workloads	Data Size	Local Mem	BPExt Size	TempDB Size	Concurrency
RangeScan	110GB	32GB	128GB	8GB	80
Hash+Sort	227GB	256GB	N/A	320GB	1
TPC-H	840GB	64GB	256GB	64GB	5
TPC-DS	900GB	64GB	256GB	64GB	5
TPC-C	168GB	16GB	32GB	8GB	2000

of-rack Mellanox FDR MSX6036F-1BRR Infiniband Switch. We explore remote memory for scenarios where the DBMS is designed to use disks. Therefore, the performance of the I/O subsystem will have a significant impact on the benefits of using remote memory. We therefore use a few different configurations to control the bandwidth and access latencies of the I/O subsystem. We use a hardware RAID controller and a RAID 0 setup where data is striped across multiple drives. We vary the number of HDDs, or spindles, in a RAID 0 partition as 4, 8 and 20, respectively, and show how the performance of I/O subsystem affects the throughput/latency of the workloads. Unless otherwise specified, the default HDD set up has 20 spindles. For brevity, we refer to the RAID 0 I/O subsystem as HDD to differentiate from an SSD which is configured with its stand alone partition and used in some configurations.

5.2 Workloads

We use various micro benchmarks to test the scenarios in isolation, and then use standard benchmark workloads to quantify the overall benefits of using both optimization for more complex workloads. Table 4 summarizes the workloads that are evaluated in this paper. We measure native performance of the I/O subsystems and remote memory accesses using a standard benchmark SQLIO [38]. We use a ***RangeScan*** workload to micro benchmark buffer pool scenarios of extension and priming, and a ***Hash+Sort*** workload which is designed to stress TempDB by generating temporary data with Hash and Sort operators on large amounts of data. We also use a number of standard benchmark workloads: two decision support benchmarks (TPC-H [47] and TPC-DS [48]) and one on-line transaction processing benchmark (TPC-C [46]). Each benchmark has a scale-factor that determines the data sizes. The scale for TPC-H is 200, and the scale for TPC-DS is 300. For TPC-C, the number of warehouses is used to scale the database, which we set to 800. For the decision support workloads, we used SQL Server's Database Engine Tuning Advisor (DTA) to tune the physical design to avoid unnecessary I/Os and ensure we have a reasonably-tuned system. We use the query generator tools provided with TPC-H and TPC-DS benchmarks to generate the workload. DTA uses this workload to recommend index structures which we subsequently built. For TPC-C, we use the indexes as per the benchmark specification. The final database sizes after building the indexes is shown in Table 4. We now describe the two micro benchmarks in a bit more detail.

5.2.1 RangeScan

This benchmark is designed to stress the BPExt and priming scenarios. This workload generates queries that scan a range in a synthetically-generated `Customer` table that has the same schema as the `Customer` table of TPC-H. This table contains 500 million rows, and the average size of each row is 245 bytes. A clustered index is built on the `custkey`. The range scan query is of the form:

```
SELECT sum(acctbal) FROM customer
WHERE  custkey >= @start
AND    custkey <  @start + @range
```

We use two variants of this query: a read-only variant which calculates the average account balance for a range of rows starting from a start key, and an update variant which updates the account balance for the range selected. Since the table has an index on `custkey`, it results in seeks on that index. We fix `@range` to 100 to simulate short read-only queries which access three database pages on average. The `@start` parameter is randomly generated from a uniform distribution. With up to 80 concurrent threads issuing this query, the uniform access pattern creates a lot of churn in the buffer pool by accessing almost the entire database which does not fit in local memory. As a result, this query is suited to stress the BPExt by frequently moving pages to and from the extension. If the parameter is selected from a skewed or a hotspot distribution, it results in a small working set, which if in memory results in low tail latencies, but if not in memory, results in longer tail latencies. This setting is suited to test the priming scenario.

5.2.2 Hash+Sort

This query is designed to stress TempDB. It joins the `lineitem` and `order` tables from the TPC-H benchmark and returns the top $100,000$ rows sorted by `extendedprice`.

```
SELECT   top 100000*
FROM     lineitem l JOIN orders o
ON       l.orderkey  = o.orderkey
ORDER BY l.extendedprice
```

We built a clustered index on the `orderkey` of each table; Figure 2 shows the resulting execution plan for the query where the tables are joined with a hash join, which builds a hash table that will spill to TempDB if enough memory is not available, followed by a sort on the joined results, which will again use an external sort algorithm using TempDB if enough memory is not available. We assign 256 GB local memory to the database server which is enough to cache the data scanned by the query, thus making the reads and writes to TempDB during the Hash and Sort the bottleneck.

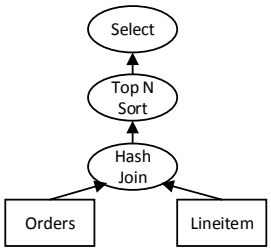


Figure 2: Execution Plan for Hash+Sort Query

5.3 Alternative Designs Evaluated

Table 5 summarizes the various design alternatives we consider for our empirical evaluation. The ***HDD*** and ***HDD+SSD*** are two baselines in which there is no remote memory available and the DBMS server uses local HDD or SSD for unmet memory demand. In the HDD setup, we disable BPExt since storing the BPExt on HDD which also has the data files degrades performance due to lower random read performance of HDD compared to SSD. For the data analysis workloads (such as *Hash+Sort*, TPC-H and TPC-DS), most of the I/Os are sequential. Considering that the HDD setup uses data striped across multiple disks (varied as $4, 8, 20$) in a RAID 0 configuration, the sequential throughput of the HDD setup can reach more than one GB/s, which is higher than the SSD's random access throughput. If the BPExt is enabled, sequential accesses are transformed to random accesses to the extension on the SSD. Therefore, for data analytics workloads, we disable BPExt for the *HDD+SSD* as well. For OLTP workloads (such as *RangeScan* and TPC-C), as most of the I/Os are random, and the SSD's random I/O throughput is higher than the HDD, we enable the BPExt for *HDD+SSD* and store it on SSD.

Table 5: Design alternatives evaluated.

Designs	Data Files	TempDB	BP Ext	Protocol
HDD	HDD	HDD	N/A	N/A
HDD+SSD	HDD	SSD	SSD	N/A
SMB+ RamDrive	HDD	Remote Memory	Remote Memory	SMB
SMBDirect +RamDrive	HDD	Remote Memory	Remote Memory	SMB Direct
Custom	HDD	Remote Memory	Remote Memory	NDSPI
Local Memory	HDD	SSD	N/A	N/A

Our abstraction of in-memory files can be implemented using various protocols to access remote memory and off-the-shelf components. We consider two such designs: ***SMB+RamDrive*** and ***SMBDirect+RamDrive***. We use a third party software that creates a fully-functional RamDrive on the remote server. SMB accesses RamDrive using TCP/IP (*SMB+RamDrive*). SMBDirect leverages RDMA for data transfer (*SMBDirect+RamDrive*). ***Custom*** is our implementation of the lightweight file API with data transfer using NDSPI (see Section 4.2).

Finally, to determine the best possible performance, we also consider a design where memory of equivalent size as remote memory is available locally at the server. We call this design ***Local Memory***. This alternative helps us to determine the overheads of using remote memory compared to when it is available locally.

6. EXPERIMENTAL RESULTS

We now present a thorough empirical analysis of our implementation to evaluate the effectiveness of remote memory in a cluster setting where a database service is deployed. Our experiments use a modified un-optimized build of SQL Server and hence our performance numbers should not be treated as official benchmark results. We first use a micro benchmark to evaluate the I/O throughput and latency for the different alternatives (Section 6.1), followed by micro benchmarks for BPExt (Section 6.2), TempDB (Section 6.3), semantic caching (Section 6.4), and buffer pool priming (Section 6.5). Appendix B presents additional experiments demonstrating the end-to-end benefits using industry-standard TPC benchmarks, impact of varying the amount of local memory, and varying the number of database servers accessing a memory server. Since the benefits of remote memory are significant only when local memory is insufficient to meet the workload's demands, our evaluation also focuses on configurations with memory pressure. Following are the key takeaways:

Custom* significantly outperforms *HDD+SSD. In all experiments when the workload's memory demand exceeds the available local memory, *Custom* significantly outperforms *HDD+SSD* in both throughput and latency. For some TPC-DS queries, *Custom* achieves up to $100\times$ improvement in terms of latency.

Custom* outperforms *SMBDirect+RamDrive. Compared to *SMBDirect+RamDrive*, which requires no code changes, *Custom*'s results in around $3.4\times$ improvement in random I/O throughput. For end-to-end SQL workloads, the improvements are in the range of $10 - 40\%$. This is mainly because for complex SQL workloads, CPU activity of query processing interleaves data accesses, which does not generate enough demand for remote memory to benefit from the lower overheads of *Custom*. That is, the workload are CPU-bound even with *SMBDirect+RamDrive*. Note that using CPU-efficient technologies such as column store and vectorized processing will further improve the benefits of remote memory, which is an interesting direction of future work.

Custom* and *SMBDirect+RamDrive* outperform *SMB+RamDrive. The experimental results also show that both *Custom* and *SMBDirect+RamDrive* outperform *SMB +RamDrive*, which demonstrates the benefits of accessing remote memory via RDMA compared to TCP. This experiment highlights the importance of the choice of protocol for RDMA transfers.

RDMA has negligible performance impact on remote server. Accessing available memory via RDMA results in negligible impact on performance of workloads executing on the remote server providing the memory, compared to accessing memory over TCP which results in $\sim 10\%$ overhead on both throughput and latency of workload on remote server.

Custom* approaches performance of *Local Memory. For most of workloads, the performance of *Custom* is within $10 - 20\%$ of *Local Memory*. That is, even though accessing remote memory via RDMA (10μsec) is slower than accessing local memory (0.1μsec), for classic database systems, the difference of accessing data cached in local memory compared to remote memory is surprisingly small. The reason is that classic memory-optimized RDBMSs are not as optimized as purely in-memory databases and hence cannot effectively saturate the memory bandwidth. Analyzing RDMA's impact on in-memory technologies is beyond the scope of this paper.

Comparable improvements with multiple servers. In a cluster where a database server uses memory pooled aggregated from multiple servers or when multiple remote servers access memory available at one server, the overheads are negligible compared to when memory is obtained from one server.

6.1 I/O Micro benchmark Performance

We now present the read I/O throughput and latency using a micro benchmark. We first analyze the performance all alternatives with two servers, one database server (DB_i) and another memory server (M_j). We subsequently evaluate the performance of *Custom* with one DB_i accessing multiple M_js, and one M_j serving multiple DB_is. In a cluster with multiple servers, a database server might be accessing memory of multiple remote servers. Similarly, multiple database servers can concurrently access memory available at one server. Our multi-server experiments characterize performance for both scenarios.

For the two server setting, we evaluate the read performance for HDD, SSD, *SMB+RamDrive*, *SMBDirect+RamDrive* and *Custom*. For HDD, we evaluate the performance of our RAID 0 configuration with 4, 8, and 20 spindles, respectively. We evaluate both the random read and sequential read performance using SQLIO, a disk subsystem benchmark tool [38]. For random read, 20 threads issue the read requests of 8 KB pages concurrently. For sequential read, 5 threads issue read requests for blocks of size 512 KB.

Figure 3 plots the read throughput and Figure 4 plots the I/O latency for the different alternatives. As can be seen in this figure, the sequential read throughput of *Custom* is almost equal to that *SMBDirect+RamDrive*, but is higher than *SMB+RamDrive*. The sequential read throughput of both HDD and SSD is slower than the sequential read of the three remote memory designs, which shows the opportunity in exploiting remote memory for databases. In addition, since the HDDs on our server are configured as RAID 0, its sequential read throughput is much higher than the SSD. For random read, the throughput of our custom implementation is much higher than that of *SMBDirect+RamDrive*, which may be due to the overhead that is introduced by the RamDrive and the SMBDirect protocol. Access latencies for remote memory designs are also considerably lower than that of designs using locally-attached HDDs or SSDs. In addition, the low overheads of *Custom* is also evident in the lower latencies, especially for random page accesses.

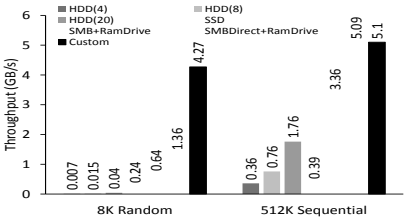


Figure 3: I/O micro benchmark throughput.

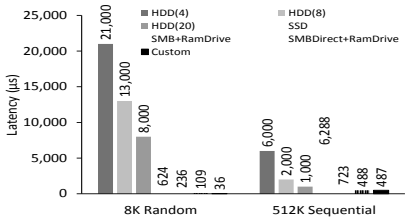


Figure 4: I/O micro benchmark latency.

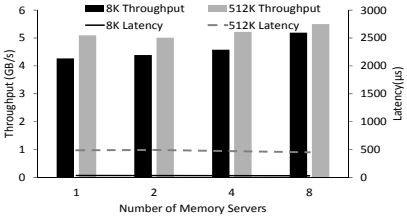


Figure 5: I/O performance with multiple memory servers.

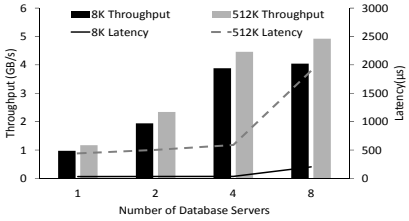


Figure 6: I/O performance with multiple database servers.

Figure 5 plots the I/O throughput and latency when one database server DB_i accesses remote memory from multiple memory servers M_j. The total amount of remote memory remains constant as we vary the number of memory servers from 1 to 8 which is plotted in the x-axis. The bars correspond to the throughput plotted on the left vertical axis and the lines correspond to latency plotted on the right vertical axis. As is evident from the figure, even when the network bandwidth of DB_i is saturated (which is the case even with one M_j), there is negligible impact on throughput and latency for accessing remote memory as we vary the number of M_js. In fact, with the higher parallelism available with 8 M_js, the latency of random I/Os is lower. This demonstrates our implementation can effectively leverage memory pooled across multiple servers.

Figure 6 plots the I/O throughput and latency when multiple DB_is access memory from on M_j. Each DB_i is accessing a fixed amount of memory and the number of accesses per second is tuned such that the NIC's bandwidth at M_j is saturated with four DB_is; we increase the number of DB_is from 1 to 8 which is plotted on the x-axis on Figure 6. The bars plot the aggregate throughput of all DB_is along the left vertical axis and the lines plot the average latency of all DB_is along the right vertical axis. There are two important observations. Before M_j's NIC was saturated, the aggregate throughput scales almost linearly with negligible change in latency as we increase number of DB_is. Once the NIC is saturated, increasing the number of DB_is results in higher latency, which is expected since contention for the resources increases. With four DB_is, the latency is comparable to that of Figure 4 where one DB_i was saturating the bandwidth of one M_j. When the link is saturated with 8 servers, the peak throughput for random I/Os in Figure 6 is comparable to that in Figure 3. Note that in this experiment, our goal was to saturate the NIC with a few servers to help us demonstrate linear scaling before saturation and the effect of contention after saturation. However, in a real cluster setting, it is unlikely that all remote database servers will continuously and concurrently overload the memory server, thus potentially allowing even more database servers to use memory available at an M_j.

6.2 Impact of Remote Memory on BPExt

6.2.1 Performance of RangeScan

We first measure the performance of *RangeScan* when 20% of the requests are updates; Figure 7 plots the throughput and Figure 8 plots the latency. Performance of the different alternative designs follow the trend observed with I/O performance discussed in Section 6.1: all the three scenarios that utilize remote memory have higher throughput compared to storing BPExt on local SSD. Furthermore, the throughput of *Custom* is only 10% lower compared to the ideal *Local Memory* scenario. It shows that with RDMA, caching pages in remote memory is a significantly better alternative when local memory is unavailable and the DBMS uses disks. Updates cause appends to the transaction log which is stored in the HDD. Therefore, as the number of spindles increases, the throughput increases and the latency decreases.

To eliminate this impact, we repeat the experiment without any updates; Figures 9 and 10 plots the throughput and latency for the corresponding experiment. Data is buffered into either SSD, remote memory or local memory depending on the alternative considered, and reading data from these structures is independent of the HDD performance. As a result, the throughput of *HDD* increases as the number of spindles increases, but the throughput of other methods remains the same in different spindle settings.

To better explain the observed behavior, we present a drill down on this experiment in terms of the I/O throughput, latency, and CPU utilization for *HDD+SSD*, *SMBDirect+RamDrive* and *Custom* for a 100 second period during the experiment (Figure 11). As seen in Figure 11(a), the I/O throughput is about 900 MB/s for *Custom*, which is much lower than the raw I/O throughput (∼5 GB/s). The reason is that during the execution of this workload, as reading a data page from remote memory is so fast, I/O is no longer the bottleneck, instead it shifts to CPU. As shown in Figure 11(b), the CPU is almost entirely utilized for both *SMBDirect+RamDrive* and *Custom*, while the CPU utilization is only about 20% for *HDD+SSD*. Figure 11(c) shows the I/O latency of reading a page from BPExt during the executing of the workloads. The benefits of the *Custom* implementation is evident where the I/O latency of *Custom* is much lower (13μsec) compared to *SMBDirect+RamDrive* (272μsec). The higher latency can be attributed to how I/O is treated and overheads of a fully functional file system and the SMB Direct protocol for RDMA transfers. Without any code changes, SQL Server treats accesses to BPExt as an asynchronous I/O since BPExt is typically stored in a disk. For asynchronous I/Os, even though the I/O can be completed within 10μsec, the thread that issues this I/O must wait to be scheduled. By contrast, *Custom* treats remote memory accesses as synchronous operations.

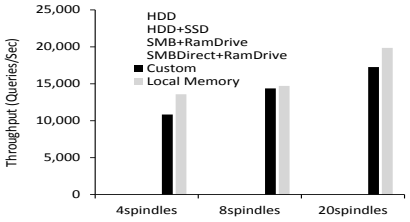


Figure 7: RangeScan throughput (20% updates).

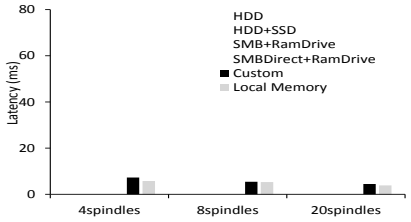


Figure 8: RangeScan latency (20% updates).

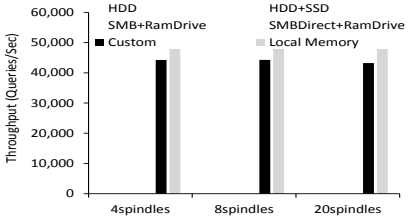


Figure 9: RangeScan throughput (no updates).

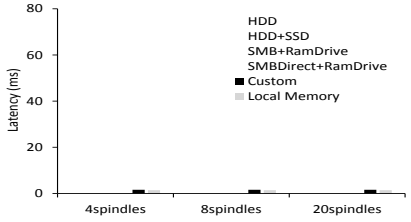
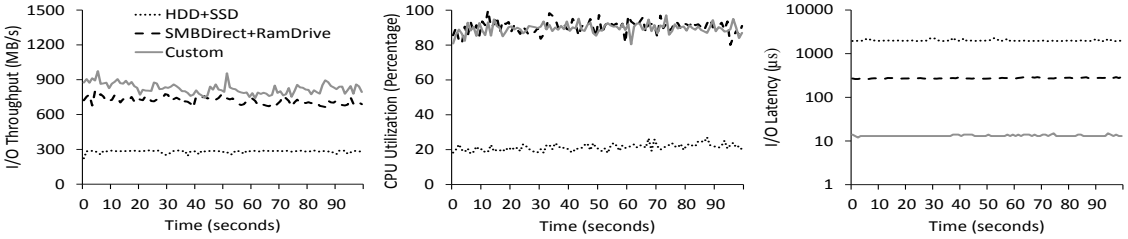
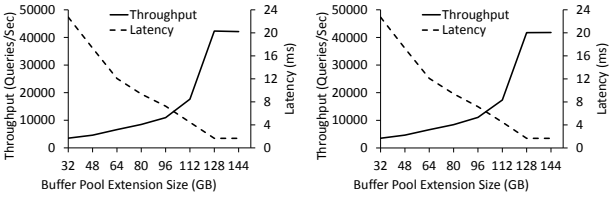


Figure 10: RangeScan latency (no updates).



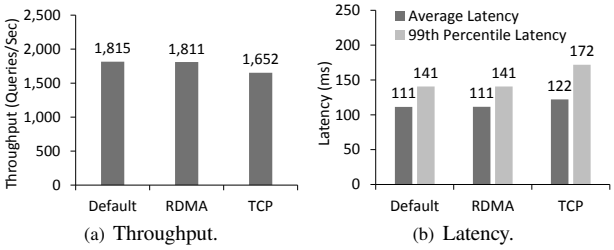
(a) I/O throughput (b) CPU utilization (c) I/O latency

Figure 11: Drill down of RangeScan.



(a) One memory server. (b) Multiple memory servers.

Figure 12: Impact of varying BPExt size.



(a) Throughput. (b) Latency.

Figure 13: Impact on remote server.

6.2.2 Impact of Memory Size

The benefits of remote memory is dependent on the available local and remote memory. In this experiment, we keep the local memory at the server constant and vary the amount of remote memory allocated to the BPExt. Appendix B reports the experiment where we vary the amount of local memory available. When varying the remote memory, we report the results for two settings: when all the remote memory is on one server (Figure 12(a)) vs. the memory is spread evenly on multiple servers (Figure 12(b)) such that larger the size of the BPExt, higher is the number of M_js accessed. We vary the number of M_js from 2 to 9 where each server contributes 16GB of remote memory. The throughput and latency numbers depend on how much remote memory is available and independent of the number of servers contributing the memory, which is not surprising given the I/O performance reported in Figure 5. Each figure plots the throughput on the left vertical axis and latency on the right vertical axis, while the x-axis plots the increase in BPExt size. Both figures look identical, which demonstrates that remote memory continues to be equally beneficial even when it is pooled from multiple servers. In SQL Server, the size of BPExt is not allowed to be smaller than the local memory size. Thus, the size of BPExt starts from 32 GB. Because the range scan query selects the start of the range from a uniform distribution that spans the table, it accesses all data with equal probability. Therefore, until the point where the entire data is cached in (local or remote) memory, the workload can always benefit from more remote memory. That is, as we increase remote memory, the throughput increases and the latency decreases. The rate of improvement is lower for BPExt sizes below 96 GB when I/O is still the bottleneck. As the BPExt size reaches close to the data size, most of the data pages are cached, resulting in significant throughput and latency improvements.

6.2.3 Impact on Remote Server

One important benefit of RDMA, in contrast to TCP, is that memory accesses and transfers to do involve the CPU of the remote server, thus potentially having minimal impact on any workload executing on the remote server. In this experiment, we quantify how the performance of the server with available memory is affected when another server accesses the memory. In this experiment, we run workloads on two servers: S_A and S_B. S_B's unused mem-

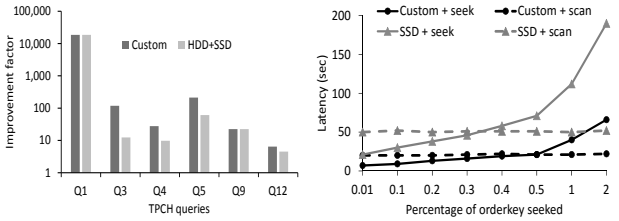
ory is used to store S_A's BPExt, accessed through either RDMA or TCP. Both servers are executing the *RangeScan* workload without updates. We measure the performance of the workload on S_B. S_A's workload corresponds to the default configuration which generates high memory demand. Since we want to measure the impact of RDMA's ability to bypass the CPU, S_B's workload uses different parameters: (1) S_B uses 128 GB memory for the RangeScan workload, which is large enough to hold the entire data set in memory; and (2) the range size for *RangeScan* is set to $10,000$. These changes make S_B's workload to be CPU-intensive (with close to 100% CPU utilization). Figure 13(a) shows the throughput of the workload on S_B. *Default* corresponds to the scenario where S_A does not use S_B's available memory. As can be seen from the figure, the throughput is unaffected when remote memory is accessed via RDMA. In contrast, when S_B's available memory is accessed by TCP, its throughput degrades by about 10%. Figure 13(b) shows the average and the 99^{th} percentile latency. The average latency with *TCP* is 10% higher than *Default* and *RDMA* and the impact is as high as 20% when considering the 99^{th} percentile latency. That is, when S_B's memory is accessed using TCP, there is a noticeable performance degradation due to the CPU consumed to process the transfers, and secondary effects such as context switches and polluting the processor cache line. On the other hand, there is no noticeable impact when memory is accessed via RDMA.

6.3 Impact of Remote Memory on TempDB

The goal of this experiment is to quantify the benefits of remote memory on TempDB by using the *Hash+Sort* micro benchmark designed to stress TempDB. Since this is a long-running query, we only report the execution times (Figure 14(a)). As evident from the figure, the query running in the *HDD+SSD* configuration is about $5\times$ slower than *Custom*. Observe that HDD is faster than *HDD+SSD* because, as noted earlier, the sequential throughput of HDD is higher than that of the SSD and this workload results in large sequential reads and writes to TempDB. However, *HDD* is still significantly slower than leveraging remote memory with RDMA to store the TempDB. For *Hash+Sort*, the execution time of *SMBDirect+RamDrive* is roughly the same as *Custom* since, as shown in Figure 3, the sequential read throughput of *SMBDirect+RamDrive* is roughly the same as that of *Custom*.

Figure 14(b) shows the I/O throughput of TempDB reads and writes during query execution. The dashed line shows the throughput of TempDB reads and writes for *HDD+SSD* and the solid line plots that for *Custom*. The darker lines correspond to reads and the lighter lines correspond to writes. Recall that the execution of the *Hash+Sort* query consists of two phases (see Figure 2 for the execution plan). The first phase scans the data and builds the hash tables for the join. Since the hash table is too big to fit in memory, this build phase also writes data to TempDB. In the second phase where the join operator starts to generate the joined results, data is read from TempDB to compute the join and the sorted results are written to TempDB for the external sort.

In Figure 14(b), the first phase can be identified where only reads are happening, while the second phase has both reads and writes. As is evident, the phases are considerably shorter for when TempDB is stored in remote memory; the query ends at around 400 sec with *Custom* compared to 1800 sec for *HDD+SSD*. In phase 1, *Custom* achieves a throughput of around 400 MB/s mainly because this phase is CPU-intensive as the query scans the data pages cached in memory, builds the in-memory hashtable based on `orderkey`, and serializes the hash table to TempDB. Phase 2 is I/O-intensive dominated by TempDB reads and writes, where the aggregate throughput of reads and writes is close to 5 GB/s. The benefit of fast TempDB I/Os is evident from the CPU utilization during query execution which is shown in Figure 14(c). As the I/O throughput of *Custom* is much higher, the CPU utilization is much higher, especially in phase 2.



(a) Benefits of semantic cache (b) Seeking vs. scanning

Figure 15: Semantic caching.

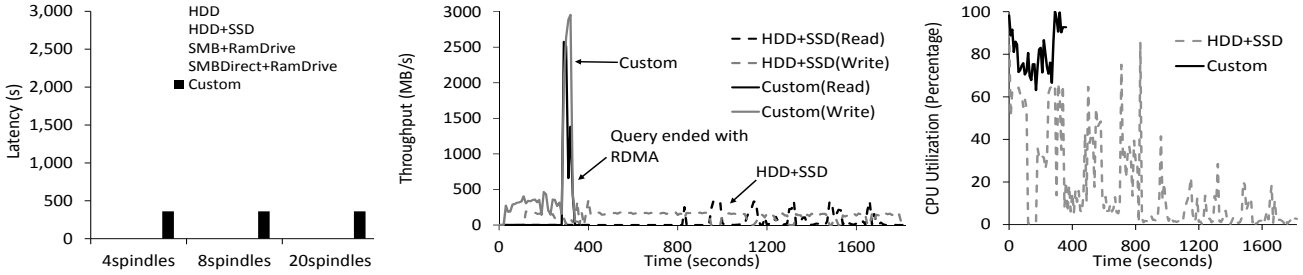
6.4 Remote Memory and Semantic Caching

The goal of this experiment is to evaluate the benefits of opportunistically building a semantic cache to improve query performance (Section 3.3). We use the TPC-H workload for this experiment. We consider materialized views (MV) which can often significantly improve performance for certain queries. We used SQL Server's DTA to generate MV recommendations for TPC-H queries; seven queries benefited from an MV. Since one of the goals of semantic caching is to not contend for buffer pool or other memory requirements of the RDBMS, these MVs are pinned in remote memory and the RDBMS accesses them through a semantic cache broker which is separate from the buffer pool. In the absence of remote memory, these MVs could still be created and stored in *HDD+SSD* to avoid contending for buffer pool memory, which is our baseline. Figure 15(a) plots the multiplicative factor improvement of the seven TPC-H queries compared to when they were executed without MVs using indexes tuned for the queries. As is evident from the figure, using MVs can result in one to four orders of magnitude improvement in query latency even when it is stored on *SSD+HDD*. Pinning the semantic cache in remote memory can result in another order of magnitude improvement. The relative improvement of the semantic cache being stored in *HDD+SSD* vs. remote memory depends on the size of the MV; pinning larger MVs to remote memory results in higher benefits.

Next we use a non-clustered index to demonstrate the fact that the query optimizers can benefit further with the knowledge that structures in the semantic cache are pinned in memory. We use an adapted version of TPC-H Q12 which joins `lineitem` with `orders`. We build a non-clustered index on `orders` which can either be seeked or scanned depending on whether an index nested-loop join (INLJ) or a hash join (HJ) plan is used. We vary the selectivity of the predicate on `lineitem` which determines the number of rows needed from `orders` for the join. Figure 15(b) plots the query latency as the percentage of `O_OrderKey` rows selected from `orders` changes. The dashed lines correspond to a HJ plan and the solid lines correspond to the INLJ plan. As expected, for high selectivity, the INLJ plan outperforms the HJ plan, and vice versa. However, the crossover points depend on whether the index is in remote memory or in SSD. Depending on whether the index is in the semantic cache or is accessed through the buffer pool (where it can be in memory or on disk), the optimal plan might vary. Thus, the optimizer's cost model must be tuned to select the optimal plan in the presence of such remote semantic cache.

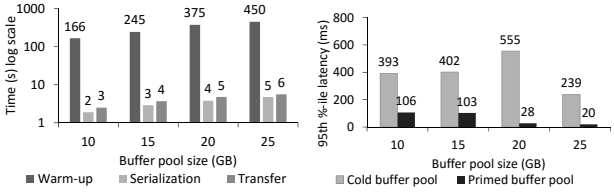
6.5 Buffer pool priming

This experiment evaluates the costs and demonstrates the benefits of proactively priming the buffer pool of a newly-elected primary (S_2) using the warm buffer pool of the old primary (S_1) (Sec-



(a) Latency (b) I/O throughput drill down (c) CPU utilization drill down

Figure 14: Performance of Hash+Sort



(a) Drill down of priming time. (b) Impact of priming on tail latencies.

Figure 16: Drill down of priming time vs. time to warm-up the buffer pool and its impact on tail latencies.

tion 3.4). We use the *RangeScan* workload with a database of about 100GB. The start key of the scan is selected from a hotspot distribution where 99% of the operation access about 20% of the data. Each query scans $5,000$ keys. Figure 16(a) plots the time in seconds (in log scale) to warm up the buffer pool at S_1 when executing the workload, scan through the buffer pool at S_1 and serialize the contents in an in-memory file to prepare for the priming transfer, and transfer the pages from the in-memory file into the buffer pool at S_2. We vary the size of the buffer pool from 10GB - 25GB. As is evident from Figure 16(a), the time to prime the buffer pool is about two orders of magnitude smaller than the time it takes to warm-up the buffer pool during the normal course of workload execution. Figure 16(b) plots the 95^{th} percentile latency of the scan queries starting with a cold buffer pool during warm-up phase at S_2, and the latency of the same workload when the buffer pool was primed with pages from S_1. Unsurprisingly, a primed buffer pool results in $4 \times -10\times$ lower tail latencies, thus significantly reducing the impact of such planned primary-secondary swaps.

7. CONCLUDING REMARKS

We considered the setting of a cluster of RDBMS servers connected with RDMA-enabled NICs where some servers are experiencing memory pressure while other servers have available memory. We presented the abstraction of exposing unused memory on remote servers using a lightweight file API that allows a mature an SMP RDBMS to significantly improve performance of memory-intensive workloads with modest changes to the DBMS engine. We implemented this remote memory abstraction in Microsoft SQL Server and demonstrated the potential performance benefits using a variety of configurations and a combination of micro-benchmarks as well as the industry-standard TPC benchmarks. Compared to using disks when memory is insufficient, our abstraction improved the throughput and latency of queries with short reads and writes by $3\times$ to $10\times$, while improving the latency of multiple TPC-H and TPC-DS queries by $2\times$ to $100\times$.

While our work presents the first in-depth study of the end-to-end benefits of RDMA in an SMP RDBMS, we also highlight some potential areas of future research to further extend our abstraction and implementation. For instance, supporting flexible memory brokering policies, ensuring fairness across multiple workloads, and being more adaptive to the flux in memory requirements are important aspects for deploying this abstraction in a production service. The DBMS engine can also benefit from making remote memory as a first-class concept, and consider it similar to other sources and consumers of memory. For instance, remote memory can be exposed as a new type of memory broker, and techniques to detect a workload's memory requirements, such as Storm et al. [41], can be used to dynamically assign remote memory among the different consumers. In addition to the four novel scenarios of leveraging remote memory and fast RDMA transfers, in Appendix C, we highlight other potential scenarios where an SMP RDBMSs can leverage available resources on remote servers. Developing these techniques are interesting directions of future work. Furthermore, since today's data centers employ some form of virtualization. If the RDBMS is hosted within the guest OS, the performance implications of virtualization on the proposed abstraction needs to quantified. Finally, there are security implications since any remote process can DMA into a memory address once it is registered with the NIC. When processing an RDMA request, NICs have minimal support for authentication, checking privileges, or accounting for resources. Therefore, it is important to explore approaches for secure long-term memory registration or to efficiently encrypt data stored in remote memory.

Acknowledgements

We would like to thank Miguel Castro, Aleksandar Dragojević, and Dushyanth Narayanan for providing the code to implement RDMA transfers using NDSPI which we use in our custom implementation. We also thank Justin Levandoski, Ryan Stutsman, and Lidong Zhou for the useful feedback that helped improve this paper.

8. REFERENCES

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, Feb 1996.

[2] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, pages 1463–1475, 2015.

[3] Microsoft SQL Server: Buffer Pool Extension. http://msdn.microsoft.com/en-us/library/dn133176.aspx.

[4] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD Bufferpool Extensions for Database Systems. *Proc. VLDB Endow.*, 3(2):1435–1446, 2010.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *SOSP*, pages 152–164, 1991.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[7] S. Chaudhuri, U. Dayal, and V. R. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54(8):88–98, 2011.

[8] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings.*, 2005.

[10] S. Dar, M. J. Franklin, B. Þ. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

[11] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8):494–505, 2011.

[12] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *NSDI*. USENIX, April 2014.

[13] P. Frey and G. Alonso. Minimizing the Hidden Cost of RDMA. In *ICDCS*, pages 553–560, June 2009.

[14] A. Gupta and I. Mumick. *Materialized Views:Techniques, Implementations, and Applications*. MIT Press, 1999.

[15] J. Huang, X. Ouyang, J. Jose, M. W. ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-performance design of hbase with rdma over infiniband. In *IPDPS*, pages 774–785, 2012.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[17] IBM DB2 Pure Scale. http://www.ibm.com/developerworks/data/library/dmmag/DBMag_2010_Issue1/DBMag_Issue109_pureScale/.

[18] Mellanox Infiniband. http://www.mellanox.com/.

[19] Infiniband Trade Association. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex A16: RDMA over Converged Ethernet (RoCE), 2010.

[20] Internet Engineering Task Force. RFC 5040: A Remote Direct Memory Access Protocol Specification, 2007.

[21] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High Performance RDMA-based Design of HDFS over InfiniBand. In *SC*, pages 1–12, 2012.

[22] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *ICPP*, pages 743–752. IEEE, 2011.

[23] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *SIGCOMM*, pages 295–306, 2014.

[24] J. Liu, J. Wu, and D. Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[25] Querymemoryresourcenotification function. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366799(v=vs.85).aspx.

[26] Microsoft Analytics Platform System. http://www.microsoft.com/en-us/server-cloud/products/analytics-platform-system/.

[27] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, pages 29–41, 2011.

[28] Oracle's Exadata Smart Flash Cache. http://www.oracle.com/technetwork/database/exadata/exadata-smart-flash-cache-366203.pdf.

[29] Oracle RAC over InfiniBand. http://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/vframe-ib-software/prod_white_paper0900aecd8056d64c.html.

[30] Oracle Temporary Tablespaces. http://docs.oracle.com/cd/B28359_01/server.111/b28310/tspaces002.htm\#ADMIN11366.

[31] Actian Matrix MPP Analytics Database. http://www.actian.com/products/analytics-platform/matrix-mpp-analytics-database/.

[32] List of RAM drive software. http://en.wikipedia.org/wiki/List_of_RAM_drive_software.

[33] RDMA in RamCloud. https://ramcloud.atlassian.net/wiki/download/attachments/6848659/rdma_slides.pdf?version=1&modificationDate=1339212592571&api=v2.

[34] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[35] P. Seshadri and A. Swami. Generalized partial indexes. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 420–427, 1995.

[36] Server Message Block. https://en.wikipedia.org/wiki/Server_Message_Block.

[37] SMB Direct. http://technet.microsoft.com/en-us/library/jj134210.aspx.

[38] SQLIO Disk Subsystem Benchmark Tool. http://www.microsoft.com/en-us/download/details.aspx?id=20163.

[39] Install sql server with smb fileshare. http://msdn.microsoft.com/en-us/library/hh759341.aspx.

[40] M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, Dec. 1989.

[41] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *VLDB*, pages 1081–1092, 2006.

[42] Microsoft SQL Server: TempDB. http://msdn.microsoft.com/en-us/library/ms190768.aspx.

[43] Teradata selects MellanoxTechnologies's InfiniBand interconnect solution. http://cloud-computing.tmcnet.com/news/2012/12/19/6804961.htm.

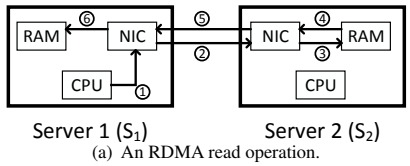
[44] Big Data Analytics for Teradata Aster. http://www.teradata.com/Aster-Big-Analytics-Appliance/#tabbable=0&tab1=0&tab2=0.

[45] Timesten in-memory database and timesten application-tier database cache. http://www.oracle.com/technetwork/database/database-technologies/timesten/overview/index.html.

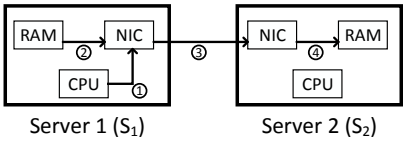
[46] TPC-C Benchmark. http://www.tpc.org/tpcc/.

[47] TPC-H Benchmark. http://www.tpc.org/tpch/.

[48] TPC-DS Benchmark. http://www.tpc.org/tpcds/.



(a) An RDMA read operation.



(b) An RDMA write operation.

Figure 17: Illustration of RDMA read and write operations.

[49] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop YARN: yet another resource negotiator. In *SOCC*, pages 5:1–5:16, 2013.

[50] J. Wu, P. Wyckoff, and D. Panda. PVFS over InfiniBand: design and performance evaluation. In *ICPP*, pages 125–132, 2003.

APPENDIX

A. RDMA BACKGROUND

RDMA allows computers in a network to exchange data from main memory of one server to the main memory of another server without involving the CPU on the remote server. RDMA bypasses the OS kernel, avoids the overhead of network protocol stacks, and performs DMA using only the NIC without copying data from the application's buffer to the OS buffers. Since the CPU is not involved in an RDMA transfer, it avoids context switches and cache line pollution. Figure 17 illustrates RDMA read/write operations between Server 2 (S_2, provider of remote memory) and Server 1 (S_1, accessor of remote memory).

The first step in initiating RDMA is setting up the network connection between the client (S_1) and the server (S_2). This setup also involves *registering* a part of the main memory, called a *memory region* (MR), with the respective NICs on both S_1 and S_2. During registration with the NIC, the device driver *pins* the registered memory pages to physical memory and stores the virtual to physical page mapping in a page table. Registering a memory region is a expensive operation (relative to RDMA transfers) since it involves the CPU and the OS kernel. There are also limits on the size of each MR (2 GB for our Mellanox NIC) and the number of registered MRs (around 130K for our NIC). The NICs also have limited memory for the page tables and uses the main memory to store this information. The NIC's memory is used as a cache and the NIC issues a DMA if there is a cache miss accessing the page table. Carefully managing registration of MRs and the available memory is crucial to leverage the performance benefits of RDMA [13].

RDMA communication is based on a set of queues. The send queue and receive queue, together called the *queue pairs*, process the RDMA transfer, and a completion queue notifies the application of the transfer's completion. RDMA requests are sent over a reliable channel and network failures are exposed as terminated connections. The NIC implements in hardware all the logic of the RDMA protocol, flow control, and reliability.

Figure 17(a) illustrates an RDMA read operation. When an application running on S_1 decides to read a memory location on S_2, S_1's CPU issues an RDMA request to the local NIC (Step ①) which in turn issues the request to the NIC on S_2 (Step ②). The request will specify the MR, the offset within the MR, the size of data read, and a memory location at S_1 where the remote memory contents will be transferred. S_2's NIC issues a DMA to access the pages in the memory (Steps ③ and ④) and transfers the contents to S_1's NIC (Step ⑤) which in turn performs another DMA to write the data to S_1's memory (Step ⑥). The steps for an RDMA write are also similar and are illustrated in Figure 17(b).

RDMA has many implementations such as in InfiniBand [18], RDMA over Converged Ethernet (RoCE) [19], and iWARP [20]. InfiniBand is a switched fabric link that provides RDMA capabilities. RoCE allows RDMA over Ethernet with data center bridging extensions where the connection state is managed in hardware providing a reliable communication channel without software protocols like TCP. iWARP implements RDMA on top of TCP/IP. It has fewer requirements on the network but the performance is worse compared to that on InfiniBand. RDMA protocols bypass the OS and avoid the overheads of TCP/IP. However, the protocols for RDMA transfers are specific to the platform. Section 4 discussed protocols in the Windows ecosystem. In the Linux ecosystem, alternative approaches include User Direct Access Programming Library (uDAPL) and the Sockets Direct Protocol (SDP). uDAPL defines a set of user-level APIs for all RDMA-capable transports. SDP is an Infiniband-specific upper-layer protocol that defines a standard wire protocol to support stream sockets networking over Infiniband allowing applications with minimal changes to leverage the benefits of the Infiniband protocols.

B. ADDITIONAL EXPERIMENTS

B.1 TPC Benchmarks

TPC-H. We now quantify the benefits of remote memory for both TempDB and BPExt using industry-standard benchmark workloads to analyze the holistic benefits for complex database workloads. Figure 18 shows the throughput of TPC-H with the different alternative designs. The experiment is repeated for three different configurations with 4, 8, and 20 HDD spindles. As can be seen in this figure, *Custom* outperforms both *HDD+SSD* and *SMBDirect+ RamDrive*. An interesting observation is that the throughput of *Custom* is even higher than *Local Memory* which has the 256 GB memory locally. This behavior is an artifact of SQL Server's query execution engine's admission control policy which does not assign all available memory to one long-running query, thus leaving memory available for subsequent queries and prevent runaway long-running queries from hogging the server's memory. Since memory provided to the operators are not enough for the data sizes, the execution resorts to using the TempDB to spill data. As a result, two of the TPC-H queries, $Q10$ and $Q18$, spill to TempDB. Since *Custom* stores the TempDB in remote memory, it is much faster than spilling to disk, thus resulting in *Custom* outperforming *Local Memory* for $Q10$ and $Q18$. For the other queries, most of the performance improvement comes from BPExt since their execution plans do not contain memory-intensive operators. Figure 19 plots a histogram of latency improvements of *Custom* compared to *HDD+SSD* for 20 spindles. As can be seen in this figure, the improvements are massive for TPC-H: $2\times$ for 8 queries, $2\times$ to $5\times$ for 10 queries, and $5\times$ to $10\times$ for 3 queries.

TPC-DS. Similar improvements can be observed for the TPC-DS benchmark which has a more diverse set of queries. Figure 20 plots the throughput and Figure 21 plots the histogram of latency improvements of *Custom* compared to *HDD+SSD* for 20 spindles. The major difference from TPC-H results is that *Custom* has slightly lower throughput compared to *Local Memory* since queries do not

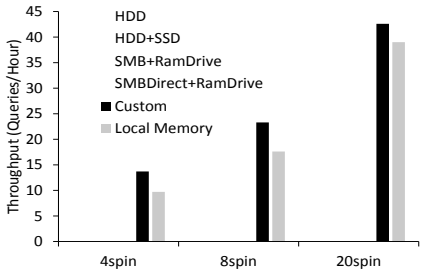


Figure 18: TPC-H throughput.

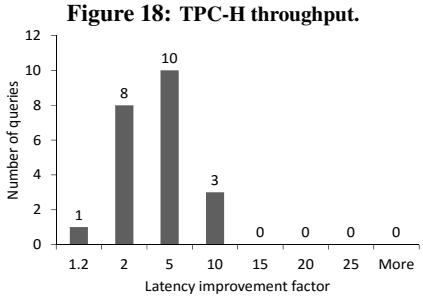


Figure 19: TPC-H latency improvement.

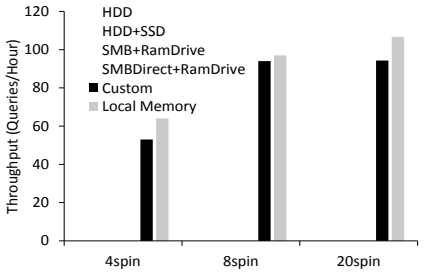


Figure 20: TPC-DS throughput.

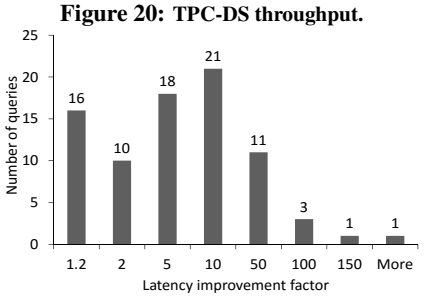


Figure 21: TPC-DS latency improvement.

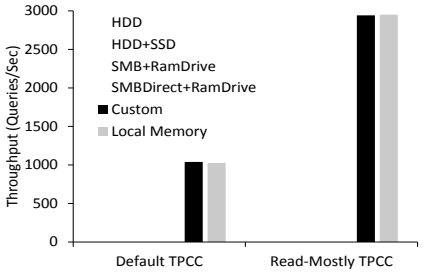


Figure 22: TPC-C throughput.

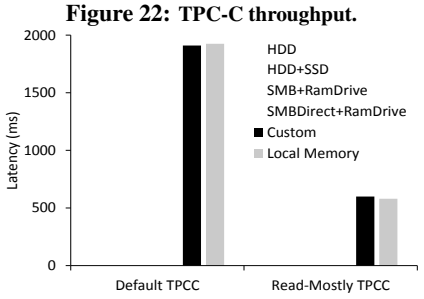
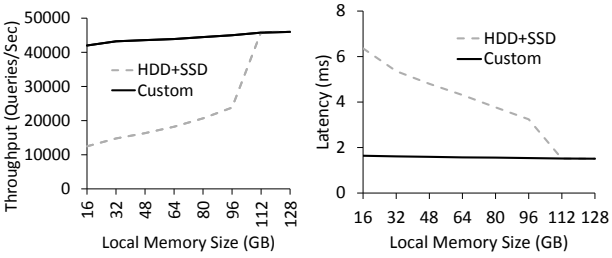


Figure 23: TPC-C latency.



(a) Throughput. (b) Latency.

Figure 24: Performance impact of varying available local memory.

spill to TempDB for the *Local Memory* setting. In addition, we can see a much more significant improvement in terms of latency: 18 queries get $2\times$ to $5\times$ improvement, 21 queries get $5\times$ to $10\times$ improvement, and 11 queries get $10\times$ to $50\times$ improvement. Some TPC-DS queries can even get more than $100\times$ improvement in latencies by leveraging remote memory.

TPC-C. OLTP workloads rarely have heavy demand for TempDB since more requests are short reads and writes. The RangeScan with update workload demonstrated some benefits of remote memory for OLTP workloads with short reads and writes. We now present results with the TPC-C workload. We report the throughput and latency of TPC-C workloads with two different parameter settings: (i) the default TPC-C transaction mix (Default TPCC); and (ii) where the read-only `StockLevel` transaction comprises 90% of the workload mix (Read-Mostly TPCC). We use an HDD setup with 20 spindles for both the experiments. As shown in Figure 22, the default TPC-C transaction mix does not benefit from remote memory even when the local memory does not cache the entire data. This is because in TPC-C, the `NewOrder` transactions create new orders which the `Payment`, `Delivery` and `OrderStatus` transactions access, which comprises 96% of the queries in the workload. Therefore, the working set for TPC-C workload is small and keeps changing as new orders are added to the database. Even with 32 GB local memory (the *Local Memory* setting), the throughput cannot be improved compared to *HDD+SSD* with 16 GB local memory. This experiment is an example where remote memory is not beneficial since there is not enough demand for memory. By contrast, with the read-mostly mix of TPC-C, most of the queries are `StockLevel` queries that also access the old data, accessing more database pages, resulting in a larger working set and more demand for memory. Hence, the alternatives with more available memory, whether local or remote, have higher throughput. Figure 23 shows the latency of different methods. For Read-Mostly TPCC, the latency of *HDD+SSD* is lower than the latency of the methods with remote memory. This is because with remote memory, the database is able to process more TPC-C queries (as seen from the throughput graph), which results in higher contention for resources, such as locks, resulting in higher latency. When the throughput of the different alternative design is set to the same target, the latency of *HDD+SSD* is higher compared to *Custom* with remote memory. Also, since this workload had lower demand for remote memory, even *SMB+RamDrive* has performance comparable to *SMBDirect+RamDrive*.

B.2 Varying available local memory

The benefits of remote memory is dependent on the available local memory and the workload's demand for additional memory. In this experiment, we vary the amount of local memory on the database server. We use the RangeScan workload used to stress the BPExt and store the BPExt on a fixed amount of remote memory sufficient to accommodate the working set. Figures 24(a) and 24(b) plot the throughput and latency of *Custom* and *HDD+SSD* as we vary the amount of local memory available from 16 GB to 128 GB. As is evident from the figures, as the amount of memory available locally increases, *Custom*'s benefits over *HDD+SSD* decreases. This decrease continues until the local memory is sufficiently large to cache the entire database. When the database is cached entirely in local memory, the throughput and latency of both alternatives are equal. In addition, as the amount of local memory increases, the throughput of *Custom* also increases slightly, since local memory is about two orders of magnitude faster than remote memory.

B.3 Multiple database servers

In the experiments presented in Section 6 we characterized the end-to-end workload performance in one multi-server setup where

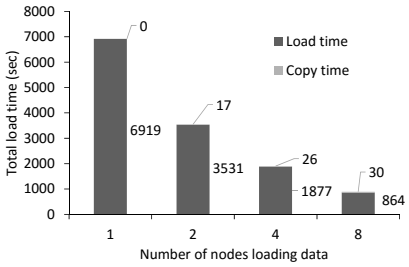


Figure 27: Parallel loading.

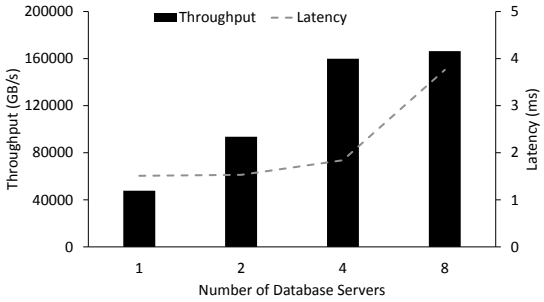


Figure 25: Performance of RangeScan when multiple database servers access remote memory at one server.

Figure 26: Recovering an index after remote node failure.

one database server accesses memory of multiple remote servers. In this section, we consider the complementary experiment where multiple database servers are accessing the memory on one remote server. We use the same RangeScan workload, though with a smaller database size (125 million rows) so that there is sufficient memory on one M_j to serve 8 DB_i servers. Each database server is configured with about 7GB local memory and 30GB remote memory. Figure 25 plots the throughput and latency as we increase the number of database servers which is plotted on the x-axis. The bars plot the aggregate throughput (queries per second) of all the database servers along the left vertical axis and the dashed line plots the average latency (ms) along the right vertical axis. Similar to the behavior we observed with the I/O micro benchmark, until the NIC at M_j saturates, the aggregate throughput increases almost linearly with the number of database servers, with negligible increase in latency. Once the NIC is saturated, increasing the concurrent load results in a noticeable increase in latency without much improvement in aggregate throughput.

B.4 Recovering the Semantic Cache

Since semantic cache is stored in remote memory and our remote memory abstraction is best effort, the remote server failure will completely wipe out the cache. Based on a user-specified policy, the queries can continue using the base indexes and ignore the cache. Alternatively, the cache can be rebuilt on another server. Even if we periodically checkpoint the cache, there might still be some trailing updates which might not be present in the checkpoint and need to be recovered from the transaction log. As a benefit of the cache being part of the RDBMS, we can use the REDO recovery logic to rebuild the index. Figure 26 plots the recovery time as a function of the amount of dirty pages in the non-clustered index since the last checkpoint. As is evident from the figure, the recovery time increases almost linearly with the size of dirty data. Less than a GB of dirty pages can be recovered in tens of seconds and about 16 GB of data can be recovered within four minutes. That is, if the data in the semantic cache is not being updated frequently, the cache can be recovered very fast. Therefore, even though we use remote memory with best-effort fault-tolerance guarantees, remote node failures only result in a small period of performance degradation since the cache can be reconstructed within minutes. Note that storing the semantic cache in HDD or SSD will not require recovery. Hence this experiment reports time to recover the cache into remote memory from the transaction log.

C. ACCELERATING DATA LOADING

In addition to the four scenarios discussed in Section 3, our lightweight file abstraction is also beneficial for other scenarios. For instance, efficiently loading new partitions of data into a data warehouse is crucial to enable querying newly-arriving data in a timely manner [7]. Parallel data loading into an RDBMS server is a CPU and I/O-intensive process which can be bottlenecked by the *single* server into which data is being loaded. Data often arrives as flat file which needs to be parsed, compressed, and converted into native database format. Each of these steps are CPU-intensive operations. If some remote database servers on the cluster have idle CPU along with unused memory, an additional scenario is to *offload* and parallelize the data load on multiple remote servers. Our key idea is to enable the remote database servers to load data for a set of partitions into local in-memory files. The CPU and memory-parallelism of multiple idle servers in the cluster can greatly speed up the data loading. Once the remote servers finish loading the partitions, the destination server *pulls* these loaded data partitions from the remote in-memory files using the API we support. Since the step of pulling data using RDMA is very fast relative to the actual data loading step, the overall data loading time can be dramatically reduced.

Figure 27 reports results from an experiment to demonstrate the benefits of such parallel loading. We consider the case where 160GB of raw data is being loaded into an existing TPC-H database. The data files are arriving from various sources and have 80 splits of approximately 2GB average size. The figure plots the time to load all the input files as we vary the number of servers loading the files. The bar with 1 server corresponds to data being loaded into the destination server using standard parallel loading tools. Since there is no copy needed, all the time is spent in loading. As we increase the number of servers, the data load is followed by a copy of the in-memory file to the destination server. As is evident from Figure 27, the data loading time reduces almost linearly, while the copy time continues to be negligible compared to the load time, thus allowing almost linear speedup in data loading. For instance, one server takes $6,919$ seconds to load the data while eight servers take 894 seconds, resulting in $\sim 7.7\times$ speedup. In addition to memory brokering which is implemented in our system, this scenario requires brokering of idle remote CPUs on database servers. A detailed design and implementation of such CPU brokering is an interesting area of future work, which can enable other interesting scenarios for CPU and memory offloading in a cluster of RDBMS servers.