CSCI 2980

Master's Project Report

Rethinking Distributed Indexing for RDMA - Based Networks

by

Sumukha Tumkur Vani

stumkurv@cs.brown.edu

Under the guidance of

Rodrigo Fonseca

Carsten Binnig

Submitted in partial fulfillment of the requirements for the award of the degree of

Master of Science in Computer Science

BROWN

Computer Science

Department of Computer Science

BROWN UNIVERSITY

Providence RI USA 02912

Spring 2018

© Copyright 2018, Sumukha Tumkur Vani

Abstract

Distributed database systems have always been designed keeping in mind that the underlying network is the bottleneck. Today we see that network bandwidth is catching up with memory channel bandwidth. With more and more research happening in the area of high-performance networks, it is time we changed the perspective of how these networks influenced the design of database architectures. Remote Direct Memory Access, an advanced feature offered by high performance networks has the ability to read and write to remote memory without involving the remote CPU. This transport mechanism can revolutionize the way systems interact with database systems. But the traditional database systems which are designed to use software-based transport cannot harness the full benefits of RDMA. Index structure and data storage scheme have to be redesigned and tailored for RDMA networks.

In this project we present two novel distributed in-memory database architectures designed using RDMA as the transport mechanism. *Hybrid* is designed using two-sided RDMA for index lookups and one-sided RDMA for data fetch. *Pure1* is designed using one-sided RDMA for both index-lookups and data fetch. Our experiments with 100 million key-value pairs and 240 clients show that these two models can be highly scalable at the expense of 75% lesser CPU compared to traditional distributed database architectures. For single-key and lower selectivities *pure1* outperforms traditional design and hybrid and for higher selectivities both *pure1* and *hybrid* perform equally better compared to traditional design.

Acknowledgment

I am very grateful to my advisors Prof Rodrigo Fonseca and Prof Carsten Binnig for their enormous guidance and support. I am very thankful to Tobias Zeigler of TU Darmstadt for guiding me in the development of the project and to Michael Markovitch for providing valuable feedback during the design phase. This project would not have been possible without the support of the BrownSys (Systems and Networking Research Group at Brown University) and the Database Research Group at Brown University.

I am thankful to Viswanath Krishnamurthy and Arvind Kini for introducing me to the exciting field of high-performance networks. I am grateful to Prof Venugopal K R of Bangalore University for identifying the research potential in me and encouraging me to pursue ScM degree. Most importantly, I would like to thank my parents, Jagadeesh Tumkur Vani and Savitha Jagadeesh and my brother Samartha Tumkur Vani for their continuous support and encouragement throughout the course of the ScM program.

Contents

1 Introduction **1**

2 Background **4**
2.1 Infiniband 4
2.2 RDMA 4

3 Motivation **6**
3.1 Problem 6
3.2 Opportunity 7

4 Design **9**
4.1 Data Store 9
4.2 Index Structure 9
4.3 Variant 1: Pure 2 Sided Model (pure2) 11
4.4 Variant 2: Hybrid Model (hybrid) 11
4.5 Variant 3: Pure 1 Model (pure1) 13

5 Implementation and Evaluation **15**
5.1 Implementation 15
5.2 Experimental Setup 15
5.3 Performance 17
5.3.1 Throughput 17
5.3.2 Lookups 18
5.3.3 Latency 18
5.3.4 CPU Utilization 19

6 Related Work **22**

7 Conclusion and Future Work **23**
7.1 Conclusion 23
7.2 Future Work 23

Chapter 1

Introduction

Modern applications such as social networks and search engines demand high throughput and low latency. As millions of clients constantly query large amounts of data in these applications, the volume of data that needs to be processed rise up to the extent of petabytes. Hence, network becomes one of the primary factors that influence the design of Database systems that support such heavy applications. The focus of distributed database systems has been to reduce the amount of data transferred over the network and perform complex processing [6]. The reason for such a design decision is the assumption that network is the bottleneck. The progress made in the area of network technologies is tremendous and we see a steady growth in the bandwidth achieved by all the network technologies. However, there is not enough change in the database architecture to harness this. Database architectures continue to be designed to work efficiently over poor performing networks. It is time to redesign database systems without the assumption that the underlying network is the bottleneck.

The state-of-the-art high performance network technologies offer bandwidth that is comparable with the bandwidth offered by memory channel. High speed interconnects such as Infiniband and Intel Omnipath are in the forefront of high performance network arena. Infiniband HDR 4x offers a bandwidth of up to 200 Gbps [12] per link while the Intel Omnipath is claiming a bidirectional bandwidth of 25 GBps [9] per port. DDR4 memory offers a bandwidth of 25GBps [7] while DDR3 memory offers a bandwidth of 6GBps. Although the network speeds can never equal the memory channel speeds, they are definitely catching up. Two generations old, Infiniband FDR 4x, itself was offering 1.7 GBps per port. This growth in network bandwidth in such a short span of time. Figure 1.1, shows how close the race is with the memory channel bandwidth.

The conventional socket-based software transport uses a lot of CPU to run its transport logic. Whereas, the transport logic for Remote Direct Memory Access (RDMA) runs entirely on the Network Interface Card (NIC), bypassing the CPU. Although Direct Memory Access (DMA) has been embraced well by a wide range of applications, RDMA on the other hand has not been put to full use. RDMA over Infiniband can achieve bandwidths very close to that of memory channel. RDMA is also offered on Ethernet, which is ubiquitous, as RDMA over Con-

verged Ethernet (RoCE) [18]. Though RDMA has been around for quite some time, database systems are yet to tap into its full potential. Some of the latest databases just support RDMA as a feature. Database systems have always been designed for the TCP/IP stack. In order to have them switch over to RDMA entirely, although a complete redesign is required, it is definitely beneficial. Since RDMA is available on Ethernet (cite RoCE), redesign efforts do not call for any additional investment in hardware.

Two types of RDMA (i) One-sided (ii) Two-sided are available. One-sided RDMA performs remote READ, WRITE and atomic operations without invoking the remote CPU. The entire transport operation is handled by the Network Interface Card (NIC). This is a very powerful feature of the one-sided RDMA compared to conventional software transports as sending and receiving data over socket-based TCP/IP can result in huge CPU overhead. The Two-sided RDMA is more like a Remote Procedure Call (RPC) which involves the utilization of the CPU on the remote machine.

Redesigning systems to transfer large amounts of data over one-sided RDMA will help reduce a lot of load on the CPU. Although data transfer might seem beneficial, if we were to traverse a complex data structure like a B-Tree on the remote machine with One-sided RDMA, we will have to spend multiple round trip times (RTT) to do so. The compute and memory power of a NIC is also very little and we cannot run complex logic on it. Alternatively, using two-sided RDMA can save us on RTTs by having the entire tree traversal done on the remote machine using its CPU. This is what makes one-sided RDMA not desirable for developing large and complex distributed database systems.

FaSST [10] proposes a highly scalable design using two-sided RDMA. However, a distributed database architecture specifically for one-sided RDMA (which significantly reduces the load on CPU) is very necessary. In order to realize such a design, the index structure and the data must be stored in a fashion easily accessible through one-sided RDMA. The design of the index structure should be such that the client machine should be able to traverse it quickly to find the location of the data that it is interested in. Also, the data should be stored in a linked fashion. While performing a range query, the client machine should do a single lookup for the first key in the range, then it should be able to fetch the remaining data by following the links. Providing information regarding how much to read along with address of the location to find the required data to the client machines will be highly favorable.

In this project, we propose two new designs for distributed in-memory database systems that use RDMA as the transport mechanism. The first design is a hybrid model which uses both one-sided RDMA and two-sided RDMA. The two-sided RPC based RDMA is used for index traversal on the server and the one-sided RDMA is used for performing the process of fetching the data. The second design is entirely based on one-sided RDMA. In this design, initially one-sided RDMA is used for traversing the index on the server, and then the process of fetching the data is also performed using one-sided RDMA. The data is stored in chunks called *pages* which are connected forming a large linked-list and the index structure is a modified ver-

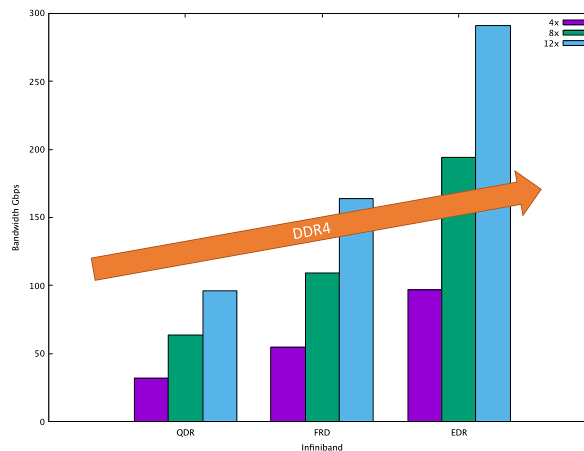


Figure 1.1: Infiniband Network Bandwith vs Memory Channel Bandwidth (DDR4)

sion of a *skip-list*. Along with the address of the *page*, the index will also contain information regarding the fillgrade of each *page*. We compare these two models against the typical design of using two-sided RPC based RDMA for both index traversal and fetching data. This model is also similar to the database systems that use traditional software-based transport. Both designs proposed by us outperform the typical design by multiple folds when scaled. The magnitude of reduction in the load on the CPU is quite significant. The performance of the entirely one-sided model is even more interesting. The latency caused by traversing an index structure designed specifically for one-sided RDMA is negligible. With increased throughout and reduced CPU load, we can emphasize that, it is time for database system architects to stop assuming that the network is the bottleneck.

Chapter 2

Background

2.1 Infiniband

Infiniband [8] is an input/output architecture standard for designing interconnects for communication between servers and storage. It has been widely used in supercomputers and high-performance compute clusters. It prevents congestion based losses and bit-error losses in the link. Infiniband follows the switched fabric architecture and supports both IP [5] and RDMA stacks. It offers reliable messaging service and RDMA capabilities with no software intervention in the data path. This is what helps achieve high bandwidth and very low latency. It is an industry standard interconnect architecture which is ideal for carrying both data plane traffic and control plane traffic over a single connection. Some of the advantages of Infiniband include:

- Extremely low end to end latency of 1μ second
- Direct support for RDMA
- Consolidate control plane and data plane traffic on to a single connection there-by saving power significantly
- Highly reliable and lossless I/O interconnect

2.2 RDMA

Remote Direct Memory Access is a protocol to perform Direct Memory Access (DMA) over the network. Access to memory of the remote machine is provided by the Network Interface Card (NIC). The transport protocol is implemented completely in the NIC hardware. There is no involvement of the CPU in performing the transport. The operating system kernel does not play any role in the data transfer process. Because a dedicated hardware, here the NIC, handles the entire task, RDMA offers extremely low latencies and very high throughput. The host and client applications register a part of their respective memory with the NIC for RDMA purposes.

The interface card, then transfers the contents from the host memory to the client memory over the network without copying it over to the operating system buffers. This is defined as zero-copying. Since there is a kernel bypass and there is no copying of data to the OS buffers, the CPU utilization is very low, thus offering low latency.

An RDMA connection consists of a pair of queues called Queue Pairs (QP). A Queue Pair comprises of a send queue and a receive queue. A completion queue is also associated with a connection which is used to track the completion of each RDMA tasks which are also called *verbs*. All the queue pairs are maintained in the NIC on either side of the connection. Whenever a machine wants to send or receive data over RDMA, it will either post a work request on to the send queue or to the receive queue. Upon completion of the task, a notification element is posted on to the completion queue. The application can poll the completion queue to know the status of the task. The *verbs* supported by RDMA include SEND, RECEIVE, READ and WRITE. The SEND and RECEIVE are two-sided while READ and WRITE are one-sided RDMA verbs. The term two-sided is used because, each SEND work request requires a RECEIVE work request on the other end. This is possible only with the intervention of the remote CPU. One-sided READs and WRITEs are performed directly without involving the remote CPU from the RDMA registered memory.

RDMA offers both connection oriented and connectionless transport. One-sided RDMA tasks are performed over connection oriented transport guaranteeing high end-to-end reliability. Due to the limited memory available on the interface card, the QP states cannot be cached. This does not permit a QP from communicating with multiple QPs. This is the disadvantage of one-sided RDMA. Two-sided RDMA is performed over connectionless transport which is not reliable. The in-order delivery of the data is not guaranteed and the communicating machines have to reorder the data at the expense of their CPU. This is very similar to TCP, where the sending and receiving machines have to synchronize and the respective CPUs have to do work to reorder the packets.

Chapter 3

Motivation

3.1 Problem

Todays most popular applications such as Netflix, Facebook and Google are highly compute intensive. Millions of clients are connected to these servers and are constantly querying data. The queries could be as small as a comment or as a large as a high definition movie. These servers have to do a lot of work in order to process these queries and serve the clients with the data. That is not all that these servers are doing. They are performing complex machine learning tasks such as recommending movies, showing targeted ads or suggesting restaurants nearby which demand more CPU cycles. With artificial intelligence entering all walks of life, the next generation data applications are going to demand more and more CPU power. If we can design database systems which offer high throughput while consuming limited CPU cycles, then the performance improvement will propagate to the applications which will in-turn become much more faster.

Data transmission over the network using software based transports consume CPU. The primary communication mechanism in distributed database systems is software based transport. This has a great impact on the throughput of the database system. The database server has to do a lot of work to transfer data over the network. Furthermore, the database fraternity has always thought that network is the bottleneck. Therefore, database architecture is always such that the data being transferred over the network is as minimal as possible. This is made possible by performing complex processing on the query results [15] before transferring data over the network to the client. The assumption that the network is the bottleneck is resulting in more and more load on the CPU of the database server. By using state-of-the-art high-performance networks, we no longer have to consider the network as the bottleneck. This in turn will reduce the load on the CPU significantly thereby permitting these CPUs to do more complex tasks.

One-sided RDMA has the ability to perform remote READs and WRITEs without involving the CPU. This is very helpful to traverse a flat data structure or contiguous memory locations. Most database indexes will be in some form of trees or hash tables. To reach a particular node in a tree using one-sided RDMA, multiple READs have to be performed starting from the head

along the direction of the interested node. This will cost us multiple RTTs which are undesirable. On the other hand, we could just perform an RPC using two-sided RDMA and request the remote server to process the query and return the data we are interested in. Here the remote server will have to traverse the tree structure, fetch the data and return it to the client with just one RTT. But as the number of clients increase, the number of threads processing the query on the server will also increase, thereby consuming a lot of CPU. This will result in a bottleneck at the server and increases the latency of each query significantly.

Having a central Master server dedicated to traverse the index and return the location of the data is not a feasible solution. This architecture is not efficient for two reasons. Firstly, as the number of clients increase, the Master server will be overloaded with requests to process. Secondly, the Master server has to regularly update its index by communicating with all the servers. While the index is being updated, the task of processing the queries has to be blocked. Further, the process of regularly updating the index will again add substantial traffic to the network. Caching the index structure on the client [21] is another possibility to reduce the queries sent to the Master server. This technique introduces another challenge of dealing with stale index on the clients. The clients have to either update their indexes regularly or have to use stale index and update them lazily. Updating cached indexes regularly will again introduce additional traffic into the network on top of the traffic already in the network because of queries. On the other hand, using a stale index, trying to fetch the data, then realizing that the data is not present in the location and then updating the index will prove costly. The frequency of updates in hot areas in the data will be very high and the probability of the cached index being stale will also be significantly high. This technique will be inefficient unless a good cached index update strategy is adopted.

3.2 Opportunity

Distributed architecture which offers high throughput and low latency is the need of the day. Using RDMA, we can get high network bandwidth and also reduce the load on the CPU. We need to redesign the index structure and the data storage structure to facilitate the use of RDMA as the transport mechanism. An index structure that is linked such as a tree or linked list can be traversed using one-sided READs. A similar approach in storing data is also going to be helpful for fetching using one-sided READs.

Having the index structure on the server seems to be viable. Since the server houses the data, it can update the index locally at regular intervals. A local update is not going to be as expensive as an update over the network. Further in this direction, a range partitioned index is going to be even more beneficial. The server can host the only index meant for the range of data it houses. A range partitioned index is also beneficial in reducing the load on each server and facilitates the distribution of the load amongst the servers. Even when the index is range partitioned, the hot areas in the data will introduce significant load on some servers. Round

robin distribution of the index might also be beneficial in distributing the load. In such cases, no matter what the hot areas are, to traverse the index, the client has to READ from all the servers. This approach is not viable for a database system using two-sided RDMA as each server does not have the entire view of the index structure.

Storing data in a way that is suitable for one-sided RDMA operations is of prime importance. The individual chunks of data must be stored in a linked fashion. Since the goal is to fetch the data using one-sided READs, the client should be able to follow the links as it keeps fetching the data. Range partitioning the data is not suitable as the load on the link for servers with hot areas in data will be more compared to the other servers. A round robin distribution of the data will result in even distribution of the load on the links even when there are hot areas in the data. Majority of the queries in real world systems are going to request very small amounts of data compared to the entire data in the database. This also applies to updates and modifications to the data. It is very uncommon for queries to request more than 10-15% of the total data and most updates and deletes are going to be single entry or single key. With this in mind, the design should be highly scalable for small to medium range queries.

Chapter 4

Design

4.1 Data Store

The data is sorted first and stored in a linked fashion in chunks called *pages*. This can be visualized as a very large LinkedList. This makes scanning from the first key to the last key very similar to traversing a list. Each *page* contains the key-value pairs in order, pointer to the next *page*, lock and version number. The reason for storing the key-value pairs in a sorted manner is that, if a client reads the wrong *page* and does not find the key-value pair, it is looking for, then it can easily decide to move forward or backward. At the time of creation of a *page*, it is not completely filled with key-value pairs. Only one half of the *page* is filled and there will be slots available to future additions of new key-value pairs. The pointer to the next *page* called *PagePointer* is a very important part of the *page*. It contains the Node ID of the server on which the next *page* resides, the RDMA offset of that *page* in the servers RDMA registered memory and the fillgrade of the *page*. Fillgrade of a *page* is the size of the *page* without any unallocated slots of key-value pairs. This is a very important piece of information which tells the client the exact size of memory to be read to fetch only the filled part of the *page*. This prevents the client from reading any unwanted memory areas thus saving the bandwidth on the link. The lock on the *page* is available to be taken when the *page* is being updated or modified. The version number of the *page* is maintained to track updates to the *page*. Every update in the *page* will result in the increment of the version number.

4.2 Index Structure

The index structure is a modified version of the skip-list. Each node in this list is an *index-page* and is linked to the next *index-page*. It contains all the components of a normal *page* as defined above. The difference is that, the value for each key in the *index-page* is a *PagePointer* to the actual *page* which contains the key-value pair. The *PagePointer*s in each *index-page* point to the succeeding *index-page*. At the time of creation, the *index-page* is also partially filled, so that new entries can be added in future, when a *page* is added between two existing *pages*. The

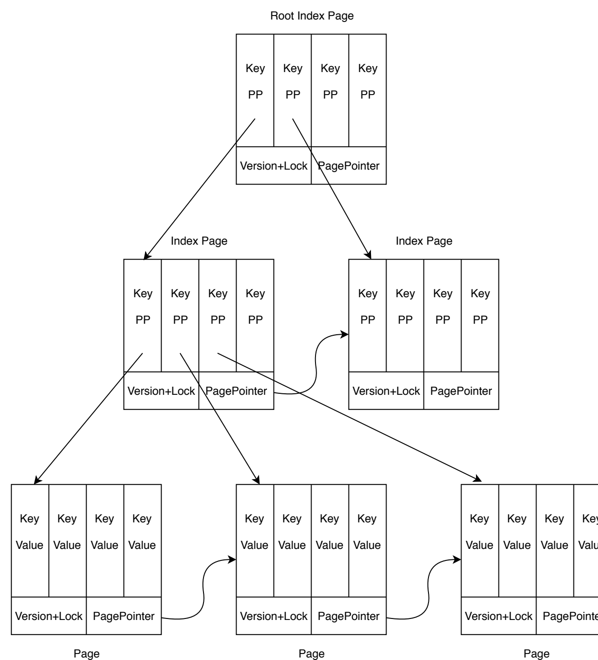


Figure 4.1: Data Store and Index Structure

index-page can be locked during updates. The version number is provided to know whether the *index-page* has been modified.

The first key in each *page* is indexed. Since the data is stored in a sorted manner, indexing the first key of each *page* will make the index also sorted. The index structure is built over multiple levels. The bottom most level will contain the *PagePointer*s to the actual *pages* containing the data. These *PagePointer*s will also contain the fillgrade information of the *pages* they are pointing to. The upper levels of the index will contain the first key of each *index-page* in the level below it. The top most level will contain one *index-page* which is called the *root-index-page*.

While performing a lookup on the index, we start from the *root-index-page*. If we find the key we are interested in, then we follow the *PagePointer* to the *index-page* in the second level. We then lookup for the *PagePointer* to the *index-page* in the third level for the key here. We continue this way until we reach the bottom most level which will contain the *PagePointer* to the *page* which has the key. Supposing we do not find the key we are interested in the *root-index-page*, then starting from the first entry, we take every two successive entries k1 and k2 and check if the key is greater than the k1 and lesser than k2. When we find such a pair k1 and k2, we take the *PagePointer* for k1 and go to the *index-page* in the second level. Here we perform the same check to find the next pair and reach the bottom most level. In the bottom-most level k1 will be the first key of the *page* which will contain the key that we are interested in. Following the *PagePointer* for k1 will give us the *page* we are looking for.

4.3 Variant 1: Pure 2 Sided Model (pure2)

This is the baseline design based entirely on Two-sided RDMA. The database server will be listening on a particular port for incoming RDMA connections from clients. The clients can connect to the database server and query it. To perform the query, the clients will have to send the query as a message to the database server, whether it is a single key query or a range query. The database server will receive the query message and perform a lookup in its index. If it finds the requested key, it will send the key-value pair to the client. In case of a range query, it will send the entire range of requested key-value pairs as multiple *pages* to the client. If it does not find the requested key or range, a failure message is sent to the client.

For insert and update operations, the client will send a message to the database server with the key-value pair to be inserted or modified. The database server, upon receiving the message, will perform a lookup in the index to find the location for inserting or updating the given key-value pair and then perform the operation. The data loading scheme is such that each *page* can accommodate new key-value pairs in future. Therefore, every insert operation will not require an index modification. Only when a *page* is completely filled, and a new *page* is created, the index structure is also suitably modified. Upon completion of the operation, a status message is sent to the client as a response.

In a typical database management system, the client will establish a TCP connection to the database server, send the query over the connection. The database server on the other end, will process the query and respond appropriately. Apart from processing the query, the CPU utilization for sending the response is high in the traditional architectures. *Pure2* is similar to the existing database architectures that use software based transports for communications. In this model, upon receiving the query message, the server will perform a lookup in the index structure, then fetch the data and send it across to the client. Apart from the process of looking up in the index, the database server has to spend a lot of CPU cycles in constructing the *pages* to be sent over. Insert and update operations are more expensive because, apart from performing the lookup, CPU is utilized for making suitable modifications to the *page*. If creation of a new *page* is necessary then, the CPU will perform the task of updating the appropriate *PagePointer*s and then rebuild the index. The task of rebuilding the index is a significant load on the CPU. When the number of clients increase, this model essentially puts a lot of load on the CPU thus impacting the performance. Hence, we consider this to be the baseline model against which we compare our next two designs.

4.4 Variant 2: Hybrid Model (hybrid)

In this model, we use both One-sided and Two-sided RDMA. Similar to *pure2*, the database server will be listening on a particular port waiting for clients to connect. When a client wants to perform a query, it will setup a RDMA connection to the database server, then send the query message over two-sided RDMA. The database server receives the query message, performs a

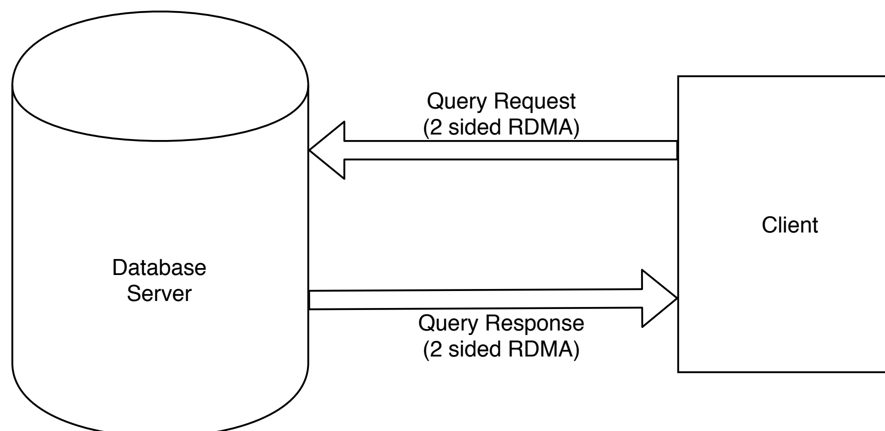


Figure 4.2: Pure2 Architecture

lookup in the index and responds to the client with the *PagePointer* to the *page* containing the key. If the lookup is unsuccessful, an appropriate failure message is sent to the client. The client receives the *PagePointer* and then fetches the *page* by performing one-sided RDMA READ from the database server. The fillgrade information present in the *PagePointer* will enable the client to only read the filled part of the *page* thus not wasting any bandwidth on the link. In case of a range query, the database server will respond to the client with the *PagePointer* to the *page* containing the key at beginning of the range. The client fetches this *page* initially over one-sided READ and then follow the link (*PagePointer*) to fetch the subsequent *pages* over one-sided READ until it has found the end of the range. Since the data is *sorted* and *stored*, following the link from the beginning of the range will lead to the end of the range.

To perform an insert or update operation, the client has to send the insert or update message to the database server over two-sided RDMA. The database server, upon receiving the message, will perform a lookup in the index and respond to the client with the *PagePointer* to the *page* in which the update or insert operation has to performed. Referring to the *PagePointer* received from the database server, the client will first fetch this *page* from the database server over one-sided READ. It will then insert or update the key-value pair and write the updated *page* back to the database server to the same location where the original *page* was using one-sided WRITE. In case a new *page* has to be inserted, the client will fill the *page* with the required key-value pairs and write the *page* to the database server over one-sided WRITE. The original *page* is also rewritten over one-sided WRITE with the modified *PagePointer* pointing to the new *page*. It will then send a message over two-sided RDMA to the database server indicating that a new *page* has been inserted and the index has to be updated. The database server will rebuild the index to accommodate the new *page* added by the client.

In this model, index lookup performed using two-sided RDMA is the most CPU demanding task. Insert and update operations are not very expensive because they involve the CPU to rebuild the index only in cases where a new *page* has been created and not for every operation. The advantage of this model is the low CPU overhead. There is a substantial decrease in the

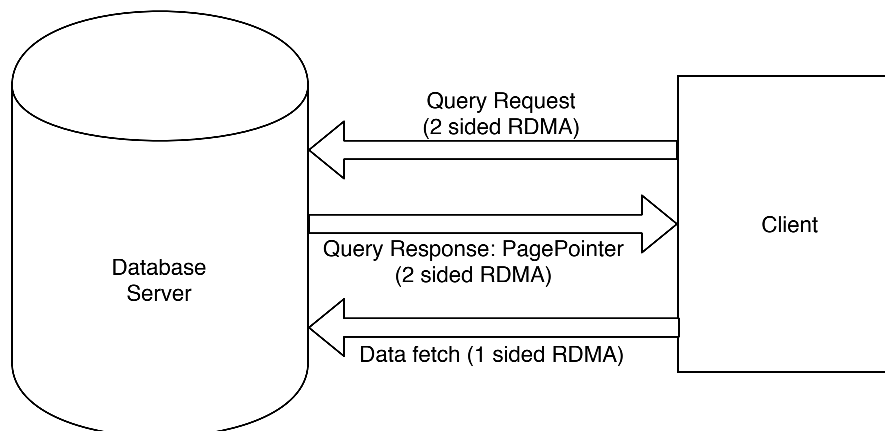


Figure 4.3: Hybrid Model Architecture

load on the CPU in this model, because the CPU is not involved in the process of sending the data to the client as compared to *pure2* model. In scenarios where the number of one-sided operations exceed the number of two-sided operations per query, we expect this model to perform better. For single key queries, which have equal number of two-sided and one-sided RDMA operations per query, this model is not suitable. For range queries, which have more one-sided RDMA operations than two-sided RDMA operations per query, this model is recommended. Although fetching each *page* over one-sided READ costs the client one RTT, since it is performed entirely using the NIC, it is much faster than having all the *pages* sent to the client over two-sided messages.

4.5 Variant 3: Pure 1 Model (pure1)

This model is designed to run using only one-sided RDMA. The database server does not respond to clients in this model. It is responsible only for storing the data and the index. With least CPU utilization, its only task is to stay online allowing clients to connect to it. The clients will establish a RDMA connection to the database server. First, they will traverse the index structure using one-sided RDMA. The root node of the index structure will be stored at a predefined address (offset) in the RDMA registered memory on the database server which the clients will be aware of. The clients will always begin index traversal from this memory address. Once the location of the interested data has been determined from the index, the client will then fetch the data using one-sided READ. The client has to fetch the *page* to know if the key-value pair it is interested in is present or not. Unlike other models, the client cannot be notified about an invalid query beforehand. In case of range queries, the client will determine the location of the starting key by traversing the index, fetch the *page* and follow the links (*PagePointer*) in each *page* to fetch the remaining *pages* until the last key of the range is found. Since the data is *sorted* beforehand and stored, following the *PagePointer* in each *page* will lead to the end of the range.

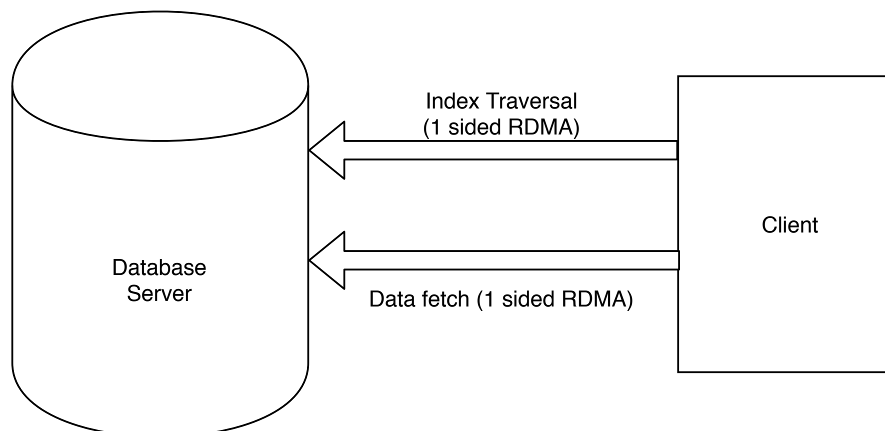


Figure 4.4: Pure1 Model Architecture

To insert a new key-value pair or modify an existing key-value pair, the client has to first lookup the index for the location of the *page* to perform the operation. The process of looking up is by traversing the index using one-sided RDMA as described above. Once the location of the required *page* has been determined, the client will fetch the *page*. The insert or update operation on the *page* is performed on the client side. Once the operation is completed, the updated *page* is written back to the same address overwriting the original *page* using one-sided WRITE. In case the original page is full, a new *page* will be created at the client side, filled with the new key-value pairs, then written to the database server over one-sided WRITE. The original *page* will also be rewritten with updated *PagePointer* pointing to the new *page* using one-sided WRITE. The process of updating the index is also performed using one-sided WRITEs. The *root-index-page* will be first locked using one-sided WRITE. Then the client will update the index structure and then release the lock on the *root-index-page* using one-sided WRITE.

This model is expected to have the least CPU overhead because the database server is not doing any work apart from just staying online. It consumes minimum CPU to be alive. All tasks are performed by the clients over one-sided RDMA. This model should scale better as long as there is bandwidth remaining on the link. The concern in this design is that each query will involve multiple RTTs for performing index lookup and then more RTTs to fetch the data, when compared with the previous two models where the cost of index lookup for the client was just one RTT. We believe that, the additional cost incurred in *pure1* will be negligible compared to having the index lookup performed by the database server itself. The reason being, one-sided operations are handled entirely by the NIC. A dedicated hardware is performing the task here which will be much faster than the CPU on the database server performing it. This will become more evident when the system scales, at which point this design will not have additional complexities of context switch and thread switch at the database server unlike the other two models. For the same reasons we also expect the throughput of this model to be somewhat similar for both single key queries and range queries.

Chapter 5

Implementation and Evaluation

5.1 Implementation

The database server and the clients have been implemented in *C++ 11*. The RDMA functionalities in this project uses the verbs API offered by OFED *2.3.1* driver [3]. The database server is a generic class called *OrderedStoreServer*. The client is also a generic class called *OrderedStoreClient*. Each database server is assigned a unique *Node ID* which the client will use for identifying the database server. A *message* is a structure (*struct*) used in two-sided RDMA communication. It will contain a command field (*uint8_t*) used to specify the type of query and status of an operation, start (*int*) and end (*int*) fields for single key and range queries, *Node ID* (*uint8_t*), RDMA offset (*size_t*) and size (*size_t*) which are used to send *PagePointer* information as query response.

A *page* is a generic C++ class which can hold any data-type key-value pairs. The configuration allows the user to choose the capacity of each *page* in terms of the number of key-value pairs and the number of slots to be left empty for future inserts. The *PagePointer* is a structure (*struct*) which consists of Node ID (*uint8_t*), RDMA offset (*size_t*) and fillgrade (*size_t*). The *version* and *lock* are combined together into one entity (*uint8_t*). The version number takes up the first 7 bits and the lock bit is the last bit (LSB). *PagePointer*, *version* and *lock* are packed before the key-value pairs in the page. When the client READs the fillgrade size from the beginning of the page, it will be able to fetch all the meaningful parts of the *page*. The *index-page* is an object of the page class with the type of value being a *PagePointer*.

The project implementation is available in the Brown github repository `https://github.com/BrownBigData/istore2`.

5.2 Experimental Setup

All the experiments have been conducted on an 8 node cluster. Each machine in the cluster has a 40 core *Intel Xeon E5-2660* processor with 256GB memory. The operating system in each

of these machines is the *Ubuntu 14.04 LTS*. All the machines are equipped with the *Mellanox Connect IB FDR 4x dualport Host Channel Adapter* which offers a full duplex bandwidth of 6.8GBps per port. The driver running on all the machines is the *OFED 2.3.1*. The 4 database servers run on 2 machines on the cluster with each server getting its own dedicated port. We use 100 million key-value pairs as the data in total in all the 4 database servers. We run 40 clients on each of the other 6 machines on the cluster which sum up to 240 clients. All experiments are performed scaling the number of clients from 1 to 240 with uniform and skewed workload. For skewed workload, the distribution of queries is 80%, 12%, 5% and 3%. The query selectivities for the experiments are single key, 0.1%, 1% and 10%.

Model	Data	Index
pure2	Range	Range
hybrid	Round-Robin	Range
pure1	Round-Robin	Round-Robin

Table 5.1: Data and Index Partition

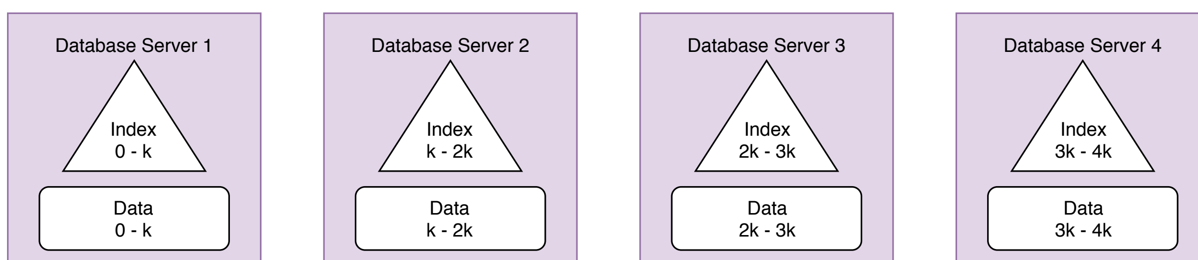


Figure 5.1: Pure2 Experimental Setup

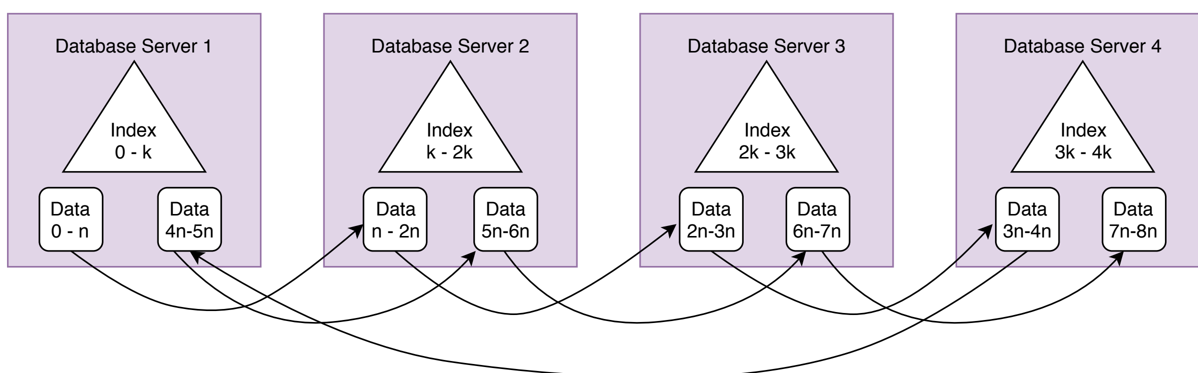


Figure 5.2: Hybrid Experimental Setup

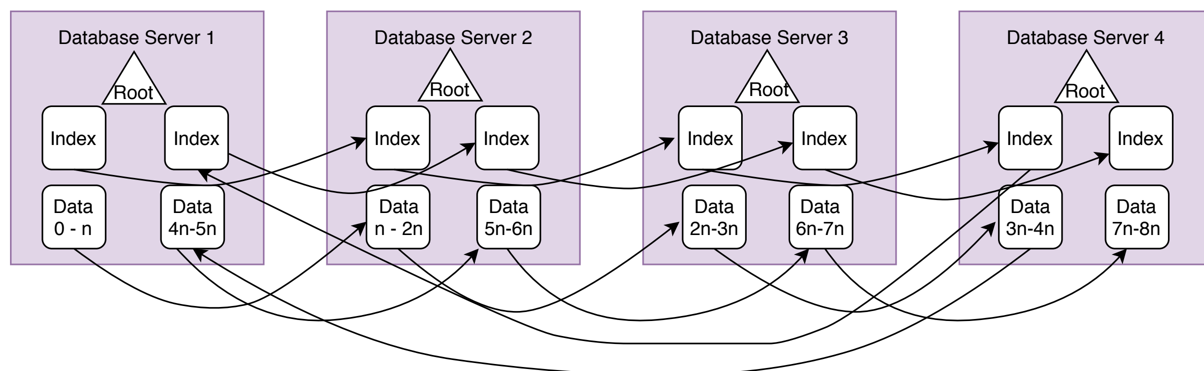


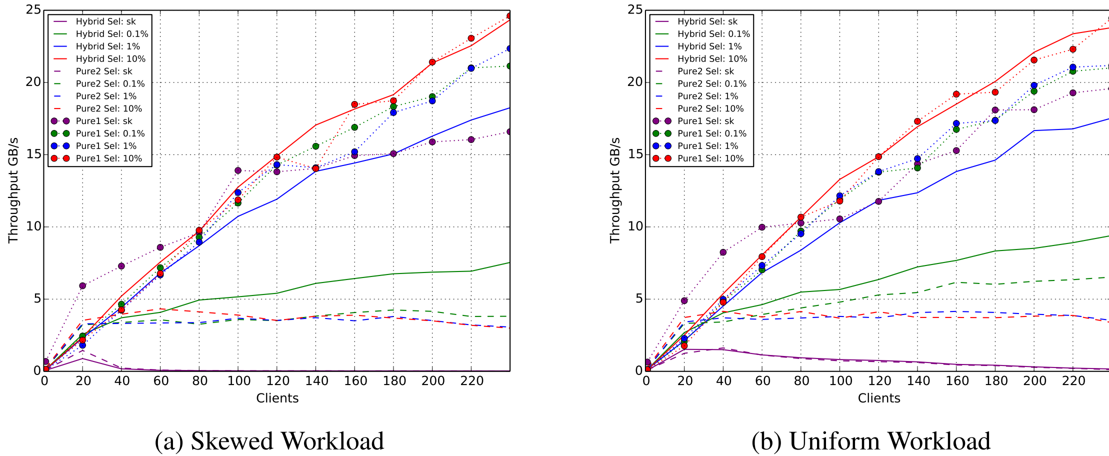
Figure 5.3: Pure1 Experimental Setup

5.3 Performance

5.3.1 Throughput

The throughput of the three models for various selectivities is shown in Figure 5.4. First lets discuss the plots for skewed distribution of workload shown in Figure 5.4a. For single key queries we see that *pure2* and *hybrid* have very low throughputs with the increase in the number of clients. Although *hybrid* uses One-sided RDMA for fetching data, since the number of one-sided operations is equal to the number of two-sided operations in single key queries, the benefit of one-sided RDMA is not maximized. Its interesting that *pure1* outperforms both *hybrid* and *pure2* in single key queries. Although *pure1* has to spend lot of RTTs in index traversal, because it is completely one-sided operation handled by the NIC, we see very high throughput. For range queries we see that the *hybrid* model performs better than the *pure2* when scaled. Since the number of one-sided operations is greater than the number of two-sided operations in range queries, the benefits of one-sided RDMA is maximized. As the selectivity increases, since data fetch happens over one-sided RDMA in *hybrid*, we see it achieves higher throughput compared to *pure2*. There is not much difference in the performance of *pure1* and *hybrid* in case of range queries because in *pure1* except for index traversal, data fetch happens exactly the same way as *hybrid*. The plots for *pure1* with various selectivites run close to one another because of the complete utilization of one-sided RDMA.

Lets take a look at the plots with uniform distribution of workload shown in Figure 5.4b. The main difference we see here when compared to skewed workload is that, when the workload is uniformly distributed, *hybrid* performs slightly better for single key queries and lower selectivity range queries compared to *pure2*. When the load gets distributed evenly, the database servers can handle requests efficiently thereby serving more requests.



(a) Skewed Workload

(b) Uniform Workload

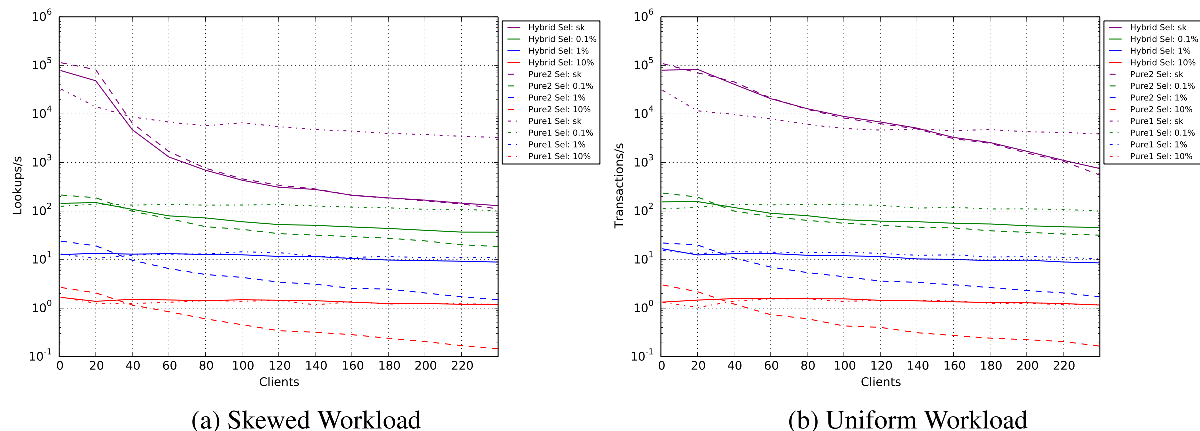
Figure 5.4: Throughput

5.3.2 Lookups

The lookups/s plots for all the three models are shown in Figure 5.5. The log-scale used in the plots will clearly show the performance improvements that *hybrid* and *pure1* have over *pure2*. As we increase the the clients, the drop in the lookups/s in *pure1* is less compared to the other two models. This is the advantage of not having the CPU involved in the index traversal. Since the operating system is not involved in the operation, this model scales extremely well as the number of clients increase. Beyond 20 clients, its evident that the performance of *pure2* drops significantly. This is because each of the database servers have 20 cores. Since one thread is created per client, after 20 clients, the CPU has to do a lot of work in handling thread switch. This will in turn impact on the performance of the model. In case of *hybrid*, this is a problem only for single key queries and range queries with lower selectivities. Since data fetch happens over one-sided RDMA in *hybrid*, we see that for range queries, there is not much drop in the performance as the number of clients increase.

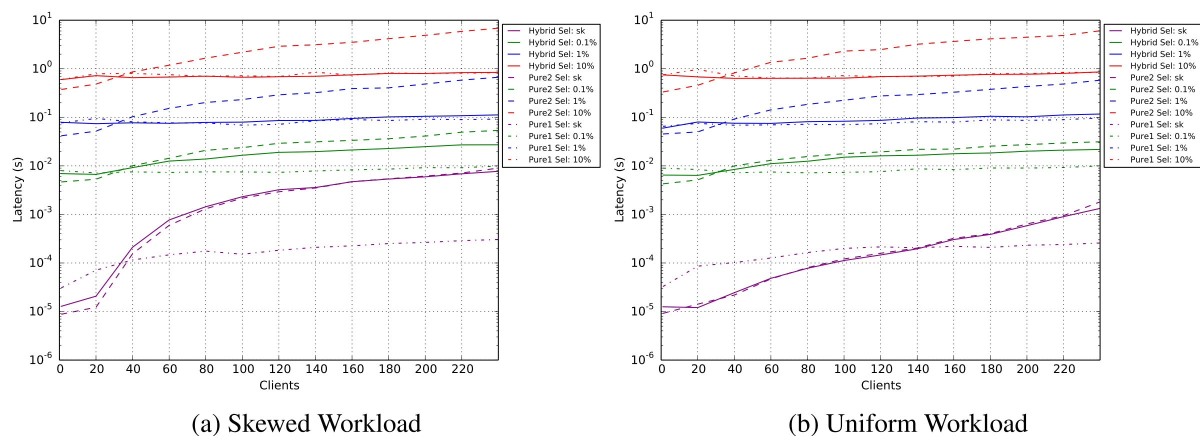
5.3.3 Latency

It is evident from the plots shown in Figure 5.6 that latency of each query increases significantly for *pure2* for all selectivities as the number of clients increases. In case of *hybrid*, we see substantial increase in latency for single-key queries and lower selectivity range queries only. As the selectivity increases meaning more one-sided operations compared to two-sided operations, there is not much change in the latency. *Pure1* always has the least latency compared to *pure2* and *hybrid* across selectivities when scaled. As the selectivity increases the behavior of *hybrid* becomes similar to that *pure1* and we can that their latency plots tend to merge eventually. Comparing the performance of *hybrid* for uniform workload against skewed workload, we can see that under uniform workload, for single-key and lower selectivity queries, the increase in



(a) Skewed Workload (b) Uniform Workload

Figure 5.5: Lookups/s



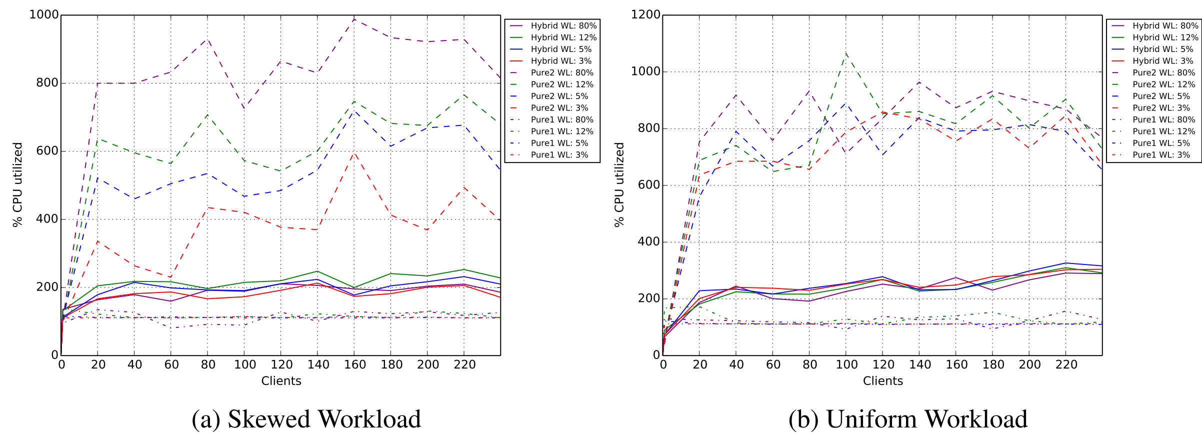
(a) Skewed Workload (b) Uniform Workload

Figure 5.6: Latency

latency is less as the number of clients increases. This is because of better distribution of load across all database servers.

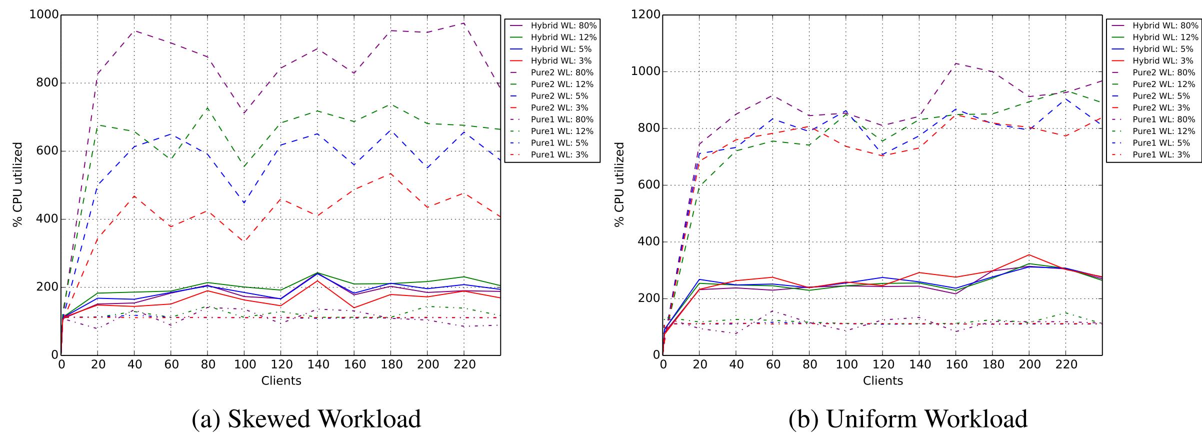
5.3.4 CPU Utilization

The CPU utilization is very high in *pure2* mainly because of the involvement of CPU in performing the lookup and sending data to the client. Managing the threads serving the clients will incur significant cost under *pure2* compared to *pure1* and *hybrid*. With no involvement of CPU, *pure1* consumes least CPU and is almost the same irrespective of the selectivities. For *hybrid*, in case of single-key queries and lower selectivities, we see slightly higher CPU utilization compared to higher selectivity range queries. This is because of the CPU utilization for index traversal. In single-key queries and range queries of lower selectivites, the index traversal happens more often than higher selectivity range queries. For both skewed workload



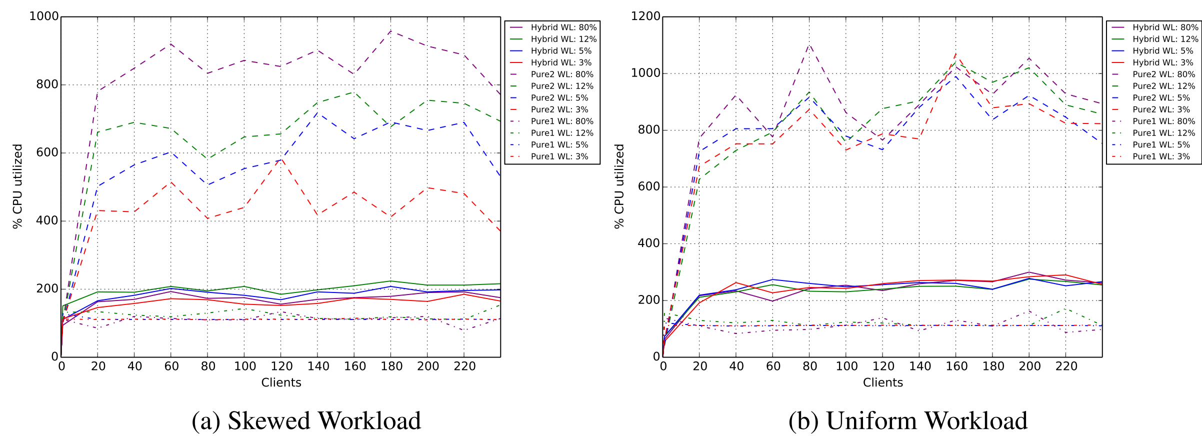
(a) Skewed Workload (b) Uniform Workload

Figure 5.7: CPU Utilization Sel: Single-Key



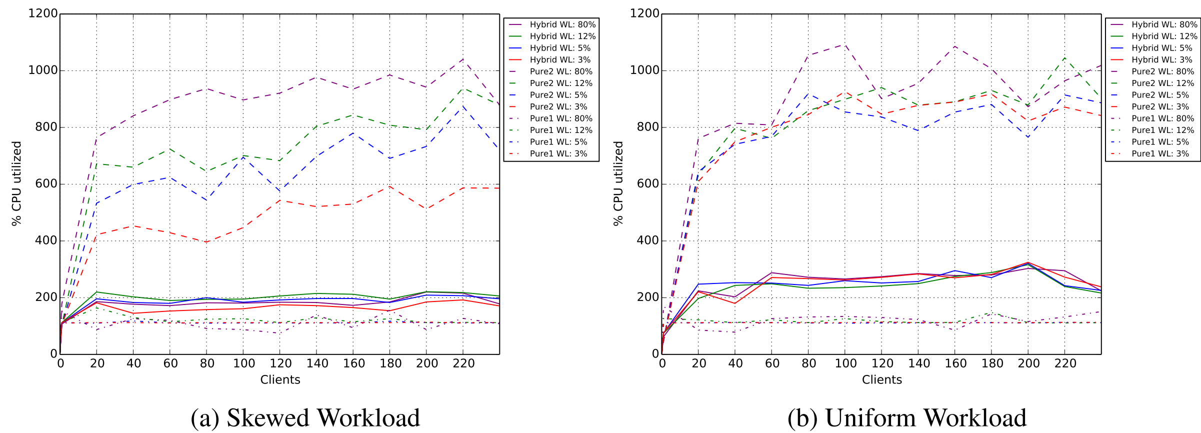
(a) Skewed Workload (b) Uniform Workload

Figure 5.8: CPU Utilization Sel: 0.1%



(a) Skewed Workload (b) Uniform Workload

Figure 5.9: CPU Utilization Sel: 1%



(a) Skewed Workload

(b) Uniform Workload

Figure 5.10: CPU Utilization Sel: 10%

and uniform workload, *pure1* and *hybrid* do not show spikes in CPU consumption as shown by *pure2*. These spikes are the result of frequent context switch and thread switch which happen in *pure2* which is much lower in *hybrid* and absent in *pure1*.

Chapter 6

Related Work

Network technologies are continuously evolving. A lot of research is being done in the area of High-Performance Networks. Intel Omnipath [22] is one of the most recent advances in this direction which aims to offer lossless and highly scalable network. The focus now is shifting towards utilization of these high-performance network technologies in applications beyond just super computers [20].

RDMA has been in use for over a long time now. Since recently, we see RDMA catching the attention of many researchers [13]. Because of its ability to bypass CPU, it is being used in large scale file-systems [19]. RDMA assisted iSCSI [2] can help improve the performance of Storage I/O significantly. Enterprise database management systems such as Oracle RAC [17] have started supporting RDMA, slowly embracing the technology. Improving the performance of RDBMS using RDMA [14] is also one of the focus areas. Some of the projects also explore query processing on RDMA [16]. At a much lower level, we see storage systems and key-value stores [1] [11] being designed with RDMA as the transport mechanism. These designs mostly incorporate the advantages of RDMA over existing design paradigms. They all treat RDMA as an afterthought instead of tailoring the design for it.

A transaction processing system designed using RPC based two-sided RDMA was proposed by FaSST [10]. According to this work, two-sided RDMA is required for better transaction processing in a distributed setup. It claims that one-sided RDMA will fail to scale. Through this project, we claim that, if the storage system can be rightfully designed for one-sided RDMA, then it can be highly scalable even in a distributed setup. A similar effort can be seen in FaRM [4] which proposes a main memory distributed transaction processing system specifically for one-sided RDMA using a message based approach. [3] forms the basis of this work. It proposes a novel architecture for in-memory database, making the full use of one-sided RDMA. Designing highly scalable transaction systems for RDMA networks [23] is another step in this direction. This novel database architecture shows that distributed transactions can indeed scale using RDMA based networks. It paves the way for discussion regarding the need for redesign in not just systems but also algorithms for distributed transactions for RDMA networks.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Through this work, we aim to show that high-performance networks can become an integral part of the design of database systems. We propose two novel in-memory distributed database designs: *hybrid* (using both one-sided and two sided RDMA) and *pure1* (using only one-sided RDMA). We compare their performance against *pure2* (using only two-sided RDMA) which is similar to traditional database architectures. It is interesting to see that *hybrid* scales linearly for larger selectivities while *pure1* scales linearly for any selectivity. Another interesting fact learnt through this project is that the excessive RTTs caused because of index traversal in *pure1* hardly impacts the performance of the system. The advantages of offloading a specific task to a dedicated hardware can be clearly seen here. Using 75% lesser CPU compared to traditional designs, both *hybrid* and *pure1* prove to be better designs than traditional ones. Thus, we can say that redesigning database architecture for RDMA networks can definitely be rewarding.

7.2 Future Work

As next steps we intend to focus on the following aspects of the project:

1. Hot-areas in the data: In a large-scale system, network can become a bottleneck when we have hot areas in the data. We plan to design the system such that hot areas can be identified by the database servers and immediately distribute them across multiple database servers. Since the *pages* are linked, moving data will only need a slight change in the *PagePointer* in the neighboring *pages*.

2. We have shown that these models use very little CPU. Having a dedicated thread running in the background to continuously update the index is not a going to be a costly investment. This will help reduce processing time of insert and update operations.

3. The index structure can be moved to the client side. This will improve the performance much more. But it will also pose us with the challenge of keeping the index up-to-

date. One of the strategies we plan to explore is to update the index lazily. RDMA can read larger blocks of memory more efficiently than smaller blocks [2]. Therefore, while performing a query, the clients can fetch neighboring pages as well at no extra cost and keep the index updated.

Bibliography

[1] Anuj Kalia, Michael Kaminsky, David G Andersen. Using RDMA efficiently for key-value services. In *Proc ACM SIGCOMM 2014*, pages 295–306, Chicago, IL, USA, Aug 2014. ACM.

[2] J. Liu, D. K. Panda, M. Banikazemi. Evaluating the impact of RDMA on storage I/O over infiniband. *SAN - 03 Workshop (in conjuction with HPCA)*, 2004.

[3] Alex Galakatos Tim Kraska Erfan Zamanian Carsten Binnig, Andrew Crotty. The End of Slow Networks: It's Time for a Redesign. In *Proc of VLDB 2016*, volume 9, pages 528–539, New Delhi, India, Sep 2016.

[4] A. Dragojevic, D. Narayanan, O. Hodson, M. Castro. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, USA, Apr 2014. ACM.

[5] J. Chu and V. Kashyap. *Transmission of IP over InfiniBand (IPoIB)*.

[6] O. Polychroniou et al. Track join: distributed joins with minimal network traffic. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1483–1494, Snowbird, Utah, USA, 2014. ACM.

[7] Transcend Information Inc. Transcend DDR FAQ.

[8] Infiniband Trade Association. *Infiniband Architecture Specification Volume 1, general specifications*.

[9] Intel Corporation. *Intel Omnipath Host Fabric Interface*.

[10] Anuj Kalia, Michael Kaminsky and David G Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (rdma) datagrams rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI Í6)*, pages 185–201, Savannah, GA, USA, 2016. ACM.

[11] T. Szepesi, B. Cassell, B. Wong, T. Brecht, X. Liu. A decoupled, client driven, key-value store using RDMA. In *IEEE Transactions on Parallel and Distributed Systems*, pages 3537 – 3552. IEEE, July 2017.

[12] Mellanox Technologies. *Introducing 200G HDR InfiniBand Solutions*.

[13] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen,T. Moscibroda. Multi-path transport for RDMA in datacenters. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, USA, Apr 2018. USENIX.

[14] Feng Li, Sudipto Das, Manoj Syamala, Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In *Proceedings of the 2016 International Conference on Management of Data*, pages 355–370, San Francisco, CA, USA, July 2016. ACM.

[15] W. Rodiger, T. Muhlbauer, P. Unterbrunner, A. Reiser, A. Kemper, T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *IEEE 30th International Conference on Data Engineering (ICDE) 2014*, Chicago, IL, USA, 2014. IEEE.

[16] W. Rodiger, T. Muhlbauer, A. Kemper, T. Neumann. High-speed query processing over high-speed networks. In *Proceedings of the VLDB Endowment*, volume 9, pages 228–239. VLDB, Dec 2015.

[17] Oracle Inc. *Oracle Real Application Cluster*.

[18] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye and M. Lipshetyn. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 conference on ACM SIGCOMM*, pages 202–215, Florianpolis, Brazil, 2016. ACM.

[19] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekhar, H. Wang, H. Subramoni, C. Murthy, D. K. Panda. High performance RDMA-based design of HDFS over infiniband. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Salt Lake City, UT, USA, Nov 2012. IEEE.

[20] X. Lu, N. S. Islam, M. W. Rahman, J. Jose, H. Wang, H. Subramoni, D. K. Panda. High-performance design of hadoop RPC with RDMA over infiniband. In *42nd International Conference on Parallel Processing (ICPP), 2013*, pages 641–650, Lyon, France, Oct 2013. IEEE.

[21] Y Chen R Chen H Chen X Wei, J Shi. Fast in-memory transaction processing using RDMA and HTM. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, USA, Oct 2015. ACM.

[22] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, Robert C. Zak. Intel omni-path architecture: Enabling scalable, high performance fabrics. In *Proc. 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI)*, Santa Clara, CA, USA, Aug 2015. IEEE.

[23] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proceedings of VLDB Endowment*, 10(6):685–696, February 2017.