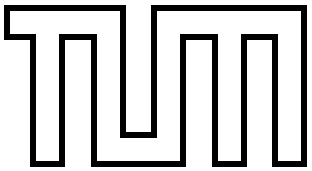


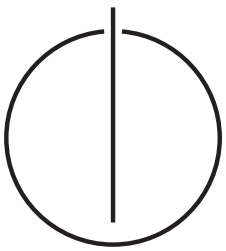
FAKULTÄT FÜR INFORMATIK

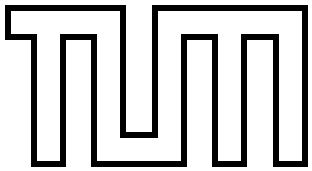
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

Efficient Distributed Join Processing on Modern Hardware

Alexander van Renen





FAKULTÄT FÜR INFORMATIK

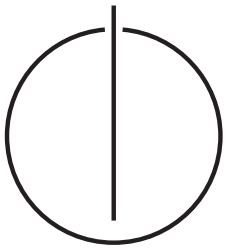
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

Efficient Distributed Join Processing on Modern Hardware

Effiziente verteilte Join-Verarbeitung auf moderner Hardware

Author:	Alexander van Renen
Supervisor:	Prof. Alfons Kemper, Ph.D.
Advisor:	Wolf Rödiger, M.Sc.
Date:	Oktober 15, 2012



I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Oktober 2012 Alexander van Renen

Acknowledgments

I would like to give my gratitude to the people of the database chair at TUM, who have earned it by supporting me in writing this thesis. At first I would like to thank Henrik Mühe, M.Sc., for introducing me to the topic of distributed join processing and further for many fruitful discussions. Second, and above all I owe a great amount of gratitude to my supervisor, Wolf Rödiger, M.Sc., who supported my work in each and every step. Many thanks for all the invaluable countless hours we spent planing, thinking and discussing. Furthermore, I would like to thank Prof. Alfons Kemper, Ph.D., who used his advanced knowledge in the field of database systems to provide me and my supervisor with new perspectives, directions and options whenever we needed them.

Abstract

The ever increasing volume of data in scientific as well as commercial information systems calls for scalable databases. Distributed database systems can satisfy this demand as they are scalable to higher performance requirements simply by adding more compute nodes. However, they require efficient ways to query distributed relations to achieve overall high performance.

This thesis describes three efficient distributed algorithms for joining two large distributed relations. Each algorithm is designed for a specific network topology. Our most efficient design – the distributed Radix Join – achieves a throughput of 80 million tuples per second on a cluster, consisting of 16 standard desktop computers. Furthermore, a detailed analysis of each algorithm is provided and it is shown that the performance of distributed join algorithms on today's commodity hardware is bound by the underlying network bandwidth.

Kurzfassung

Die ständig zunehmende Menge an Daten in sowohl wissenschaftlichen als auch kommerziellen Informationssystemen erfordert skalierbare Datenbanken. Verteilte Datenbanksysteme sind in der Lage diese Nachfrage zu befriedigen, indem sie die hohen Performance-Anforderungen einfach durch das Hinzufügen weiterer Rechnerknoten erfüllen. Allerdings erfordern diese sorgfältig entwickelte Algorithmen für die Anfrageabarbeitung auf den verteilten Relationen, um insgesamt eine hohe Leistung zu erzielen.

Die vorliegende Arbeit beschreibt drei effiziente Algorithmen für die verteilte Join-Bearbeitung auf zwei großen Relationen. Jeder Algorithmus ist für eine bestimmte Netztopologie ausgelegt. Unser bestes Design – der verteilte Radix Join – erreicht einen Durchsatz von 80 Millionen Tupel pro Sekunde auf einem Cluster, bestehend aus 16 Standard-Desktop-Computern. Ferner stellt diese Arbeit eine detaillierte Analyse jedes Algorithmus bereit und zeigt, dass die Leistung von verteilten Algorithmen auf heutiger Hardware durch deren Netzwerkbandbreite limitiert ist.

Contents

1 Introduction

A fundamental part of computer science is the development of computing systems for storing, transforming and interpreting information. Today many applications use a database management system to store their data. Database systems provide an abstraction of the underlying storage hardware. They take the responsibilities, like checking for inconsistencies, providing redundancy or even hiding a computer cluster, away from the application.

1.1 Database Systems

Besides offering a convenient way for storing information, database systems provide an easy access to the information they contain. In addition they allow the user to efficiently aggregate the data in various ways. This is achieved by carefully engineered and optimized algorithms for processing huge amounts of data.

Distributed Database Systems

Some recent studies by [4] have shown a current global IP network traffic of 43.4 exabyte per month and forecasted a growth to 110.8 exabyte per month in the year 2016. This rapid increase of global network traffic is fueled by an exponential growth rate in network bandwidth, as predicted by Nielsen's Law.

With this powerful machinery as a backbone, companies have spread out internationally with branches all over the globe. There are internet warehouses like Amazon, which are selling over a billion articles per year. Online games provide access to virtual worlds for millions over of players. Facebook created a world wide social network with over 800 million active users. Search engines like Google or Yahoo answer almost 100 million request a day.

One thing all these companies have in common is the need for distributed data management. With data being a vital part of each companies success, they have to prevent data loss. Furthermore, it is necessary to ensure fast access and, when dealing with personal data, deny access to third parties. These are some of the basic requirements, which are fulfilled by distributed database management systems.

Using a distributed database system is not only required for handling globally distributed data. It is also necessary for dealing with the sheer size of (for example) scientific data. By using the computing power and storage capabilities of hundreds of computing nodes instead of only one, the distributed database system can provide more performance. Distributed database systems have to be very robust in order to tolerate hardware faults, which are getting more and more likely with an increasing number of nodes in a system.

Main Memory Database Systems

A high end cluster from 1994 as described in [11] had a main memory size of 10 GB and cost about 1.6 million dollars back then. Today, one can buy a single computer equipped with the same amount of storage for under a thousand dollars. The capability of main memory is rapidly increasing and allowing for servers configurations with over a terrabyte of memory.

This opens the way for database systems to keep their entire data resident in the main memory of the system. Thereby avoiding expensive hard disc operations, which increases performance and erases the disc as a bottleneck. This trend is likely to continue as the increase of available main memory continues.

Intention

The idea of a distributed database system can be applied to a main memory database system thus forming a distributed main memory database system. This combination essentially provides the benefits of both architectures and is likely to be the future for distributed database systems.

The most important operation in any database system during query processing is the join. This operation combines two relations using a key attribute. Its optimization has been the focus of many researchers over the past decades and with hardware constantly evolving the title of the best algorithm is always switching.

This thesis picks up the concept of a distributed main memory database system and assumes a continuing growth in network bandwidth. It tries to develop an efficient distributed join algorithm design in this environment.

1.2 Outline

The rest of the thesis is structured as follows. In chapter 2 an overview of commonly used network topologies, a selection of standard network protocols used in this thesis and a description of basic join algorithms is provided. The next chapter – chapter 3 – describes the design of three distributed join algorithms. A detailed analysis of the experimental evaluation of each distributed join algorithm, including an in-depth comparison to each other can be found in chapter 4. The conclusion (chapter 5) presents the most important results of this work together with an overview of work related to this thesis.

2 Background

This chapter describes the characteristics of different network topologies, networking protocols and basic join algorithms, which are important for the development of distributed join algorithms. Each section provides a general overview of its topic and highlights the most important aspects. The first section describes different network topologies, in which the join algorithms operate. It is important to understand their different behaviors, because the distributed join algorithms, developed in this thesis, depend strongly on the structure of the underlying network. The next section provides an insight into the transport protocols used to send data across the networks. The last section of this chapter describes two of the fundamental local join algorithms – the sort-merge join and the hash join.

2.1 Network Topologies

This section discusses common ways to connect nodes in a network. Let n be the number of nodes in the network. Figure 2.1 shows an overview of the discussed network topologies.

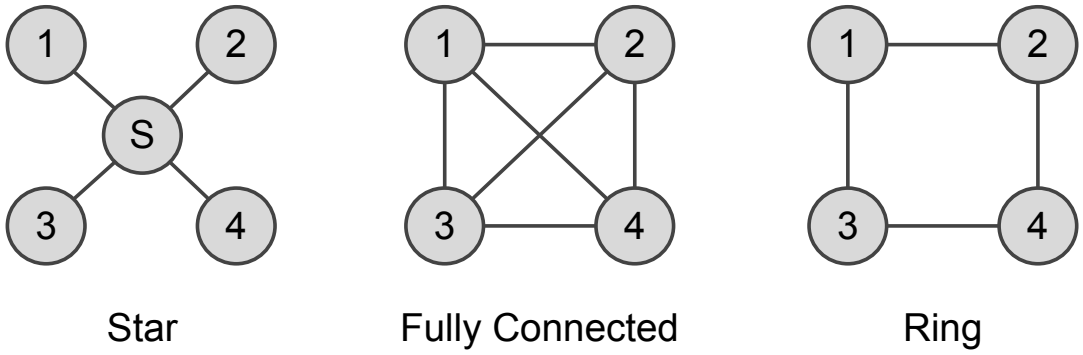


Figure 2.1: Common network topologies.

Star

A star topology consists of one special node (called server), which is connected to all other nodes (called clients). This topology does not directly map to hardware, as the server would require a massive amount of connections. Usually the server is connected with a fast connection to a switch. Each client is connected to this switch using a normal connection. As the size of the network grows the clients are no longer directly connected to the server switch but through other switch, thus forming a tree with the server switch as the root.

In theory the clients do not know each other and only communicate with the server, as in our implementation of the Broadcast Join in section 3.2.

Fully Connected

In a fully connected network each node is able to send data to every other node. A node requires $n-1$ connections and a total of $\frac{n^2-n}{2}$ connections is needed in the network. In practice this becomes very soon to expensive and is solved by using a star topology, where each node is connected to the same switch. Note that this only changes the physical topology to a star, the logical topology is still a fully connected network.

With each node having a fixed up and down link (each $u\frac{bit}{s}$) it is necessary to avoid cross traffic. Consider a network with three nodes, where node two and three are sending data to node one. The problem is that node one is only able to receive $u\frac{bit}{s}$, thus slowing down the send rate of node two and three to $\frac{u}{2}\frac{bit}{s}$ each. This effect is studied in more detail in section 6.1.

Ring

In a ring topology a node is only connected to its neighbors. This topology can be mapped directly to hardware, as each node only needs two connections, and therefore only two cables. A problem arises when node one needs to send a message to node three, because theses nodes are not directly connected. This problem is usually handled by passing data in one direction through the ring. Hence a message from node one to node three has to cross node two. The answer would cross node four.

This uncommon behavior makes designing an efficient join algorithm for this network an interesting problem. We implemented the Cyclo Join on top of this network in section 3.1 as described in [8] and [9].

2.2 Network Protocols

When one is designing an efficient distributed join algorithm, it is important to make best usage of the underlying network in order to achieve maximal performance. Hence, one has to be aware of the available network transport protocols. This section provides an overview of the three different protocols used in this thesis.

The first one is the Transmission Control Protocol (TCP). It takes care of delivering packets in an order preserving manner and handling lost packets. Due to its nice properties it is the default choice for many networking applications. It is supported by any networking card and uses the CPU to process the incoming packets.

The next protocol used, is the User Datagram Protocol (UDP). It is widely used in games, video streaming and other areas where low-latency and high throughput is necessary. The protocol does not guarantee to preserve the order of send packets or that sent packets arrive at the destination. It provides the option to broadcast a packet to a network, which can help a server to distribute large amounts of data to many clients. Like TCP it processes the incoming packets on the CPU and can be used with almost every networking card.

The last protocol, used in this thesis, is the Internet Wide Area RDMA Protocol (iWARP). It uses a new technology, called Remote Direct Memory Access (RDMA). As the name suggests this technology enables an application to directly copy main memory data to a connected nodes main memory. By using a so called RDMA enabled Network Interface Controller (RNIC) this network protocol does not require any involvement of either computers CPU or operating system. This makes iWARP a good choice for high performance clusters, but due to the need of expensive RNICs and other issues (see 2.2.3) inapplicable for personal computers.

2.2.1 Transmission Control Protocol (TCP)

Today the Transmission Control Protocol (TCP) as described 1981 in [20] is the most popular network protocol. Due to its age it has a view shortcomings when applied in todays high speed networks. This subsection describes the design of TCP in order to infer these shortcomings.

TCP Properties

TCP is a connection oriented protocol, which means that two nodes have to create a connection before they are able communicate. TCP handles lost packets by resending them and it guarantees that the order in which the packets are send by the server is preserved. This incurs an overhead each time a packet is lost or damaged, because the client has to wait until the server sends it again. But with the chance of losing a packet approaching zero on standard short distance cable connections, this is no real problem for our networks.

Establishing a TCP Connection

To establish a TCP connection between two nodes, one node acts as a server and the other as a client. The server node sets up a server socket and listens on it. The client node sets up a clients socket and connects to the server socket. Thus creating a connection between the two nodes. This connection is bidirectional, which means that both – the server and client node – are able to send and receive data on it.

The server node can accept multiple connections on the same server socket, hence the TCP protocol supports the Server-Client-Pattern. When sending data to all clients the server node has to send the data on all connections. Therefore when publishing data to 100 clients the server has to each packet 100 times.

Note that in a fully connected network each node has to create a server socket and accept a connection from each other node.

Data Transfer in TCP

Figure 2.2 illustrates the data transfer from a server node (left hand side) to a client node (right hand side) using TCP. In this scenario the server wants to send the content of a chunk of memory (represented by 1) to the client. Therefore the server calls the system function *write*, which switches to kernel mode, copies the memory to a kernel buffer and packs it into TCP packets (represented by 2). The Network Interface Controller (NIC) accesses

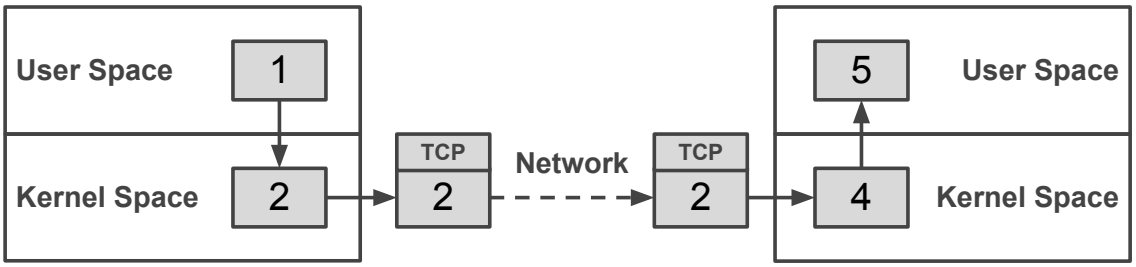


Figure 2.2: Data flow in TCP.

those using Direct Memory Access (DMA) and takes care of pushing the packets to the physical cable. Note that the data crosses the memory bus three times.

- Read data from application buffer (done by CPU).
- Write data to kernel buffer (done by CPU).
- Read data from Kernel buffer (done by NIC).

After the data crossed the network, the same happens on client side in reversed order. First the data is placed in the kernel buffer by the NIC using DMA (represented by 4). When the user calls *read* on the socket, the CPU will unpack the TCP packets and place the data into the user supplied buffer (represented by 5). After that the application on the client node is able to use the data.

Problems in TCP

This top level and slightly simplified view of TCP illustrates three problems, which one has to be aware of.

- The CPU overhead is the most obvious. To process the TCP stack (packing and unpacking the TCP packets, including checksums) CPU time is required. Thus limiting the available CPU time for other applications. As a rule of thumb: 1 GBit/s network needs 1 GHz of CPU power.

- A second source of overhead is the expensive copying behaivior. The reason for this is the specification of the socket interface. As shown before, the data has to cross the memory bus three times on both sides – client and server. Hence 3.75 GB/s of main memory bandwidth is used when sending or receiving data with a 10 GBit/s link. With a standard DDR3-1600 random access memory having a theoretical bandwith of 12.8 GB/s, this is a considerable number.

- The last and most subtle overhead are interrupts. The NIC needs to notify the CPU when new data is available by sending an Interrupt, thus suspending the currently active process. Theses issues are discussed in more detail in chapter 2.1 of [9].

Observations

With us only having a 1 GBit/s ethernet at our disposal, these issues are of lesser importance here. These problems only become relevant when using faster networks, as discussed in [8].

TCP provides a reasonable utilization of the available network bandwidth and takes the responsibility to ensure the order and the completeness of the data stream off the programmer. All join algorithms presented in this thesis use TCP for control and synchronization messages. Moreover the Radix Join uses TCP to distribute the actual data.

2.2.2 User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) as described 1980 in [19] provides no features to ensure a complete transmission of the data. But therefore offers a higher throughput and a lower latency than TCP. Both protocols were developed at roughly the same time in the same hardware environment. Therefore UDP uses the same socket based interface as TCP to move data between the network and the applications (described in 2.2.1). Consequently UDP has the same three issues as TCP, when applied to high speed networks. This subsection provides an overview of UDP.

UDP Properties

UDP is not connection oriented, consequently there is no need to establish a connection between two nodes. The server sends data to a specified port on the client's node. If there is an application on the client's side reading on this port it receives the data.

In addition UDP neither guarantees to preserve the order of the packets nor to ensure a complete transmission (packets may get lost). When using TCP and the client receives a damaged packet or a packet gets lost, the clients TCP implementation has to stall until the packet is resend. Thus stalling the application waiting for the data. In UDP this is not necessary and provides a huge potential for performance increase, as the application never gets stalled. Of course this is only helpful if the client handels the dataloss appropriately, like in video streaming, where it does not matter if one image gets lost, or in our implementation of the Broadcast Join, where the order does not matter (see section 3.2).

One can use UDP for broadcasting data to a subnet. Using this feature it is possible to distribute data from one server to a arbitrary number of nodes (assuming all nodes are in the same subnet) while sending only once.

Observations

UDP provides the broadcasting feature, which is very interesting for distributing huge data chunks to multiple targets. On the downside the potential lost of data is unacceptable in distributed database systems and needs to be handled properly. We show how to solve this, without implementing a TCP clone on top of UDP in section 3.2, which covers the broadcast join.

2.2.3 Internet Wide Area RDMA Protocol (iWARP)

Using the Internet Wide Area RDMA Protocol (iWARP) applications are able to access the main memory of another node. By providing, an entirely new interface iWARP overcomes the issues of TCP (and in advance UDP). It allows the RNIC to directly place data into the main memory, without any CPU involvement.

RDMA

Remote Direct Memory Access (RDMA) is a technology to access the memory of a remote system. It is possible to transfer data from one computer's memory into an advertised buffer on another computer without using the CPU of either system. The iWARP protocol is based on RDMA. Therefore it combines a TCP offloading Engine (TOE), which allows to process the TCP stack on the NIC, and a DMA engine to move the data from the NIC to the main memory of the system. Thus eliminating all three TCP issues discussed in section 2.2.1. With iWARP being the standard protocol when using RDMA the two terms – RDMA and iWARP – are used interchangeable throughout this thesis.

RDMA was originally used with InfiniBand [22] but has been adapted for Ethernet [1], therefore no special network is required for this setup.

Softiwarp

Softiwarp is a software implementation of the iWARP interface by [9], used in systems without a RNIC. This obviously does not provide the benefits of a hardware solution, thus having the same performance drawbacks as TCP. But with RNICs being an expensive pieces of equipment Softiwarp helps in the development process, e.g. for testing. Moreover the two systems are interchangeable, which increases the portability of the software. In addition, it is possible to connect the Softiwarp system with a hardware iWARP system. This can be used in a client server architecture, where the server uses iWARP and the clients use Softiwarp. Hence the server benefits from the advantages of iWARP and the clients do not need the expensive RNICs.

Using iWARP

RDMA/iWARP is a connection-based protocol. When connected, the nodes use send and receive requests to transmit data. A request contains the address and size of a the memory region, which should be send or received. After placing a request the RNIC asynchronously starts working on it and creates a completion notification when finished, which can be checked by the application.

When sending or receiving data the RNIC uses DMA to fetch or place the data in the main memory of the system, thus directly reading the data from the user space buffer and avoiding the extra memory bus passes of TCP. During this time it is necessary that the data does not move (for example the operating system is not allowed to swap the data to a hard disc). To ensure this the data has to be pinned by the application before creating the send or receive request. Pinning data requires kernel activity, which makes it an expensive operation. It is important to reuse already pinned buffers when using RDMA/iWARP. The data transfer is done without any CPU involvement. By using the

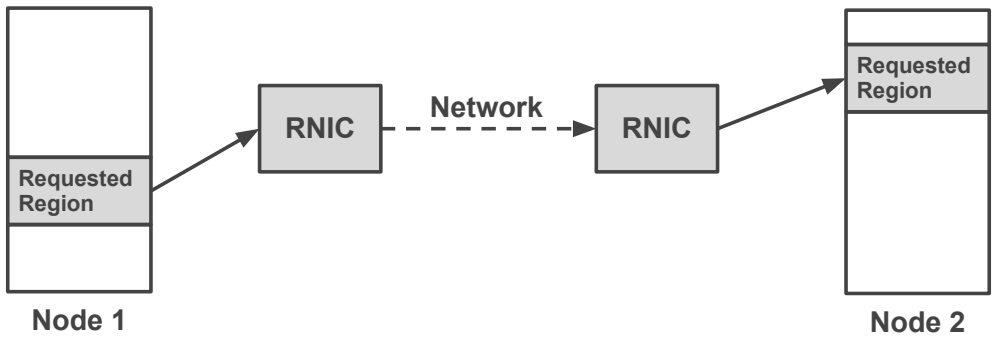


Figure 2.3: Data flow in iWARP.

request-based interface to transfer chunks of memory other than single packets there is no need for using interrupts during communication.

Note that the RDMA/iWARP approach for data transfer avoids all three problems of TCP or UDP described in section 2.2.1, thus making it an excellent candidate for high speed networks. But as one might expect there are a few drawbacks, besides the new interface, which has to be learned. The first one, as noted above is the need for pinning the memory regions, which is an expensive operation and limits the available memory of the system. In order to benefit from RDMA/iWARP it is necessary that the application is able to use its internal buffers directly for the data transfer, otherwise – when the application has to create a buffer, pin the buffer and copy its data into the buffer – it is just a TCP clone (with a TCP offloading Engine (TOE)) on top of RDMA, by moving the additional memory bus passes in the application code. The last issue is the RDMA set up time (about $200ms$ see [9]), which makes it inapplicable for applications, which use the connection only for a short data transfer.

Observations

RDM provides an efficient way to transfer data in a high speed network, by shifting CPU work to the RNIC. As explained above it is necessary that the application is able to use the internal buffers directly for sending. This, if possible, requires a thoughtful design. Because of the time required to create a connection, it is useful to have a static network, which gets reused in the application. For example a database system could set up an RDMA network once and use it for all operations.

For further reading on RDMA see [9]. A sample code using directly the C interface can be found in [7]. A C++ wrapper class implementation for easy an usage is provided in this thesis.

2.3 Local Join Algorithms

In database systems it is often necessary to combine two relations. Consider a database system representing a warehouse. This database probably contains a relation for customers

and one for orders.

$$\text{customers} := \{(\text{id},\ \text{name}) : \text{id} \in \text{Integer}, \text{name} \in \text{String}\}$$
$$\text{orders} := \{(\text{id},\ \text{id}_{\text{customer}},\ \text{date}) : \text{id},\ \text{id}_{\text{customer}},\ \text{date} \in \text{Integer}\}$$

A typical question is to find all customers, who placed an order in the last month. This requires information of the customer relation and the orders relation. A naive idea is to create the cross product of the two relations and then select all tuples where the customer id matches.

$$\text{cross\_product} := \text{customers} \times \text{orders}$$
$$\text{result}_{\text{naive}} := \{(\text{id}_{\text{customer}}, \text{name}, \text{id}_{\text{order}}, \text{id}_{\text{customer}}, \text{date}) \in \text{cross\_product}\}$$

This approach creates the cross product of the two relations, which is of the size of $\Theta(n^2)$. This is even for relations of moderate sizes inapplicable. The solution is to combine the two steps.

$$\text{result}_{\text{join}} := \{(c,\ o) : c \in \text{customers} \wedge o \in \text{orders} \wedge c.\text{id} = o.\text{id}_{\text{customer}}\}$$

This operation is call a join. By combining the two steps it is possible to reduce the complexity of the join operation to $\mathcal{O}(n \cdot \log(n))$ for the sort merge join and to $\mathcal{O}(n)$ for the hash join (assuming realistic inputs, otherwise the complexity remains $\mathcal{O}(n^2)$). Both – the sort merge join and the hash join – are well known in centralized systems. The next two sections proved a brief review of both techniques. For more details and a state of the art implementation see [15].

2.3.1 Sort Merge Join

The sort merge join is a standard techniques for implementing a join operator in a database system. The basic idea is rather simple, at first both relations (called *R* and *S*) are sorted. After that the two relations are processed in a sequential order, starting with the lowest elements of both relations. If the current element of *R* is less than the current element of *S* no match is found and the algorithm moves only in *R* to the next element (vice versa in the inverse case). If the current elements are equal a match is found. With the keys being not necessarily unique, it is possible that the current *S* tuple has more join partners in *R*. Therefore the algorithm needs to loop over all succeeding elements in *R* with the same key and append theses matches to the output. After all matches of the current *S* tuple are found, the algorithm moves only in *S* to the next element and remains at the first matching element in *R*.

```
// sort both relations
S.sort();
R.sort();

// indices
int i, j = 0;

while(i<S.size() AND j<R.size()) {
   // S is smaller
   if(S[i] < R[j])
```

```
      i++;

   // R is smaller
   else if(R[j] < S[i])
      j++;

   // R equals S
   if(S[i] == R[j]) {
      int marker = j;
      while(S[i] == R[j]) { // loop over all mathes
         Output.add(S[i], R[j]);
         j++;
      }
      j = marker;
      i++;
   }
}
```

All implementations follow this basic idea and improve the single steps. At first it is necessary to use an efficient sort algorithm, because the sorting phase dominates the overall time in practice. One reason for this is that the scaling properties of the sorting phase are worse (in general $\mathbb{O}(n \cdot \log(n) + m \cdot \log(m))$ than those of the merge phase ($\mathbb{O}(n+m)$). With modern CPUs providing multiple cores it is important to exploit this parallelism, in order to create an efficient sort merge join, as demonstrated in [15] on a single processor machine or in [2] for an non uniform memory access (NUMA) architecture.

2.3.2 Hash Join

The other common algorithm for implementing a join operator is the hash join. During the past years the hash join has been the faster one, but as hardware is evolving very fast this might change in a few years as discussed in [15]. The basic idea of the hash join is as simple as the one of the sort merge join. Let there (again) be two relations *R* and *S*. The hash join algorithm decomposes in two phases: the build and the probe phase. The build phase constructs a hash table for the relation *R*. After that, during the probe phase, the algorithm does a lookup in the hash table for each tuple in *S*. If the lookup is successful a match is found, if not there is no math.

```
// build phase
hashTable = buildHashTable(R);

// probe phase
foreach(s in S) {
   // match
   r = hashTable.lookup(s);
   if(r != nullptr)
      Output.add(r,s);
}
```

One of the main problems when implementing a hash join – besides using efficient hash tables – is the memory access pattern. With the looked up addresses during the probe phase depending on the hash value of the current element, it is basically random. This introduces a lot of cache misses, which slow down the algorithm. Modern implementations try to overcome this problem by partitioning the relations in cache fitting partitions, see [15] or [17].

3 Distributed Join Algorithms

This chapter describes the three distributed join algorithms implemented during this thesis from an algorithmic standpoint. A comparison is not provided in this chapter but can be found in the next one (chapter 4). Each algorithm is tuned for a specific network topology and network protocol, giving it unique properties and advantages.

The first section presents the Cyclo Join algorithm. It uses a ring topology as a network, on which one of the input relations is rotated. The data is transmitted between the nodes of the network with RDMA. The next section describes the so called Broadcast Join, which uses a star topology. The central node, which is connected to all other nodes, servers as a server and broadcasts one of the relations to all client nodes using UDP. The last implemented distributed join algorithm is the Radix Join. In this approach both relations are radix clustered and each node processes a number of those partitions.

All distributed join algorithms presented in this thesis use two input relations called $\mathcal{R}$ and $\mathcal{S}$. One can imagine $\mathcal{S}$ standing for *Shared*, because this relation gets distributed (except in radix join, where both relations are distributed). The number of used computer nodes is referred to as n. Furthermore let $\mathcal{N}_\mathcal{R}$ denote the number of tuples in $\mathcal{R}$ and likewise $\mathcal{N}_\mathcal{S}$ for $\mathcal{S}$. In addition if a relation is distributed over several nodes then $\mathcal{R}_i$ refers to the relation resident on node i (same for $\mathcal{S}$). The size of the tuples is referred to as *tuple_size* and *network_link_bandwidth* denotes the bandwidth of a single connection in the used network.Table 3.1 provides an overview of the used notation.

$\mathcal{R}$	The first input relation.
$\mathcal{S}$	The second input relation.
$\mathcal{N}_\mathcal{R}$	Number of tuples in $\mathcal{R}$ per node.
$\mathcal{N}_\mathcal{S}$	Number of tuples in $\mathcal{S}$ per node.
$\mathcal{R}_i$	The part of relation $\mathcal{R}$ resident on node i.
$\mathcal{S}_i$	The part of relation $\mathcal{S}$ resident on node i.
tuples_size	Size of one tuple in byte.
network_link_bandwidth	Network bandwidth of a single link.
K	Prefix for kilo (2^{10}).
M	Prefix for mega (2^{20}).
G	Prefix for giga (2^{30}).
B	A capital B means byte.
b	A lower case b means bit.
n	The number of computing nodes.

Table 3.1: Notation Overview.

3.1 Cyclo Join

The first distributed join algorithm implemented in this thesis is Cyclo Join. It is adopted from [8] and [9]. This section describes the idea of the algorithm in detail and highlights some of the interesting design decisions.

Setup

Cyclo Join uses a ring topology as the logical structure for the underlying network. The first step is to set up the network. For this a special node is required, the coordinator node, where all nodes sign up. This node is used to distribute the setup data and to gathering the results at the end. Note that the coordinator node is not used during the join, hence the Cyclo Join only requires a ring structure. After all nodes are signed up, the coordinator assigns an identifier to each node and publishes the connection information, which contains the identifier, the IP and port of each connected node.

The nodes use this information to establish the ring network. Therefore each node creates an RDMA connection to the node with the preceding identifier and the one with the succeeding identifier. Thus forming a ring structure, as shown in Figure 3.1.

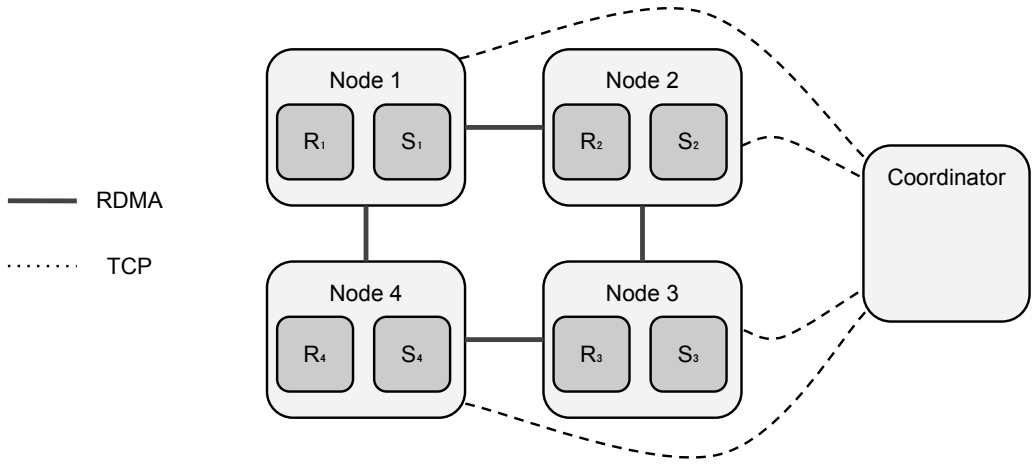


Figure 3.1: The network after setup.

The last step in the setup is to create the two input relations $\mathcal{R}$ and $\mathcal{S}$. In this join each node has a part of each relation. In practice the input would be supplied by the distributed database system, but as we are focusing on distributed join algorithms in this thesis, we use randomly created data.

Work Phase

Each node i starts by sorting the relations $\mathcal{R}_i$ and $\mathcal{S}_i$, which are locally available (sort phase). When sorted the two relations are joined (merge phase). After that the relation $\mathcal{S}$ gets send around in the ring. Therefore each node i sends its part of the relation $\mathcal{S}$ to the next node $(i + 1)\ mod\ n$ and does the join with the part of the relation $\mathcal{S}$, received from

the previous node $(i-1)\ mod\ n$. This is repeated until each client has seen the complete relation $\mathcal{S}$ and therefore done the join $\mathcal{R}_i$ with $\mathcal{S}$. Figure 3.2 illustrates this process.

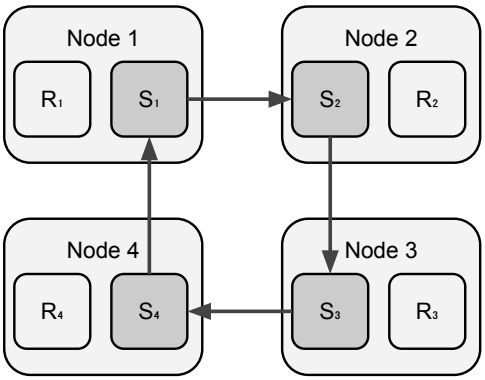


Figure 3.2: Communication Schema.

Note that $n-1$ send-join cycles are sufficient, because each node has one local relation already available and therefore only needs the parts of the $n-1$ other nodes. Executing the join algorithm changes the location of the parts of $\mathcal{S}$. The relation S_i, which was located at the node i is located at node $(i-1)\ mod\ n$ after the join. When summing up, each node has to send and receive $(n-1)\cdot\mathcal{N}_\mathcal{S}$ tuples. Therefore a total of $n\cdot(n-1)\cdot\mathcal{N}_\mathcal{S}$ has to be send in the complete network, which consists of n links. Consequently the factor n cancels itself out. The expected time spent sending data can be calculated as follows:

$$\text{send\_time} = \frac{n\cdot(n-1)\cdot\mathcal{N}_\mathcal{S}\cdot\text{tuple\_size}}{n\cdot\text{network\_link\_bandwidth}} = \frac{(n-1)\cdot\mathcal{N}_\mathcal{S}\cdot\text{tuple\_size}}{\text{network\_link\_bandwidth}}$$

Note that the send and receive bandwidth are independent, thus a node is able to send data with $\text{network\_link\_bandwidth}$ and receive data with $\text{network\_link\_bandwidth}$ at the same time. Therefore the same formula applies for receive_time.

Optimizations

With a remarkable increase of available network bandwidth over the previous years, the bottleneck in distributed algorithms is shifting from the network bandwidth to the CPU and the available memory bandwidth of the systems (studied in [8]). Sadly, as one can observe in the Evaluation, our network is too slow and still remains the bottleneck of this algorithm. Cyclo Join in high speed networks – for which it is originally designed – is studied in [8] and we hope to be able to reproduce this results when having a faster network available.

With the network being the bottleneck three aspects are important to optimize:

- The join algorithm has to start sending data as soon as possible, in order to avoid wasting valuable network time.
- The amount of data to send has to be minimized.

- The sending and processing phases have to be concurrent, in order to avoid introducing overhead after the data has been sent.

It is possible to minimize the time until the first data is send. This is done by starting a send task for the relation $\mathcal{S}_i$ as soon as it is sorted. Note that the sorting of $\mathcal{R}_i$ and the join between $\mathcal{R}_i$ and $\mathcal{S}_i$ is not affected, because the send process only reads the relation. The same trick can be used during every join phase, to overlap the sending and joining. Thus using the network the full time, because the join phase is much faster than the network phase.

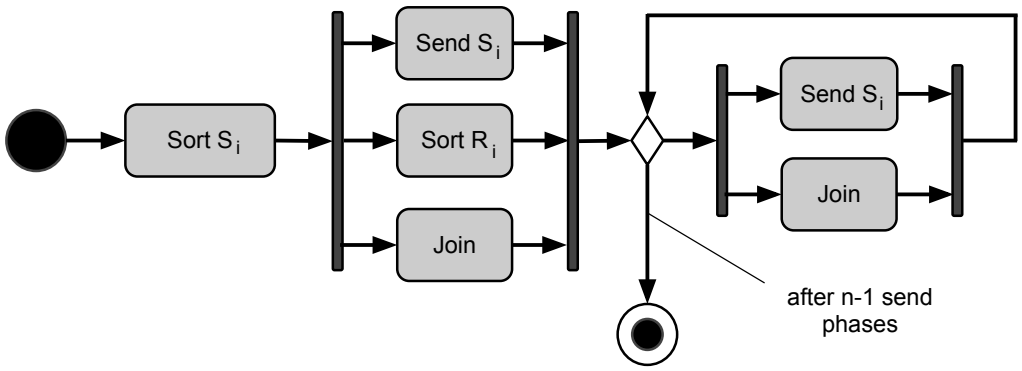


Figure 3.3: Concurrency on one Cyclo Join node i.

The resulting algorithm introduces concurrency to increase the performance as shown in Figure 3.3. In a faster network, where the send/receive operations have to wait for the merge phase, it is possible to increase the merge performance by using multi threading.

RDMA and Buffer Management

As Cyclo Join uses a rather simple sort-merge-based join it is possible to use the relation memory directly for sending and receiving. This makes Cyclo Join an ideal candidate for RDMA. Thus shifting work from the CPU to the RNIC and reducing the needed memory bandwidth. Therefore allowing the merge phase to fully utilize the CPU and memory bus. This is important in high speed networks, which otherwise use large fractions of the CPU and memory bandwidth.

With RDMA requests being asynchronous, overlapping the merge and send phases is very easy. As a trade-off the algorithm needs to organize the buffers for the network. Each node uses two buffers, which are large enough to fit $\mathcal{S}_i$. The first buffer contains the current part of relation $\mathcal{S}$, which gets joined with the local part of $\mathcal{R}$ and is send to the next node. The second buffer is used to receive the next part of $\mathcal{S}$ from the previous node. After the next part of the relation $\mathcal{S}$ is received and the current one is sent and joined, the roles of the two buffers are swapped.

As described earlier each node has two RDMA connections, one to the previous node and one to the next node. Both buffers need to be pinned, to ensure that the pages are not

modified by the operating system. For reasons beyond us a buffer has to be pinned to the connection on which it is sent or received. Consequently it is not possible to simply swap the first and the second buffer, because the first buffer – used to send the current part of $\mathcal{S}$ to the next node – can not be used for receiving from the previous node, because it is pinned to the connection to the next node. Vise versa for the second buffer.

To solve this problem, there are two options available:

- Pin/Unpin: Unpin the memory of both buffers and then pin it to the other connection.
- Copy: Copy the memory from the first buffer, which contains the next relation, to the second buffer, which containing current relation, which is no longer needed at this point.

operation	Intel ® i5-2467M (1.6GHz)	Intel ® Q6700 (2.66GHz) inside a virtual machine
Pin/Unpin	$62.8ms$	$136.6ms$
Copy	$116.5ms$	$263.2ms$

Table 3.2: Experimental evaluation of pin/unpin and copy.

The experimental evaluation in Table 3.2 shows that the pin/unpin approach is about twice as fast and is consequently used in our implementation of Cyclo Join. This is no large overhead in comparison with the overall time of the join, but as the pinning operation has – to our best knowledge – nothing to do with a specific RDMA connection, it could be avoided.

Observations

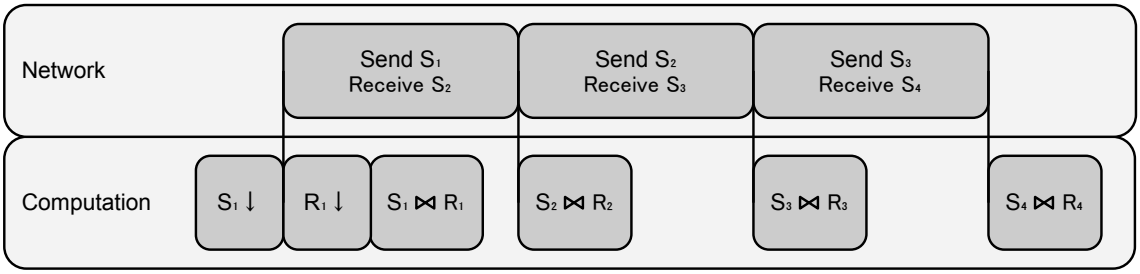


Figure 3.4: Network and processing overlap (exemplary for node 1).

The previous section has shown techniques to overlap the phases of this join algorithm in order to make best use of the available network. It tries to use the network as soon as possible and overlaps the processing phases with the network phases as shown in Figure 3.4.

Cyclo Join – originally designed for high speed network – uses intentionally no methods to reduce the network traffic, in order to preserve all available CPU time for the merge

phases. In addition it uses RDMA to further unload the CPU. Therefore it is no ideal candidate for a slow network, but is very interesting in a high speed network ring as shown in [8].

3.2 Broadcast Join

The second distributed join algorithm this thesis presents is Broadcast Join. As the name suggests this join makes use of the broadcasting feature of UDP to send one relation to a number of clients, which perform the join with their parts of the other relation. The following section explains how this is done in detail.

Setup

Broadcast Join uses a star topology. The architecture consists of one data server, holding the complete relation $\mathcal{S}$, which has $n \cdot N_S$ tuples, and a number of worker clients, each holding a part of $\mathcal{R}$ of the size $\mathcal{N}_\mathcal{R}$. As before the relations $\mathcal{S}$ and $\mathcal{R}$ are created using a random number generator. All clients sign up at the server node using TCP and start listening on the TCP port for control information. The clients do not know each other, consequently there is no need for the server to distribute any connection information, like in Cyclo Join or Radix Join.

The server opens an UDP port for broadcasting and the client nodes start listening on this port. It is necessary that all client nodes are in the same subnet and the server is able to broadcast into this subnet. The state after the setup is shown in Figure 3.5. The left side shows an arbitrary number of clients, which are connected to a server node, called coordinator.

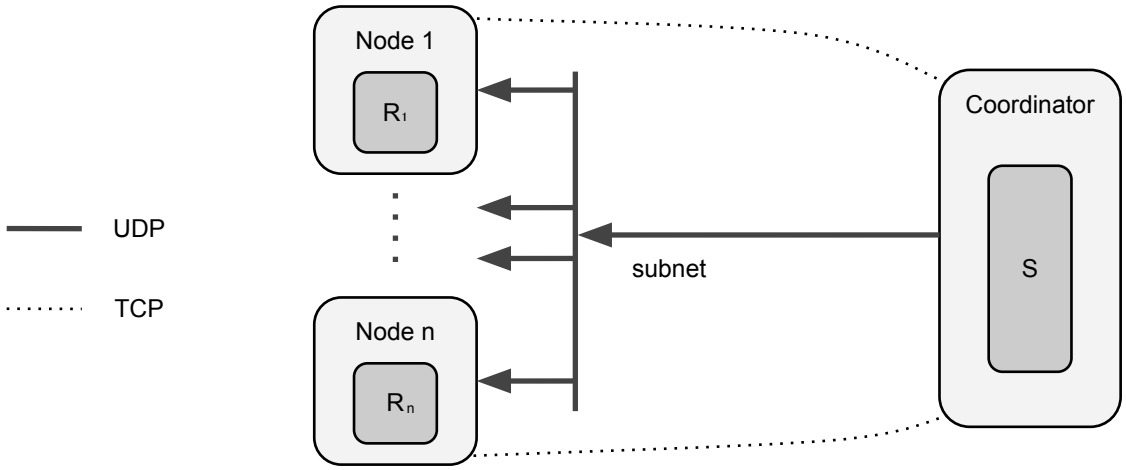


Figure 3.5: The network after setup.

Broadcasting Schema

When all clients are connected and ready, the coordinator starts by broadcasting the relation $\mathcal{S}$ to the clients. This is done in several phases. In each phase a fixed number p of UDP packets, containing tuples of the relation $\mathcal{S}$, is sent to the clients. After p packets are sent, the coordinator sends a *sync* request to each client using TCP. The clients respond with a TCP message containing all identifiers of packets, which they did not receive. These packets are resent together with new ones (additional to the p packets). This procedure is repeated until each client has received every packet. This mechanism is best described using a sequence diagram, as shown in Figure 3.6.

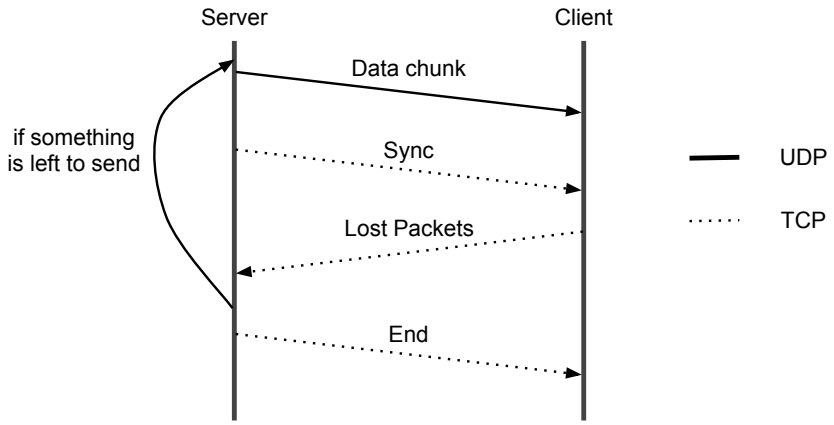


Figure 3.6: The broadcasting phases.

Missing packets are detected by using identifiers. The relation $\mathcal{S}$ is divided into several partitions. Every partition has a unique ID, which increases continuously from the first to the last partition . A partition fits together with its ID into a single UDP packet. The coordinator sends the partitions one after the other starting with the first one – with the lowest ID – and ending with the last one – with the highest ID.

A *sync* request – sent by the coordinator – contains the highest ID of all sent partitions and therefore implies that all partitions with an ID less or equal are already sent. Consequently the client is able to determine which packets were lost and can request those from the coordinator. Note that the order, in which the packets arrive do not affect the join result.

Join Algorithm on a Client Node

The client nodes have to process the incoming stream of tuples of the relation $\mathcal{S}$, faster than they arrive to archive perfect overlapping of communication and computation. In addition the time after the last tuple arrives until the join is done has to be minimized. We decided against a hash join algorithm as it uses usually more memory and is harder to implement. It also introduces a lot of page table faults, which are hard to avoid without partitioning the relations (as described in [17]), which is not possible, because the tuples of $\mathcal{S}$ arrives continually during the join. In advance, as shown in the evaluation (chapter 4),

our sort-merge approach is able to process the incoming data fast enough. Note that the choice between sort-merge and hash join for processing the data stream has to be reviewed when using a faster network.

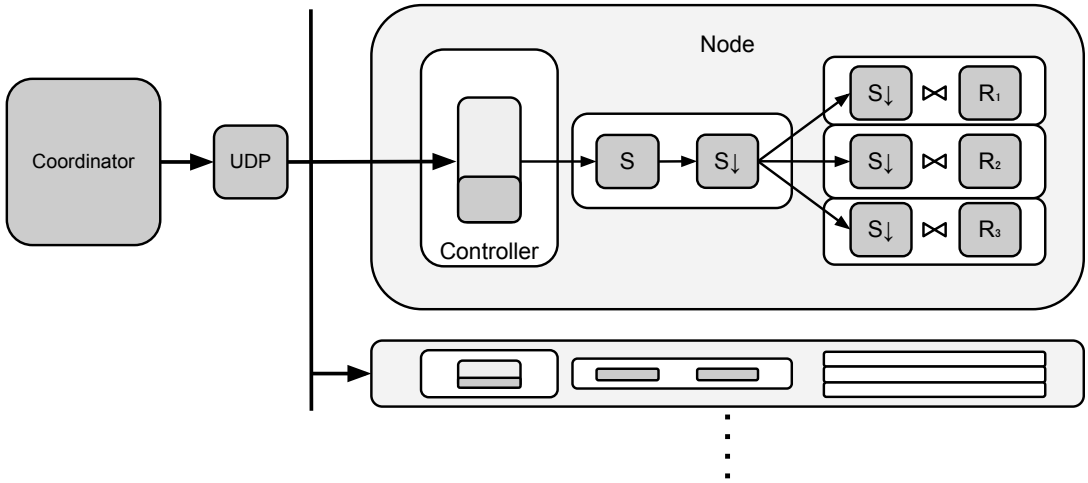


Figure 3.7: Stream based processing a client node.

In our sort-merge-join-based implementation, each client node has a controller thread, which takes care of the communication with the coordinator. It receives the tuples, sends lost packet IDs to the coordinator and assigns the incoming tuples to the worker threads. Each client node has a fixed number of worker threads. After a UDP packet arrives the controller thread appends the content to a buffer. When the buffer is full it gets passed to a worker threads which will sort the tuples. After that each worker thread does a join between the sorted buffer and an evenly sized part of the relation $\mathcal{R}$, which was assigned to the worker during setup. This process is illustrated in Figure 3.7

Network Traffic

A UDP packet consists of a header field and a data field. The header field contains a total of 8 byte, 2 of those encode the length of the packet. Therefore an UDP packet has a maximum length of $2^{16} - 1$ byte, including the 8 byte UDP header and the 20 byte IP header. Broadcast Join needs to assign an ID to each packet, in order to be able to identify the lost packets. Consequently the data length of an UDP packet is limited to

$$\text{data\_length} = 2^{16} - 1 - 8 - 20 - 8 \text{ byte} = 65499 \text{ byte}.$$

The number of tuples per UDP packet can be calculated as follows.

$$\text{tuples\_per\_packet} = \frac{65499\text{byte}}{16\frac{\text{byte}}{\text{tuple}}} = 4093.6875 \text{ tuples}.$$

Consequently Broadcast Join is able to send 4093 tuples per UDP packet. By not splitting

tuples over two UDP packets 11 byte are unused in each packet (about 0.01% per packet)[1]. This very small overhead is easily defensible by the reduction in the complexity of the algorithm.

With the server node being the only node sending data the time minimal needed for sending the complete relation $\mathcal{S}$ calculates as follows:

$$\text{send\_time} = \frac{\mathcal{N}_\mathcal{S} \cdot n \cdot \text{tuple\_size}}{\text{network\_link\_bandwidth}}$$

Observations

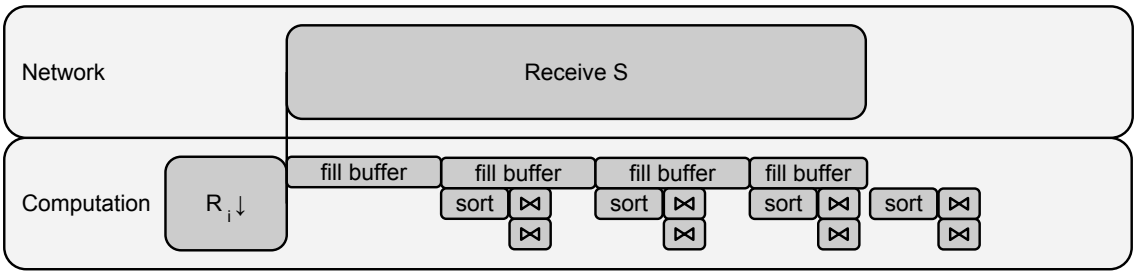


Figure 3.8: Stream based processing an exemplary client node.

Broadcast Join exploits UDP broadcasting transmissions for sending the complete relation $\mathcal{S}$ to all clients over a single link making it inapplicable for big $\mathcal{S}$ relations on slow networks. By using UDP it is able to distribute data to an arbitrary number of client nodes. Since only the completeness and not the order of the tuples matter, this section described how ensure continuous processing on the client nodes using UDP. A TCP implementation in contrast would need to wait for lost packets.

When dealing with $\mathcal{S}$ relations of moderate size Broadcast Join is an interesting algorithm, because the size of the relation $\mathcal{R}$ can be compensated by using more client nodes. In advance it uses a rather simple network topology with a fast setup time. The client nodes of Broadcast Join use multiple threads to allow concurrent receiving and processing of the data, as the diagram in in Figure 3.8 shows.

3.3 Radix Join

The last distributed join algorithm covered by this thesis is a distributed Radix Join. It adepts the local Radix Join presented in [15] to a distributed system. Radix clustering is applied to partition the relations and then gather related partitions on nodes, which then perform the join. The following section describes the inner workings of this algorithm.

[1] Note that not even this amount is directly wasted bandwidth, as the 11 byte are never send. This only incurs the transmission of an additional UDP packet every 5954 packets. Only the 36 byte header of this packet is wasted bandwidth. Consequently only $\frac{36}{5954} = 0.006$ byte are wasted per packet (about $10^{-5}\%$).

Setup

The logical topology of the underlying network is a fully connected network (the actual physical topology is a star topology using a switch to which all nodes connect, as described in section 2.1). Therefore the setup is a lit bit more complicated. First all client nodes connect to a dedicated node, the coordinator, which is later used for synchronization between the client nodes. The coordinator gathers the IP addresses of the nodes and assigns a unique port number to each client node. This information is distributed to all clients. Then the client nodes create a TCP server socket on their assigned port and connect to all other client nodes using the given IP and port numbers.

After the fully connected network is setup the client nodes load the relational data. This join algorithm uses random data like the ones previously discussed. Each node loads the same number of tuples $\mathcal{N}_\mathcal{S}$ for relation $\mathcal{S}$ and $\mathcal{N}_\mathcal{R}$ for relation $\mathcal{R}$. The structure after the setup is shown in Figure 3.9.

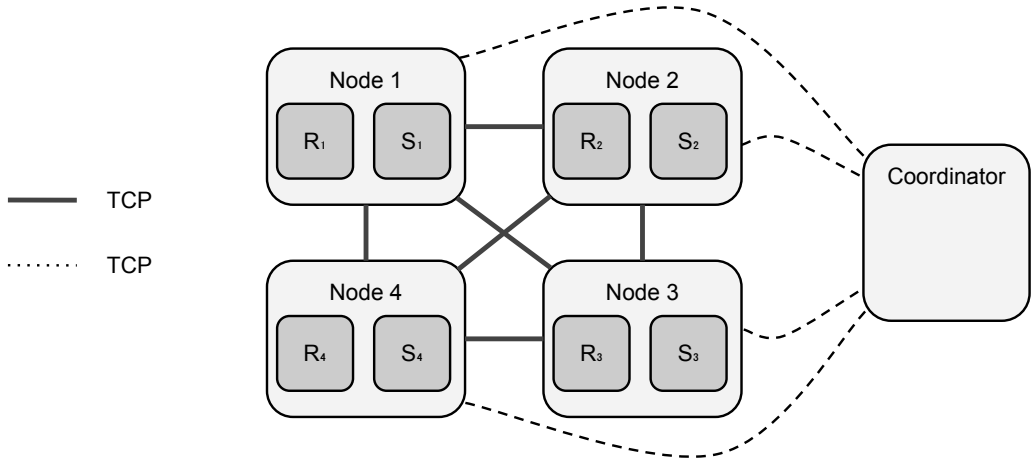


Figure 3.9: The network after setup.

Histogram Phase

The actual join algorithm starts with the nodes creating histograms for both input relations and sending them to the coordinator node, which combines them. At first each node i creates a histogram of the locally resident input relation $\mathcal{S}_i$. Therefore the relation $\mathcal{S}_i$ is evenly divided into one part for each worker thread. The workers iterate over their assigned part and create a histogram using the last 16 bit of the key of the tuples. When the workers are done these worker-local histograms are combined to a node-global histogram $\mathcal{H}_{\mathcal{S}_i}$ for the relation $\mathcal{S}_i$ on each node. The histograms $\mathcal{H}_{\mathcal{S}_i}$ contain the distribution of the relation $\mathcal{S}_i$ of the node i. Each node sends its node-global histogram $\mathcal{H}_{\mathcal{S}_i}$ to the coordinator node. After that the same procedure is repeated for the relation $\mathcal{R}$.

The coordinator node uses the node-global histograms $\mathcal{H}_{\mathcal{S}_i}$ and $\mathcal{H}_{\mathcal{R}_i}$ to create the global histograms $\mathcal{H}_\mathcal{S}$ and $\mathcal{H}_\mathcal{R}$, which contain the number of tuples for each complete partition. The distributed Radix Join gathers all tuples of one partition on one single node, which then performs the join of this partition. For that a partition mapping array is used, which

contains the node ID for each partition. This structure is created by the coordinator node and then published to all client nodes. After that the coordinator distributes the node-global histograms $\mathcal{H}_{\mathcal{S}_i}$ and $\mathcal{H}_{\mathcal{R}_i}$ to all nodes.

Partitioning Phase

In the next step each node i partitions the local relation $\mathcal{S}_i$. This means that all tuples, whose keys have the same last bits get copied into the same chunk of memory. In order to speed up this operation it is important to avoid TLB misses, as elaborated in [15]. The number of partitions has to be limited to fit into the TLB, thus keeping all frequently accessed pages in it. This effect is studied in Figure 3.10. The x-axis denotes the number of bits used for the partitioning and therefore the number of partitions, because x bits lead to 2^x partitions. The y-axis shows the time needed for the operation in milliseconds. A table containing $64M$ tuple with evenly distributed $64bit$ keys and $64bit$ payloads was used in in this experiment.

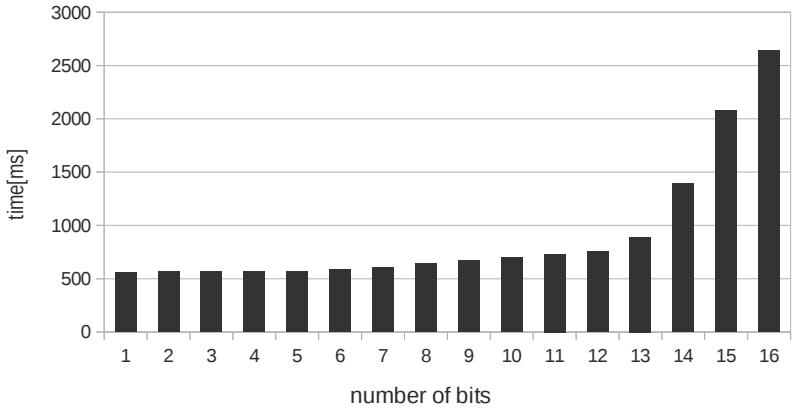


Figure 3.10: Time for partitioning.

The partitioning is divided into two passes, because two partitioning passes with 8 bit each are faster than one with 16 bit. The first pass for relation $\mathcal{S}_i$ is done by the node i, the second pass is done by the node, which receives this partition. Therefore the relation $\mathcal{S}_i$ is divided into $\mathcal{N}_W$ even parts and assigned to the worker threads, as in the case for the histogram phase. The worker threads then partition their assigned part into a pre-allocated buffer. Synchronization between the worker threads can be avoided by using the previously calculated worker-local histograms for the single parts of $\mathcal{S}$. For example when a worker thread is processing the third part of the relation $\mathcal{S}$ and encounters a tuple for partition 12, the worker can calculate the position for the tuple in partition 12. For that the worker uses the histograms for part one and two (L_{S_1} and L_{S_2}), which contain the number of tuples for partition 12 of part one and two. When adding these two values the worker knows where the first tuple for partition 12 of part three goes. By keeping track of the number of already inserted tuples for each partition the worker is always able to find the right spot for a tuple. At the end of this phase the relation S is partitioned into $2^8 = 256$

partitions, each containing tuples with the same last 8 bit.

Transmission and Join Phase

After the partitioning of $\mathcal{S}_i$ on every node the algorithm receives the partition mapping – the array containing the node IDs for each partition – from the coordinator. Each node now starts to distribute the partitions of $\mathcal{S}$ to the corresponding nodes. During the transfer of $\mathcal{S}$, the relation $\mathcal{R}_i$ gets partitioned on each node just like the relation $\mathcal{S}_i$. By overlapping these two tasks – the sending of the partitions of $\mathcal{S}$ and the partitioning of $\mathcal{R}_i$ – no system resources are wasted. When $\mathcal{R}$ is partitioned, the partitions are also send to the nodes, using the partition mapping. Each node always sends the partition of either relation – $\mathcal{R}$ or $\mathcal{S}$ – with the lowest partition ID first. Hence the partitions with the lowest IDs are gathered at a node very soon. As soon as all parts of a partition are gathered at a node the node can perform the join between those two partitions and therefore produce results. Consequently the distributed Radix Join is able to produce results very fast.

At this point each node is sending/receiving partitions of $\mathcal{R}$ and $\mathcal{S}$ to/from the other nodes of the network. Moreover each node has received the partition mapping and the local histograms of the other nodes from the coordinator. Each arriving message contains all tuples of one $8bit$ partition of the sender node. The incoming partitions with $8bit$ are partitioned one more time into sub-partitions of $16bit$ by the node. This is done by passing the message to a worker thread, which will copy the data into preallocated memory chunks for each sub-partition. Note that by using the node-global histograms – like the workers used the worker-local histograms before – there is no need for synchronization between the worker threads.

When all tuples of one partition are received the node can perform the join of this partition. Therefore a task for each sub-partition is created and executed by one of the worker threads. For the actual join of a sub-partition pair, the sub-partition of $\mathcal{S}$ is radix partitioned one last time (total of $24bit$). Then the worker iterates over the tuples of the sub-partition of $\mathcal{R}$ and looks up the keys.

Network Traffic

The same number of partitions is assigned to each node. Consequently each node needs to send and receive roughly the same number of tuples to and from every other node. There are n nodes and the tuples are assumed to be uniformly distributed, each node has to send about $\frac{n-1}{n}$ tuples of its local relation $\mathcal{S}_i$ and $\mathcal{R}_i$ (which are of the size of $\mathcal{N}_\mathcal{S}$ and $\mathcal{N}_\mathcal{R}$). When summing up each node of the distributed Radix Join is sending a total of $\frac{n-1}{n} \cdot (\mathcal{N}_R + \mathcal{N}_S)$ tuples.

$$\text{send\_time} = \frac{\frac{n-1}{n} \cdot (\mathcal{N}_\mathcal{R} + \mathcal{N}_\mathcal{S}) \cdot \text{tuple\_size}}{\text{network\_link\_bandwidth}}$$

For performance reasons, it is important to avoid cross traffic in a fully connected network (section 2.1), in order to archive maximum performance. This is done by synchronizing the send phases of the nodes. A comparison of different data distribution schemes in a fully connected network is giving in section 6.1. Our implementation of the Radix Join

uses the synchronized distribution schema, which has the best performance of the ones evaluated.

Observations

The distributed Radix Join sends data as soon as possible and overlaps the data distribution with the processing of the incoming data. The concurrent activities are demonstrated in Figure 3.11. This allows a very good utilization of the network.

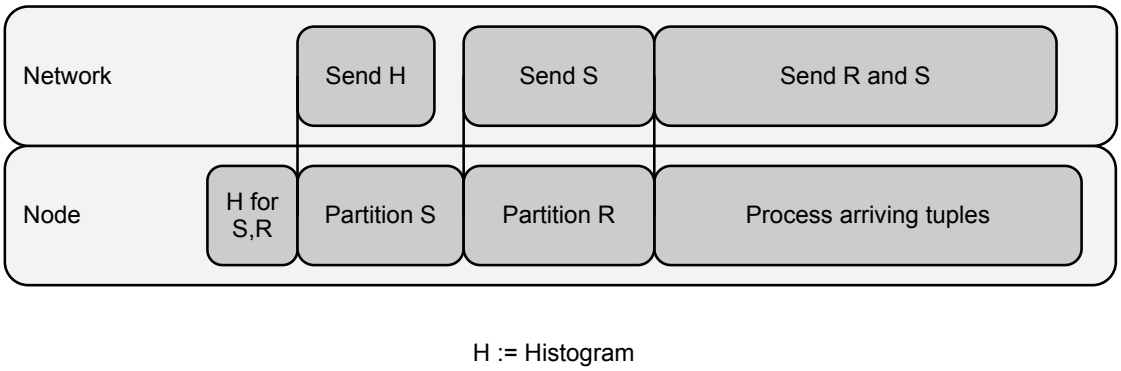


Figure 3.11: Network and processing overlap on an exemplary node.

The algorithm uses a complicated network topology. By sending the data directly – not over several hops, like the cyclo join – to the destination on n connections simultaneously – unlike the broadcast join, which uses only one connection–, it has the lowest expected network time. The algorithm is an excellent candidate for smaller clusters where it is possible to have a fully connected network.

4 Evaluation

All algorithms are implemented using C++11 and compiled with the open source compiler GCC-4.6. The experiments are executed on a 64 bit server edition of the GNU-Linux operating system Ubuntu 12.04. We used a a homogeneous cluster containing 16 machines, which are all connected to a switch with a theoretical bandwidth of 1 GBit/s (in practice around 870 Mbit/s). Hence a fully connected network is formed, where every node is able to send and receive data with 1GBit/s each. The computers use an Intel ® Core™2 Quad Q6700 processor, which supplies four hardware threads clocked at 2.66 GHz. Each of the four cores has a 64KB L1 cache (32KB data, 32KB instructions). In addition there are two 4MB L2 caches, each of which is shared between two cores. A 4-way set associative TLB with 256 entries is used to speed up the page table lookup. The systems are equipped with 8GB RAM of main memory. It is interesting to note that the CPU is already five years old and is still fast enough for our evaluation, but could be replaced to satisfy a faster network.

In our experiments we assume that the data is fully resident in the main memory of the systems. This is a fair assumption considering the still growing size of today's memory and the trend towards main memory database systems [13]. The experiments all use two input relations R and S. Both relations consist of a 64bit integer key and a 64bit payload. A join of two relations with larger tuples can be reduced to this core problem using payload splitting as described in section 6.2. We execute the following equi-join:

```
SELECT count(*)
FROM   R,S
WHERE  R.key=S.key
```

This query allows to ignore the costs introduced by materializing the results, which are irrelevant for the actual join performance. The two input relations are loaded using a random number generator. In practice the input would be supplied by a distributed database system.

The following sections summarize our results. In section 4.1 a detailed runtime analysis of each join algorithm is given. After that we compare the algorithms scale out behavior and examine their ability in dealing with different cardinalities, skewed data and join selectivity (section 4.2).

4.1 Analysis

This section shows the runtime of the components of each distributed join algorithm individually. It demonstrates that the runtime of the implemented join algorithms is bound by the available network bandwidth. The joins are examined in the same order as they are described in chapter 3. During the individual analysis we assume a number of 16 nodes and a relation size of $100M$ for $\mathcal{R}$ and $\mathcal{S}$ per node (with $M = 2^{20}$).

Cyclo Join

Each node of Cyclo Join has to forward the locally available part $n-1$ time, as explained in section 3.1. Therefore the expected network costs sum up to:

$$\text{send\_time} = \frac{(n-1) \cdot \mathcal{N}_\mathcal{S} \cdot \text{tuple\_size}}{\text{network\_link\_bandwidth}}.$$

When plugging in the values the expected time for the network evaluates to:

$$\text{send\_time} = \frac{15 \cdot 100M \cdot 16\text{Byte}}{870\frac{\text{Mbit}}{s}} = 220.7s.$$

The chart in Figure 4.1 shows the performance measurements. Each bar demonstrates the time needed for the respective phase. The first bar ($overall$) shows the time needed for the complete execution of the join algorithm, which takes about $239s$, without the loading phase. The three following bars show the individual phases of the algorithm.

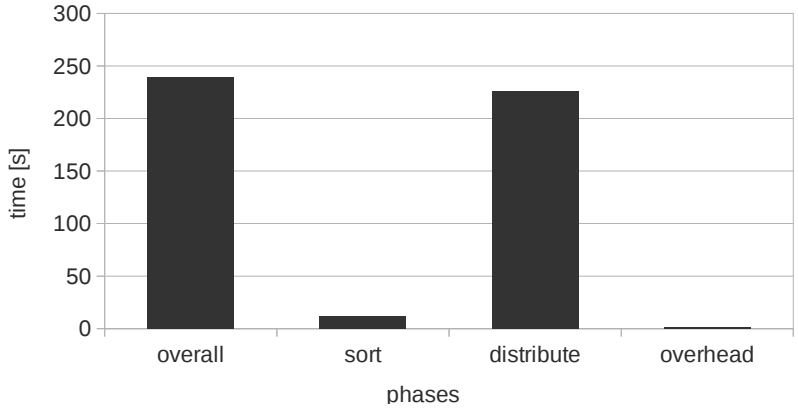


Figure 4.1: Cyclo Join details.

The first one ($sort$) shows the time needed to sort the relations $\mathcal{R}$ and $\mathcal{S}$ (about $11s$). The bar entitled $distribute$ represents the main work phase of the join algorithm. This takes about $226s$. Recall that each node has one worker thread, which performs the join of $\mathcal{S}_j$ with $\mathcal{R}_i$, while the RNIC is sending $\mathcal{S}_j$. Thus the sending and processing is completely parallel. While the worker threads are only working 11% of the time, the network is used the whole time during this phase. Consequently this phase is bound by the network bandwidth.

The sort phase can be avoided by sending the $\mathcal{S}$ relation without sorting. But this approach introduces more computation, as each node has to sort the arriving parts of $\mathcal{S}$ before the merge phase. In a slow network – like the one used in this thesis – the version without sorting increases the join performance, but in high speed networks – where the join is not limited by the network – it decreases the performance.

The reason for the overhead of about $1.2s$ at the end is, that each node has to perform a join between the last arriving part of relation $\mathcal{S}$ and its local part of $\mathcal{R}$. This could be minimized by dividing the relation $\mathcal{S}$ into smaller parts or by parallelizing the merge phase.

Broadcast Join

Broadcast Join (section 3.2) uses one server node, which transmits the complete relation $\mathcal{S}$ to all clients. With the relation $\mathcal{S}$ containing $n \cdot \mathcal{N}_\mathcal{S}$ tuples, the expected network time can be calculated as

$$\text{send\_time} = \frac{n \cdot \mathcal{N}_\mathcal{S} \cdot \text{tuple\_size}}{\text{network\_link\_bandwidth}}.$$

When using the same values as above this evaluates to

$$\text{send\_time} = \frac{1600M \cdot 16\text{Byte}}{870\frac{\text{Mbit}}{s}} = 235.4s.$$

As shown in Figure 4.2 the Broadcast Join takes an overall time of about $240s$ (first bar). The next three bars show the individual phases of the Broadcast Join. The bar entitled $init$ denotes the time needed to sort the input relation $\mathcal{R}$ on each client node (about $6s$), this process can be overlapped with the network by just buffering the incoming tuples of $\mathcal{S}$. The next bar ($distribute$) shows the time spend receiving data and processing it in parallel. This is the main phase and takes about $239s$. The last bar illustrates the overhead for processing the last tuples (about $1s$).

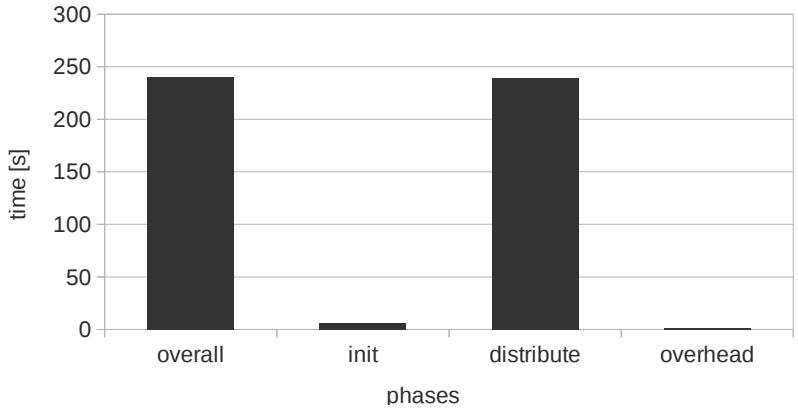


Figure 4.2: Broadcast join details.

The workers on each node use 24% of their time for sorting the tuples of $\mathcal{S}$, 54% for joining them with the parts of $\mathcal{R}$ and about 22% is spent on waiting for more input. Consequently the Broadcast Join is also bound by the network, as the tuples do not arrive fast enough. When using a faster network it is necessary to optimize the local worker threads, which could be done by a more sophisticated sort-merge join implementation or a hash join. With our approach being fast enough to process the incoming packets, we did not investigate this issue further.

Radix Join

When assuming evenly distributed data, each node of Radix Join needs to send $\frac{n-1}{n}$ of its local available tuples of both relations, because there are $n-1$ other nodes. Therefore, the expected network time is

$$\text{send\_time} = \frac{\frac{n-1}{n} \cdot (\mathcal{N}_\mathcal{R} + \mathcal{N}_\mathcal{S}) \cdot \text{tuple\_size}}{\text{network\_link\_bandwidth}}.$$

The equation evaluates to

$$\text{send\_time} = \frac{\frac{15}{16} \cdot (100M + 100M) \cdot 16\text{Byte}}{870\frac{\text{Mbit}}{s}} = 27.6s.$$

Radix Join has the lowest expected network costs. This is easily explained, because every node uses its network link – unlike the Broadcast Join, where only one link is used – and the data is always placed directly where it belongs – unlike the Cyclo Join, where several hops are required. The experimental results are presented in Figure 4.3.

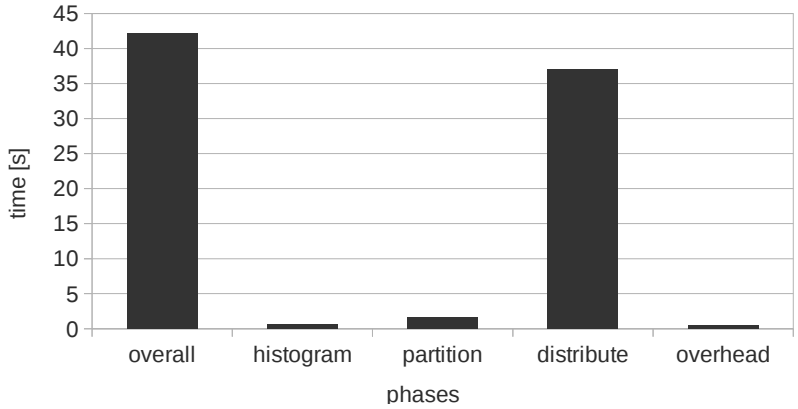


Figure 4.3: Radix Join details.

The first bar denotes the overall time ($42.2s$) of Radix Join. The following bars show the individual phases. The bar entitled $histogram$ is the time needed to create the histograms for $\mathcal{R}$ and $\mathcal{S}$ (about $0.6s$). The next bar ($partition$) is the partition phase for the relation $\mathcal{S}$, which takes about $1.6s$. The partitioning of $\mathcal{R}$ is, as explained in section 3.2 completely overlapped with the network phase, which is shown in the bar entitled $distribute$ (about $37s$). The last bar shows the overhead for processing the last received relation, which takes about $0.5s$.

It is interesting to note that the experimental result for the network phase of this join algorithm is slower than expected. This effect is due to the need for synchronization to avoid cross traffic as explained in section 6.1.

It is very likely that Radix Join would perform well in a faster network. First, it is possible to adapt it to use RDMA, because we are already placing the received data directly into the work buffers, whose sizes are described in the histograms. Second, the local worker threads are not fully utilized, each of the two threads used (on our somewhat older CPUs) is idle 65 % of the time. Third, our implementation of the worker threads is rather simple which leaves room for optimizations.

Summary

With the network being the bottleneck in our hardware setting, the performance of each join algorithm is dominated by the network phase. Hence, the fastest join algorithm is the Radix Join, which has the lowest expected network costs and therefore the shortest network phase. Note, that this join also uses the most complicated network topology, when applying it to a ring or a star it would also perform worse. In a faster network – where the CPU or memory bandwidth is the bottleneck – this ranking is likely to change, as the Cyclo Join uses less local resources.

4.2 Comparison

The following section compares the algorithms scale out behavior, their skew resilience and their ability to profit from not non-equally sized relations.

4.2.1 Scale Out

The reason for using distributed algorithms is to benefit from the computing power of more than one machine. Therefore an important criteria for the quality of a distributed algorithm is how well it performances on a growing number of nodes. The following section studies this effect in a cluster with the node number increasing from 2 to 16. The size of both relations is also increasing with $|\mathcal{S}| = |\mathcal{R}| = 100M \cdot n$. Throughout this section the helper variable $c := \frac{\text{tuple\_size}}{\text{network\_link\_bandwidth}}$, which is a constant in our scenario, is used.

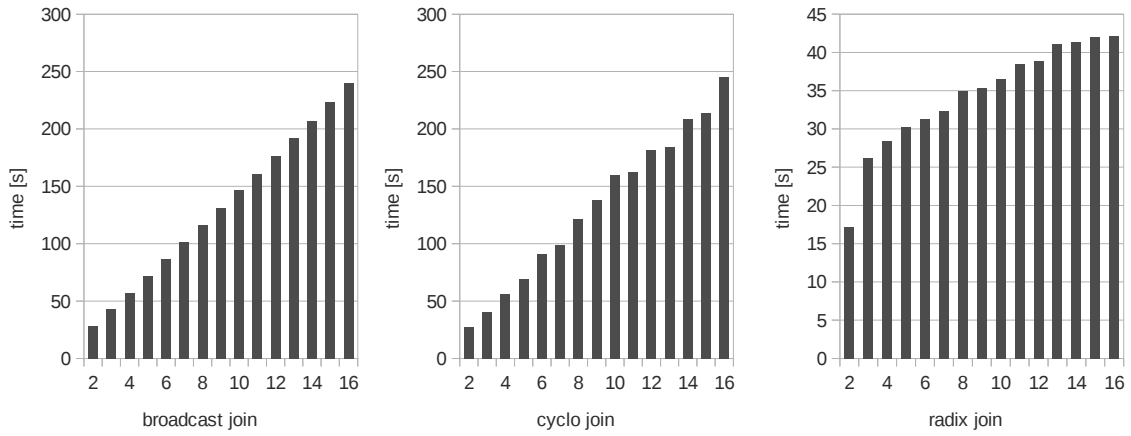


Figure 4.4: Scale out behavior.

Figure 4.4 shows the experimentally obtained scale out behavior of Broadcast Join, Cyclo Join and Radix Join (in this order). The x-axis denotes the number of nodes used and the y-axis the execution time in seconds. By looking at the functions, which express the expected runtime of the algorithms it is possible to get a better understanding of their behavior. With the runtime of each algorithms being dominated by their network phase it is legit to use

the expected network time as an indicator for their complete execution time. Note that each function verifies the experimentally obtained results in the chart.

Broadcast Join

The network costs for Broadcast Join is derived in chapter 3 as follows:

$$\text{network\_cost} = n \cdot \frac{|\mathcal{S}|}{n} \cdot c = |\mathcal{S}| \cdot c.$$

This function calculates the expected time for Broadcast Join. With n (the number of nodes) canceling itself out, the result depends solely on the size of $\mathcal{S}$. On the one side this is good, because the expected time is independent of $\mathcal{R}$ [1]. On the other side this is very bad, because the function does not depend on the number of computing nodes used. This means that installing more computing machines does not increase the algorithms performance.

Cyclo Join

Sadly, the situation in Cyclo Join is even worse than in Broadcast Join. The expected time can be calculated with:

$$\text{network\_cost} = (n-1) \cdot \frac{|\mathcal{S}|}{n} \cdot c = (1 - \frac{1}{n}) \cdot |\mathcal{S}| \cdot c.$$

This formula means that the expected time for the algorithm increases when using more computing nodes – because more hops are required – or a larger relation $\mathcal{S}$ is used – because more data has to be send. Consequently when using Cyclo Join it is necessary to use only the required number of nodes to be able to process the local joins between the parts of $\mathcal{S}$ and $\mathcal{R}_i$ fast enough. Thus avoiding the costs for additional hops in the circle.

Our experiments in Figure 4.4 showed that the execution time increases linear in our setup. The equation shows the same behavior when we insert the size of $|\mathcal{S}|$ as $n \cdot 10M$:

$$\text{network\_cost} = (1 - \frac{1}{n}) \cdot |\mathcal{S}| \cdot c = (n-1) \cdot 10M \cdot c.$$

Radix Join

In Radix Join each node uses one connection for sending data – unlike Broadcast Join, where only the server is sending data – and the tuples are only send once to the node, which needs them – unlike Cyclo Join, where each tuple is send $n-1$ times. Hence Radix Join has different network costs:

$$\text{network\_cost} = \frac{n-1}{n} \cdot \frac{|\mathcal{R}| + |\mathcal{S}|}{n} \cdot c = \frac{n-1}{n^2} \cdot (|\mathcal{R}| + |\mathcal{S}|) \cdot c.$$

[1] Note that the size of $\mathcal{R}$ does not matter as long as the client nodes are able to process the incoming network stream fast enough.

Increasing the size of either relation will increase the expected runtime of Radix Join. But when using more computing nodes the expected runtime will decrease. In our sample scenario where $|\mathcal{S}| = |\mathcal{R}| = n \cdot 10M$ the algorithm scales as follows:

$$\text{network\_cost} = \frac{n-1}{n^2} \cdot (|\mathcal{R}| + |\mathcal{S}|) \cdot c = (1 - \frac{1}{n}) \cdot 20M \cdot c.$$

Summary

The observations provided in this section only apply to a setup, where the network costs of the join algorithm are the dominating costs and can therefore be used to express the overall runtime. The three discussed distributed join algorithm showed very different behaviors. In a network-dominated setup Cyclo Join and Broadcast Join are unable to deal with larger relations. Radix Join on the other side promises a good scale out behavior, because the runtime can be controlled by using more computing nodes when dealing with large relations.

4.2.2 Skew

Broadcast Join and Cyclo Join are both using a simple sort-merge join variant and no mechanism for partitioning the input data. Therefore skewed data can not result in unevenly sized partitions in either algorithm. Without larger partitions causing a load imbalance between worker threads neither of these algorithms can suffer a performance penalty from skewed data. Consequently these algorithms are not considered in this section.

Radix Join in contrast uses radix clustering to partition the input relations. One partition could be larger than the others and therefore take longer to send and process, hence slowing down the involved nodes. But the partitions are created using a simple hash functions, consequently normal data skew, where a range of values is represented more often, has little to no effect on the radix join as shown in Figure 4.5.

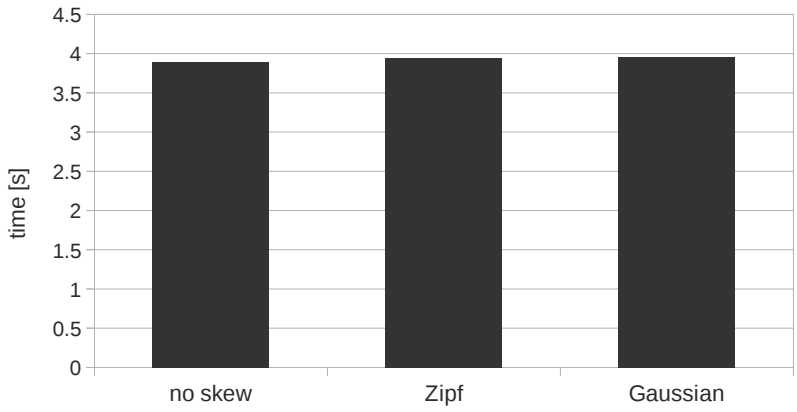


Figure 4.5: Data skew.

The first bar serves as a baseline and shows the time needed to distribute a uniformly distributed dataset. The same experiment is conducted with a dataset with a Zipf distribution (bar two) and a Gaussian distribution (bar three). The radix clustering assigns two consecutive values into two different partitions therefore the ranges with frequently occurring tuples are almost evenly distributed over the partitions. Nevertheless it is possible to construct input data, which leads to unevenly sized partitions and therefore to load imbalance. We examined two solutions for dealing with skewed data:

- The first approach distributes the partitions in a smart way across the nodes. As described in section 3.3 all client nodes send their local histograms of both input relations to the coordinator, which combines those to a global histogram. Using this histogram the coordinator is able to evenly distributed the partitions to the client nodes in a way that each client gets roughly the same amount of tuples to process. In this approach for dealing with skew it is important to keep in mind that the most expensive operation is the movement of data between the nodes. Therefore the applied objective function has to minimize the expected network traffic. This technique for dealing with skew in a hash based join algorithm, which was inspired by [23], was not tested in the distributed radix join.
- A second solution, to avoid the problems of skewed data in Radix Join, is the usage of a better hash function. Hashing the keys distributes them, when using a good hash function, evenly. Hence destroying the skew patterns and therefore avoiding unevenly sized partitions. With the hash function having collisions, this approach is producing false positives as explained in section 6.2.

4.2.3 Multiplicity

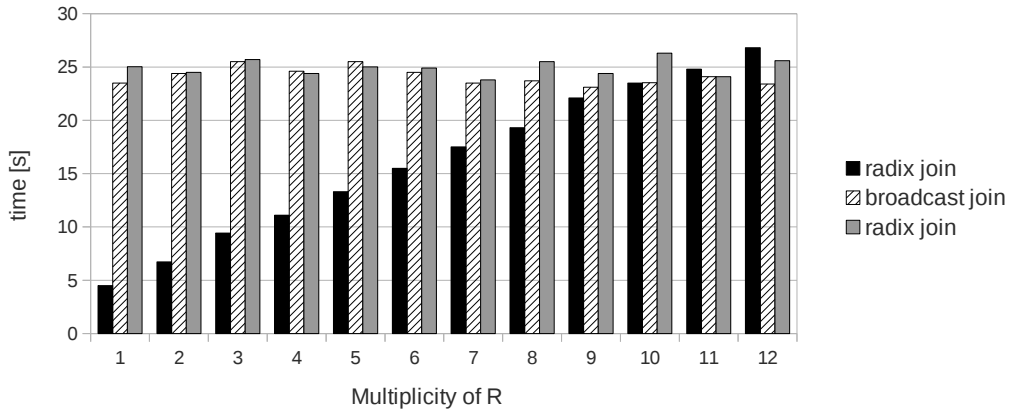


Figure 4.6: .

We conducted a series of experiments to test the behavior of the join algorithms when one of the two input relations is larger than the other. The results are shown in in Figure 4.6. The expected runtime of Cyclo Join and Broadcast Join depends mostly on the size of the

shared relation $\mathcal{S}$, which gets distributed over the network. Therefore we fixed the size of $\mathcal{S}$ to 10 M tuples and scaled $\mathcal{R}$ from one to twelve times the size of $\mathcal{S}$ (scale factor is shown on the x-axis). The three differently colored bars in the chart show the three join algorithms. The y-axis denotes the execution time.

Cyclo Join and Broadcast Join are only sending the shared relation $\mathcal{S}$, which has a fixed size. Therefore their execution time remains constant. But with an increasing amount of tuples in the private relation $\mathcal{R}$, the local operations on the nodes become more complex.

The network costs of Radix Join, in contrast, depend on $\mathcal{R}$ and $\mathcal{S}$. Therefore the execution time for Radix Join increases with a larger relation $\mathcal{S}$. Hence there exists a ratio between $\mathcal{R}$ and $\mathcal{S}$ such that the Broadcast Join outperforms the Radix Join.

This break even point can be observed in the chart at bar 11 (or more clearly at bar 12). It is also possible to calculate this point (for example for Broadcast Join) by examining the equations which describe their network costs.

$$\text{send\_time}_{\text{BroadcastJoin}} = \text{send\_time}_{\text{RadixJoin}}.$$

$$\frac{n \cdot \mathcal{N}_{\mathcal{S}} \cdot \text{tuple\_size}}{\text{network\_link\_bandwidth}} = \frac{\frac{n-1}{n} \cdot (\mathcal{N}_{\mathcal{R}} + \mathcal{N}_{\mathcal{S}}) \cdot \text{tuple\_size}}{\text{network\_link\_bandwidth}}.$$

This can be reduced to:

$$(\frac{n^2}{n-1} - 1) \cdot N_{\mathcal{S}} = N_{\mathcal{R}}.$$

Which evaluates for the sixteen used nodes to:

$$16.1 \cdot \mathcal{S} = \mathcal{R}.$$

Therefore $\mathcal{R}$ has to be about sixteen times as large as $\mathcal{S}$. Our experimental results show that this ratio is reached earlier in practice. This is perfectly valid, considering that both join algorithms not only depend on the network phase, as assumed in the calculations. For example Radix Join needs more setup time for partitioning the relations than Broadcast Join, which has almost no setup time.

5 Conclusion

This chapter summarizes the most important results and contributions of this thesis to the field of distributed join processing. In addition it provides a section containing an overview of relevant work in this field.

5.1 Contributions

This thesis describes three distributed join algorithms and evaluates their performance on a hardware cluster containing 16 nodes. Each algorithms provides an efficient implementation for a certain network topology. In contrast to approaches like Hadoop++ [6], our algorithms are designed for a high per node efficiency.

Radix Join with a throughput of 80 million tuples per second provides a very efficient distributed join algorithm on a fully connected network. When compared to the peak performance (100 million tuples per second) of the highly optimized local radix join described in [15], one has to consider three things:

- The hardware used in our cluster is a mid-end configuration from five years ago, where the hardware in [15] is only three years old and was a high-end setup at this time.
- The distributed Radix Join uses 128 bit tuples, instead of 64 bit tuples like the local radix join. First, this increases the performance of the local radix join and second, this slows down the distributed radix join, as it almost doubles the network costs. This is a crucial penalty in the distributed Radix Join, because its execution time is dominated by the network phase.
- In our implementation it is easy to increase the performance by using more nodes (as shown in subsection 4.2.1 it scales linear to the number of nodes).

This shows that Radix Join is an efficient design. In addition the linear scale out behavior makes Radix Join an excellent candidate for larger networks, which are fully connected. Furthermore it has been shown that Radix Join is able to satisfy a faster network, because the CPU of each node is not fully utilized (only about 50%).

Cyclo Join is specialized for a specific network topology – a ring – and was originally developed by [8]. It is designed for a high speed network and as shown in our evaluation the nodes are waiting most of the time (89%) for more data, which indicates that Cyclo Join would be able to process the data in a much faster network (as shown in [8] this is the case). In addition Cyclo Join has shows a way how to apply iWARP/RDMA to a join operator. In our opinion Cyclo Join is not suited for larger clusters. As shown in the evaluation Cyclo Join does not benefit, but suffer from an increasing number of nodes. This makes Cyclo Join only useful for smaller clusters.

Broadcast Join uses the broadcasting feature of UDP for sending data in a distributed join algorithm. It uses the connection between the server and the client nodes to transmit $\mathcal{S}$, one of the two input relations, completely. Hence its performance strongly depends on this connection and the size of the $\mathcal{S}$. Broadcast Join is an interesting join, because the size of the other relation $\mathcal{R}$ – the one, which is not transmitted – is almost irrelevant. By using broadcast messages, an arbitrary number of nodes, with parts of $\mathcal{R}$, can be supplied with tuples of $\mathcal{S}$.

The analysis of the algorithms has shown that each one is bound by the network bandwidth. This means that each algorithm is spending most of its time sending and receiving data and would benefit by a faster network. This result indicates that the join performance in a distributed database – even on a not state of the art hardware – is limited when using a 1 GBit/s network. Consequently these networks still require joins which try to reduce the amount of transmitted data ([12, p. 489-493]). As [8] postulates this changes in faster networks.

5.2 Related Work

Distributed join algorithm always include two essential parts. First, the distribution of the data, which can be done in various ways (as shown in chapter 3). And Second, after the distribution the input data has to be combined.

Networking

With the available network bandwidth increasing over the years the use of iWARP/RDMA will become more and more important for efficiently moving data between computer nodes. As described in [9], the computation overhead and the extra memory bus passes introduced by the TCP protocol are wasting a considerable amount of system resources. The original proposal of Cyclo Join [8] showed that in a 10 GBit/s network using RDMA the join performance is limited by the available main memory bandwidth. This shift from the network as a bottleneck to the main memory should be more dramatic when using today's highest achievable network bandwidth, e.g., 40 GBit/s with InfiniBand.

High speed networks open new ways for implementing distributed database systems [14]. The creators of Cyclo Join showed in [10] how to build one under the assumption of a fast network with RDMA. The so called Data Cyclotron consists of a small cluster connected with a ring structure. A hot set of the relational data is continuously rotated in the ring and used to process the queries, which are assigned to the nodes.

Local Join Algorithms

At some point each distributed join algorithm has to perform a local join on some gathered data. This problem – local join processing – is well known and extensively studied over the last decades. Cyclo Join for example uses a simple sort-merge approach to join the parts of the input relations. This could be extended by using more advanced techniques, like in [15], which makes extensive use of thread and data-level parallelism to gain performance. Another interesting approach is the Massively Parallel Sort-Merge Join (MPSM) [2]. The

MPSM is an efficient transition of a sort-merge join to a Non Uniform Memory Access (NUMA) architecture.

Over the past years hash joins were the dominant alternative to sort-merge-based join algorithms. Most of todays hash join algorithms use partitioning to fit the data into the cache [21], thus reducing cache misses during the probe phase. [18] proposed to use several partitioning steps in order to reduce the number of subtables in a single partitioning step, hence avoiding TLB misses. The researches from Intel and Oracle [15] created a hash join algorithm based on this observations. They used a two passes radix clustering as proposed in [18]. Our work adopts this idea and transfers it to a cluster, creating a distributed Radix Join.

Other Distributed Join Algorithms

The MapReduce [5] framework from Google inc. is designed to provide non-expert users an easy access to the computing power of large clusters. It achieves great scalability and fault tolerant at the costs of per node efficiency, which gets compensated by simply using more computing nodes. This approach got very popular for query processing over the last years in systems like Hadoop [3], which was optimized in Hadoop++ [6].

Another way for distributed join processing is the Symmetric Join, which is a pipeline-based approach. In contrast to the classic hash join algorithms, this algorithm builds a hash table for both relations. Each new tuples gets probed into the hash table of the other relation and then inserted into the hash table for its relation. Consequently this algorithm produces very early results, but is in our opinion not feasible for high speed networks, because of the increased resource requirements for building two hash tables and probing each tuple individually.

6 Appendix

6.1 Data Distribution

This section describes three ways to distribute data in a fully connected network. With the network being the limiting factor, it is important to use a schema, which avoids cross traffic in order to get the best performance. Cross traffic occurs whenever two nodes in a fully connected network are sending data to the same receiver node. In this case the two up links can only send with halfe of their maximum capacity, because they are limited by the single down link of the receiver node. Hence the network is not fully utilized during this time.

In order to study this problem we created a test scenario. The scenario consists of 16 nodes, each node has to send $1GB$ of data to each other node. Consequently each node has to send $15GB$, which takes about $141.2s$, using a $870\frac{Mbit}{s}$ network.

Random Distribution

In the first schema the data is distributed randomly. Therefore each node creates a work set containing all other nodes. In each step a node sends 1 MB to a random node in the work set. If a node has sent 1 GB in total to a node of its work set then this node is dropped from the work set. If the work set of a node is empty then this node is ready. This procedure ensures that each node sends exactly 1 GB to each other node. This is a rather simple approach, which only serves as a comparison baseline. The chance that there is no cross traffic at any given point of time during the distribution is as low as $\frac{16!}{16^{16}} = 1.134 \cdot 10^{-6}$.

Synchronized Distribution

The synchronized distribution schema is illustrated in Figure 6.1. The boxes illustrated the nodes (only four nodes in order to keep the picture simple) and the arrows demonstrate data flow. In each phase j each node i sends data to the node $(i + j) \bmod n$. After each phase the nodes wait for each other, before they start with the next phase. There is no cross traffic during the phases – because each node has a unique target – and by using a synchronization step the phases do not overlap, hence this approach is cross traffic free. As a trade-off a synchronization overhead is introduced.

Unsynchronized Distribution

The unsynchronized distribution schema is probably the most obvious approach to this problem. It is very similar to the synchronized schema, but without the synchronization step after each phase. Therefore each node i sends 1GB to its successor (node $(i+1) \bmod n$).

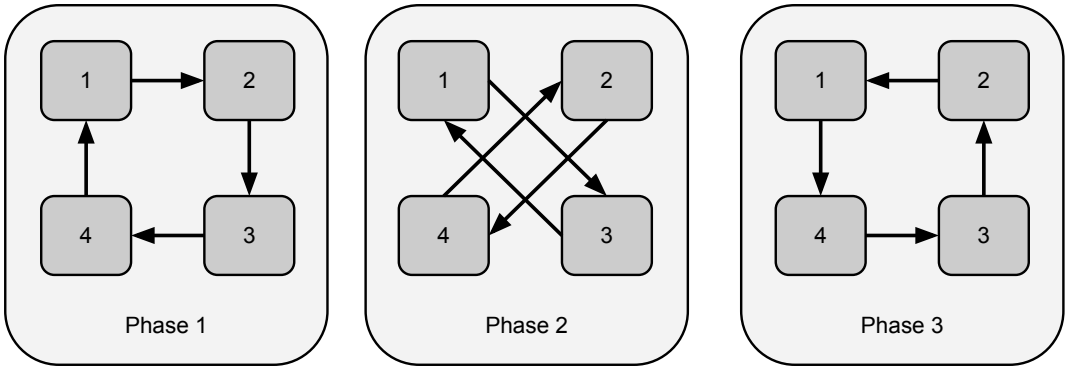


Figure 6.1: Data distribution schema.

When done, the node continues with the successor of its successor. And so on until it has send to all other nodes, without ever waiting for other nodes.

Note that in a real hardware setting this approach is not cross traffic free. If one node finishes a little bit earlier than the other nodes it continues by sending to the successor of its last target, which is still receiving data. Thus slowing both senders down and amplifying this effect.

Summary

Figure 6.2 shows the experimental evaluation. The x-axis shows the nodes by id and the y-axis marks the time needed by each node to finish.

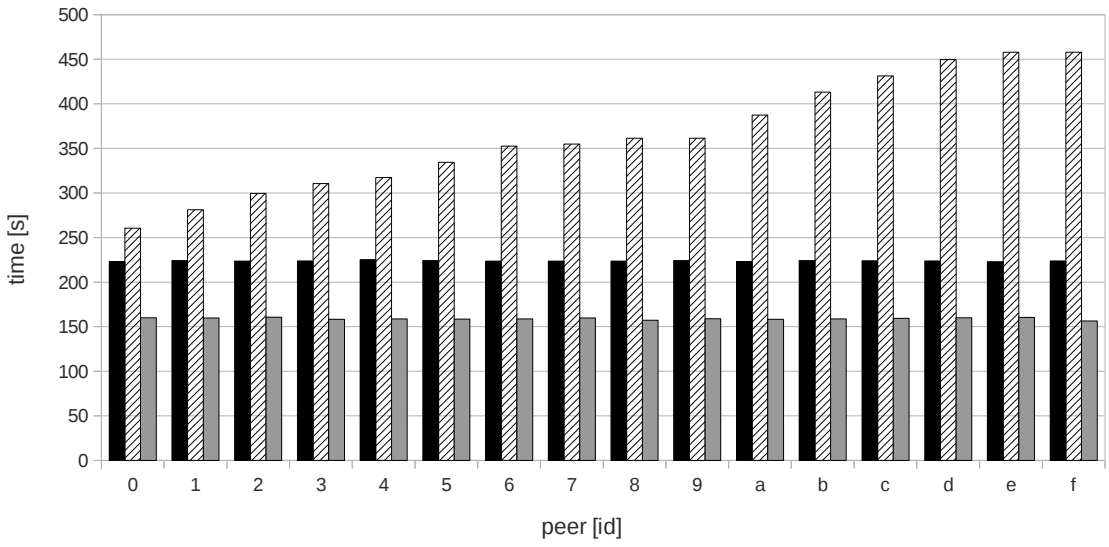


Figure 6.2: Data distribution performance.

The unsynchronized distribution schema has the wost performance results with $457s$ and a bandwidth usage of only $268\frac{Mbit}{s}$ (about 31%). It is interesting to see how the execution time with this approach is growing from node *0* to node *f*. This can be explained with the jam effect. Assume that node j is sending to another node i and for some hardware or operating system related reasons it takes a little bit longer and before the transmission is finished another node k (the predecessor of j) starts sending to i. Consequently the receive bandwidth from i is shared between j and k, thus delaying the start of the transmission from j to $i+1$ and reducing the time until the start of the transmission from k to $i+1$. Consequently the time where j and k share a connection grows. In addition the nodes before k also suffer, because k also needs more time. Thus creating the increase in required time. We called this a jam effect, because a similar effect can be observed on a highway, when one car is slowing down and a traffic jam forms behind it with no real reason.

This jam effect is so bad, that even the random approach with about $223s$ and an effective network bandwidth of $551\frac{Mbit}{s}$ is faster.

The best technique (of the presented ones) is the synchronized distribution schema, without any cross traffic. It takes about $160s$ to transmit the data and allows $768\frac{Mbit}{s}$ of bandwidth. Note that in this schema about 7% of the time per node is wasted for waiting on other nodes.

6.2 Payload Splitting

Payload splitting is used to reduce the network traffic in distributed join algorithms. The following section describes the idea of this method and shows that its application is independent of the actual join algorithm used.

Overview

In distributed database systems it is a common practice to use a method called payload splitting to reduce the size of tuples when executing joins (as studied in [16]). Payload splitting removes the payload of each tuple and replaces it with a record identifier. The record identifier uniquely identifies a tuple. Therefore it has to contain information about the location of the tuple it is referring to. In our implementation we used 16bit to determine the node on which the tuple is resident and another 48 bit to encode the position in the relation on that node.

By replacing the arbitrary long payload of a tuple with a fixed size 64 bit record id, it is possible to reduce the tuples size. In common scenarios like in the TPCH benchmark a table consist of fairly large tuples (over 50 Byte). This can dramatically reduce the amount of network traffic, making payload splitting almost a standard procedure when evaluating distributed joins.

Pre Processing

As shown in Figure 6.3, payload splitting can be implemented around the actual join algorithm. Consequently the main focus for developing distributed join algorithms is the optimization of the actual join algorithm, without payload splitting.

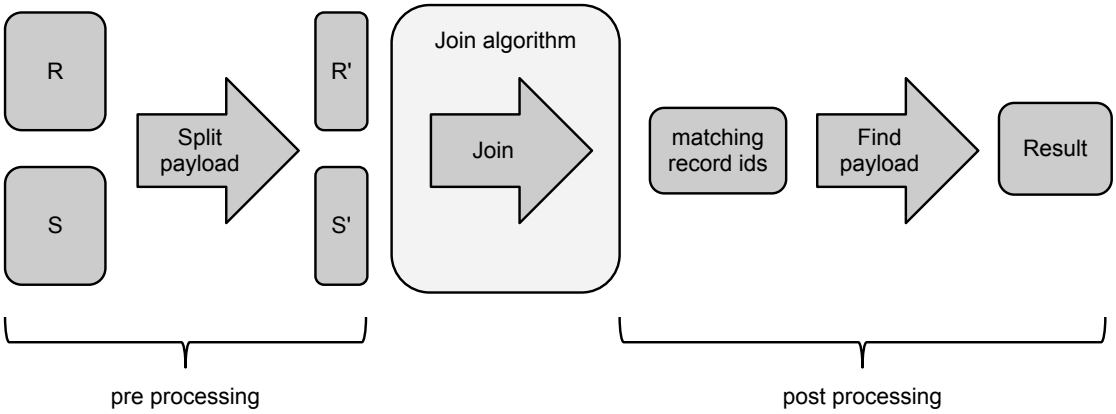


Figure 6.3: Payload splitting overview.

In a preprocessing step the input relations $\mathcal{R}$ and $\mathcal{S}$ are transformed to $\mathcal{R}'$ and $\mathcal{S}'$ by replacing the payload with a record identifier. Thus forming tuples with a key and a 64 bit payload. The key is usually transformed to a 64 bit hash value using a hash function, which provides a good trade-off between speed and collision (for example the murmur hash). Thus avoiding negative effects of data skew (except for input, specially made to harm the performance). In addition this reduces the size of the tuples further if the key of the relation is larger than 64 bit. The end result of the preprocessing step are two relations $\mathcal{R}'$ and $\mathcal{S}'$ containing a 64 bit key and a 64 bit payload, which encodes the record identifier.

Post Processing

The result of the distributed join algorithm is a set of tuples. Each tuple contains the key of $\mathcal{R}'$, the record identifier of the actual tuple of $\mathcal{R}'$, the key of $\mathcal{S}'$ and the record if of the actual tuple of $\mathcal{S}'$. The payload splitting algorithm needs to find the corresponding tuples using the record identifier. This is done in two phases. First the tuples are sent to the node where the tuple of the relation $\mathcal{R}$ is resident and then, second, the tuples are sent to the node where the tuple of the relation $\mathcal{S}$ is resident.

When using a hash function on the keys it is possible to get false positives, because two keys could map to the same hash value. Thus the output of the join is only a set of potential matches. The false positives are detected after the second phase, because when looking up the tuples using the record identifier it is possible to check the keys.

Performance Evaluation

Figure 6.4 shows the performance of a straight forward implementation with no special optimizations. The first bar shows the time needed for the preprocessing step, where the payload gets replaced by a record identifier and the key gets hashed to a 64 bit value using the Murmur hash. The next bar shows the actual join algorithm (radix join) and the last bar denotes the time needed for the post processing.

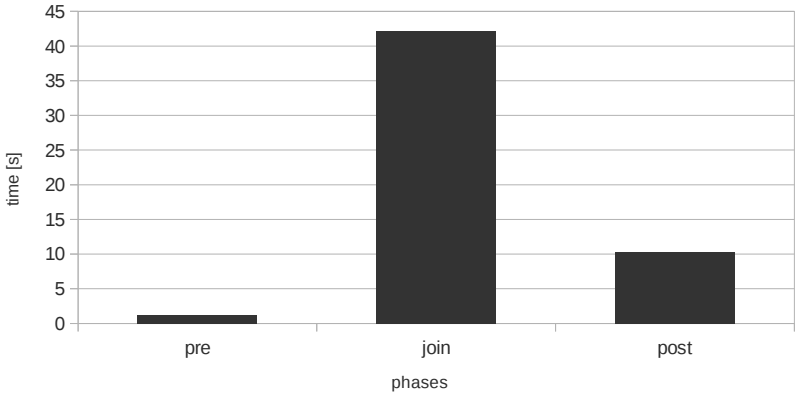


Figure 6.4: Payload splitting performance.

The post processing step is a considerable amount of work, but is strongly dominated by the network (because a lot of data has to be send). In addition it is highly influenced by the join selectivity, because only the matching tuples have to be send. Let ρ be the join selectivity and $\mathcal{N}_\mathcal{S}$ the number of tuples in $\mathcal{S}$ per node. Then the expected time for the post processing phase can be calculated with

$$\text{phase}_1 := \frac{\frac{n-1}{n} \cdot \rho \cdot \mathcal{N}_\mathcal{S} \cdot 2 \cdot (\text{record\_id\_size} + \text{key\_size})}{\text{network\_link\_bandwidth}}$$

$$\text{phase}_2 := \frac{\frac{n-1}{n} \cdot \rho \cdot \mathcal{N}_\mathcal{S} \cdot (\text{record\_id\_size} + \text{key\_size} + \text{tuple\_size})}{\text{network\_link\_bandwidth}}$$

$$\text{ovderall} := \text{phase}_1 + \text{phase}_2 = \frac{\frac{n-1}{n} \cdot \rho \cdot \mathcal{N}_\mathcal{S} \cdot (3 \cdot (\text{record\_id\_size} + \text{key\_size}) + \text{tuple\_size})}{\text{network\_link\_bandwidth}}.$$

The experiment of Figure 6.4 used $\mathcal{N}_\mathcal{S} = 64M$ and $\rho = 25\%$, which evaluates to

$$\text{ovderall} = \frac{\frac{16-1}{16} \cdot 0.25 \cdot 64M \cdot (3 \cdot (8\text{byte} + 8\text{byte}) + 16\text{byte})}{870\frac{\text{Mbit}}{\text{s}}} = 8.8s$$

This shows that our implementation is almost as fast as the theoretical performance limit.

Bibliography

[1] A. Romanow, J. Mogul, T. Talpey, and S. Bailey. *Remote Direct Memory Access (RDMA) over IP Problem Statement*. 2005.

[2] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, June 2012.

[3] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 975–986. ACM, 2010.

[4] Cisco visual networking index: Global mobile data traffic forecast update, 2011-2016.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[6] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, September 2010.

[7] Roland Dreier. *Writing RDMA applications on Linux: Example programs*. 2007.

[8] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. Spinning relations: high-speed networks for distributed join processing. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, pages 27–33, New York, NY, USA, 2009. ACM.

[9] Philip Werner Frey. *Zero-Copy Network Communication: An Applicability Study of iWARP beyond Micro Benchmarks*. 2010.

[10] R. Goncalves and M. Kersten. The data cyclotron query processing scheme. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 75–86, New York, NY, USA, 2010. ACM.

[11] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, SIGMOD '94, pages 243–252, New York, NY, USA, 1994. ACM.

[12] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 8. Auflage*. Oldenbourg, 2011.

[13] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[14] Martin L. Kersten. The database architecture jigsaw puzzle. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 3–4, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, August 2009.

[16] Changkyu Kim, Jongsoo Park, Nadathur Satish, Hongrae Lee, Pradeep Dubey, and Jatin Chhugani. Cloudramsort: fast and efficient large-scale distributed ram sort on shared-nothing cluster. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 841–850, New York, NY, USA, 2012. ACM.

[17] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. Technical report, Amsterdam, The Netherlands, The Netherlands, 1999.

[18] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. Technical report, Amsterdam, The Netherlands, The Netherlands, 1999.

[19] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[20] John Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[21] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[22] InfiniBand Trade Association. InfiniBand Architecture Specification. *http://www.infinibandta.org*.

[23] Joel L. Wolf, Philip S. Yu, John Turek, and Daniel M. Dias. A parallel hash join algorithm for managing data skew. *IEEE Trans. Parallel Distrib. Syst.*, 4(12):1355–1371, December 1993.