FAKULTÄT FÜR !NFORMATIK

Implementierung und Analyse von Parallelen, Verteilten Datenbank Operationen auf Service-orientierten Architekturen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Bakk. techn. Michael Koitz
Matrikelnummer 0227547

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: **Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Erich Schikuta**
Mitwirkung: **Univ.-Ass. Mag. Peter Paul Beran**

Wien, 11.07.2008

Abstract

This thesis covers the optimization and analysis of parallel database operations in a heterogeneous environment. It focuses especially on sort and join operations, because of their relevance in database system. For the implementation of the sort operations the Parallel Binary Merge Sort and the Block Bitonic Sort have been chosen because of their increased performance in parallel processing. For the implementation of the join operations we focused on the Merge Join, Nested Loop Join and the Hash Join.

All these algorithms are implemented as Web services to make them easy transferable from one node (a computer) to another. As framework for implementation SODA has been used because of its flexibility and expandability. SODA is written in Java and uses the Apache Tomcat, a container for the Web services.

The work is motivated by the idea taking advantage of the varying resource characteristics (e.g. network bandwidth) of the heterogeneous environment. The approach was implemented and justified by a speedup and scale-up analysis.

Zusammenfassung

Diese Diplomarbeit befasst sich mit der Optimierung von Parallelen Datenbank Operationen in heterogenen Umgebungen. Der Fokus ist auf Sortier und Join Operationen gelegt, wegen ihrer Relevanz in fast allen Datenbank Systeme. Für die Implementierung der Sortier Operationen wurden der Parallel Binary Merge Sort und der Block Bitonic Sort ausgewählt, weil sie eine gute parallele Effizienz bieten. Für die Implementierung der Join Operationen wurde der Fokus auf den Merge Join, Nested Loop Join und den Hash Join gelegt.

Alle diese Algorithmen wurden als Web Services implementiert um sie einfach zwischen den Nodes (Computer) zu transferieren. Als Framework wurde SODA wegen der Flexibilität und Erweiterbarkeit ausgewählt. SODA ist in Java geschrieben und benutzt Apache Tomcat als Container für die Web Services.

Die Arbeit basiert auf der Idee die variierenden Ressource Charakteristiken (z.B. Netzwerkbandbreite) von heterogenen Umgebungen auszunutzen. Diese Herangehensweise wurde implementiert und mittels Speedup und Scale-up Analyse bestätigt.

Table of Contents

1. Introduction

Databases always have been of great importance in the software engineering domain because hardly any software has to deal with masses of data, so a database is necessary. Therefore database performance is always a big issue. One way to improve the performance of a database is to use several computers for executing operations on a database. These computers can be built cheaper and can be placed in different locations too. Sometimes it is even necessary to have databases on different locations. Ways have to be found to query these databases in an efficient way.

1.1 Motivation

In the beginning Werner Mach (a college from Siemens) told me about his dissertation, on the field of parallel database operations in grids, and that he had a major breakthrough within his work. He figured out that in his equations the network speed was far more important for the overall execution time than someone might think. He wanted to practical proof his theory and therefore was looking for someone who was willing to take the dangerous and stony path to develop an application which fits the requirements. So I took this chance and began to walk the burdensome road. In a first meeting I meet Professor Erich Schikuta who told me what had to be done and how this target could be reached, he also introduced me to Peter Paul Beran, a research assistant, who had developed SODA [4] in his own diploma thesis, which is a framework for the provision and execution of service-oriented database operations. So SODA was taken as the base framework and used to implement the sort and join algorithms that had to be analysed.

1.2 Acknowledgements

First of all I want to thank my parents and my whole family who supported my from the day on I decided to study informatics. Many thanks go to my love Silvia who never had a taught that I could not make it. Thanks for his believe in me goes to Werner Mach who had chosen me to prove his ideas, and for his help. I also want to thank the people at the department of knowledge and business engineering, especially Erich Schikuta and Peter Paul Beran who guided me through the whole process of writing this thesis. Peter also helped me in main parts of the implementation of the operations and so he gets a second thanks from me.

1.3 Definition of Terms

NAT:	Network Address Translation
SOAP:	Simple Object Access Protocol
SODA:	Service Oriented Database Architecture
CSV:	Character/Comma/Colon Separated Values
HDD:	High Density Disc
DBMS:	Database Management System
TPC:	Transaction Processing Performance Council
SQL:	Structured Query Language
GB:	Giga Byte
SF:	Scale Factor
CBQ:	Class-Based Queuing
I/O:	Input/Output
DB:	Database
JVM:	Java Virtual Machine
CPU:	Central Processing Unit
URI:	Uniform Resource Identifier

2. Service-Oriented Environment

Today the focus for new applications lies on the accessibility by other applications and to be available from anywhere. With the Internet the availability from any place in the world is nearly granted.

Service-Oriented Environment is an architectural style for creating and using business processes, resembled as services. Moreover it allows different applications to exchange data and participate in business processes. One of the main aspects in this area is that it separates functions into distinct units called services, which can be accessed over a network and can be combined and therefore reused to create new applications. Here the services communicate with each other by passing data (messages) from one service to another.

Services can run simultaneous on different computers and communicate over a network so this result in a form of parallel computing. Parallel computing is most commonly used to describe program parts running simultaneously on multiple processors on the same computer, of which services also take advantage. Both types of processing require a program that can be divided into parts, running simultaneously, but services often deal with heterogeneous environments, network links of varying latencies, and unpredictable failures in the network transfer or in computer states.

The idea of Mach and Schikuta [1] was to use sort and join operations in a database, transform these into services and design a workflow that takes advantage of the heterogeneous environment. This is done in a way that every node, where an operation can be run, knows its processing power, disc performance and network speed, so a broker who collects this data can easily decide which node(s) to use. The analysis of Mach and Schikuta also shows that the most important factor is the network bandwidth. Generally every workflow - for a specific database operation - can be restructured to perform quicker if the network bandwidth is taken into account.

The Service-oriented environment consists of three base paradigms shown in Figure 1. These paradigms are:

- **Service Registry**
  A service registry is a network-based directory that contains the addresses of available services. It is an entity that accepts and stores information about service providers and provides those contracts to interested service consumers.

- **Service Consumer**

  The service consumer is an application, service or some other type of software that requires a service. It is the entity that initiates the locating of the service in the service registry. After the successful retrieving of the service address it is binding the service over the network and executing the service function(s). The service consumer executes the service by sending a request formatted in the according messaging format.

- **Service Provider**

  The service provider is a network-addressable entity that accepts and executes requests from service consumers. It can be a mainframe system, a component, or some other type of software system that executes the service request. The service provider publishes its information about available functions, address and availability in the service registry to provide access by service consumers.

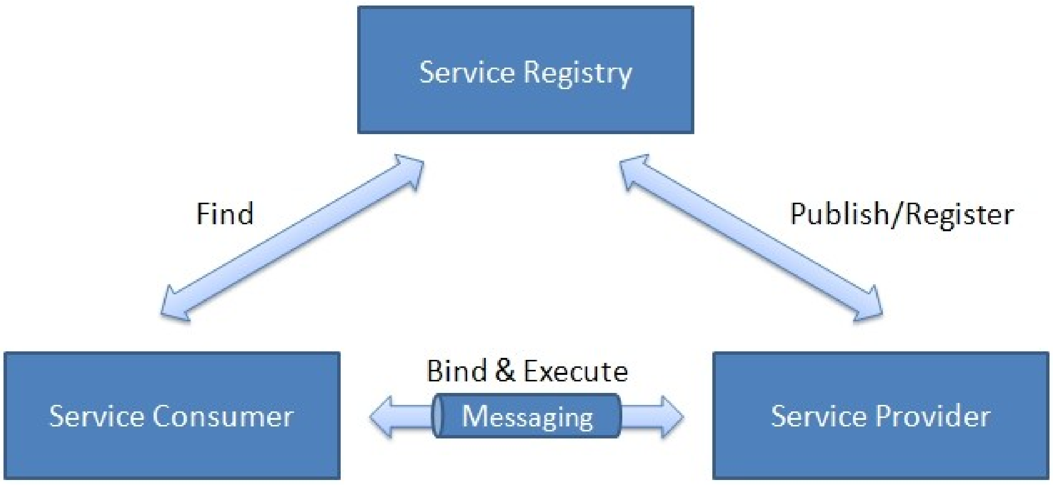


Figure 1: Service-Oriented Environment Schema

3. Sort Operations

Sort operations are a main part of each database system. They are a critical fact when performance is measured, so their optimization is very important.

There are many already known sort algorithms and each of them have their pros and cons. For this thesis the focus lies especially on algorithms that perform faster in parallel processor systems, so they can be adapted to a service oriented environment. Therefore the Parallel Binary Merge Sort and the block bitonic sort are covered in detail.

3.1 Parallel Binary Merge Sort

The Parallel Binary Merge Sort is split into four phases [1]. These phases are the prepare phase, the suboptimal phase, the optimal phase and the postoptimal phase. In Figure 2 and Figure 3 a demonstration of the operation is illustrated. The processing nodes *p* are used to sort *nk* tuples, where *n* denotes the number of pages containing the dataset and *k* is the number of tuples of each page. For a realistic scenario we assume that there are much more pages *n* than processing nodes *p* (*n* >> *p*), also the size of the data set that is processed is much larger than the available main memory of the nodes.

1. **Prepare Phase**

The prepare phase is needed because in a general case the data is not distributed equally across the mass storage of the nodes. Moreover it cannot be assumed that the tuples of the data set are sorted by the sorting criteria of the pages. So the prepare phase distributes the pages to the nodes equally and orders the tuples inside every page by its sort criteria and writes them back to the persistent storage of the node. After the prepare phase is finished *n*/*p* pages are assigned to a specific node and the tuples of the pages are sorted in the nodes.

2. **Suboptimal Phase**

Inside every node the pages are merged. This happens by merging pages of longer and longer runs. A run is an ordered sequence of pages. In each step the length of the runs is twice as large as the preceding run. At the start each node reads two pages, merges them into a run of two pages and writes them back to the disk. This is repeated until all pages are read and merged into two-page-runs. If the number of runs exceeds 2*p*, the suboptimal phase continues with merging two-page-runs to a sorted four-page-run. This continues until all two-page-runs are merged. The phase ends when the number of runs is 2*p*. At the end of the suboptimal phase on each node two sorted files of length *n*/2*p* exist. In the suboptimal phase (see Figure 2) the nodes work independently in parallel, and every node accesses its own data only. [1]

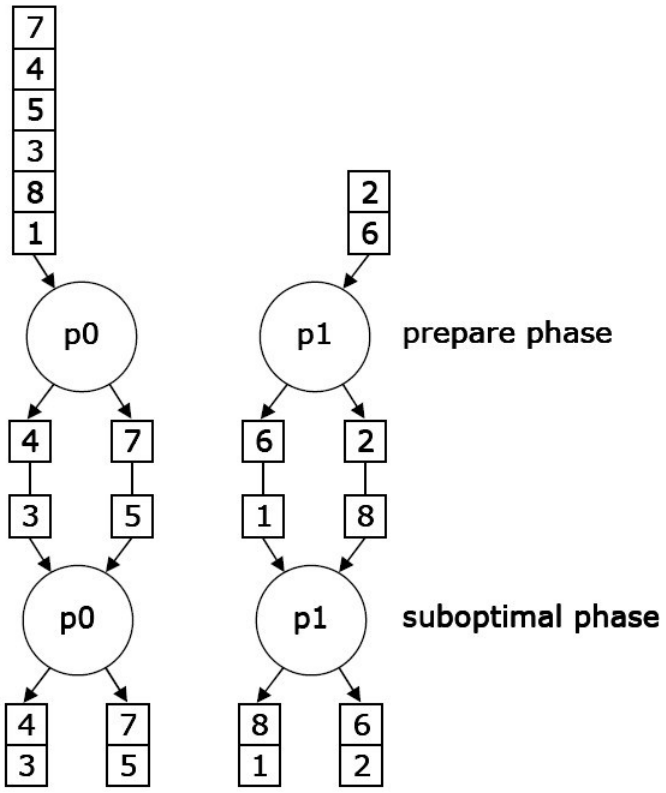


Figure 2: Prepare and Suboptimal Phase

3. Optimal Phase

In the optimal phase each node merges its two runs of length *n*/2*p* and sends them to a target node. Therefore the length of this run is *n*/*p*. To calculate the target node for even source node numbers use

$$nodenr_{target} = \frac{p}{2} + nodenr_{source}$$

and for odd source node numbers use

$$nodenr_{target} = \frac{p}{2} + nodenr_{source} + 1.$$

See Figure 3 for a demonstration of the optimal phase.

4. Postoptimal Phase

The postoptimal phase merges the remaining *p*/2 runs into the final run. This final run has the length *n*. Therefore at the beginning of this phase there are *p*/2 runs. During this phase *p*/2 nodes are not used any more, each of the other nodes are used only once during this phase.

There are two forms of parallelism used in the parallel binary merge sort. First, all nodes of one step work in parallel. Second, the steps of the postoptimal phase overlap in a pipelined fashion (see Figure 3). [1]

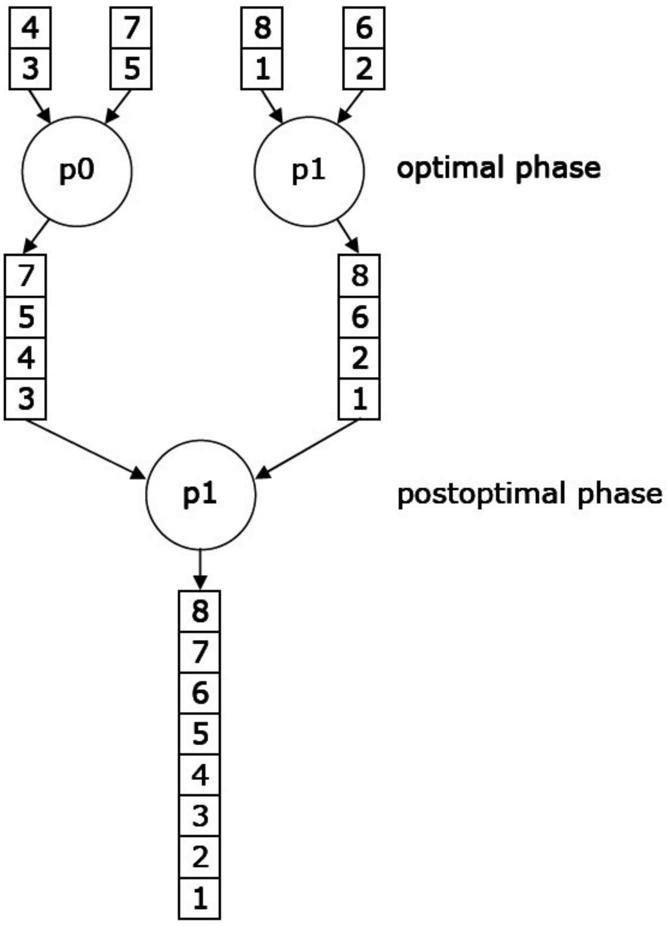


Figure 3: Optimal and Postoptimal Phase

The execution time between two steps consists of merging the first pages, building the first output-page and sending this page to the target-node. During the postoptimal phase every node is used only in one step, which means that every node is idle for a certain time.

Thus the algorithm costs are

$$\left.\frac{n}{2p}\log\left(\frac{n}{2p}\right)\right\}\ \mathit{suboptimal}$$

$$\left.+\frac{n}{2p}\right\}\ \mathit{optimal}$$

$$\left.+\log p - 1 + \frac{n}{2}\right\}\ \mathit{postoptimal}$$

which can be expressed as

$$\frac{n\log n}{2p} + \frac{n}{2} - \left(\frac{n}{2p} - 1\right)(\log p) - 1.$$

3.2 Block Bitonic Sort

The Block Bitonic Sort algorithm was developed by Batcher in 1968 [2]. It sorts *n* numbers with *n*/2 comparator modules in 1/2 *log n* (*log n* + 1) steps [2]. Every step consists of a comparison exchange at every comparator module and a transfer to the target comparator module. A comparator module is a node, which is connected by the perfect shuffle [3] (Figure 4) to each other. The perfect shuffle uses three types of comparator modules (Figure 5), which merges two pages and distributes the lower page and the higher page to two target nodes. The target nodes are defined by using a mask information.

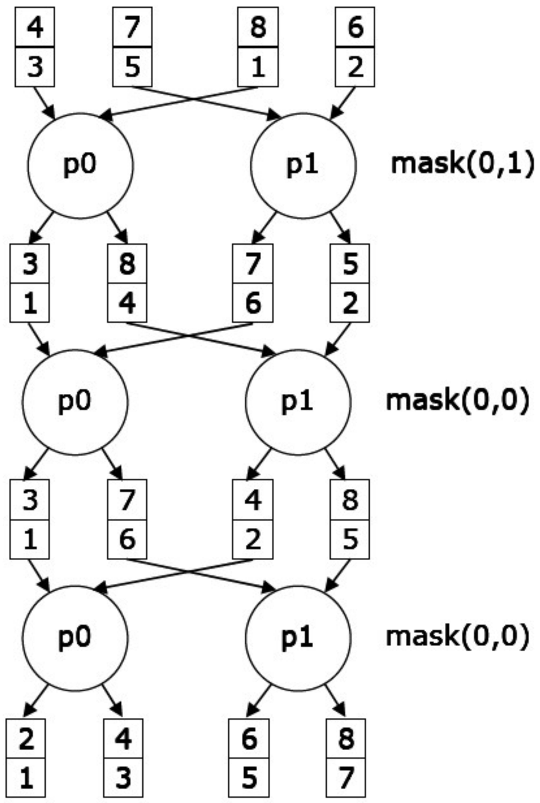


Figure 4: Block Bitonic Sort

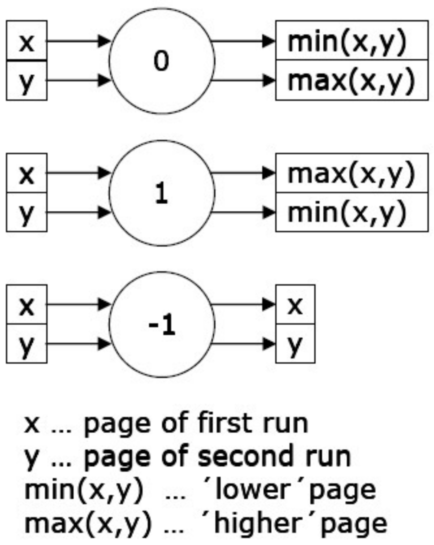


Figure 5: Perfect Shuffle

The algorithm uses three basic operations.

1. Circulate(S)

From each of the $2p$ files the pages are routed to a node through the perfect shuffle interconnection. Every node p_i creates its *MASK*(i). After that it merges its pages and distributes them according to *MASK*(i) = 0, 1, -1

2. Shuffle(MASK)

This operation shuffles the *MASK*-array from the second 2^{p-1} nodes to their target nodes in a way that the target nodes will use the same type of comparator.

3. MASK(i)

It computes the entries of the *MASK*-array. Given an integer i, an array of length $2p$ is created at this operation.

MASK = -1, -1, -1, ..., -1	if $i = -1$
MASK = 0, 1, 0, 1, ..., 0, 1	if $i <= p - 1$
MASK = 0, 0, 0, ..., 0, 0	if $i = p$

For the algorithm it is necessary to build $2p$ equally distributed and sorted runs of length $n/2p$. The prepare phase and the suboptimal phase produce the $2p$ runs. The algorithm for the perfect shuffle is defined in Listing 1.

```
foreach $i$ from $1$ to $p$ do
    Mask(-1);
    foreach $j$ from $1$ to $p-i$ do
        Circulate(S);
    Mask(i);
    foreach $j$ from $p-i+1$ to $p$ do
        Circulate(S);
        Shuffle(MASK);
```

Listing 1: Perfect Shuffle Algorithm

The total costs are:

$$\frac{n}{2p}\left(log\, n + \frac{log^2 2p - log\, 2p}{2}\right).$$

4. Join Operations

Join Operations are also a major part of each database system, therefore their optimization is as well very important.

There are many known join algorithms and each of them has its pros and cons. For this thesis the focus lies on the algorithms that perform especially well in multi processor systems, so they can be adapted to a service-oriented environment. The Nested Loop Join, the Sort Merge Join and the Hash Join are the most prominent ones.

4.1 Nested Loop Join

For this algorithm two steps are necessary to join the two participating relations. The inner relation *T* is the smaller one, and the outer relation *R* is the bigger one.

1. Initiate Phase

The first step is the initiate phase. For this phase each of the processors read a different page of the outer relation *R*.

2. Broadcast and Join Phase

The second step is the broadcast and join phase. All pages of the inner relation *T* are sequentially broadcasted to the processors. After getting the page which was broadcasted, each processor joins the page with its copy of page *R*.

Assume that *n* and *m* are the number of pages of the relations *R* and *R'*, and *n* >= *m*. We assign *p* processors to perform the join of *R* and *R'*. When *p* = *n* the execution time is

$$T_{nested-loop} = T(\text{read a page of } R) + mT(\text{broadcast a page of } R') + mT(\text{join 2 pages}).$$

The join selectivity factor *S* indicates the average number of pages produced by the join of a single page *R* with a single page of *R'*. To join the two pages they are merged, and later on the output page is sorted by the join attribute and written to the disc.

$$S = \frac{size(R \text{ join } T)}{m \cdot n}$$

If the number of processors *K* is smaller than the number of pages *n*, step 1 and 2 must be repeated *n*/*p* times. So the costs for the Parallel Nested Loop Join are

$$T_{nested-loop} = \frac{n}{p}\left[C_r + m[C_r + C_m + S(C_{so} + C_w)]\right].$$

4.2 Sort Merge Join

This algorithm is performed in two steps. First of all the two relations are sorted by the join attribute, because it can be assumed that the two relations are not already sorted. The second step is performed after the sorting. Afterwards the two sorted relations are joined together and the resulting relation is being produced.

For the first step the Block Bitonic Sort can be used, as described in section 3.2. The costs for the Sort Merge Join are

$$T = \left[\frac{n}{2p} \log n + \frac{m}{2p} \log m + (\log^2 2p - \log 2p) \frac{n+m}{4p}\right] C_p^2 + (n+m)\, C_r + max(n \cdot$$

$$m\ Cm + m \cdot n \cdot SCso + Cw.$$

Using the Parallel Binary Merge Sort the costs are

$$T = \left[\frac{n \log n}{2p} + \frac{n}{2} + \left(\frac{n}{2p} - 1\right)(\log p)\right] C_p^2 + (n+m)\, C_r + max(n \cdot m)\, C_m + m \cdot n \cdot$$

$$S(C_{so} + C_w).$$

4.3 Hash Join

The algorithm described here is based on the analysis of Valduriez and Gardarin described in [13]. The Hash Join uses Bit-arrays. The method hashes the join attribute and use the result as an address into the Boolean array. Marked Bits in the array mean that matching tuples exist. The value of the Boolean array is to eliminate most of the data not needed in the result.

This algorithm has two stages. For the first stage a cache processor is chosen and the smaller relation *T* is read into the cache memory and hashed on the join attribute. The results are tuples written into buckets of a hashed file. This hashed file contains buckets, having a variable number of linked pages. In cache memory a page frame is maintained for each bucked, so an overflow area is not needed. At the same time for each join attribute value *v*, *B*(*h*(*v*)) is set to 1, where *h* is a hashing function applied to the join attribute. When the entire relation is hashed the first stage is completed.

At the beginning of the second stage, the Boolean array is sent to *p* processors. The lager relation *R* is sequentially distributed to *p* processors. Each processor uses four buffers. Two for the input pages, one for the output page and one to store the Boolean array. Therefore each processor receives one page of the larger relation and performs the Hash Stage-2 algorithm (see Listing 2).

```
foreach page in a Bucket do
    foreach tuple in the page do
        if join attribute value $\nu'$ satisfies
        $B(h(\nu')) = 1$ then
            One bucket of the hashed file is accessed
            by specifying the key to find the matching
            tuple(s);
            The tuples of each page are then
            compared with $+\nu'$ to complete the join ;
```

Listing 2: Hash Stage-2 Algorithm

More than one hashing function is used to avoid collisions. If v_1 and v_2 are different join attribute values, we can have $h(v_1) = h(v_2)$. The Boolean array is accessed by hashing. Collisions can lead to unnecessary access to the hashed file at the second stage. To reduce this collisions, more than one hashing function $h_1, h_2, \ldots, h_q$ can be used, each is associated with a Boolean array $B_1, B_2, \ldots, B_q$. Then, for each value v, all of the corresponding bits in each B_i must be set (e.g.: $B_1(h_1(v)) = 1$, $B_2(h_2(v)) = 1$, …, $B_q(h_q(v)) = 1$). Increasing q causes the chance of collisions to nearly zero.

The execution time of the algorithm is made up of time T_1 for hashing the smaller relation by the cache processor, time T_2 for sending the larger relation among p processors, time T_3 for accessing the hashed file and time T_4 for writing the result. The time for broadcasting the Boolean arrays is ignored because it is insignificant. There are c page frames available in the cache for the join operation. Therefore the creation of the hashed file consists of creating m buckets if $c > m$, or c buckets otherwise. In the first case, the hashed file could be maintained in cache memory during the entire execution of the join operation. In the other case, the pages of the same bucket would be linked and written on disc and retrieved using a table of physical address. The time for reading a page for R, taking the ratio H into account is

$$t_1 = (1 - H)\, R_c.$$

The time for hashing a page of k tuples is

$$t_2 = k\,(C + V).$$

The time for writing the hashed file, consists of the time for writing $(m - c)$ pages, because c pages are reserved in cache memory during the join execution. Besides that, page frames may be available in cache with the probability H'. Therefore the time for writing $(m - c)$ pages from cache to disc is

$$t_3 = (m - c)(l - H')\, R_c.$$

The execution time for hashing a relation for *m* pages is

$$T_1 = m(t_1 + t_2) + t_3$$

$$T_1 = m[(1 - H)\, R_c + k\,(C + V)] + (m - c)(l - H')\, R_c.$$

The time costs for the second stage consists of:

- reading the relation *S* by *p* processors in parallel
- accessing the hashed file
- writing the result relation

Each processor reads *n*/*p* pages of the relation *S* and accesses the Boolean array for each of the tuples. So the costs are

$$T_2 = (C_r + k\,C)\,\frac{n}{p}.$$

An access to the hashed file is needed if the tuples match. The number of matching tuples is defined by the semijoin selectivity factor *SS*. Therefore each bucket of the hashed file contains $\frac{m}{c}$ pages. So, for each page of S, the number of pages read from the hashed file is

$$T_3 = \frac{n}{p}\,\frac{m}{c}\,C_r \cdot k \cdot SS.$$

The time for writing the result relation of size *m* * *n* * *JS* to disc in parallel by *p* processors is

$$T_4 = m \cdot n \cdot JS\,\frac{cw}{p}.$$

Therefore the total time *T* of the Hashing join is the sum of T_1, T_2, T_3 and T_4.

5. SODA

SODA is a Service Oriented Database Architecture developed by Beran and Habel [4]. It is a framework for distributed database operations. The idea is that everyone can add services, implement algorithms for selection, sort, join, projection, and so on to provide new functions to workflows that connect database specific services.

SODA can answer questions like:

- Does a change of the infrastructure have an impact on the query execution?
- How does a change of the number of working services affect the query execution time?
- Does a change of the performance of a single service affect the query execution time?

The basic mechanism of SODA is that the broker receives a query request and analyzes it. In this step it dynamically builds a workflow (a plan) to execute the query (see Figure 6). To do this, all available services are checked and the best available are subsequently chosen to build the final query execution tree. There must be at least one operator service registered for every operation type in the query, otherwise this operation type cannot be provided by the system. If the operation service is missing, the query cannot be performed and the execution terminates.

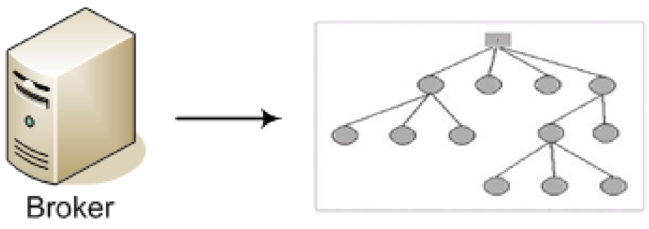


Figure 6: Broker and Execution Tree

There is also another way to execute a query. For testing and benchmarking a workflow can be built which is executed directly without the help of the broker. So operator services can be called and executed in the order which is necessary for the given query. The result is, like in the broker an execution tree.

After the generation of the execution tree, subrequests are sent to the operator services, so every service gets only the part of the query which is necessary for its execution. Beginning at the top of the execution tree all subrequest objects are distributed. All subrequests can have links to other operator services. There can be links for incoming and outgoing data. Incoming links wait for input data from a child operator service while an outgoing link sends results of an operation to a parent operator service, applied on a query execution tree hierarchy.

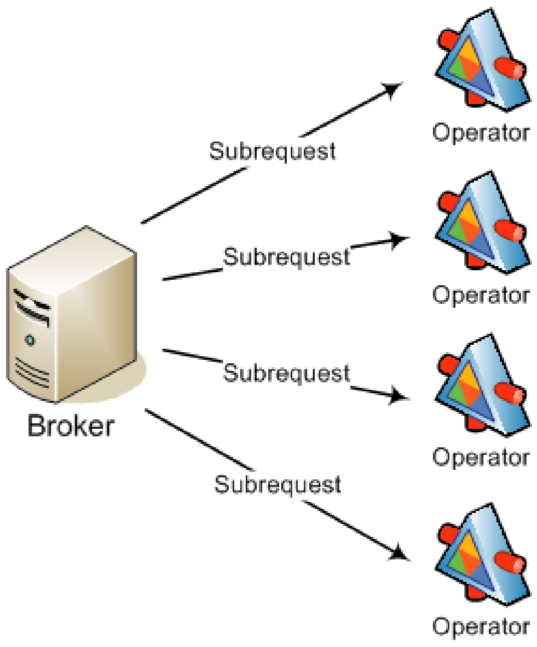


Figure 7: Distribution of Subrequests

When an operator service receives a job it starts with its execution. If the operator service is a data service, which gets data from a database or any other persistent storage, it can start the execution as soon as it is initialized. All other operator services need to wait until they get some input data. Some of the operators have to wait until all data packets are received, others can start with the processing immediately as soon as the first data packet arrives. At this point the broker service is not needed any more. When all operator services have finished their work the overall execution is finished as well. At this point the result is stored at the root node of the execution tree, a data sink operator.

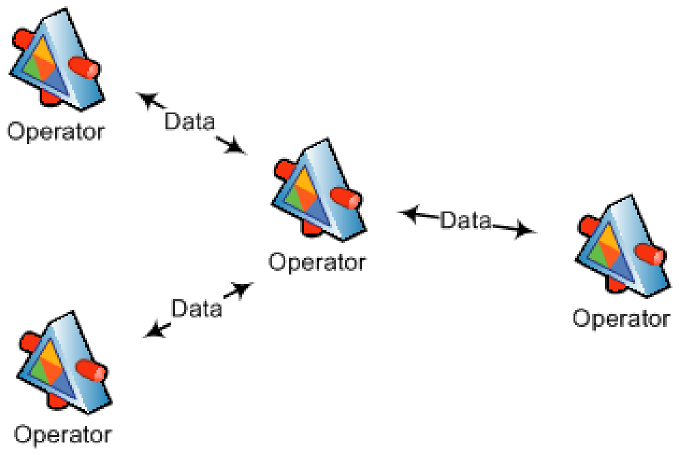


Figure 8: Operator Services at Work

SODA uses SOAP messages for communication, therefore every SODA service can be located at another system or at another location. A service can be of one of the following kinds:

- SODA Client
- SODA Broker Service
- SODA Operator Service
- Data Source (mostly a DB or a flat file)

This distributed architecture looks according to Figure 9.

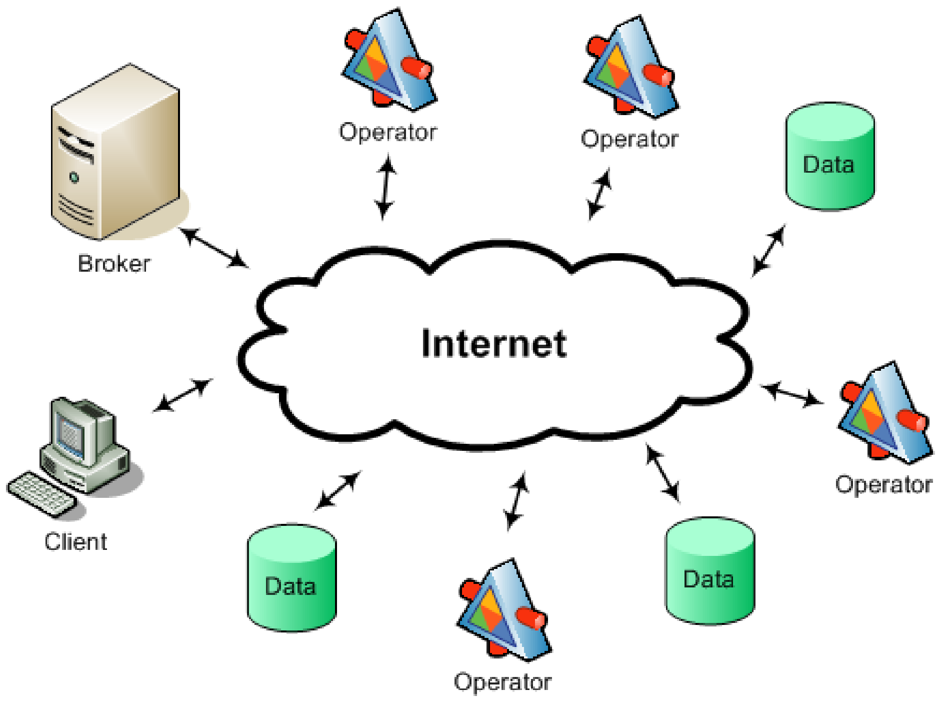


Figure 9: SODA Distributed Architecture

The SODA Client can be every type of code that can establish a connection to one of the services of the SODA system. For example it could be the SODA Broker Service or one of the SODA Operator Services.

The communication between services can be described as a kind of pipeline. Data packets are pushed into one end of the pipeline and received again at the other side of the pipeline. A pipeline must contain at least one pipe but can also have multiple pipes. Each packet in a pipe has a packet id. Therefore, if a packet is lost or the packet order is wrong, the receiver can put them back into the right order.

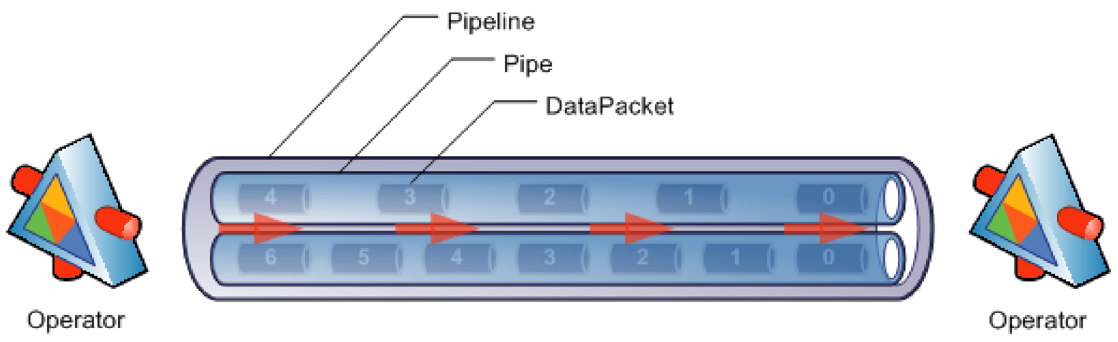


Figure 10: Schematic Pipeline Diagram

The original pipeline was not capable to deal with big data streams, so a caching mechanism had to be implemented. EHCache [12] is a Java object cache and was therefore the best choice for SODA. EHCache is used to store the data packets in memory and if the machine runs out of memory it stores the data on the HDD. The caching can be enabled or disabled. If it is enabled, various parameters like storage path or memory limit can be configured.

Because of the flexibility and the good tool support, XML acts as a well fitting transportation format for SODA. It is used to deliver the data sent from one operator service to another. XML brings flexibility and support, but there is also one big downside. XML produces a lot of overhead data (because of the used XML-tags) than for example a plain CSV file. This leads to a huge transportation overhead which is indeed not a big problem when modern broadband connections are used. Additionally the storage of the XML files, respectively the XML objects in memory and the disc cache is about five to ten times larger than for example storage space that is required to save a comparable CSV file.

5.1 SODA Environment

SODA is written in Java 1.5 [5] and needs therefore a Java Virtual Machine (JVM). It also needs the Apache Tomcat 6.0 [6] as a container for the used Web services.

For developing SODA services there is also Apache Ant 1.6.5 [7], Apache Muse 2.2.0 [8], AXIS 2.1.1 [9] (which is included in Muse) and XMLBeans 2.2.0 [10] required.

MySQL 5 [11] is recommended but not necessary for the persistent storage. It is also possible to use CSV files or any other data service can be implemented for other databases, like Oracle DB, MSSQL, PostgreSQL or any other database management system (it has not to be a relational database).

For further installation instructions please refer to the work of Beran and Habel in [4].

For the benchmarking two machines out of our pool of three are used. Some algorithms like the Bitonic Sort needs a node number which is a power of two, therefore virtualization software was used to simulate up to eight nodes.

5.2 SODA Services

For the implementation of the new Sort and Join Operators new Services had to be written.

5.2.1 Logging Service

The Logging Service (Figure 11) is used to log tracing messages which are sent by other services. It stores the message with the `Timestamp`, `RequestId`, `ClassName`, `ThreadId` and `State` to a persistent file. The message is used to identify the disc, network or CPU usage, so that the duration of the different operations at the different services can be calculated. The timestamp is taken at the service so that a timing problem does not occur.

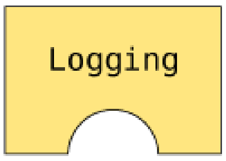


Figure 11: Logging Service

See Figure 12 for the class diagram of the following classes.

LoggingService: Web Service enabled Java class that implements the `IMyCapability` (see [4] for a detailed class description) interface and has among others these methods implemented:

- `getRB`
  Implements the `getRB` method of the `IMyCapability` interface (see [4] for a detailed class description), by returning the `LoggingService ResourceBundle`.
- `doPerform`
  Performs an incoming request, which is sent as an `org.w3c.dom.Element` and specified as a `RequestDocument`.
- `doReceive`
  Receives an incoming response, which is sent as an `org.w3c.dom.Element` and specified as a `RequestDocument`.

- `getState`

  Sends a `Message` that contains the logging messages for the given `RequestId` and `SubRequestId`.

- `makeTable`

  Is a utility function that creates the table representation of the logging Messages.

- `makeBorder`

  Is a utility function that creates the border of the logging table, it is used by the `makeTable` function.

- `makeLine`

  Is a utility function that creates one line of the logging table, it is used by the `makeTable` function.

- `duration`

  Is a utility function that calculates the duration between two `Timestamps` which are captured in milliseconds.

- `formatDuration`

  Is a utility function that formats the milliseconds returned by the `duration` function in the format *Hour*:*Minute*:*Second*.*Millisecond* (e.g.: 5:21:11.329).

LoggingService.properties: Contains the following parameters:

- `param.TYPE = LOGGING`
- `param.SUBTYPE = STORAGE`
- `param.NUMBER_OF_PARALLEL_REQUEST = 10`
- `param.TIMEOUT_FOR_REQUEST = 3600000`

LoggingManager: It calls the `LoggingService` and performs a logging action or retrieve a log from the `LoggingService` with the given `RequestId`.

- `getLog`

  Gets the formatted `LoggingMessage` for the given `RequestId`.

- `log`

  Performs the logging call. `CPU_START`, `CPU_STOP`, `NETWORK_START`, `NETWORK_STOP` or `DISC_DURATION` can be specified as the message for the

`LoggingService` which is necessary for the calculation of the CPU, network and disc I/O duration.

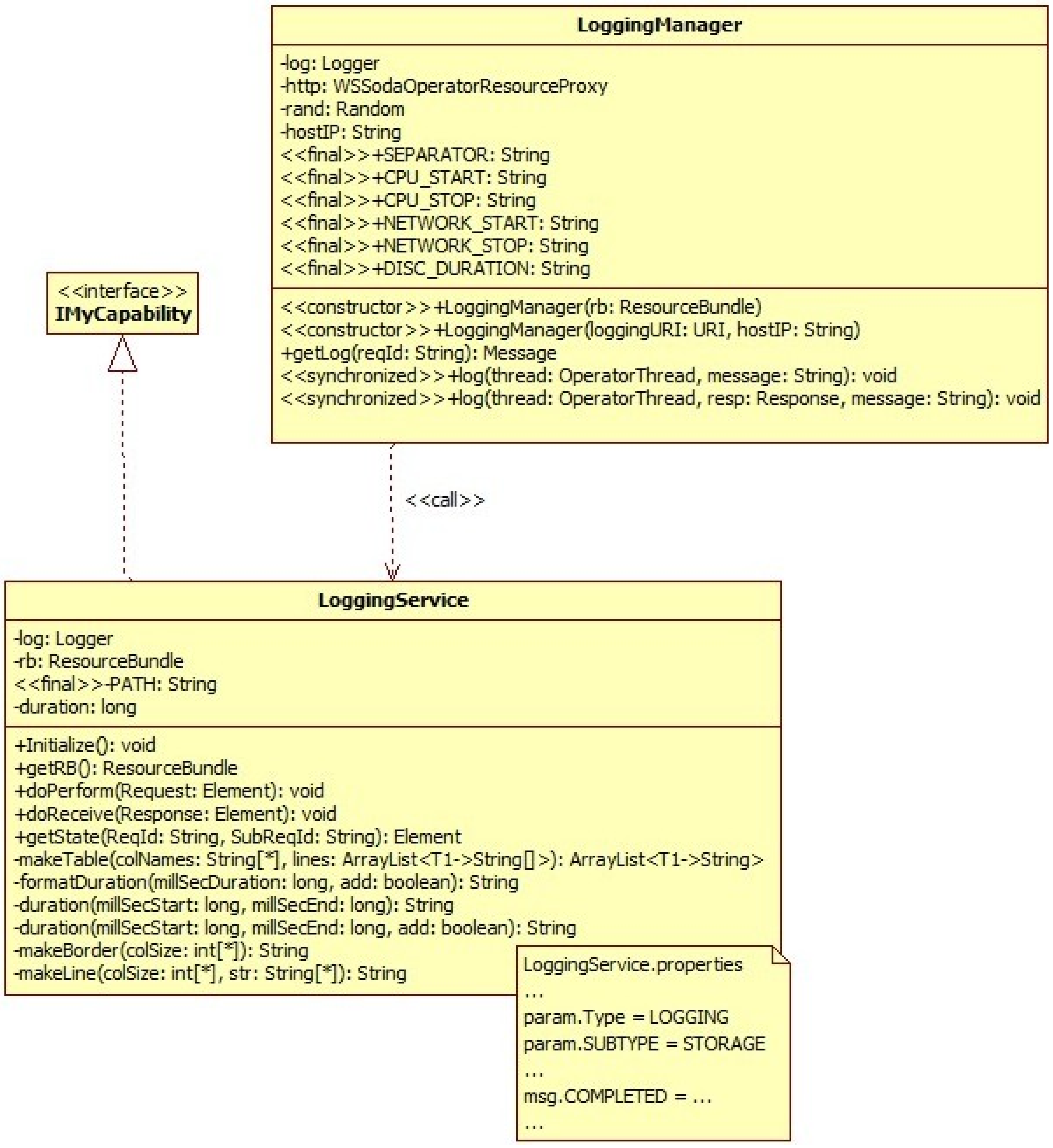

LoggingManager
-log: Logger
-http: WSSodaOperatorResourceProxy
-rand: Random
-hostIP: String
<<final>>+SEPARATOR: String
<<final>>+CPU_START: String
<<final>>+CPU_STOP: String
<<final>>+NETWORK_START: String
<<final>>+NETWORK_STOP: String
<<final>>+DISC_DURATION: String
<<constructor>>+LoggingManager(rb: ResourceBundle)
<<constructor>>+LoggingManager(loggingURI: URI, hostIP: String)
+getLog(reqId: String): Message
<<synchronized>>+log(thread: OperatorThread, message: String): void
<<synchronized>>+log(thread: OperatorThread, resp: Response, message: String): void
<<interface>>
IMyCapability
<<call>>
LoggingService
-log: Logger
-rb: ResourceBundle
<<final>>-PATH: String
-duration: long
+Initialize(): void
+getRB(): ResourceBundle
+doPerform(Request: Element): void
+doReceive(Response: Element): void
+getState(ReqId: String, SubReqId: String): Element
-makeTable(colNames: String[*], lines: ArrayList<T1->String[]>): ArrayList<T1->String>
-formatDuration(millSecDuration: long, add: boolean): String
-duration(millSecStart: long, millSecEnd: long): String
-duration(millSecStart: long, millSecEnd: long, add: boolean): String
-makeBorder(colSize: int[*]): String
-makeLine(colSize: int[*], str: String[*]): String
LoggingService.properties
...
param.Type = LOGGING
param.SUBTYPE = STORAGE
...
msg.COMPLETED = ...
...

Figure 12: LoggingService Classes

5.2.2 MultiFetch Service

The MultiFetch Service (Figure 13) fetches data from a MySQL database and splits the input data stream to multiple pipelines which can have multiple pipes, and so it distributes the data over a variable amount of endpoints (the x in Figure 13 illustrates this).

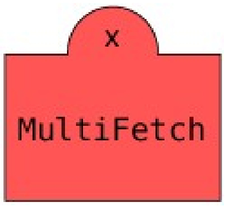


Figure 13: MultiFetch Service

See Figure 14 for the class diagram of the following classes.

MultiFetchService: Web Service enabled Java class that implements the `IMyCapability` (see [4] for a detailed class description) interface and has among others these methods implemented:

- `getRB`
  Overloads the abstract method `getRB` of the `BaseWSResourceCapability` class (see [4] for a detailed class description), by returning the `MultiFetchService ResourceBundle`.

- `doPerform`
  Performs an incoming request, which is sent as an `org.w3c.dom.Element` and specified as a `RequestDocument`.
  - Case 1: If the request arrived the first time this class creates a new `MultiFetchThread` instance and starts this thread using the `OperatorThreadController` (see [4] for a detailed class description).
  - Case 2: If the request and its `ReqId` and `SubReqId` are known (has arrived at least once before) and the created thread is still running it skips the request.
  - Case 3: If the request has already arrived and the result is available it propagates the result (`ResponesDocument`) again to all URIs listed in the `OutputTo` parts of the `RequestDocument`.

- `getMYSQLConnectionName`

Returns the value of the `mysql.connection` property declared in the `MultiFetchService.properties` file. This MySQL connection has to be created manually in the system control panel.

- `getMYSQLUser`
  Returns the value of the `mysql.user` property declared in the `MultiFetchService.properties` file. This user must be created in the database with the administration tool.
- `getMYSQLPassword`
  Returns the value of the `mysql.password` property declared in the `MultiFetchService.properties` file. This is the user's password and can be set with the database administration tool.
- `getMYSQLCatalog`
  Returns the value of the `mysql.catalog` property declared in the `MultiFetchService.properties` file. This defines the database used by this service.
- `getMYSQLConnection`
  This creates a new `java.sql.Connection` session with the above specified settings (connection, user, and password).
- `getDBSchema`
  Provides a database schema for the database accessed. To obtain a listing of all available tables in the specified database the SQL command `TABLE_CAT` is used. Each table name is retrieved using the `TABLE_NAME` command. To get the column type of a column the `TYPE_NAME` command is used.

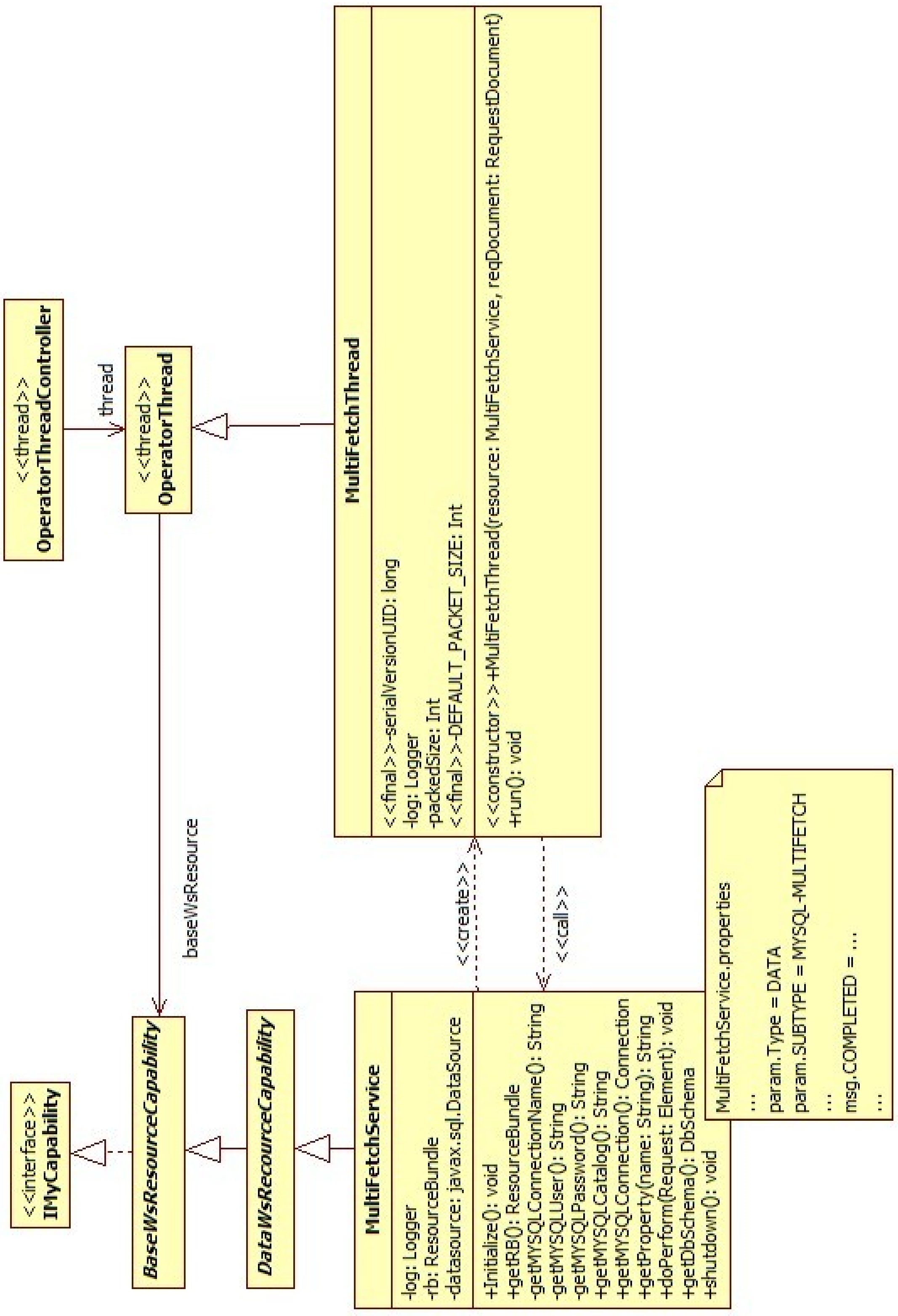
MultiFetchService.properties: Contains the following parameters:

- `param.TYPE = DATA`
- `param.SUBTYPE = MULTIFETCH`
- `param.QUANTITY = PACKETS`
- `param.PROPAGATION = EAGER`
- `param.PROPAGATION_MODE = SYNC`
- `param.PARALLELIZATION = NONE`

- `param.SNAPSHOT = ON`
- `param.NUMBER_OF_PARALLEL_REQUEST = 4`
- `param.PACKET_SIZE = 1000`
- `param.TIMEOUT_FOR_REQUEST = 3600000`
- `mysql.connection = SODA_MYSQL`
- `mysql.catalog = sodadb`
- `mysql.user = soda`
- `mysql.password = soda`

MultiFetchThread: Implements the application logic for a data operation. It retrieves the `TABLE_NAME` from the `RequestDocument`, reads out the table data and creates a new `WebRowSetDocument`. It propagates the final `WebRowSetDocument` in the pipelining mode to the defined endpoints. The thread splits the `WebRowSetDocument` into packets with a specified number of rows defined in `param.PACKET_SIZE` property. Each packet is sent in an incrementally increasing `packetId`, this ensures the correct ordering of the packets at the receiver's side.

- `run`
  Performs the fetching operation of the operation.
- throws `MissingParameterException`
  Thrown, if the request parameter `TABLE_NAME` is not specified in the `RequestDocument`.
- throws `InvalidTableNameException`
  Thrown, if the `TABLE_NAME` is a `null` string.
- throws `TableNotFoundException`
  Thrown, if the `TABLE_NAME` is not found in the database shema.
- throws `SQLException`
  Thrown, if the given SQL statement fails.



<<thread>>
OperatorThreadController
thread
<<thread>>
OperatorThread
baseWsResource
MultiFetchThread
<<final>>-serialVersionUID: long
-log: Logger
-packedSize: Int
<<final>>-DEFAULT_PACKET_SIZE: Int
<<constructor>>+MultiFetchThread(resource: MultiFetchService, reqDocument: RequestDocument)
+run(): void
<<create>>
<<call>>
<<interface>>
IMyCapability
BaseWsResourceCapability
DataWsRecourceCapability
MultiFetchService
-log: Logger
-rb: ResourceBundle
-datasource: javax.sql.DataSource
+Initialize(): void
+getRB(): ResourceBundle
-getMYSQLConnectionName(): String
-getMYSQLUser(): String
-getMYSQLPassword(): String
+getMYSQLCatalog(): String
+getMYSQLConnection(): Connection
+getProperty(name: String): String
+doPerform(Request: Element): void
+getDbSchema(): DbSchema
+shutdown(): void
MultiFetchService.properties
...
param.Type = DATA
param.SUBTYPE = MYSQL-MULTIFETCH
...
msg.COMPLETED = ...
...

Figure 14: MultiFetch Service Classes

5.2.3 MultiSink Service

The MultiSink Service (Figure 15) collects the data from multiple pipelines, which can have multiple pipes, and stores it to be fetched by a client. The x in Figure 15 illustrates this behaviour.

MultiSink
x

Figure 15: MultiSink Service

See Figure 16 for the class diagram of the following classes.

MultiSinkService: Web Service enabled Java class that implements the `IMyCapability` (see [4] for a detailed class description) interface and has among others these methods implemented:

- `getRB`
  Overloads the abstract method `getRB` of the `BaseWSResourceCapability` class (see [4] for a detailed class description) by returning the `MultiSinkService ResourceBundle`.

- `doPerform`
  Performs an incoming request, which is sent as an `org.w3c.dom.Element` and specified as `RequestDocument`.
  - Case 1: If the request arrived the first time this class creates a new `MultiSinkThread` instance and starts this thread using the `OperatorThreadController` (see [4] for a detailed class description).
  - Case 2: If the request and its `ReqId` and `SubReqId` are known (has arrived at least once before) and the created thread is still running it skips the request.

MultiSinkService.properties: Contains the following parameters:

- `param.TYPE = STORAGE`
- `param.SUBTYPE = XML-MULTISINK`
- `param.QUANTITY = FULL`
- `param.PROPAGATION = EAGER`
- `param.PROPAGATION_MODE = SYNC`

- `param.PARALLELIZATION = NONE`
- `param.SNAPSHOT = ON`
- `param.NUMBER_OF_PARALLEL_REQUEST = 10`
- `param.TIMEOUT_FOR_REQUEST = 3600000`

MultiSinkThread: The `MultiSinkThread` can be specified to store the `WebRowSetDocuments` received by the input channels as an XML file in the local file system. If it is not necessary to store the XML file, only the rows are counted and this count is saved to the local file system.

- `run`
  Performs the store operation.

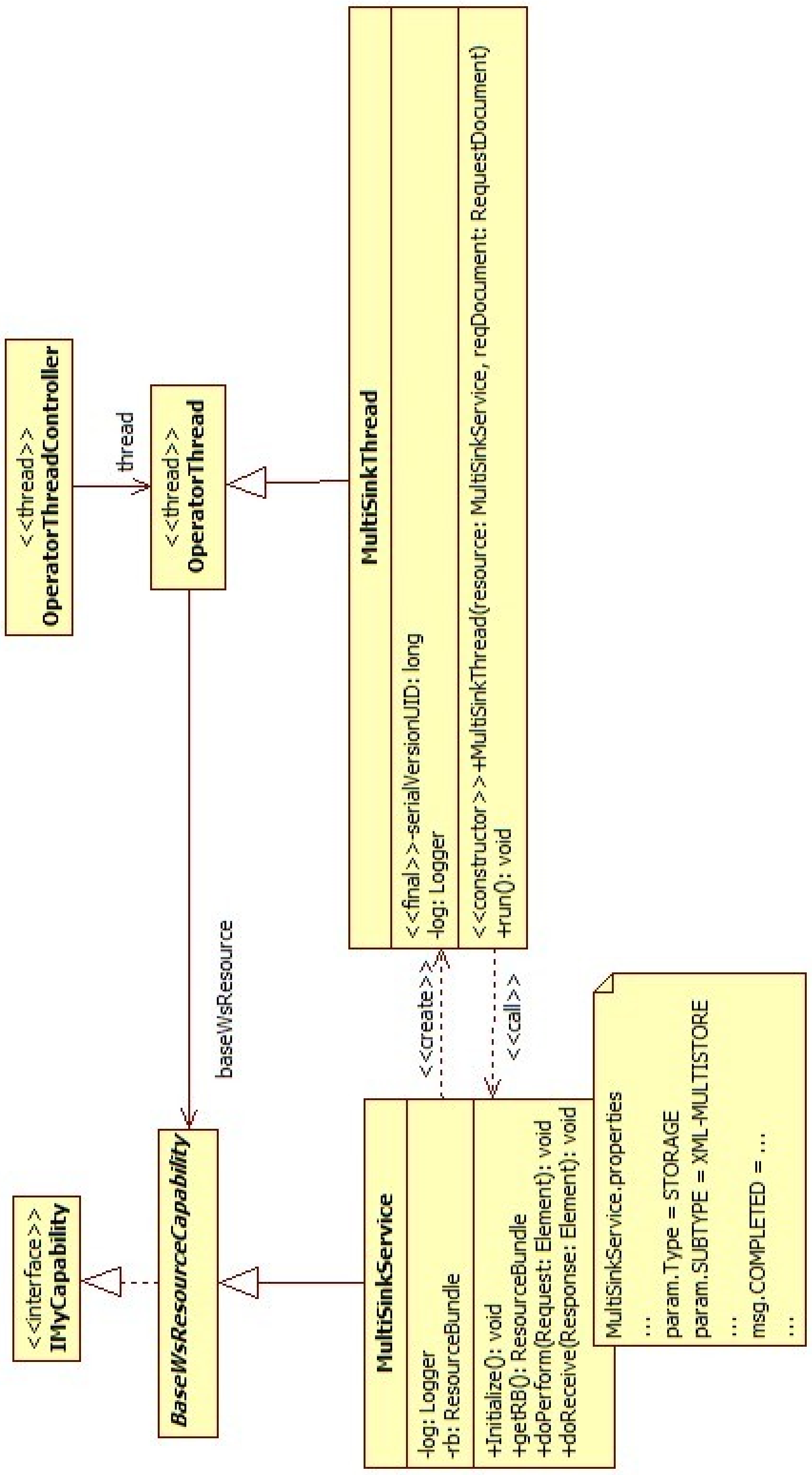


Figure 16: MultiSink Service Classes

5.2.4 MultiRelay Service

The MultiRelay Service (Figure 17) collects data from multiple services and concatenates this data. It adds data in the ordering of the pipelines and their pipes. The x in Figure 17 illustrates this behaviour.

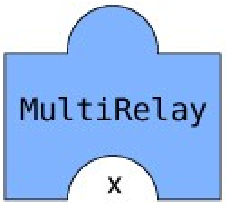


Figure 17: MultiRelay Service

See Figure 18 for the class diagram of the following classes.

MultiRelayService: Web Service enabled Java class that implements the `IMyCapability` (see [4] for a detailed class description) interface and has among others these methods implemented:

- `getRB`
  Overloads the abstract method `getRB` of the `BaseWSResourceCapability` class (see [4] for a detailed class description), by returning the `MultiRelayService ResourceBundle`.

- `doPerform`
  Performs an incoming request, which is sent as an `org.w3c.dom.Element` and specified as `RequestDocument`.
  - Case 1: If the request arrived the first time, this class creates a new `MultiRelayThread` instance and starts this thread using the `OperatorThreadController` (see [4] for a detailed class description).
  - Case 2: If the request and its `ReqId` and `SubReqId` are known (has arrived at least once before) and the created thread is still running, it skips the request.
  - Case 3: If the request has already arrived and the result is available, it propagates the result (`ResponesDocument`) again to all URIs listed in the `OutputTo` parts of the `RequestDocument`.

MultiRelayService.properties: Contains the following parameters:

- `param.TYPE = JOIN`

- `param.SUBTYPE = MULTIRELAY`
- `param.QUANTITY = PACKETS`
- `param.PROPAGATION = EAGER`
- `param.PROPAGATION_MODE = SYNC`
- `param.PARALLELIZATION = NONE`
- `param.SNAPSHOT = ON`
- `param.PACKET_SIZE = 1000`
- `param.NUMBER_OF_PARALLEL_REQUEST = 10`
- `param.TIMEOUT_FOR_REQUEST = 3600000`

MultiRelayThread: It combines the `WebRowSetDocuments` received by the input pipelines. The data is added to the output in the ordering of the pipelines. When enough data is combined to fill a packet, this packet is propagated in a new `ResponseDocument` (containing the `WebRowSetDocument`). This is done till all data has arrived on all pipelines.

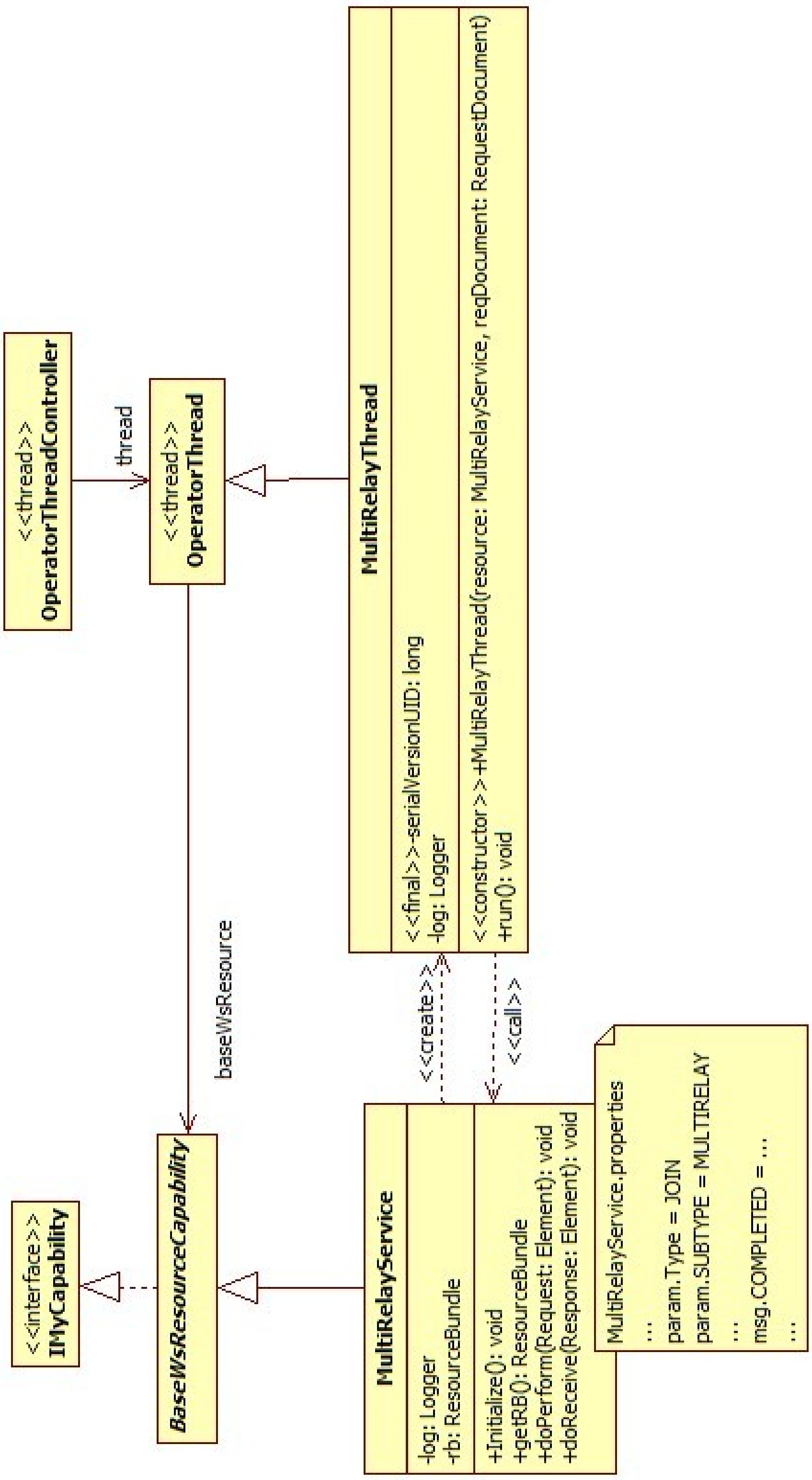


Figure 18: MultiRelay Service Classes

5.2.5 QuickSort Service

The QuickSort Service (Figure 19) implements the famous quick sort algorithm. All data has to be present for the QuickSort Service to begin its work.

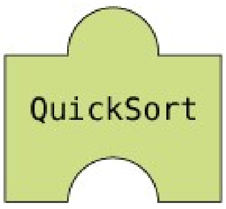


Figure 19: QuickSort Service

See Figure 20 for the class diagram of the QuickSort Service. For a detailed description refer to the thesis of Beran and Habel [4].

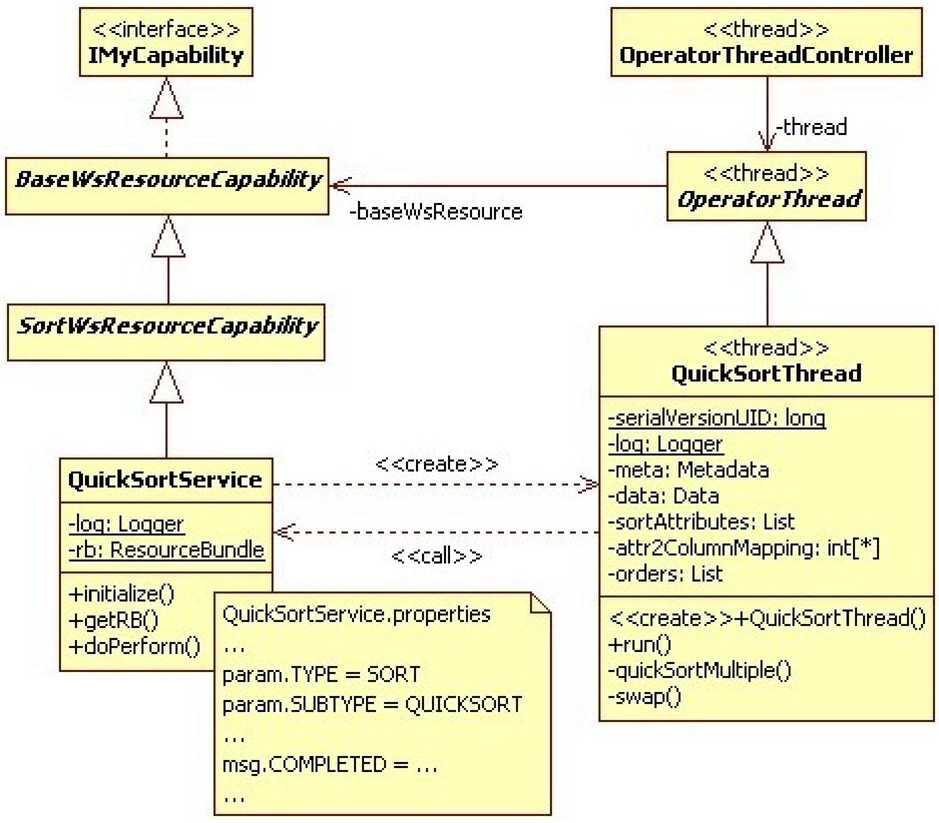


Figure 20: QuickSort Service Classes

5.2.6 ParallelBinaryMergeSort Service

This service retrieves two sorted input streams and merges them to one sorted output stream. As the first parts of data arrive at the two input channels, the ParallelBinaryMergeSort Service (Figure 21) begins its work to compare the two streams and merge them to one sorted output stream.

PBMergeSort

Figure 21: ParallelBinaryMergeSort Service

See Figure 22 for the class diagram of the following classes.

PBMSortService: Web Service enabled Java class that implements the `IMyCapability` (see [4] for a detailed class description) interface and has among others these methods implemented:

- `getRB`
  Overloads the abstract method `getRB` of the `BaseWSResourceCapability` class (see [4] for a detailed class description), by returning the `PBMSortService ResourceBundle`.

- `doPerform`
  Performs an incoming request, which is sent as an `org.w3c.dom.Element` and specified as `RequestDocument`.
  - Case 1: If the request arrived the first time this class creates a new `PBMSortThread` instance and starts this thread using the `OperatorThreadController` (see [4] for a detailed class description).
  - Case 2: If the request and its `ReqId` and `SubReqId` are known (has arrived at least once before) and the created thread is still running it skips the request.
  - Case 3: If the request has already arrived and the result is available it propagates the result (`ResponesDocument`) again to all URIs listed in the `OutputTo` parts of the `RequestDocument`.

PBMSortService.properties: Contains the following parameters:

- `param.TYPE = SORT`
- `param.SUBTYPE = PBMSORT`
- `param.QUANTITY = PACKETS`
- `param.PROPAGATION = EAGER`
- `param.PROPAGATION_MODE = SYNC`
- `param.PARALLELIZATION = INTER`
- `param.SNAPSHOT = ON`
- `param.NUMBER_OF_PARALLEL_REQUEST = 10`
- `param.PACKET_SIZE = 500`
- `param.TIMEOUT_FOR_REQUEST = 3600000`

PBMSortThread: Implements the application logic for the parallel binary merge sort operation described in section 3.1. The execution starts when at least one of the two input channels contains some data. The algorithm then continuously merges data from the first channel with data from the second channel. When enough data is merged to fill a packet, this packet is propagated in a new `ResponseDocument` (containing the `WebRowSetDocument`). This is done until all data has arrived at the two input channels.

- `run`
  Performs the merging operation of the sorting algorithm.
- throws `MissingParameterException`
  Thrown, if there is no parameter `ATTRIBUTE` specified in the `RequestDocument`.
- throws `MissingColumnNameException`
  Thrown, if there is a column definition in the `Metadata` part that does not contain a column name.
- throws `ColumnNotFoundException`
  Thrown, if one specified column cannot be found in the `Metadata` part.

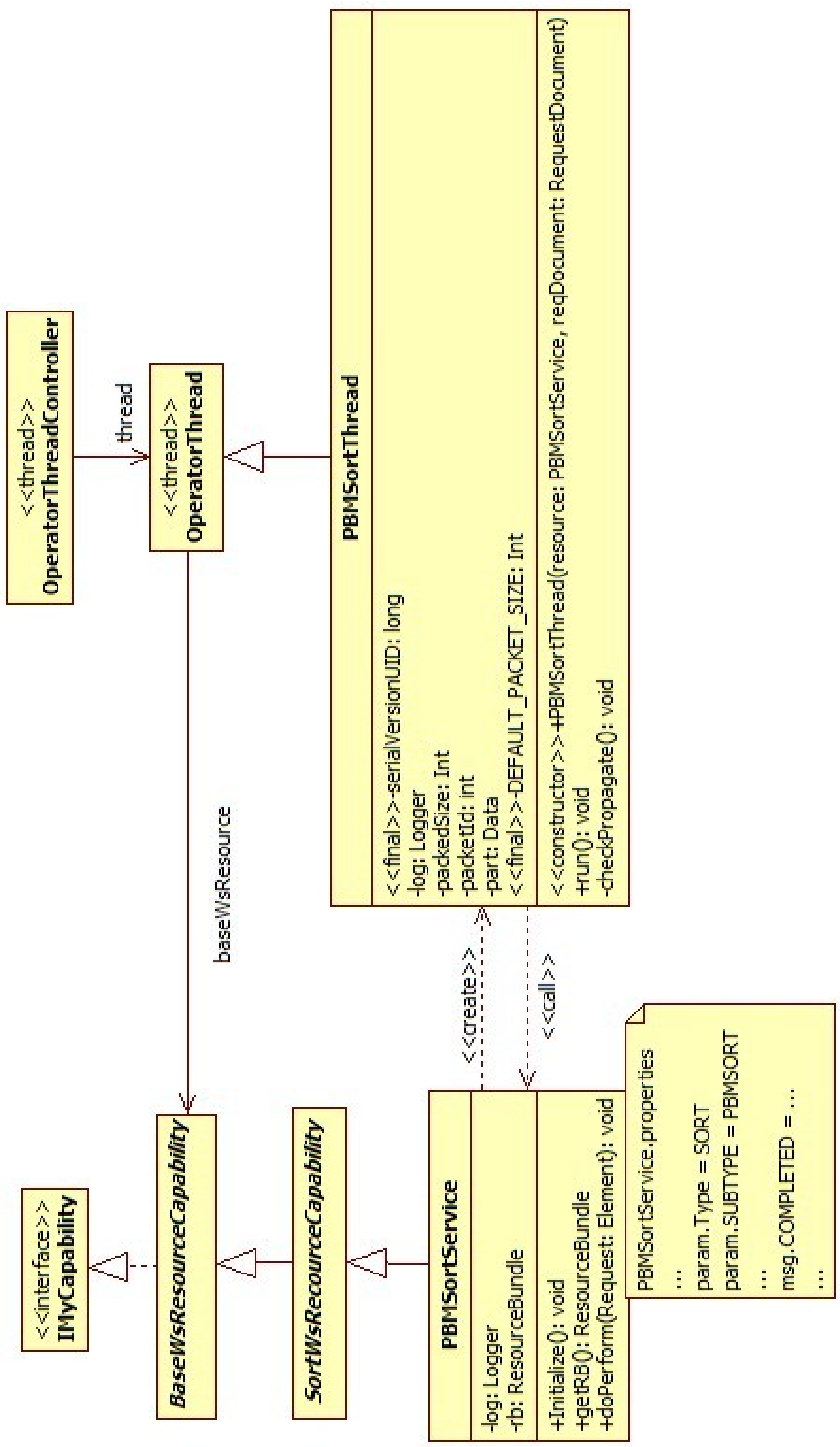


Figure 22: ParallelBinaryMergeSort Service Classes

5.2.7 BitonicSort Service

The BitonicSort Service (Figure 23) implements the Bitonic Sorter, it has two input pipelines and two output pipelines.

BitonicSort

Figure 23: BitonicSort Service

See Figure 24 for the class diagram of the following classes.

BitonicSortService: Web Service enabled Java class that implements the `IMyCapability` (see [4] for a detailed class description) interface and has among others these methods implemented:

- `getRB`
  Overloads the abstract method `getRB` of the `BaseWSResourceCapability` class (see [4] for a detailed class description), by returning the `BitonicSortService ResourceBundle`.

- `doPerform`
  Performs an incoming request, which is sent as an `org.w3c.dom.Element` and specified as `RequestDocument`.
    - Case 1: If the request arrived the first time this class creates a new `BitonicSortThread` instance and starts this thread using the `OperatorThreadController` (see [4] for a detailed class description).
    - Case 2: If the request and its `ReqId` and `SubReqId` are known (has arrived at least once before) and the created thread is still running it skips the request.
    - Case 3: If the request has already arrived and the result is available it propagates the result (`ResponesDocument`) again to all URIs listed in the `OutputTo` parts of the `RequestDocument`.

BitonicSortService.properties: Contains the following parameters:

- `param.TYPE = SORT`
- `param.SUBTYPE = BITONICSORT`

- `param.QUANTITY = PACKETS`
- `param.PROPAGATION = EAGER`
- `param.PROPAGATION_MODE = SYNC`
- `param.PARALLELIZATION = INTER`
- `param.SNAPSHOT = ON`
- `param.NUMBER_OF_PARALLEL_REQUEST = 20`
- `param.PACKET_SIZE = 1000`
- `param.TIMEOUT_FOR_REQUEST = 3600000`

BitonicSortThread: Implements the application logic for the bitonic sort operation described in section 3.2. The execution starts when the whole data has arrived at the two input channels. It combines the data from both inputs and checks the *MASK* information it gets. If the *MASK* is -1 it just splits the data into two parts and propagates two new `ResponseDocuments` (containing the `WebRowSetDocuments`). In case of 0 or 1 it sorts the data with the quick sort algorithm, if the *MASK* is 0 with ascending order otherwise with descending order. After that the data is split into two halves and each half is propagated as a new `ResponseDocuments` (containing the sorted `WebRowSetDocuments`).

- `run`
  Performs the *MASK* decisions of the sorting algorithm.
- `quickSortMultiple`
  Performs the multi-level quick sorting algorithm, which compares and sorts on the attributes on the next level if the current attributes are equal.
- `swap`
  Swaps two rows in a data document.
- throws `MissingParameterException`
  Thrown, if there is no parameter `ATTRIBUTE` specified in the `RequestDocument`.
- throws `MissingColumnNameException`
  Thrown, if there is a column definition in the `Metadata` part that does not contain a column name.
- throws `ColumnNotFoundException`
  Thrown, if one specified column cannot be found in the `Metadata` part.

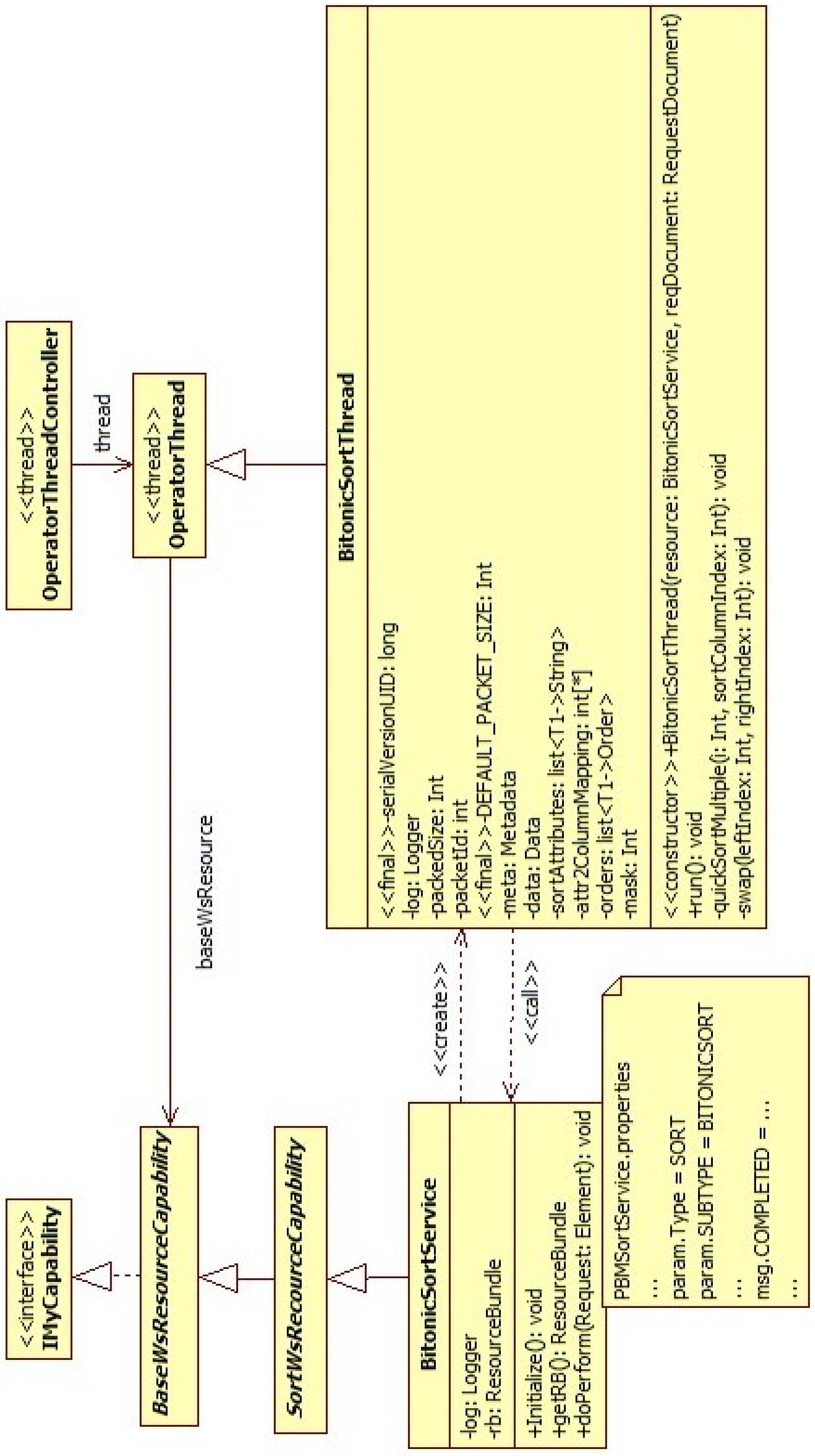


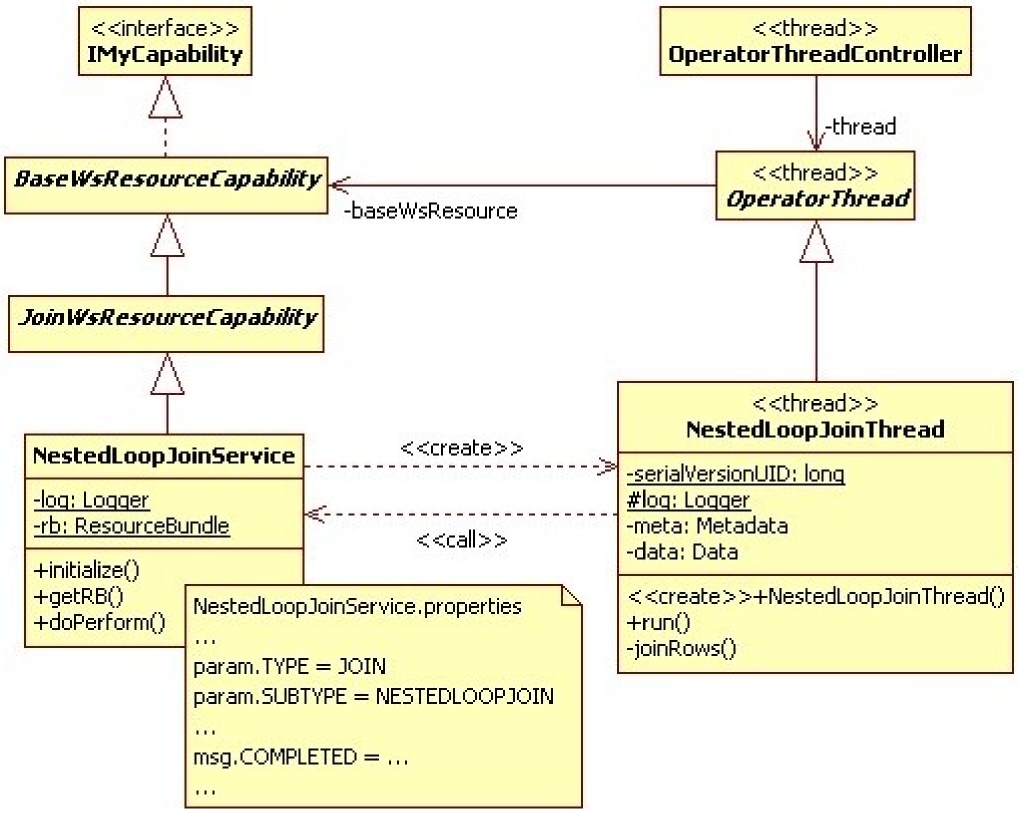
Figure 24: BitonicSort Service Classes

5.2.8 NestedLoopJoin Service

The NestedLoopJoin Service (Figure 25) implements the nested loop join algorithm described in section 4.1.

Figure 25: NestedLoopJoin Service

See Figure 26 for the class diagram of the NestedLoopJoin Service. For a detailed description refer to the thesis of Beran and Habel [4].



```
NestedLoopJoinService.properties
...
param.TYPE = JOIN
param.SUBTYPE = NESTEDLOOPJOIN
...
msg.COMPLETED = ...
...
```

Figure 26: NestedLoopJoin Service Classes

5.2.9 MergeJoin Service

The MergeJoin Service (Figure 27) implements the merge join algorithm described in section 4.2.

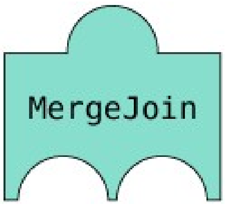


Figure 27: MergeJoin Service

See Figure 28 for the class diagram of the MergeJoin Service. For a detailed description refer to the thesis of Beran and Habel [4].

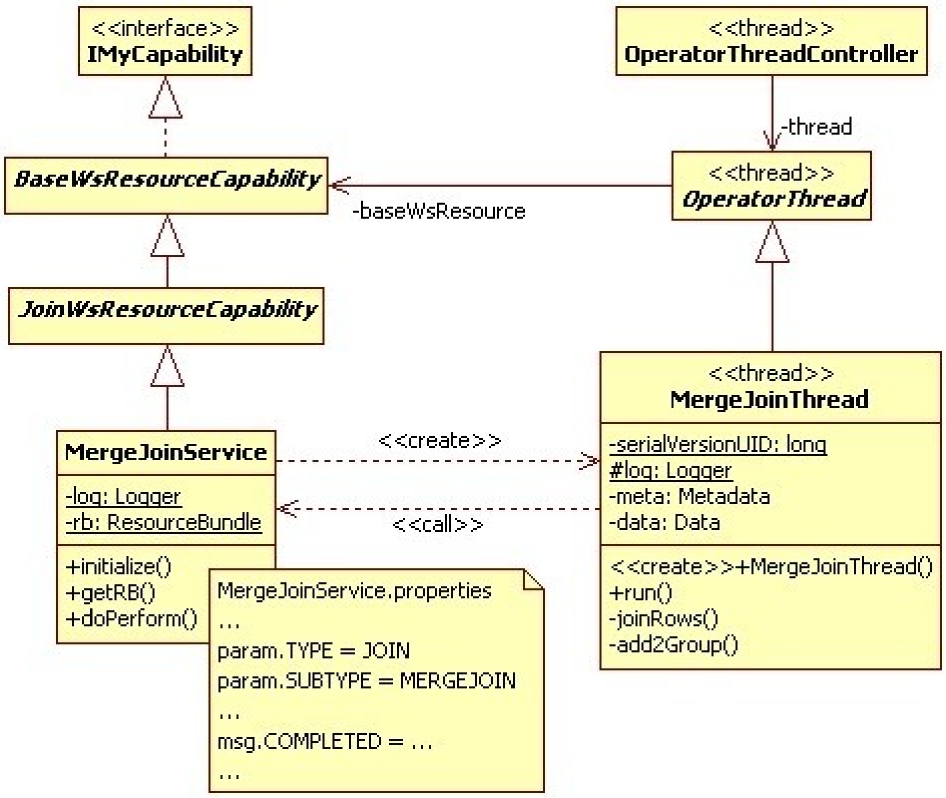


Figure 28: MergeJoin Service Classes

5.2.10 HashJoin Service

The HashJoin Service (Figure 29) implements the hash join algorithm described in section 4.3.

Figure 29: HashJoin Service

See Figure 30 for the class diagram of the HashJoin Service. For a detailed description refer to the thesis of Beran and Habel [4].

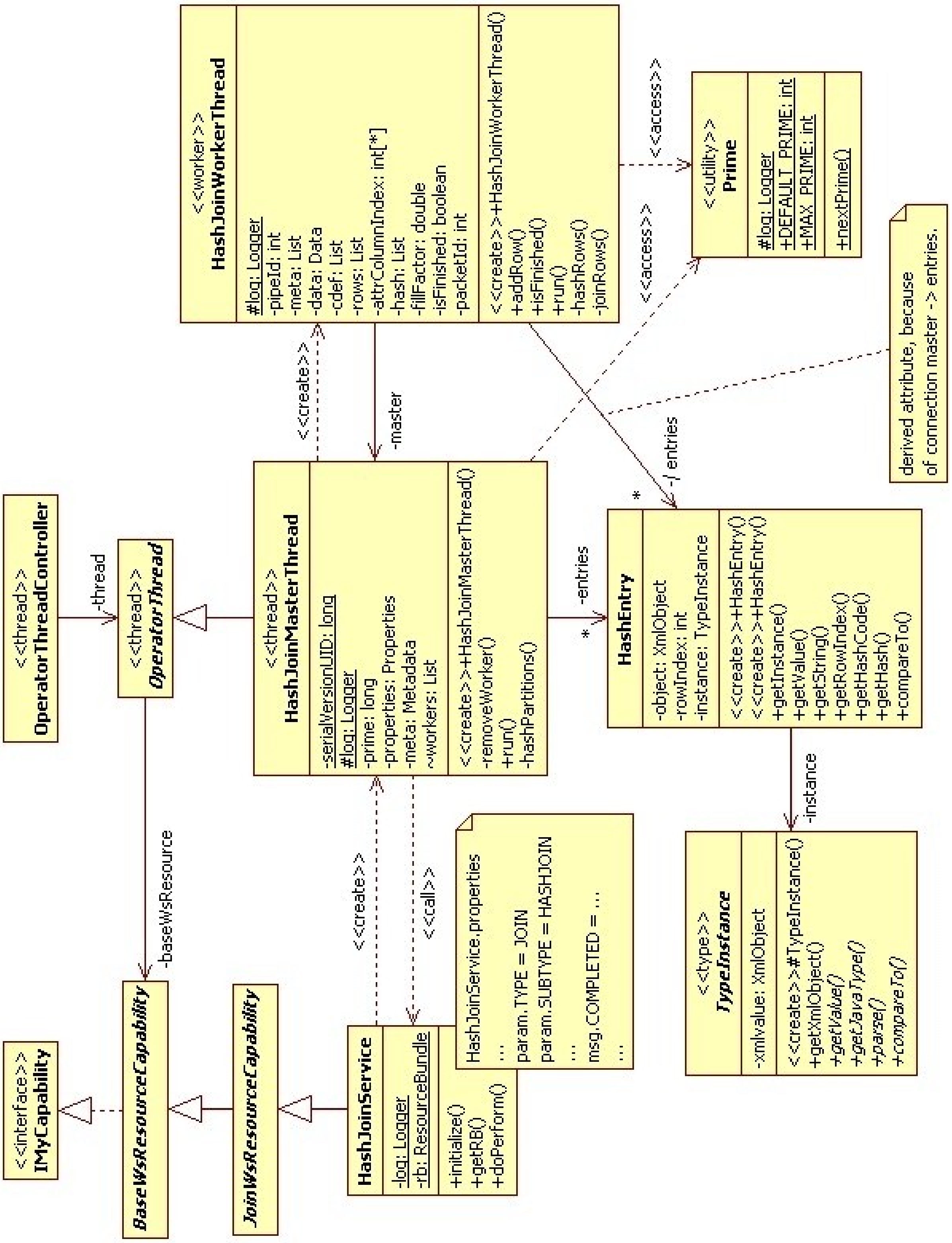


Figure 30: HashJoin Service Classes

5.3 Sort Operation Workflows

The sort algorithms described in section 3 had to be implemented as Web services, which are described in section 4. To uses these services in SODA, workflows had to be implemented, which will be described for the use of two nodes.

5.3.1 Parallel Binary Merge Sort

The theory how the Parallel Binary Merge Sort operates is described in section 3.1. The algorithm is implemented and split into services and an execution tree is build. The four phases of the algorithm named prepare, suboptimal, optimal and post optimal phase are therefore implemented in different services (see Figure 31).

The MultiFetch Service implements the prepare phase, where the data is fetched from a persistent storage and equally distributed to the participating nodes. This service fetches data from a MySQL database and splits the input data stream into two outputs.

The QuickSort Service is required to support the suboptimal phase where the data has to be presorted before the actual merging can take place. All data has to be present for the QuickSort Service to begin its work.

The ParallelBinaryMergeSort Service is the main service which retrieves two sorted input streams and merges them to one sorted output stream. As soon as the first data parts arrive at the two inputs the ParallelBinaryMergeSort Service can begin its work to compare the two streams and merge them to one sorted output stream. Compared to the theory part the service implements the optimal and suboptimal phase.

The MultiSink Service collects the data and stores it to be fetched by a client.

Workflow

The execution tree seen in Figure 31 uses the services explained earlier and is an optimized sample workflow for the execution on two machines.

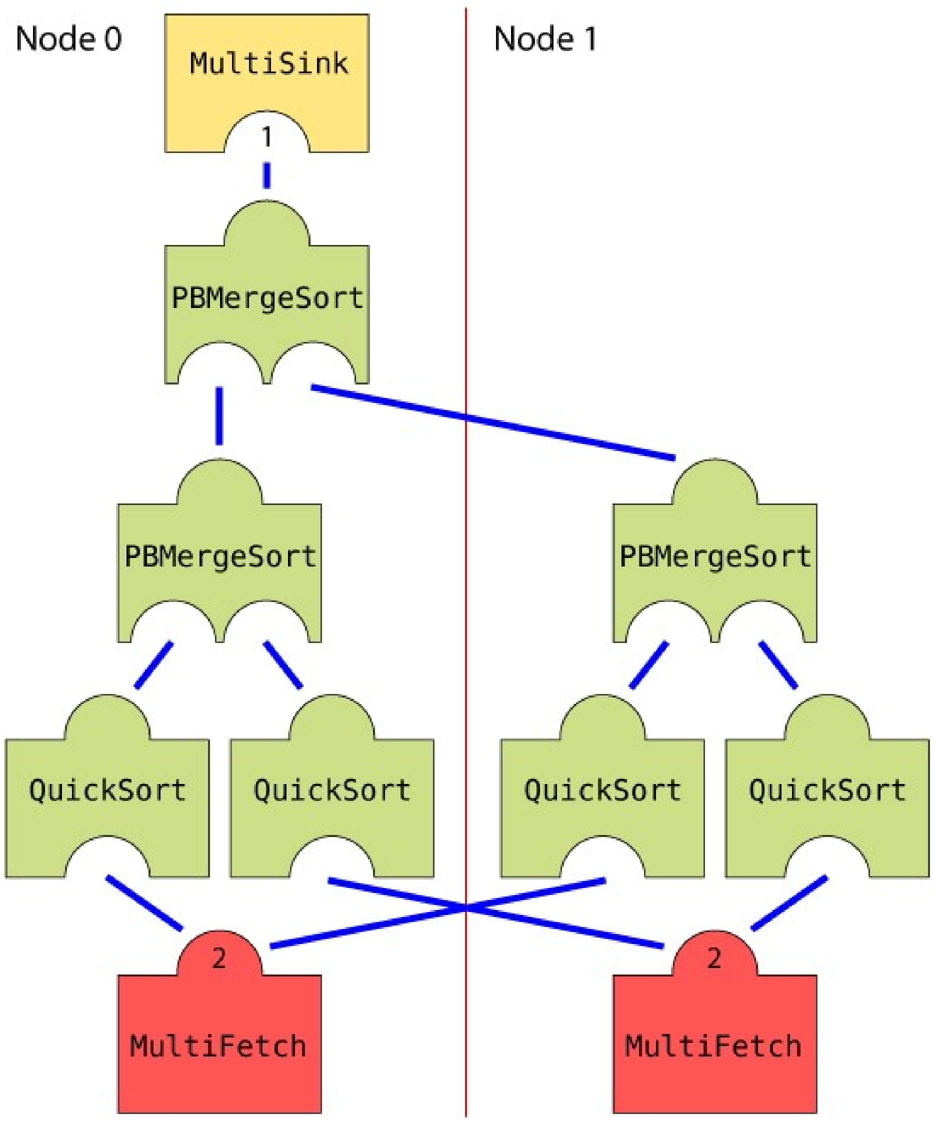


Figure 31: Parallel Binary Merge Sort Query Execution Tree

At the bottom of the tree there are two MultiFetch Services, one on each node. These services collect the data from the MySQL database they are connected to, and split it up into two output stream of the same size. Since there are two services which split their data in halves, one output stream of each service is delivered to the other node. Therefore the data is equally distributed to all participating QuickSort nodes.

In the next step the QuickSort Services fetch the input from the MultiFetch Services and sorts this input data. This step is done in parallel on each node. Because of the assembly of the machines used, which are at least dual core processors, the two QuickSort Services can also run parallel on one node. So the four existing QuickSort Services run in parallel.

After the sorting is done the optimal phase of the Parallel Binary Merge Sort begins by merging the presorted input streams. The ParallelBinaryMergeSort Services can begin with the work when data is available at the two input pipes. When the parallel execution of the ParallelBinaryMergeSort Services is finished the suboptimal phase begins. In this phase only one node can be used to merge the input streams, but as mentioned before it can begin with its work when there is data at the two input lanes available.

The MultiSink Service is used to collect the data and store it for the client.

5.3.2 Bitonic Sort

How the Bitonic Sort algorithm works in theory is described in section 3.2. The algorithm is implemented and split into services and an execution tree is built (see Figure 32).

The MultiFetch Service fetches data from a MySQL database and splits its input data stream into two output data streams.

The BitonicSort Service retrieves the data from the MultiFetch Service. According to its *MASK* information (the *MASK* can be -1, 0 or 1), it decides if it should sort the input (in case of 0 and 1) and then distribute it, or leaves the data untouched and passes it to the next service (in case of -1).

The MultiSink service collects the data from the last BitonicSort Services and merges them in the right order.

Workflow

The execution tree seen in Figure 32 uses the services explained earlier and is an optimized sample workflow for the execution on two machines.

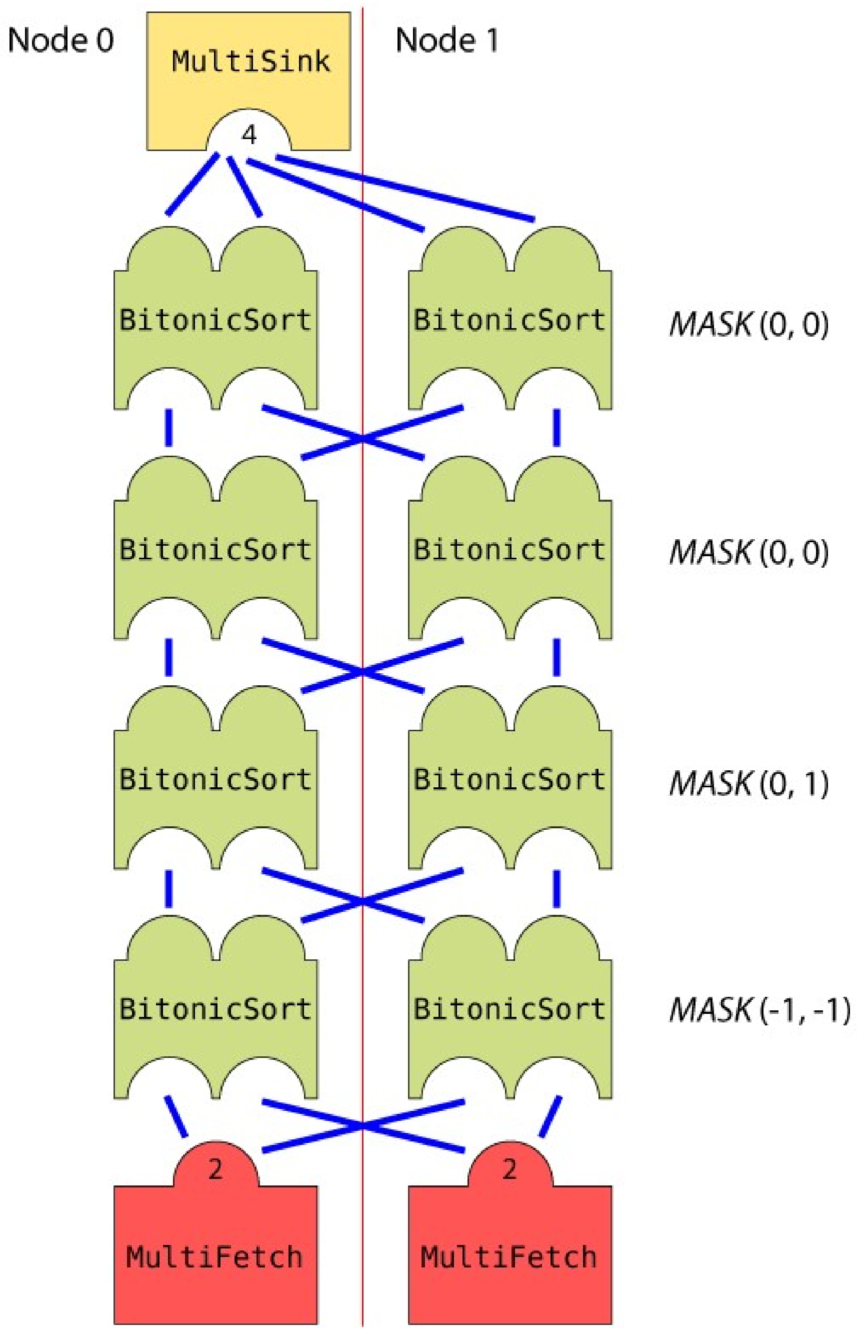


Figure 32: Bitonic Sort Query Execution Tree

At the bottom of the tree there are two MultiFetch Services, one on each node. These services collect the data from the MySQL database they are connected to, and split it up into two output stream of the same size. Since there are two services which split their data in halves, one output stream of each service is delivered to the other node. Therefore the data is equally distributed to all participating BitonicSort nodes.

In the next step the BitonicSort Services begin with their work. Starting with a MASK of (-1, -1) the two services pass the result to the next service stage. The next service on the right path has 1 as *MASK* information so it sorts the input, splits it and passes the lower part to the right output pipeline and the higher part to the left output pipeline. All remaining services have 0 as there *MASK* information, so they sort the input, split it up and pass the lower part to the left output execution path and the higher part to the right output execution path.

The MultiSink Service collects the data from the last BitonicSort Services. It has to merge the input pipelines from index 0 to 3 in the exact order because otherwise the sorting would not be correct.

5.4 Join Operation Workflows

For the join algorithms described in section 4 to work in SODA, workflows that consist of services had to be implemented. All workflows will be described for the use of two nodes.

5.4.1 Nested Loop Join

How the nested loop join algorithm works in theory is described in section 4.1. The algorithm is implemented and split into services and an execution tree is built (see Figure 33).

The MultiFetch Services fetches data from a MySQL database.

The NestedLoopJoin Service retrieves the data of relation R and T from the MultiFetch Services and performs the join on them together.

The MultiSink Service collects the data from both of the NestedLoopJoin Services and merges them together.

Workflow

The execution tree seen in Figure 33 uses the services explained earlier and is an optimized sample workflow for the execution on two machines.

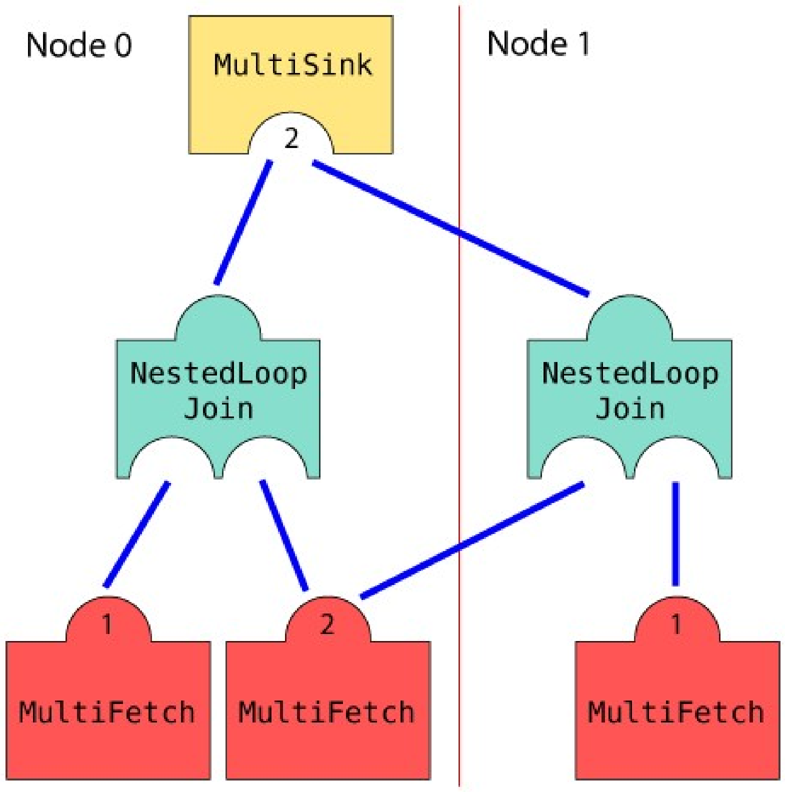


Figure 33: Nested Loop Join Query Execution Tree

At the bottom of the tree there are two MultiFetch Services on the left side and one on the right side. The MultiFetch with two outputs fetches relation *R* which is smaller than *T* and splits it into two parts. The other two MultiFetch Services fetch the same relation *T* from local MySQL databases on each node.

Each of the NestedLoopJoin Services retrieves the half of the relation *R* and relation *T* as input. Since *T* is smaller it is used as the inner loop assignment and is sequentially read.

The MultiSink Service collects the data from the two NestedLoopJoin Services and merges the input pipelines.

5.4.2 Sort Merge Join with Parallel Binary Merge Sort

How the sort merge join algorithm works in theory is described in section 4.2. The algorithm is implemented and split into services and an execution tree is built (see Figure 34).

The MultiFetch Services fetches data from a MySQL database and spits it into two output pipelines.

The QuickSort Services sort the relations on the join attribute and distribute them to the ParallelBinaryMergeSort Service.

The ParallelBinaryMergeSort Service sorts the input streams on the join attribute.

The MergeJoin Service joins the presorted inputs on the join attribute.

The MultiSink Service collects the data from the MergeJoin Service.

Workflow

The execution tree seen in Figure 34 uses the services explained earlier and is an optimized sample workflow for the execution on two machines.

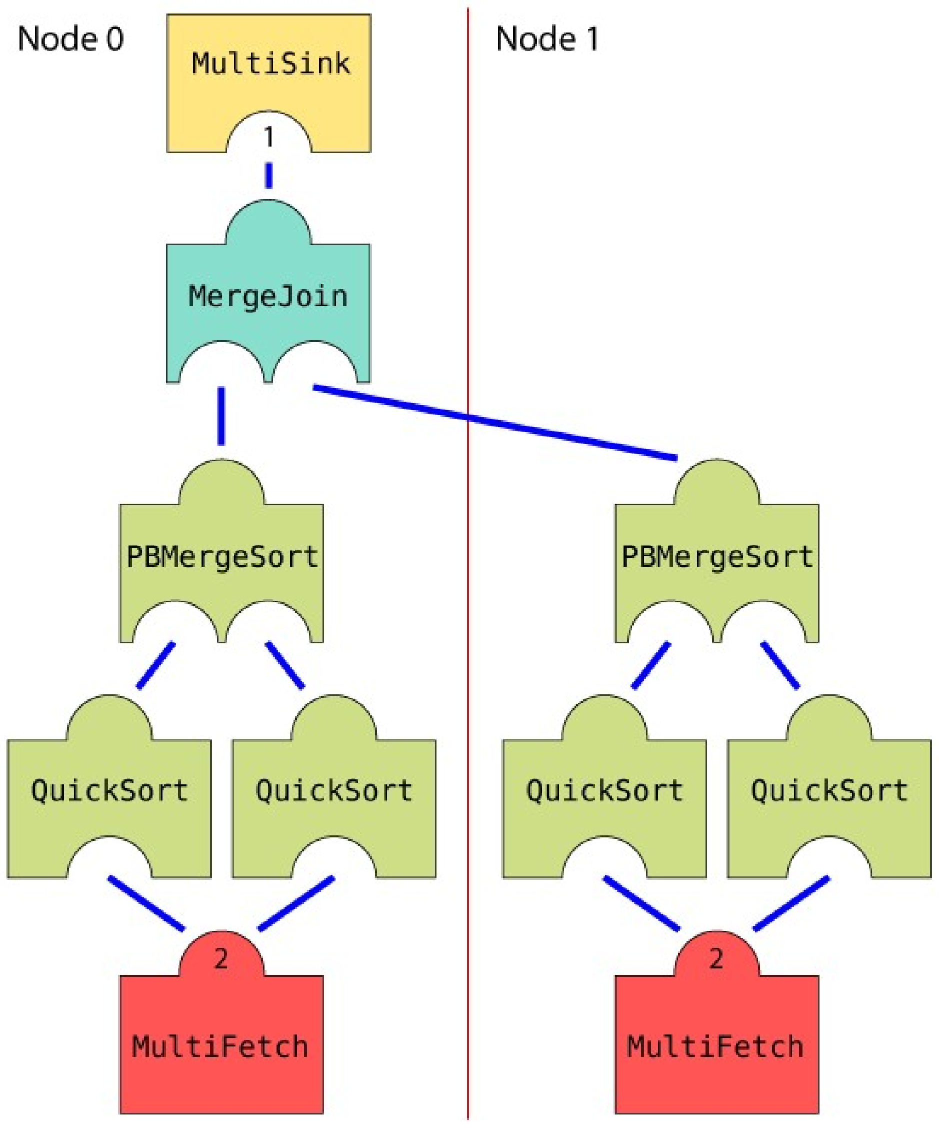


Figure 34: Sort Merge Join with Merge Sort Query Execution Tree

At the bottom of the tree there are two MultiFetch Services that fetch relation *R* and *T* and split it into two parts.

The QuickSort Services on both sides sort the two relations on the join attribute as a preparation for the following ParallelBinaryMergeSort Service.

The ParallelBinaryMergeSort Service then merges and sorts the two input streams from the QuickSort Service.

The MergeJoin Service retrieves the sorted relations *R* and *T* as input and joins them to the output relation.

The MultiSink Service collects the data from the MergeJoin Service.

5.4.3 Sort Merge Join with Bitonic Sort

How the Sort Merge Join algorithm works in theory is described in section 4.2. The algorithm is implemented and split into services and an execution tree is built (see Figure 35).

The MultiFetch Services fetches data from a MySQL database and spits it into four output streams.

The BitonicSort Service retrieves the data from the MultiFetch Service. It decides according to the *MASK* information (the *MASK* can be -1, 0 or 1), if it should sort the input (in case of 0 and 1) and then distribute it or, if the *MASK* is -1, it leaves the data untouched and passes it to the next service.

The MultiRelay Service collects the data from the last BitonicSort Services, merges it in the right order and distributes the result to the MergeJoin Service.

The MergeJoin Service joins the presorted inputs on the join attribute.

The MultiSink Service collects the data from the MergeJoin Service.

Workflow

The execution tree seen in Figure 35 uses the services explained earlier and is an optimized sample workflow for the execution on two machines.

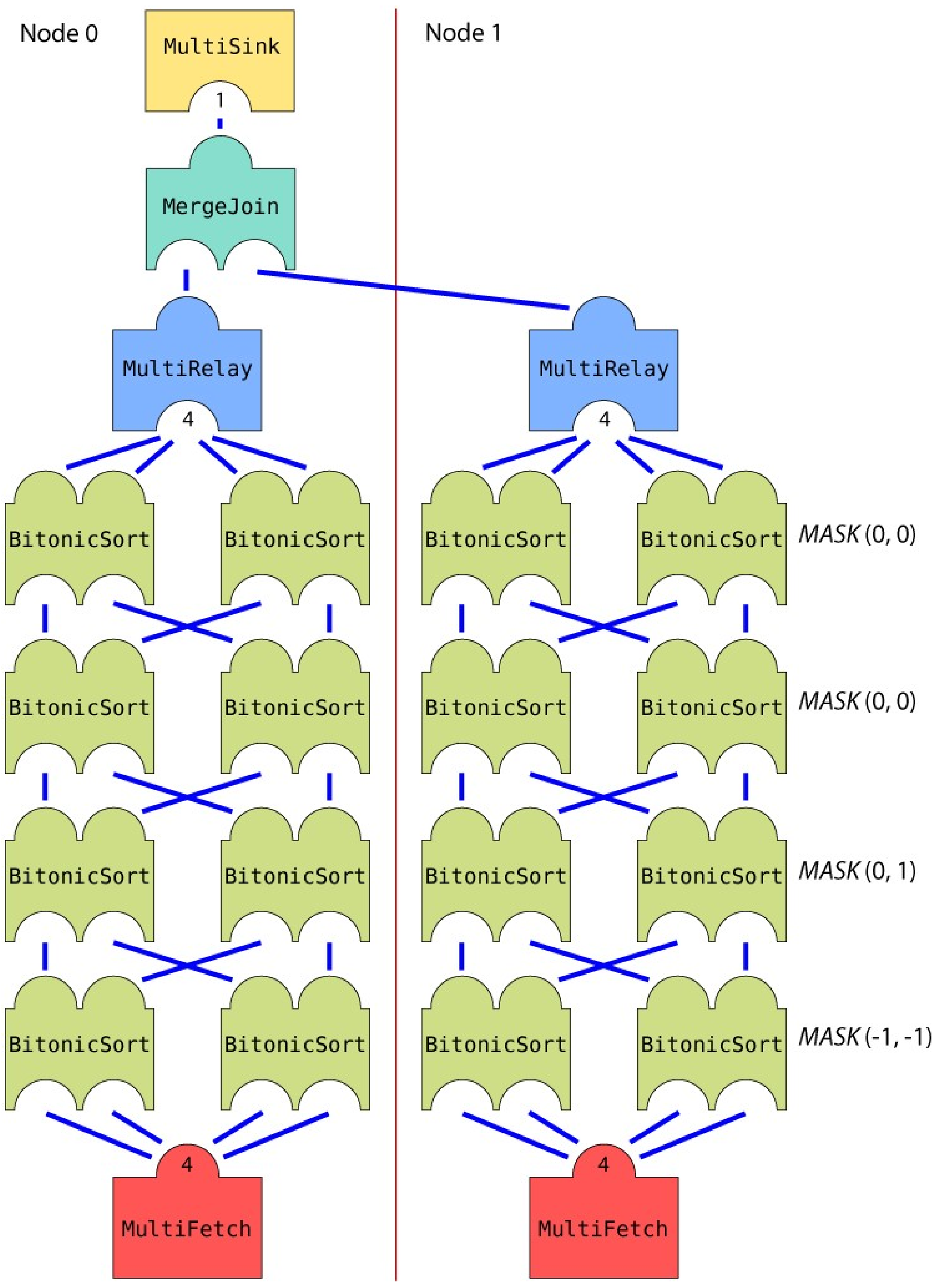


Figure 35: Sort Merge Join with Bitonic Sort Query Execution Tree

At the bottom of the tree there are two MultiFetch Services that fetch relation *R* and *T* and split it into four parts.

In the next step the BitonicSort Services begin with their work. Starting with a MASK of (-1, -1) the two services pass the result to the next service stage. The next service on the right path has 1 as *MASK* information, so it sorts the input, splits it and passes the lower part to the right output pipeline and the higher part to the left output pipeline. All remaining services have 0 as there *MASK* information, so they sort the input, split it up and pass the lower part to the left output execution path and the higher part to the right output execution path.

The MultiRelay Services collect the data from the last BitonicSort Services. They have to merge the input pipelines from index 0 to 3 in the exact order because otherwise the sorting would not be correct. After that it distributes the output to the MergeJoin Service.

The MergeJoin service retrieves the sorted relations *R* and *T* as input and joins them to the output relation.

The MultiSink Service collects the data from the MergeJoin Service.

5.4.4 Hash Join

How the Hash Join algorithm works in theory is described in section 4.3. The algorithm is implemented and split into services and an execution tree is built (see Figure 36).

The MultiFetch Services fetches data from a MySQL database.

The HashJoin service retrieves the data of relation *R* and *T* from the MultiFetch Services and performs the join on them.

The MultiSink Service collects the data from both of the HashJoin Services and merges them.

Workflow

The execution tree seen in Figure 36 uses the services explained earlier and is an optimized sample workflow for the execution on two machines.

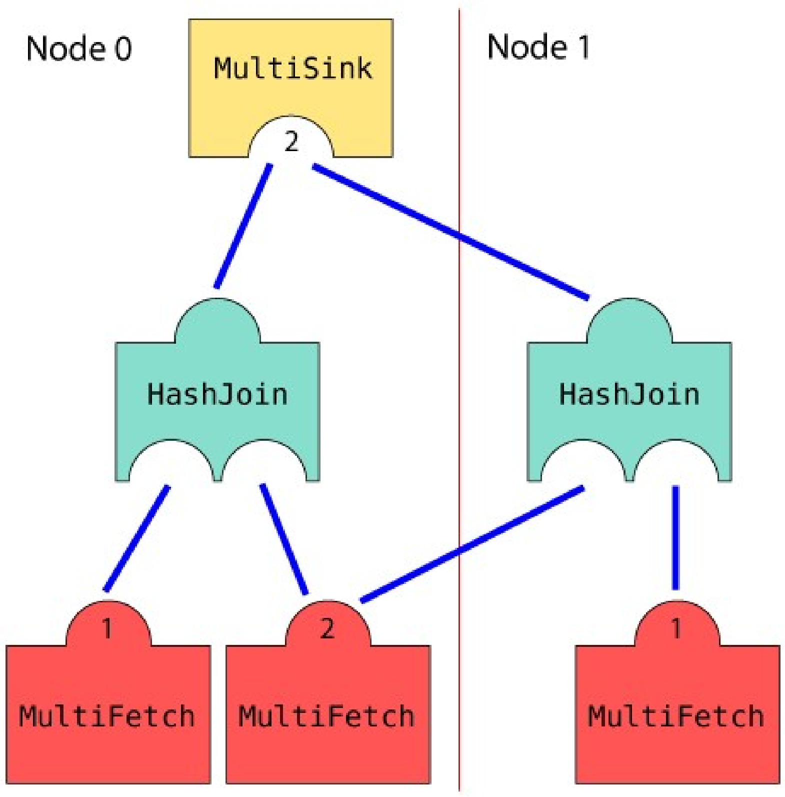


Figure 36: Hash Join Query Execution Tree

At the bottom of the tree there are two MultiFetch Services on the left side and one on the right side. The MultiFetch Service with two outputs fetches relation *R* which is smaller than *T* and splits it into two parts. The other two MultiFetch Services fetch the same relation *T* from local MySQL databases on each node.

Each of the HashJoin Services retrieves the half of the relation *R* and relation *T* as input. Since *T* is smaller it is used as the prehashed relation and is sequentially read.

The MultiSink Service collects the data from the two HashJoin Services and merges the input pipelines.

6. Benchmark

For benchmarking, database relations have to be produced. The Transaction Processing Performance Council, also known as TPC [14], provides a suite of benchmarks to test databases according their performance, measured in Transactions/Time interval. Because of the general purpose the TPC-H benchmark is designed, it was used as basis for the testing. The TPC-H tools consist of a database generator, which creates the SQL statements for inserting the tuples, and the query generator for testing the performance of the database. The overall database layout is illustrated in Figure 37.

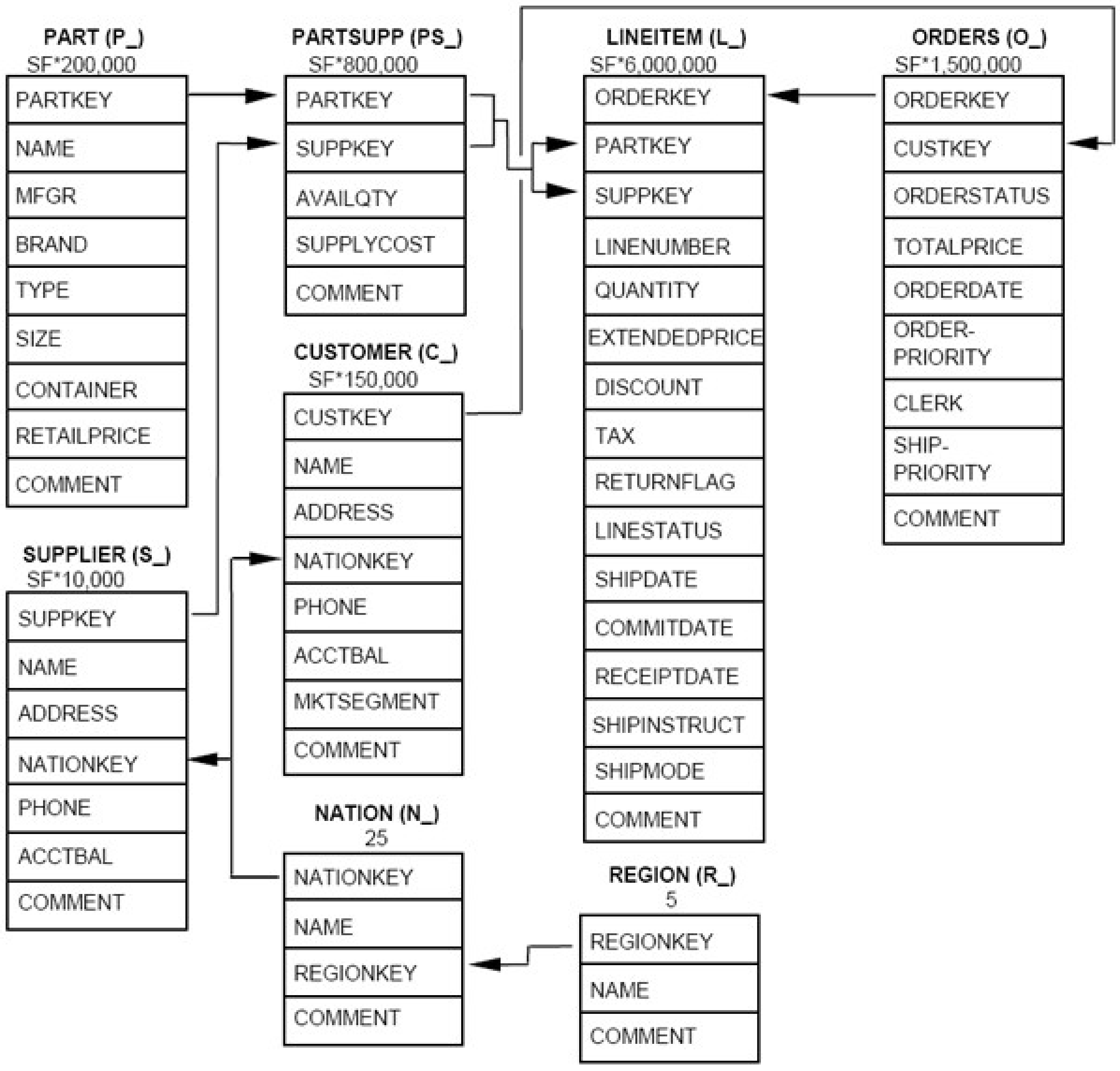


Figure 37: TPC-H Database Shema

In the case of testing the SODA operators only sorts and joins are used, therefore the database layout and the benchmark queries were adapted to their needs. Hence only the *PART* and

PARTSUPP tables are used. The sort operators query the *PART* table and the join operators join the *PART* with the *PARTSUPP* table.

For the normal TPH-H benchmark [14] a scale factor (SF) of 1 means that there is 1GB of data in all tables. Because only two of the tables are used, the SF has changed. Table 1 shows how the data is distributed among the tables.

Table Name	**Cardinality (in rows)**	**Length (in bytes) of Typical[1] Row**	**Typical[1] Table Size (in MB)**
SUPPLIER	10.000	15	2
PART	200.000	155	30
PARTSUPP	800.000	144	110
CUSTOMER	150.000	179	26
ORDERS	1.500.000	104	149
LINEITEM[3]	6.001.215	112	641
NATION[1]	25	128	< 1
REGION[1]	5	124	< 1
Total	**8.661.245**		**956**

1 Typical lengths and sizes given here are examples, not requirements, of what could result from an implementation (sizes do not include storage/access overheads).
2 The cardinality of LINEITEM table is not a strict multiple of SF since the number of lineitems in an order is chosen at random with an average of four.
3 Fixed cardinality: does not scale with SF.

Table 1: Estimated Database Size

With the data from Table 1 as fundament the new SF, if just the *PART* and *PARTSUPP* tables are used, will be approximately 7,6 for 1GB of data.

Some sample rows from the *PART* table can be seen in Table 2.

PARTKEY	NAME	MFGR	BRAND	TYPE	SIZE	CONTAINER	RETAILPRICE	COMMENT
1	goldenrod lace spring peru powder	Manufacturer#1	Brand#13	PROMO BURNISHED COPPER	7	JUMBO PKG	901.00	ly. slyly ironi
2	blush rosy metallic lemon navajo	Manufacturer#1	Brand#13	LARGE BRUSHED BRASS	1	LG CASE	902.00	lar accounts amo
3	dark green antique puff wheat	Manufacturer#4	Brand#32	STANDARD POLISHED BRASS	21	WRAP CASE	903.00	egular deposits hag
4	chocolate metallic smoke ghost drab	Manufacturer#3	Brand#42	SMALL PLATED BRASS	14	MED DRUM	904.00	p furiously r
5	forest blush chiffon thistle chocolate	Manufacturer#3	Brand#34	STANDARD POLISHED TIN	15	SM PKG	905.00	wake carefully

Table 2: Sample PART Table

Some sample rows from the *PARTSUPP* table can be seen in Table 3.

PARTKEY	SUPPKEY	AVAILQTY	SUPPLYCOST	COMMENT
1	2	3325	771.64	, even theodolites. regular, final theodolites eat after the carefully pending foxes. furiously regular deposits sleep slyly. carefully bold realms above the ironic dependencies haggle careful
1	2502	8076	993.49	ven ideas. quickly even packages print. pending multipliers must have to are fluff
1	5002	3956	337.09	after the fluffily ironic deposits? blithely special dependencies integrate furiously even excuses. blithely silent theodolites could have to haggle pending, express requests; fu
2	3	8895	378.49	nic accounts. final accounts sleep furiously about the ironic, bold packages. regular, regular accounts
2	2503	4969	915.27	ptotes. quickly pending dependencies integrate furiously. fluffily ironic ideas impress blithely above the express accounts. furiously even epitaphs need to wak
2	5003	8539	438.37	blithely bold ideas. furiously stealthy packages sleep fluffily. slyly special deposits snooze furiously carefully regular accounts. regular deposits according to the accounts nag carefully slyl

Table 3: Sample PARTSUPP Table

6.1 Environment

The environment was predefined by the infrastructure and the circumstances of the knowledge and business engineering institute. There are three servers to run the tests and to install the necessary software.

To simulate the different network speeds in a heterogeneous environment the test nodes have to be configured with a limited bandwidth. Class-Based Queueing [15] delivers this demand. A class for the network device has to be configured which can limit the throughput of the network device. To ease the configuration and the specially to manage more configurations the `cbq.init` script [16] is used.

6.1.1 Servers

Processor	Name	Operating System	RAM	HDD
Single Dual Xenon (@ 2.0 GHz)	chachacha1	Scientific Linux 5.0	3 GB	260 GB
Double Dual Xenon (@ 2.0 GHz)	chachacha2	Scientific Linux 5.0	3 GB	260 GB
Single Quad Xenon (@ 2.0 GHz)	chachacha3	Scientific Linux 5.0	3 GB	260 GB

Table 4: List of Servers

A virtualisation software is used because of the flexibility it provides in setting up different environments.

The first step was to install VMware Server 1.0.4 [18] on the three machines. The program installs a network bride so that the guest operating system can access the outside world. There are four options a guest can access the network. The bridged networking, network address translations (NAT), host-only networking and no network connection.

Bridged

Bridged Network enforces the guest to directly access the network. The guest needs to have an own IP-address and can be contacted by other devices on the network.

NAT

Network Address Translation enables the guest to access the hosts dial-up or external network connection. The guest cannot be contacted by the outside world. The guest does not get an IP-address on the external network.

Host-only

The guest is connected to the host's virtual network. This is used if the guest is not allowed to access the external network.

For our purpose bridged networking is the best choice, because the nodes need to communicate a lot and have to be accessible from the outside.

6.2 Analysis

For all six algorithms a speedup and a scale-up analysis, as described by DeWitt and Gray in [17], has been performed.

Speedup describes the effect of processing time by adding nodes to the system. Therefore the speedup is defined by the ratio of the time used to run a job, with a fixed number of tuples, on a small system and the time used to run a job, with the same amount of tuples, on a larger system.

This can be described by the following formula:

$$speedup = \frac{Small\_System\_Elapsed\_Time}{Big\_System\_Elapsed\_Time}$$

Scale-up explains the scalability of an algorithm. It determines if an increasing problem size can be compensated by increasing resources like adding nodes.

This behaviour can be expressed with the following formula:

$$scale - up = \frac{Small\_System\_Elapsed\_Time\_On\_Small\_Probe}{Big\_System\_Elapsed\_Time\_On\_Big\_Probe}$$

Moreover every algorithm has been analyzed with a naive way of distributing the services in the heterogeneous environment, in which every node has a different network bandwidth, and using a smart way of distribution. This distribution is mainly based on the analysis of each algorithm and the amount of data it has to send and receive. Therefore services with higher demand on data are placed on nodes with higher network bandwidth, but there is also the processing power and the I/O performance of a node which is taken into account. The smart way will be called “modified” in the corresponding text and figures. For this analysis all nodes have the same processing power and I/O performance.

6.2.1 Parallel Binary Merge Sort – speedup

The Parallel Binary Merge Sort speedup analysis in Figure 38 shows that the speedup is - as expected - good but it could be even better. The modified workflow performs better because it uses a workflow that takes advantage of the heterogeneous environment in a way, that a data intensive node which has to send and receive more data than other nodes possesses a higher network bandwidth. The speed for one node is the same because no network is used and the algorithm is the same, therefore the speed has to be as well the same.

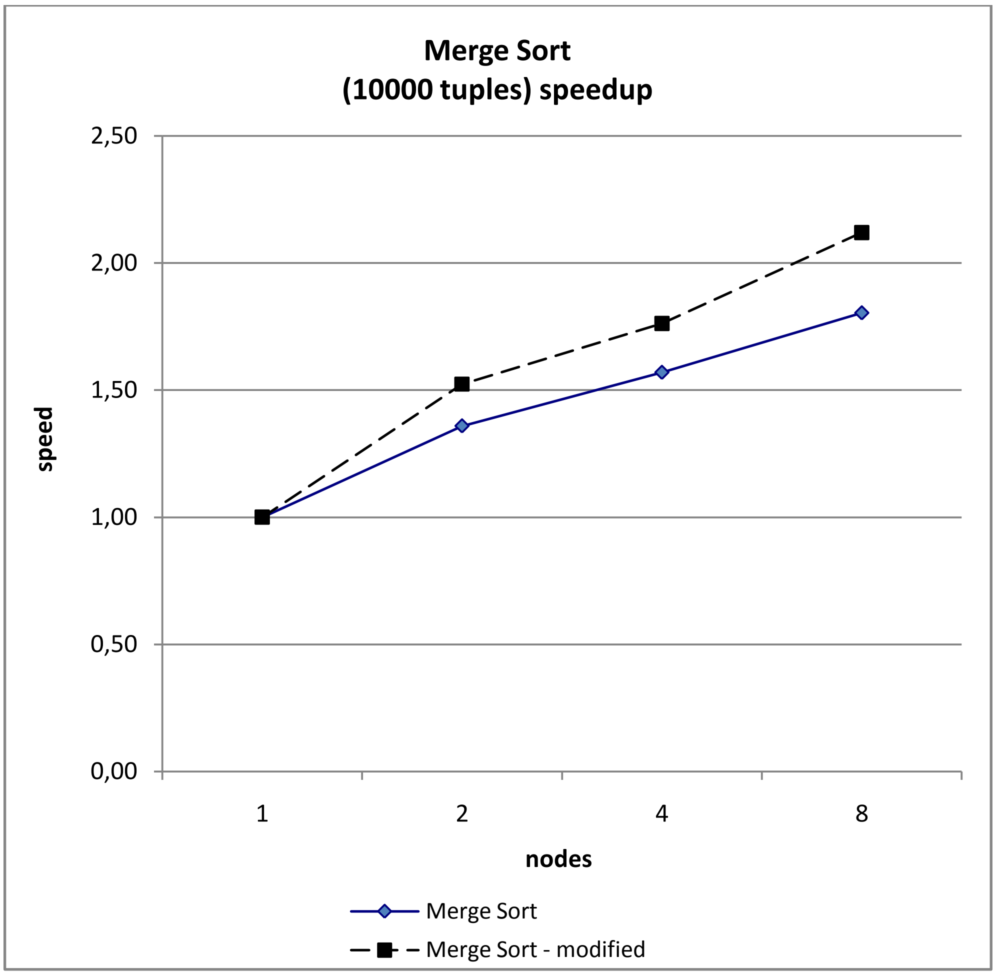


Figure 38: Parallel Binary Merge Sort – speedup

6.2.2 Parallel Binary Merge Sort – scale-up

The Parallel Binary Merge Sort scale-up analysis in Figure 39 shows that the scale-up for both workflows is more or less the same, but the modified workflow is a little bit faster. The modified workflow performs better because it uses a workflow that takes advantage of the heterogeneous environment in a way, that a data intensive node which has to send and receive more data than other nodes possesses a better network bandwidth.

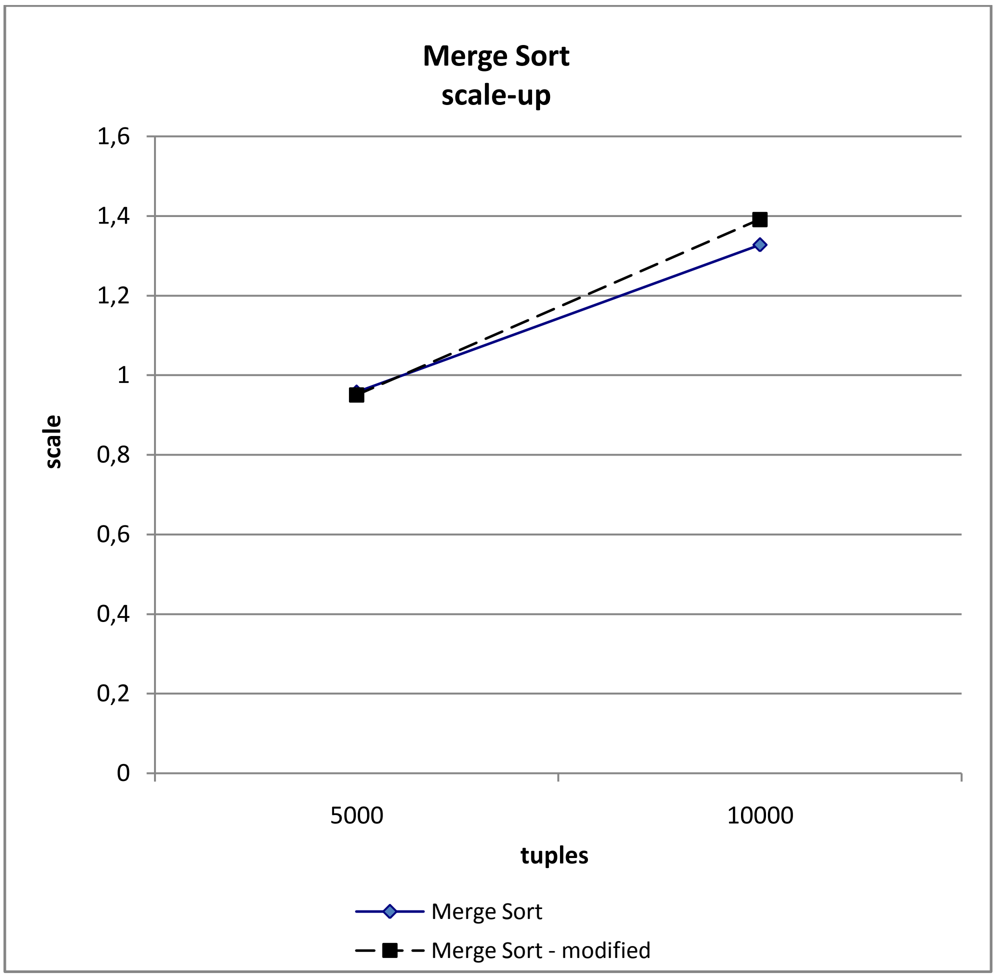


Figure 39: Parallel Binary Merge Sort – scale-up

6.2.3 Block Bitonic Sort – speedup

The Block Bitonic Sort speedup analysis in Figure 40 shows that the modified workflow performs better but all two are even worse. This is because the Block Bitonic Sort uses eight services with two nodes, 64 services with four nodes and 512 services with eight nodes, therefore the network cost is far too high to increase the overall performance. From one to two nodes a speedup can be identified, this is because one node uses eight services like two nodes. Therefore the processing power will be doubled, but on the other side the network slows this down again, therefore a speedup of about 1,68 can be achieved. With the naive workflow the network is taken much more into account so the speedup is only about 1,18.

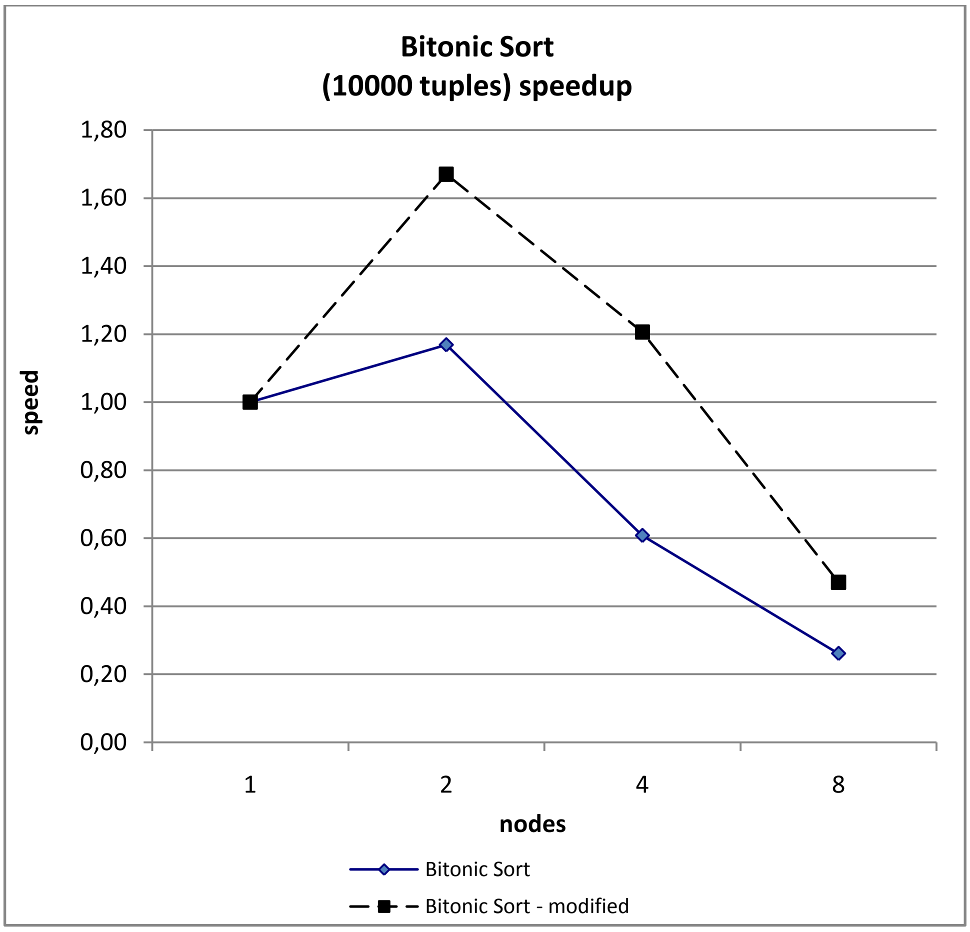


Figure 40: Block Bitonic Sort – speedup

6.2.4 Block Bitonic Sort – scale-up

Taken into account that the speedup of the Bock Bitonic Sort is really poor it is no surprise that the scale-up shown in Figure 41 is also really bad. But here the modified workflow is also a little bit more performing.

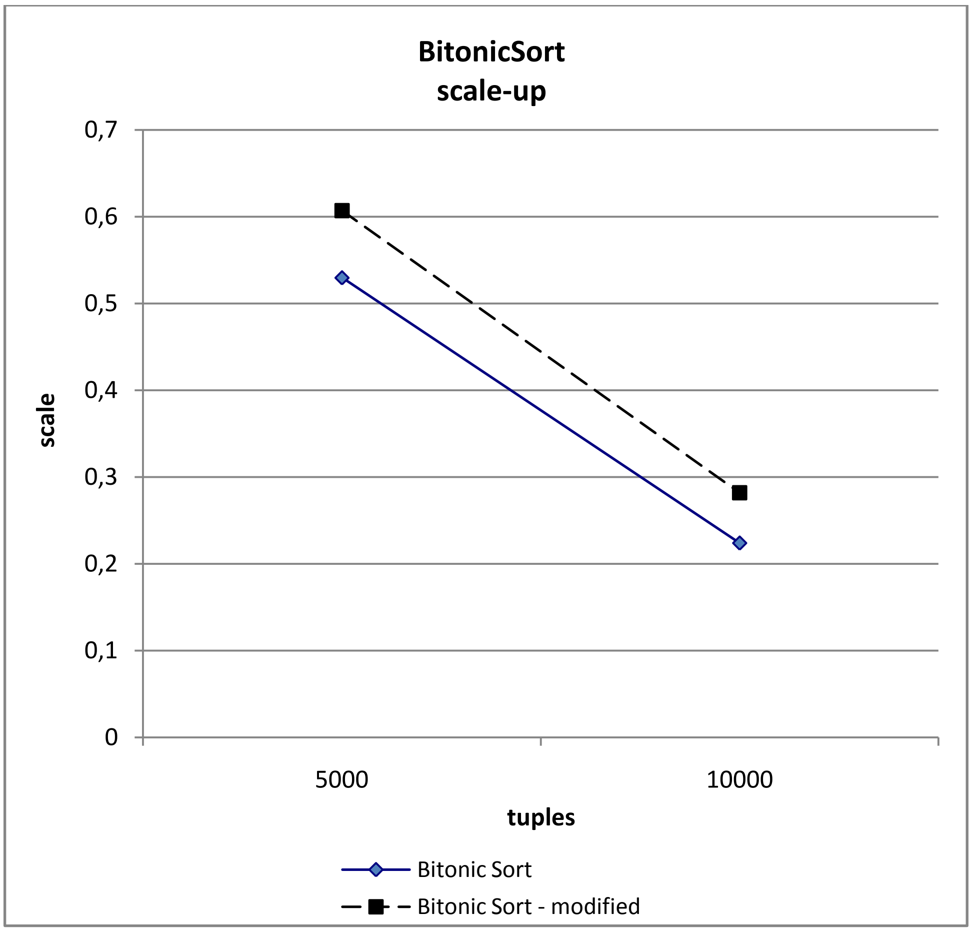


Figure 41: Block Bitonic Sort – scale-up

6.2.5 Nested Loop Join – speedup

The Nested Loop Join speedup analysis shown in Figure 42 points out that the speedup is very good for two and four nodes, but later on (using eight nodes) the speedup breaks down. This excellent speedup, which is unexpected in theory, could resemble from the fact that the nodes performing the smaller relation have to wait until the nodes which are processing the larger relation are finished. With more nodes processing the larger relation the overall time gets significantly lesser. Again the modified workflow is by far better and takes advantage of the distribution of services, also the naive workflow is not bad at all.

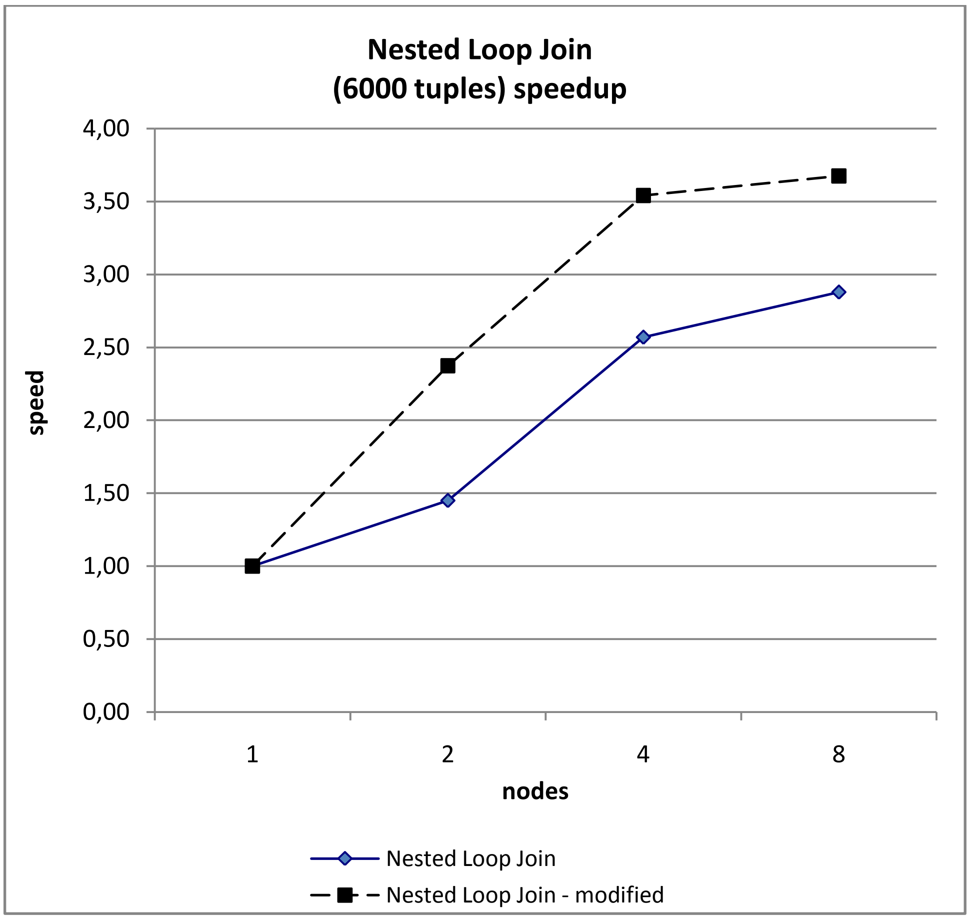


Figure 42: Nested Loop Join – speedup

6.2.6 Nested Loop Join – scale-up

The Nested Loop Join scale-up analysis in Figure 43 shows that the scale-up for both workflows is not bad at all, but the naive workflow is a bit faster, which is a unexpected because the speedup is better for the modified workflow. This could come from the fact that the speedup from four to eight nodes is better in the naïve workflow. The modified workflow performs better because it uses a workflow that takes advantage of the heterogeneous environment in a way, that a data intensive node which has to send and receive more data than other nodes possesses a better network bandwidth.

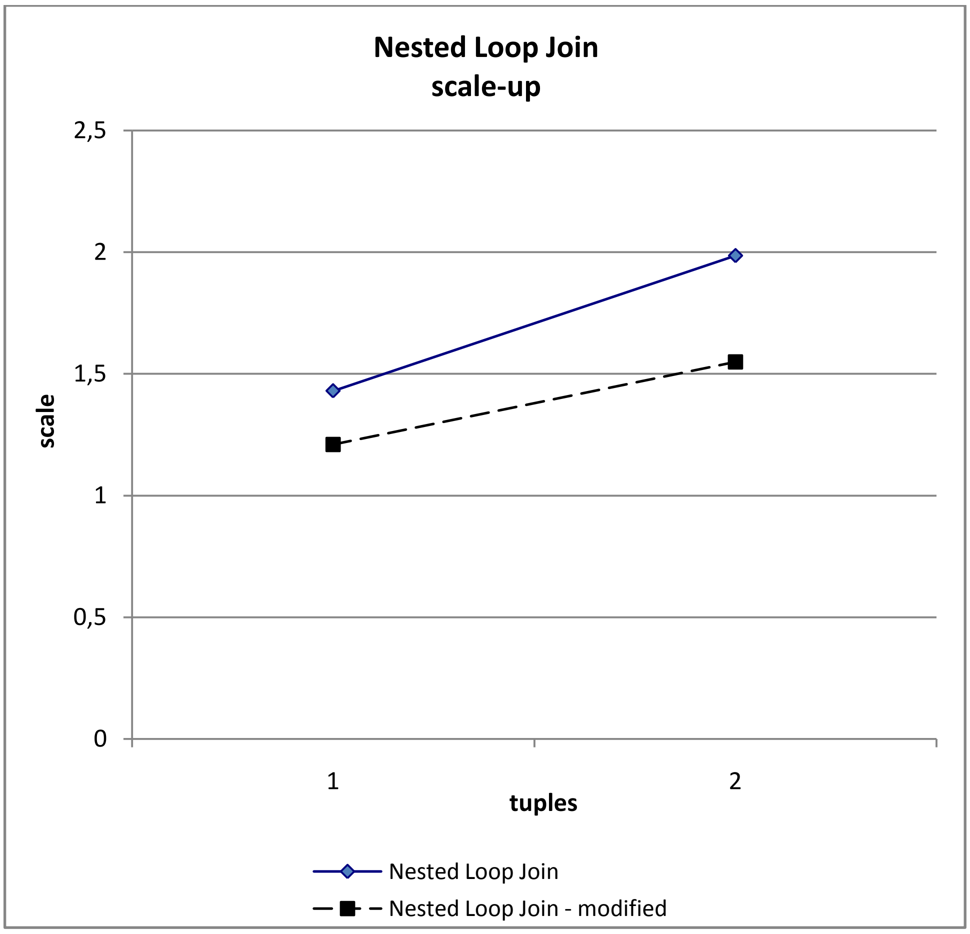


Figure 43: Nested Loop Join – scale-up

6.2.7 Merge Join with Parallel Binary Merge Sort – speedup

The Merge Join with Parallel Binary Merge Sort speedup analysis shown in Figure 44 reveals that the modified workflow is better than the naive one. The nearly same speed of one node compared to two nodes is not surprising, because the service count, the workflow execution tree of both workflows and the relation sizes that have to be processed are equal. Regarding to the fact that the network costs are not present at one node the shape of this curve can be explained.

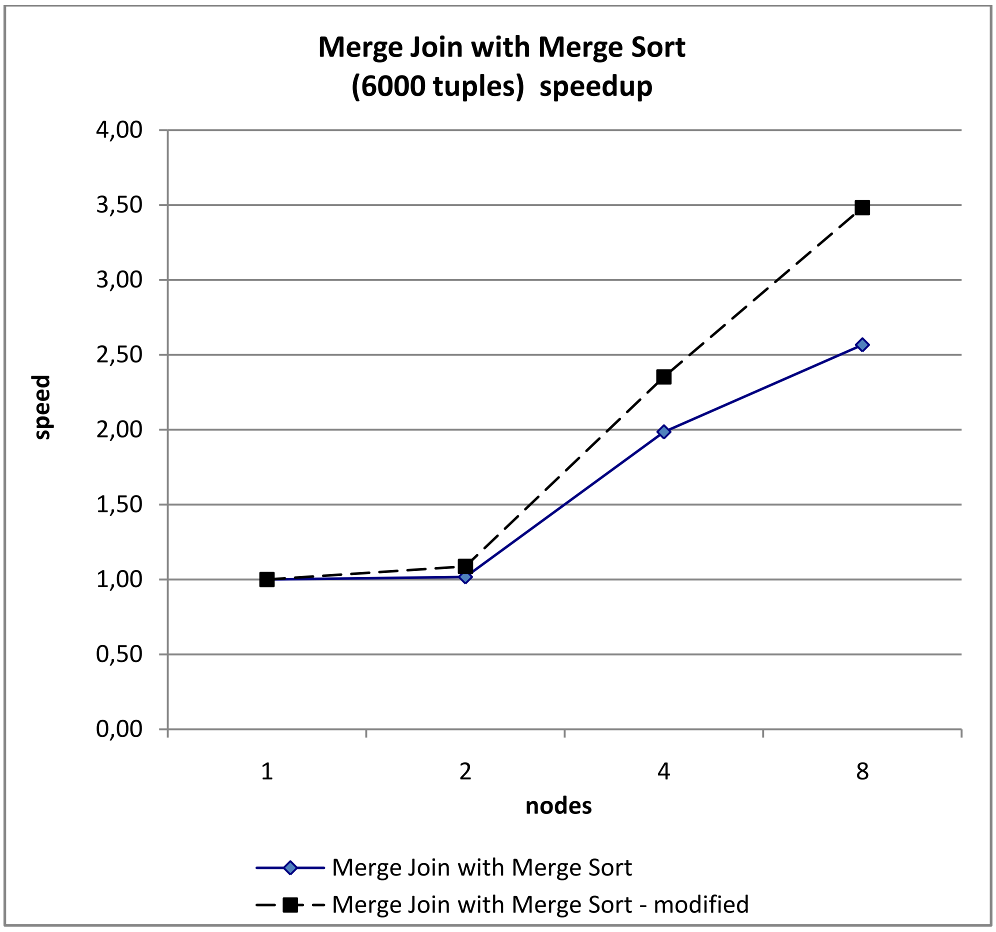


Figure 44: Merge Join with Merge Sort – speedup

6.2.8 Merge Join with Parallel Binary Merge Sort – scale-up

The Merge Join with Parallel Binary Merge Sort scale-up analysis shown in Figure 45 pointed out that the scale-up for both workflows is good, but the modified workflow is a little bit better. The modified workflow performs better because it uses a workflow that takes advantage of the heterogeneous environment in a way, that a data intensive node which has to send and receive more data than other nodes possesses a better network bandwidth.

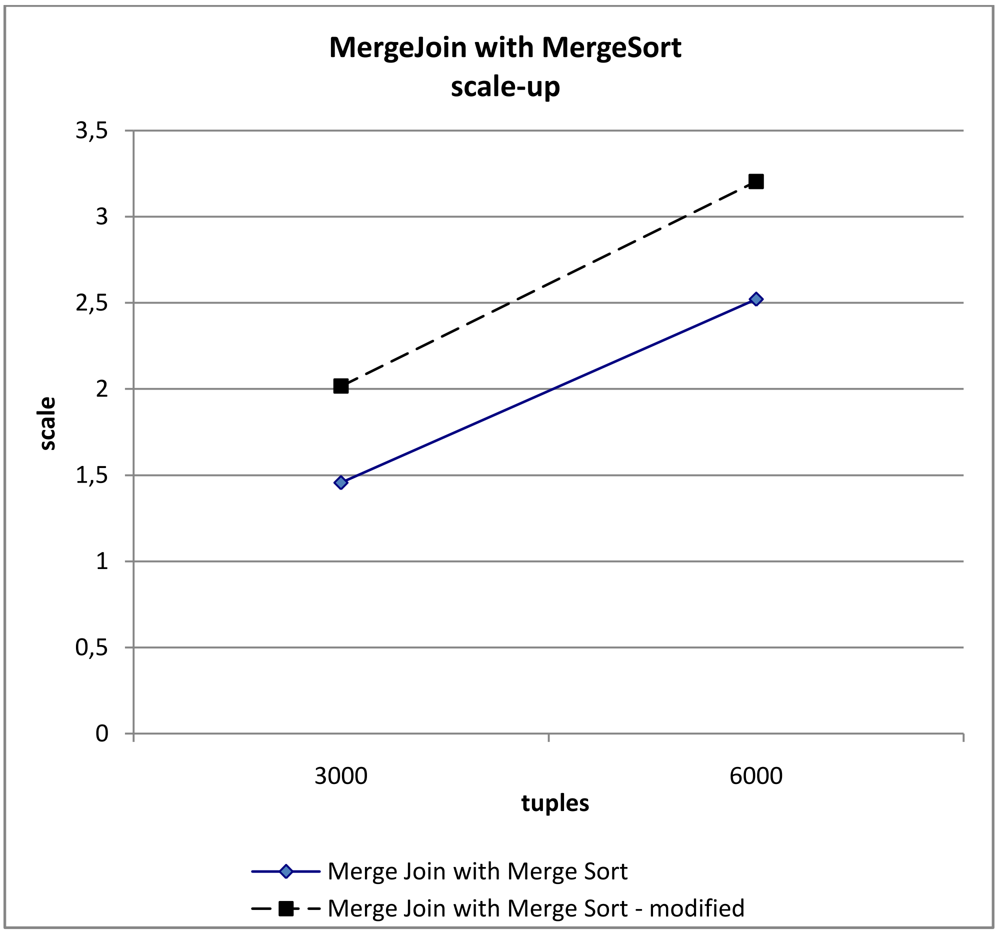


Figure 45: Merge Join with Merge Sort – scale-up

6.2.9 Merge Join with Block Bitonic Sort – speedup

The Merge Join with Block Bitonic Sort speedup analysis shown in Figure 46 illustrates that the speedup is not as good as expected. For the modified workflow the speed is a bit faster which uses a workflow that takes advantage of the heterogeneous environment in a way that a node which has to send and receive more data has a better network bandwidth. This workflow uses 16 services for one, two and four nodes and 128 services for eight nodes so it is a little surprise that the speedup of the modified workflow is that good. The worse speedup of the naive workflow can be explained by the higher network usage and the resulting delays.

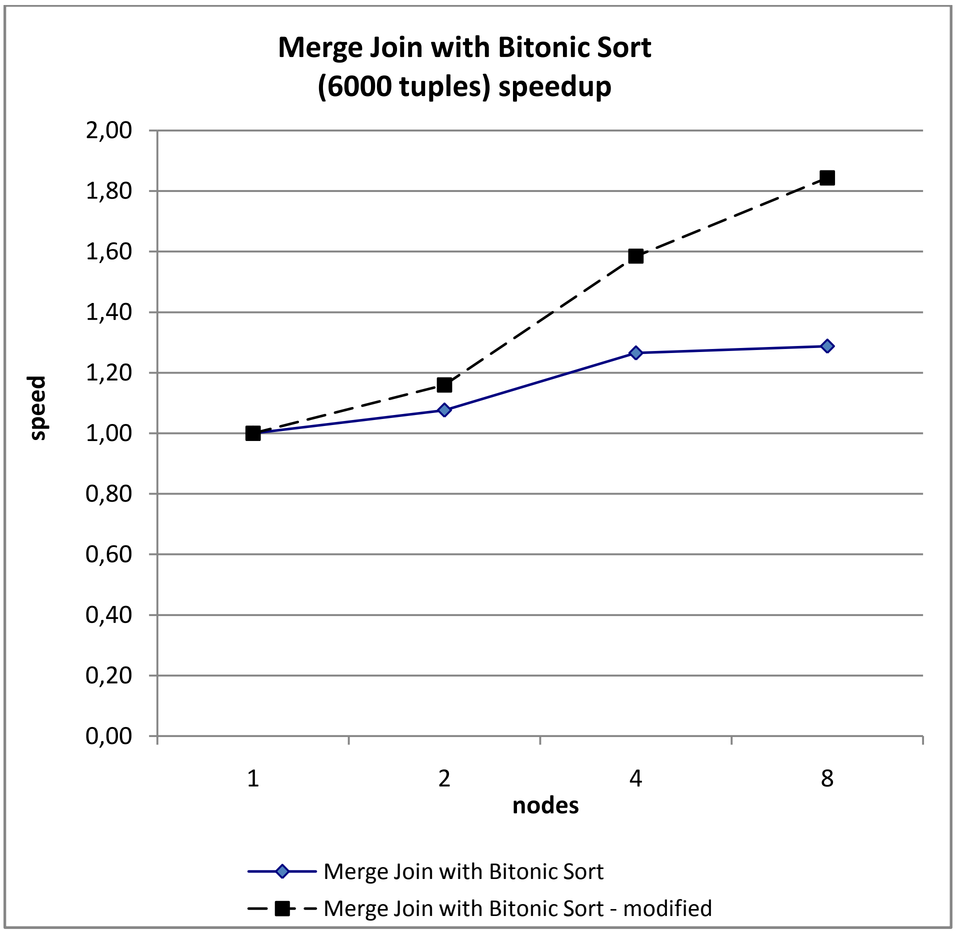


Figure 46: Merge Join with Block Bitonic Sort – speedup

6.2.10 Merge Join with Block Bitonic Sort – scale-up

The scale up analysis of the Merge Join with Block Bitonic Sort shown in Figure 47 is not good for the naive workflow, a bit better for the modified workflow but also not intoxicating. This is really no surprise because the measured speedup for this paradigm is also a very low.

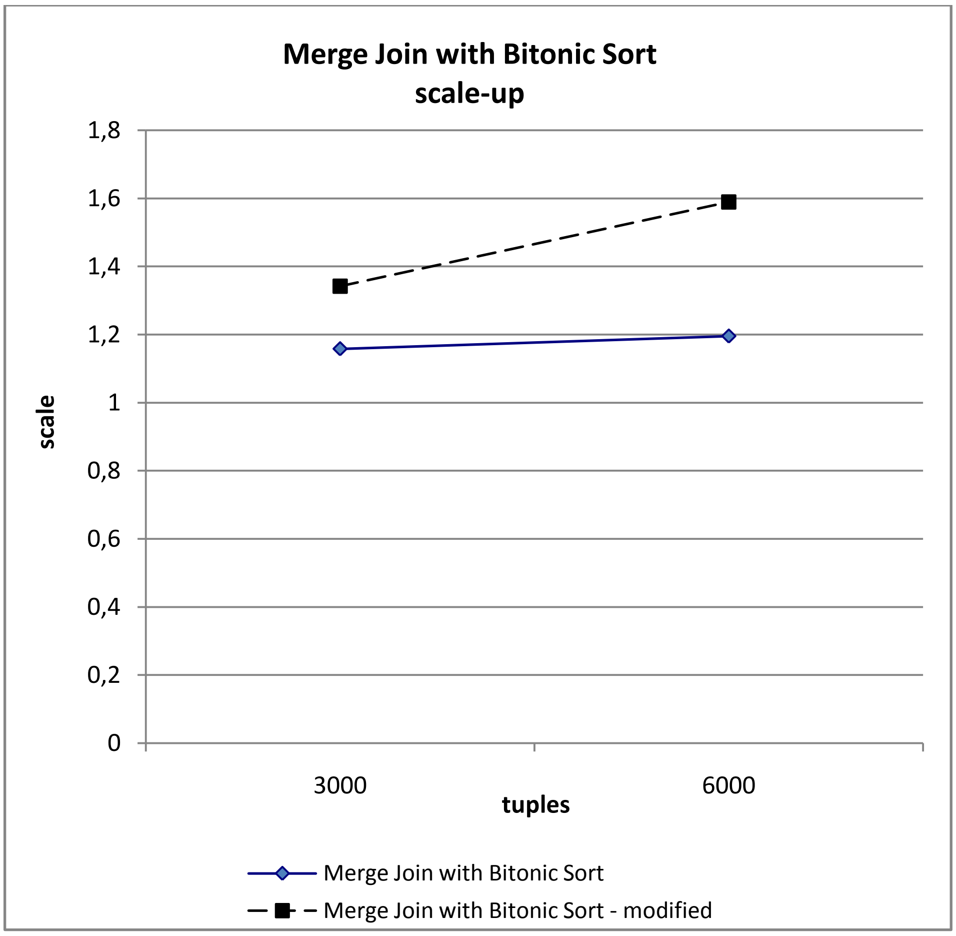


Figure 47: Merge Join with Block Bitonic Sort – scale-up

6.2.11 Hash Join – speedup

The Hash Join speedup analysis shown in Figure 48 performs very well for the modified workflow but on the other side worse for the naive workflow. The cause of this behavior could be the fact that HashJoin Services spawn worker threads and these send many small packets to the MultiFetch Service, so with two, four and eight nodes the count of the worker threads is multiplied and the network is overloaded, therefore the naive workflow is much slower. The good speedup from one to two nodes is because the two workflow execution trees are the same.

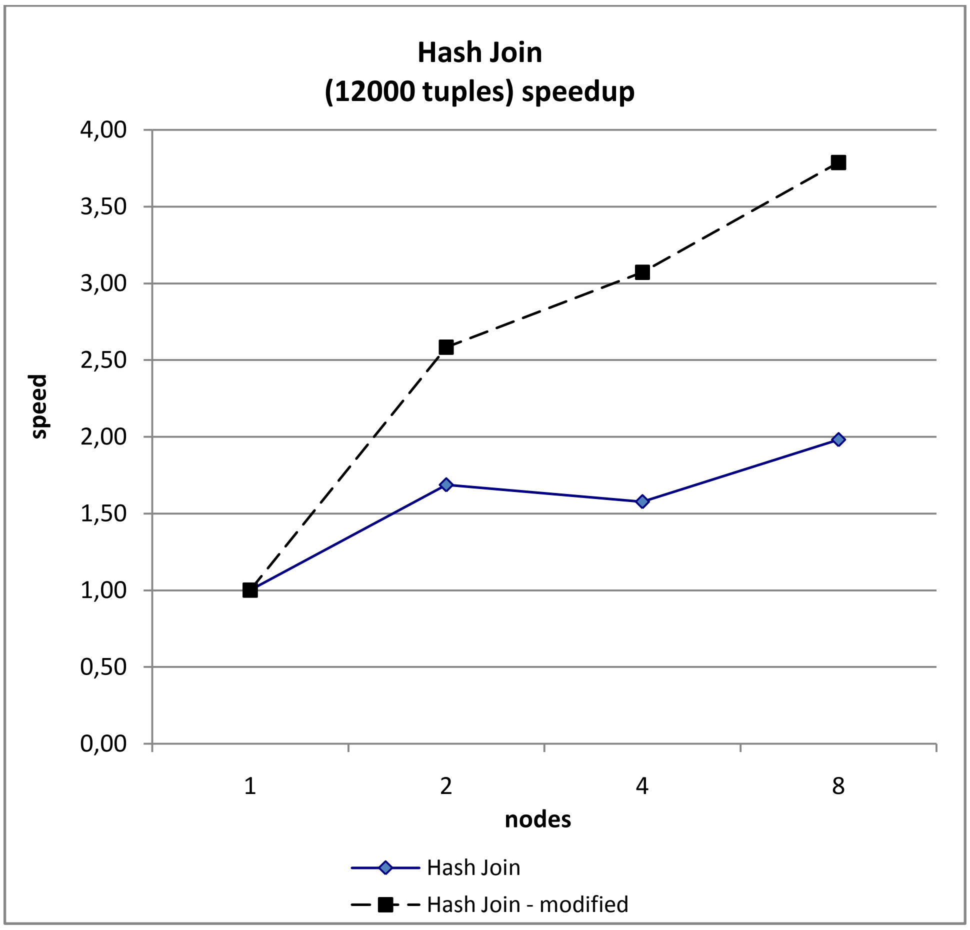


Figure 48: Hash Join – speedup

6.2.12 Hash Join – scale-up

The Hash Join scale-up analysis shown in Figure 49 illustrates that the scale-up for both workflows is not bad at all, but the naive workflow is a bit faster, which is unexpected because the speedup is better for the modified workflow. This could come from the fact that the speedup from four to eight nodes is better in the naïve workflow. The modified workflow performs better because it uses a workflow that takes advantage of the heterogeneous environment in a way, that a data intensive node which has to send and receive more data than other nodes possesses a better network bandwidth.

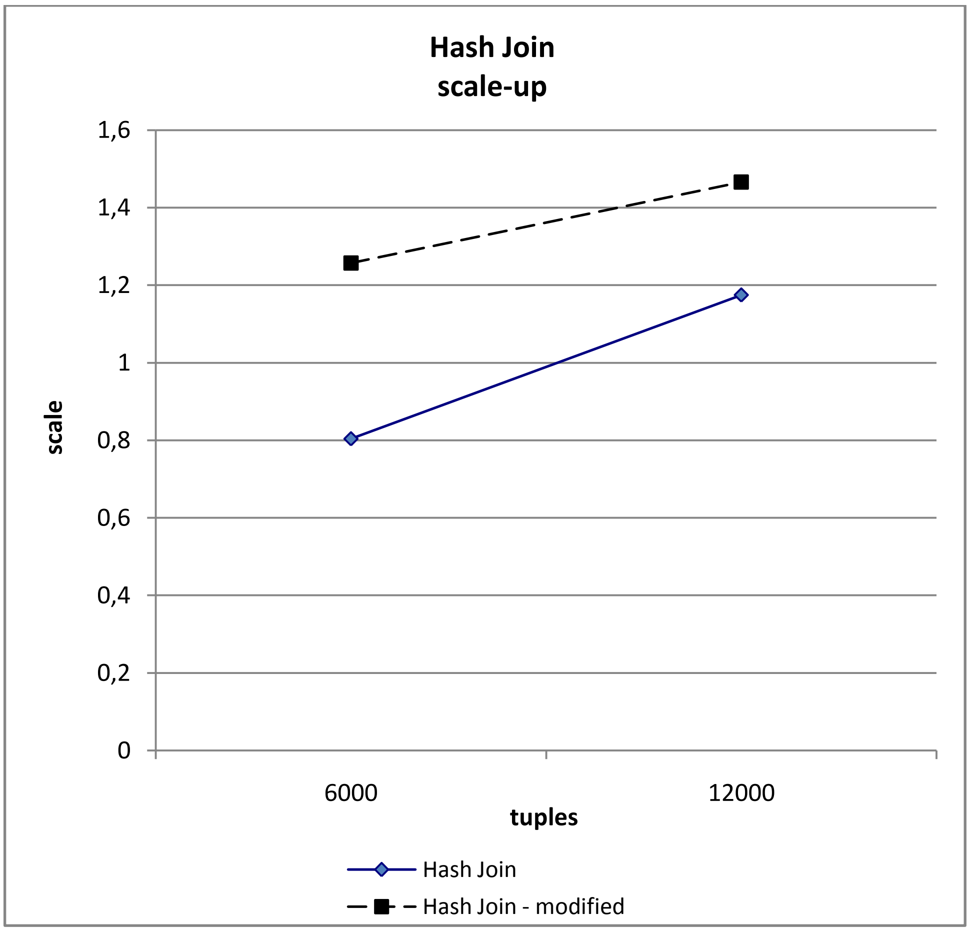


Figure 49: Hash Join – scale-up

7. Lessons learned and Future Outlook

First off all benchmarking is not fun. The benchmarks took ages and the reruns to get good results were annoying. But as everything good thing, it takes a long time.

Another point is that the virtualization used might not be the best way to analyse and build a heterogeneous environment, because of the time spend in configuring the virtual machines, a distributed application simulator could be uses like OptorSim [19], GridSim [20] or SimGrid [21]. Also the performance of the virtual machines does not reach the performance of real machines, but for the available resources it was the best choice.

So with these results, which are far from perfect, the theory of Mach and Schikuta [1], that a smart way to use nodes in a heterogeneous environment is a key to better performance, can be proven. Therefore it can be said that the network bandwidth of a node in a heterogeneous environment is more important than the processing power or the I/O performance. This is especially true if the network speed of two nodes is very different.

There are operations that benefit more from a smarter workflow than other. This can be taken to advantage by a dynamic system that can decide which operation to use when it knows the environment and it has to run the workflow. So this could be a smart broker aware of the properties like CPU, RAM, disc I/O performance, network bandwidth and other parameters which it uses to create a workflow best fitting to the execution environment.

The identification of deadlocks, where one service in the middle of a workflow execution tree is not responding or is endlessly processing, could be better in SODA. So there should be some kind of mechanism to watch the progress of the workflow, because in SODA only the data endpoint service is queried for completion.

Another point would be a dynamic adaption of the packet size for different network loads. To go a step further also the amount of data and how many receiving services are there, should be concerned for a dynamic adjustment of the packet size. With all these factors taken into account a better network performance could be achieved.

For this thesis static workflow execution trees were used. It would be better to dynamically use nodes if they have free CPU time or there network bandwidth has changed. Also a nice idea would be to run two key services, which are important for fast execution of the workflow, in parallel, if there are enough resources left, and if one is finished the other is told to stop and maybe begin with another task.

As mentioned in section 5 a main part of SODA is Muse. Sadly Muse is not in active development any more according to the Muse mailing list. So SODA has to be migrated to another Web service platform especially using another library. Maybe it would be wise to take the experience gained from SODA and start a whole new project, so any design flaws of SOAD could be found in the design state of the new project, and also new ideas which otherwise be hard to implement in SODA as it is today can be integrated easier.

With this said a future work could be the design of a middleware like SODA but with more dynamic aspects like awareness of its surrounding environment and planning the workflow execution tree according to these parameters in a runtime aware manner.

Indices

List of Figures

List of Listings

List of Tables

References

[1] Erich Schikuta, Werner Mach, “Parallel Database Sort and Join Operations Revisited on Grids*”. High Performance Computation Conference (HPCC) Houston, Texas*, 2007.

[2] K. E. Batcher, “Sorting networks and their applications,” *Proc. of the 1968 Spring Joint Computer Conference (Atlantic City, NJ, Apr. 30-May 2)*, vol. 32, 1968.

[3] H. Stone, “Parallel processing with the perfect shuffle,” *IEEE Trans. Computing*, vol. C-20, no. 2, Februar 1971.

[4] Peter Paul Beran, Gernot Habel, “Soda (Service Oriented Database Architecture)”, *diploma thesis*, 2007

[5] Java: “Developer Resources for Java Technology”. http://java.sun.com/, [last access 2008-04-15]

[6] Tomcat: “Apache Tomcat”. http://tomcat.apache.org/, [last access 2008-04-15]

[7] Apache Ant: “Apache Ant - Welcome”. http://ant.apache.org/, [last access 2008-04-15]

[8] Apache Muse: “Apache Muse - A Java-based implementation of WSRF 1.2, WSN 1.3, and WSDM 1.1.”. http://ws.apache.org/muse/, [last access 2008-04-15]

[9] AXIS2: “Apache AXIS2”. http://ws.apache.org/axis2/, [last access 2008-04-15]

[10] XMLBeans: “Welcome to XMLBeans”. http://xmlbeans.apache.org/, [last access 2008-04-15]

[11] MySQL: “MySQL :: The world's most popular open source database”. http://www.mysql.com/, [last access 2008-04-15]

[12] EHCache: “Ehcache”. http://ehcache.sourceforge.net/, [last access 2008-04-15]

[13] P. Valduriez and G. Gardarin, “Join and semijoin algorithms for a multiprocessor database machine,” *ACM Trans. Database Syst.*, vol. 9, no. 1, pp. 133–161, 1984.

[14] TPC: “Transaction Processing Performance Council”. http://www.tpc.org/, [last access 2008-04-30]

[15] Floyd, S., and Jacobson, V., “Link-sharing and Resource Management Models for Packet Networks” *IEEE/ACM Transactions on Networking*, Vol. 3 No. 4, pp. 365-386, August 1995.

[16] cbq.init: “SourceForge.net: CBQ.init traffic management script”. http://sourceforge.net/projects/cbqinit/, [last access 2008-06-09]

[17] D. DeWitt and J. Gray, “Parallel database systems: the future of high performance database systems”, *Commun. ACM*, vol. 35, no.6, pp.85-98, 1992.

[18] VMware Server: “VMware Server, Virtual Server Consolidation, Free Virtualization - VMware”. http://www.vmware.com/products/server/, [last access 2008-06-20]

[19] OptorSim: “SourceForge.net: OptorSim”. http://sourceforge.net/projects/optorsim/, [last access 2008-07-10]

[20] GridSim: “The GRIDS Lab and the Gridbus Project”. http://www.gridbus.org/gridsim/, [last access 2008-07-10]

[21] SimGrid: “Welcome to the SimGrid project!”. http://simgrid.gforge.inria.fr/, [last access 2008-07-10]