OGSA-DAI Parallel Hash Joins Using BonFIRE

Joshua Eke

August 19, 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

This study examines the phenomenon of ‘data deluge’ and the challenge of making effective use of large volumes of data. It focuses on use of OGSA-DAI and DQP and ways in which performance can be improved. The study identified a blockage in the OGSA-DAI pipeline, which it fixed. It explores whether a parallel hash join can be implemented as an efficient join method using OGSA-DAI and under what conditions this approach is effective. The study created code for the early adoption of BonFIRE. The data shows that the pipeline fix was effective, the code produced for the use of BonFIRE was successful but a parallel hash join is no faster than a serial join and BonFIRE did not provide the anticipated stable environment. A range of further work is identified.
Contents

Chapter 1 Introduction

Chapter 2 Background

2.1 Relational Algebra

2.1.1 Tuples

2.1.2 Join Operations

2.1.3 Normalisation

2.2 OGSA-DAI

2.2.1 Implementation

2.3 Parallel Hash Join

2.4 OGSA-DQP

2.5 BonFIRE

2.6 Adaptive Joins

2.7 Previous Work

2.8 Amdahl’s law

Chapter 3 Theoretical Analysis

3.1 Pipeline Model

3.2 Data Transfer Model

Chapter 4 Experiment Infrastructure

4.1 Data Transfer

4.2 Workflows

4.3 Requests
6.2.5 Cause Of Variability .................................................................................. 37
Chapter 7 Conclusion .......................................................................................... 40
  7.1 Future Work .................................................................................................. 41
    7.1.1 Additional Instrumentation ................................................................. 41
    7.1.2 Virtual Wall Emulated Network ......................................................... 42
    7.1.3 Weighting parallel hash joins .............................................................. 42
    7.1.4 Network Latency Exploration .............................................................. 43
  7.2 The Consistency And Reliability of BonFIRE Results .................................. 44
    7.2.1 Instance Size ....................................................................................... 45
  7.3 Summary ...................................................................................................... 40
Appendix A  Test Data .......................................................................................... 46
Appendix B  Code Structure ............................................................................... 48
References ........................................................................................................... 49
List of Figures

Figure 1: Parallel Hash Join With Tuples Running On Three Nodes ...................... 8
Figure 2: Hashing Of Tuples ................................................................................. 9
Figure 3: Pipeline Example .................................................................................. 12
Figure 4: Speedup Ratio ...................................................................................... 14
Figure 5: Deadlock ............................................................................................... 19
Figure 6: Tuple Generator UML .......................................................................... 23
Figure 7: Parallel Hash Join Timings ................................................................... 33
Figure 8: Repeat One Parallel Hash Join Timings .............................................. 33
Figure 9: Repeat 1 And First Run ....................................................................... 34
Figure 10: Run One Setup Times ......................................................................... 36
Figure 11: First and Second Query Batch First Tuple Arrival Time ...................... 37
Acknowledgements

I would like to thank Ally Hume for his guidance and support throughout my project. I would also like to thank Amy Krause for her contributions whilst Ally was away.
Chapter 1

Introduction

The quantity of data that is created and analysed by scientific, business and government organisations is constantly increasing. This phenomenon is sometimes referred to as the ‘data deluge’. According to The Economist [1], in 2005 mankind created 150 exabytes of data and by 2010, another 1,200 exabytes of data would have been produced.

As more sensors are becoming connected to networks and the internet, more data is becoming available for analyses in all fields. Amongst the fields where this has occurred are Astronomy, Oceanography and Physics. The LHC (Large Hadron Collider) is expected to produce 50 to 100 PB of data each year [2]. Much of Astronomy is now carried out using database queries to search different light wavelengths. The amount of data that is produced by telescopes to populate databases for this research is large.

Other data that is stored includes consumer related purchasing information. This could be purchases at a supermarket, credit card companies or online websites. This data could be used for purposes such as marketing, for example, in the case of supermarkets, or prevention of fraud.

Our ability to cope with large volumes of data depends, in part, on our ability to mine the data. This is not only true in marketing but also in scientific fields when analysing the results of sensors.

The main problems that occur with data analysis on this scale are the compatibility of the data sources, independent locations of this data and the speed of processing this data. Many of these problems can be solved by using data warehousing to consolidate the data in one location where the data is stored using compatible formats.

Data cannot always be stored in a data warehouse as it may be collected in geographically dispersed locations or produced and managed by different organisations. This problem escalates when the amount of data produced at a location is large such, as in the case of the LHC, or some of the astronomical telescopes which collect data in a digitised format.

In situations where large volumes of data are collected at a single site, the cost of transporting this data to a data warehouse increases and it becomes more economic to
store this data at a local facility. Alternatively when the data is small and managed by independent groups it may be more convenient to store the data locally.

When your databases are distributed across multiple sites there are often times when the data needs to be joined. One possible algorithm for doing this is the parallel hash join. A framework that can be used for the investigation of a parallel hash join is provided by OGSA-DAI (Open Grid Service Architecture – Data Access Integration).

OGSA-DAI is a software system that allows data from different geographic sources to be combined as a federated resource and queried by users. The federated data sources appear as one data source to the user. The data it contains becomes accessible as if it were one database. Federation of data is achieved by using the Distributed Query Processor (DQP), a component of OGSA-DAI. The DQP is responsible for handling the database queries issued by the user.

When a user issues a request to a federated data source, the data sources which make up the resource need to be individually queried. After the data sources start returning their values, any operations which the user requested in their query are performed. The data will then be returned to the user via their requested method. The DQP is responsible for coordinating all these user request associated operations and interpreting the queries.

The problem that we will be focusing on is whether a parallel hash join can be implemented as an efficient join method using OGSA-DAI and under what conditions this approach is effective. When this work has been completed, it will be possible to update the DQP to utilise a parallel hash join where it would result in better throughput to the user.
Chapter 2

Background

This section will relate the work to be conducted on the existing theoretical model of data bases, relational algebra. It then goes on to cover OGSA-DAI, the engine that is used for this investigation into parallel hash join implementations, DQP, adaptive joins, BonFIRE [3] and Zhu’s [4] previous work on the subject.

2.1 Relational Algebra

A database is similar to a spreadsheet in that there are rows, columns, worksheets and books. A relational database system has similar properties. To describe the properties of a database in a way that is implementation independent we use Relational Algebra. Relational Algebra is a theoretical model for representing data sets and their operations.

2.1.1 Tuples

Tuples are similar to a row in a spreadsheet but they have greater theoretical background and definition. Date [5] says, “Every tuple contains exactly one value (of appropriate type) for each of its attributes”. This means that for every column in a spreadsheet, there is one value per tuple.

Date goes on to say, “There is no left-to-right ordering to the components of a tuple. This property follows because a tuple is defined to involve a set of components, and sets in mathematics have no ordering to their elements”. Because there is no ordering, this tuple:

\{(Name, Bob), (age, 29), (phone, 02476513236)\}

is the same as this tuple:

\{(phone, 02476513236), (Name, Bob), (age, 29)\}.

There is also no top to bottom ordering of tuples as there would be in a spreadsheet.
Date’s final tuple property is “Every subset of a tuple is a tuple (and every subset of a heading is a heading)”. This property states that tuples can be made from other tuples for example \{(Forename, Bob), (Surname, Smith), (Address, \{(Street, Mayfair), (City, London), (Postcode, SW1 1JB)\})\}. In order for this to work, the address attribute has to be a ‘super type’ which contains the attributes Street, City, Postcode.

When describing the number of attributes a tuple contains, this is called the tuple’s degree. The degree of a tuple with no attributes is called a zero degree tuple. A tuple with three attributes like those above is called a ternary degree tuple [4].

2.1.2 Join Operations

There are eight different relational algebra operations described by Date [4]. However, we are only interested in the join operation. There are many flavours of join. The join that we are interested in here is called an equijoin. This kind of join is used to combine multiple tuples using their common attributes.

For example, if you have the sets of tuples:

C = {
    \{(id, 1), (Name, Bob Smith)\},
    \{(id, 2), (Name, John Roland)\},
    \{(id, 3), (Name, Roger Bowen)\}
}

D = {
    \{(id, 1), (Salary, 40000)\},
    \{(id, 2), (Salary, 35000)\},
    \{(id, 4), (Salary, 20000)\}
}

Then

C JOIN D = {
    \{(id, 1), (Name, Bob Smith), (Salary, 40000)\},
    \{(id, 2), (Name, John Roland), (Salary, 35000)\}
}

From the results of C JOIN D it can be seen that the tuples with common attribute values (1 and 2) have been combined and the tuples without common attribute values (3 and 4) have not been combined.
2.1.3 Normalisation

Normalisation is a process used to reduce data sets to formats where data is not repeated. When normalising data each tuple has an attribute(s) which can be used to uniquely identify that tuple. The attribute(s) are called the primary key. The primary key can be composed of more than one attribute in which case it is called a composite key.

2.2 OGSA-DAI

The OGSA-DAI [6] application works to combine data from multiple locations and databases and provide a layer of abstraction that causes the data to appear through one interface. This makes processing the data easier for users and can ease application development. It also hides the implementation details of different vendors’ RDMS (Relational Database Management Systems) from the application developer and user.

Example applications that have benefited from the use of OGSA-DAI are GeoTOD-II [7] (Geospatial Transformation with OGSA-DAI), BIRN [8] (Biomedical Informatics Research Network) and ADMIRE [9] (Advanced Data Mining and Integration Research in Europe). All these applications require data to be combined from geographically dispersed locations.

OGSA-DAI can perform queries that reduce the amount of data transferred when a user makes a query. This is done by performing local queries at each storage facility and then combining the resulting data sets. If the user performs a query that restricts the data returned by the database then the data sets returned will be smaller than the entire database stored at that database location. If OGSA-DAI were not able to selectively query databases then the entire database would have to be transferred to a central location for processing.

When OGSA-DAI servers communicate, they break their data sets up into chunks to make transportation across the network more resistant to error and to reduce the time taken to pack and unpack the datasets. As the dataset is received the server can either wait until the entire dataset has arrived or begin processing each chunk as it arrives.

OGSA-DAI uses a software pipeline to process data as it arrives. The advantages of a pipeline are that it is highly modular, it produces initial results quickly and it handles unpredictable network conditions well. If the user wishes to alter the flow of the pipeline then they only need to change its stages. Producing initial results quickly will allow the client to start processing the results sooner. Handling unpredictable network conditions well allows processing to continue when poor network conditions are present. The disadvantages of a pipeline are that the data processing speed will be limited by the slowest point in the pipeline.

We have access to an environment which will allow different network conditions to be emulated. This will be used to explore the impact that network conditions have on OGSA-DAI and to assess if the joins investigated are worth including in OGSA-DAI.


2.2.1 Implementation

OGSA-DAI is implemented in Java and uses an Object Orientated Programming (OOP) style. This makes development and maintenance of the code easier. It also makes the code more portable across multiple operating system platforms. OGSA-DAI runs as a web service inside a tomcat server.

2.3 Parallel Hash Join

Currently OGSA-DAI has a limited set of join operations implemented. Implementing a parallel hash join using OGSA-DAI will expand the range of joins available to choose from. This will make OGSA-DAI more efficient at processing data requests and more versatile.

A parallel join allows two data sets to be joined in parallel. This could potentially increase the speed of any join operation. A parallel hash join uses a hashing algorithm to distribute data sets between multiple worker nodes. These nodes then perform a join on the two data streams they receive and pass the results to a client or a server which performs further processing. This is shown in Figure 1 which shows three nodes performing the work.

There are multiple types of joins that can be performed but the join we will be considering is the equality join. The equality join combines tuples from each data stream which have the same key. A parallel hash join only works with equijoins. This is because each tuple with the same hash code must be sent to the same server, so that it can be combined with other tuples that have that hash code.

An example of tuple hashing is shown in Figure 2. In this example, the id attribute of the tuple is hashed and then has a modulus division performed on it. Because in Java the hash code of a number is the number itself, it can be assumed the id attribute value will be divided and used to allocate the tuple to a server.

The expectation in using a parallel hash join is that by distributing the work of the join across multiple machines, the load on each server will be reduced and the time to perform the join will reduce. However, by distributing the load across multiple machines, OGSA-DAI will transfer more data which could increase the time which a join takes. The data transfers that do occur will be performed concurrently. This could offset the time taken by the transfer of data to the parallel join worker nodes. The parallel join should provide better performance gains when a greater quantity of data is being processed by the servers.

We are going to investigate both theoretical and practical aspects of the PHJ algorithm. There will be a theoretical analysis of the parallel hash join performance performed. This will result in a model of the system’s performance. This model will explain the performance of the join in respect to the pipeline stages and data transfer times. After a practical implementation of the parallel hash join has been implemented and performance tested, the results will be compared to the theoretical model to determine if the model was accurate.
The infrastructure of a parallel hash join could be setup in multiple ways. There will always be two servers producing data from a data source. Each of these servers will process a fraction of the data locally. The rest of the data will be shared out to the worker nodes.

When working with databases, the attributes of a tuple can be called keys. When performing a join the common attributes are called the join keys. When implementing the parallel hash join we assume there will only be one join key.

The hash split uses the join key to distribute the data over multiple worker nodes. It hashes the key value and uses the result to ensure that keys with the same value from each table are sent to the same worker node. Therefore the distribution between nodes will only be as good as the distribution of the keys.

If the performance of the data transfer slows the processing of the data, the hash split could be weighted. Weighting could be varied based on the processing speed or network bandwidth. It could also be varied based on the number of rows in a table and the locality of the data source.
Figure 1: Parallel Hash Join With Tuples Running On Three Nodes
Figure 2: Hashing Of Tuples
2.4 OGSA-DQP

The OGSA-DQP is used to split the workload of a query between servers. Currently the DQP does not have the capabilities to analyse a query and distribute it in the most efficient manner.

Part of this project will be to analyse the effectiveness of the OGSA-DAI parallel hash join algorithm and under what circumstances it proves to be effective. If a parallel join proves efficient, DQP could be altered to process queries more effectively when processing join operations.

The OGSA-DQP framework will not be used in this project but the processing it would do, after the initial decoding of the query, will be simulated by the parallel hash join implementation provided. This implementation will remove the overhead that would be incurred when using the OGSA-DQP framework.

2.5 BonFIRE

BonFIRE [3] is a cloud infrastructure test bed. Multiple virtual machines can be setup with an image on them. OGSA-DAI or a client can then be installed on the image. A network can then be configured between the virtual machines. There is a BonFIRE test bed facility called the Virtual Wall that allows network conditions to be emulated. Conditions such as additional latency, packet delay and bandwidth can all be controlled. The Virtual Wall will be used to test the effectiveness of the parallel hash join under different network conditions similar to those encountered when using data sources from different geographical locations.

It will be of interest to see how well the parallel hash join performs when items of data are transferred using different network conditions. This situation could be optimised by using a weighted hash split algorithm.

2.6 Adaptive Joins

There is a class of join known as an adaptive join. This class of join algorithm is designed for joining datasets streamed across a network. The type of join that is executed in parallel will be an adaptive join as it is most appropriate for the network conditions encountered by this project.

2.7 Previous Work

A parallel hash join has previously been studied formally and informally by members of EPCC (Edinburgh Parallel Computing Centre). Zhu [4] studied parallel hash joins as part of his dissertation. At the time of his study, he could not produce stable results in relation to parallel hash joins. This could have been because the computing nodes he was using were dispersed across multiple geographic locations which were connected using a network that did not perform in a stable manner. This was a realistic environment, however, it might not have been stable enough to produce consistent
results. It was anticipated that by using BonFIRE, a stable network environment could be guaranteed for the experiments to be conducted.

Anecdotal reports suggest that because the joins take very little time to complete, not much speedup can be achieved by using a parallel mechanism. However, since these are not formal studies, no figures are available and repeating the work in a more formal manner is worth doing and could yield useful results. These results will show when a parallel hash join is worth using and under what conditions.

2.8 Amdahl’s law

It should be noted that the execution time will be limited by Amdahl’s law [9]. The time to execute a query will be proportional to the amount of work that can be performed in parallel and the amount of work that must be performed in serial. In the case of the parallel hash join, the retrieval of data from a database and the division of data between different servers will all be performed in serial. The joins of the data from different databases can all be performed in parallel.
Chapter 3

Theoretical Analysis

The following section summarises the theoretical model of a streamed parallel hash join algorithm implemented using OGSA-DAI. Analysing the algorithm before implementation will allow a feasibility assessment. The performance of the join algorithm will be estimated using time proportional to the number of tuples processed.

3.1 Pipeline Model

The parallel hash algorithm acts as a load balancer, dividing the data between multiple nodes. In a cloud infrastructure, data transfer times could add considerably to the application’s runtime. To compensate for the time taken to transfer the data from one node to another, the time taken to process the data must outweigh the time to transfer the data.

As the number of tuples that pass through a node increases, the time to perform the join should also increase. When a node receives a tuple, the tuple must be stored in memory so that it can be compared to tuples that have yet to arrive. An increase in the number of tuples stored causes an increase in the number of comparisons that have to be made, hence an increase in time. This could be significant as running the join in parallel will reduce the number of comparisons made per node. However, for the purposes of this analysis we will not be concerned with the time it takes to execute individual joins and how many comparisons they make.

![Pipeline Example](image)

**Figure 3: Pipeline Example**

The example pipeline in Figure 3 shows that if the Hash Split can process one tuple every millisecond, then for every tuple after the first there will be a four millisecond wait until the Data Transfer stage can receive the next tuple. The join will then be idle for three milliseconds whilst it waits for the data transfer to complete.
It can be concluded that a pipeline is limited by the slowest stage of the pipeline, and that the interval at which the pipeline produces results is the time of the slowest stage. Therefore when considering a pipeline the time taken for completion of n tuples will be:

\[ T_s = S_1 + S_2 + S_3 + (n - 1) \max(S_1, S_2, S_3) \]

Where \( T_s \) is the total serial time and \( S_i \) is the time taken for each stage to complete. This works for a general model of a pipeline, however, does not account for the parallelisation. The time for parallel execution of a stream will be:

\[ T_p = S_1 + S_2 + S_3 + \frac{(n - p) \max(S_1, S_2, S_3)}{p} \]

Where \( T_p \) is the time for one processor to complete. In OGSA-DAI’s implementation each of the pipeline stages has a twenty block buffer connecting them, however, we will not be concerned with this when analysing the time taken to perform a join and data transfers.

Grama et al [9] define speedup as “the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm”. The parallel speedup is therefore:

\[ R = \frac{S_1 + S_2 + S_3 + (n - 1) \max(S_1, S_2, S_3)}{S_1 + S_2 + S_3 + (n - p) \max(S_1, S_2, S_3)} \]

Some of the delay created could be mitigated by performing operations using chunks of data. For example, the data transfer could be performed on a chunk of data to reduce the overhead associated with it. I would predict that the data transfer will be the most expensive part of the pipeline as it incurs more overheads than any other stage of the pipeline.

Figure 4 shows the speedup that could be expected according to this model. Lower values of \( n \) show less speedup while with higher values of \( n \), a near linear increase in speedup occurs. Although the times for each stage used to calculate Figure 4 were not realistic it shows that by using the pipeline model an increase in speedup could be expected but that for larger numbers of tuples the speedup would be greater than for lower numbers of tuples. The more machines used to processes the tuples in parallel the greater the speedup that can be achieved. This is particularly relevant to the case where \( n = 1000 \) where a near linear speedup is predicted.
The constant units of time used when calculating Figure 4 were $S_1 = 8$, $S_2 = 16$, $S_3 = 32$.

### 3.2 Data Transfer Model

This section explores how much data will have to be transferred between servers and clients when a parallel hash join occurs. In the standard parallel hash join model there is a point at which the data that has been processed in parallel needs to be merged together. Because the merging of data items requires more network communication there is an increased overhead. It is hoped that the increased overhead of network communication will be offset by the savings in time that performing a parallel hash join achieves. Every tuple that does not reside on the server performing the merge must be transferred across the network. This merge step more than doubles the volume of data that has to be transferred when performing a parallel hash join.

If the network is quick the cost of transferring the data will be low. If the network is slow then the cost of transferring data across the network will be high. It will only be beneficial to transfer data if the cost of processing it is greater than the cost of transferring it. This could be what happens with a parallel hash join.

A hash split can execute and produce an even division of data or an uneven division. If there is an even division of data then the data transferred to each client will be the same. If there is an uneven division of tuples by the hash split then an uneven balance occurs. In the case of an even distribution of tuples:

$$c = 2^{j-2}$$
Connections occur where $j$ is the number of joins that occur and $c$ the number of connections, this only holds true for a parallel join, not a serial join. If the scenario is investigated where two databases are combined which contain the same number of tuples, and where the tuples are the same size, the total data transferred from a data producing node will be:

$$D_T = \frac{n(p - 1)}{p}$$

This is because a fraction of the data is not transferred across the network. Instead this data is processed locally by the data producing node. The total data transfer that happens between a data producing node to any other node is:

$$D_w = \frac{n}{p}$$

Because the data transfer could happen in parallel the time to transfer $D_w$ could be less than the time to transfer $D_T$. This means that there could be a saving in time made by the parallelisation of the data transfer between nodes. The next section goes on to talk about the experiment infrastructure.
Chapter 4

Experiment Infrastructure

This section covers the way that OGSA-DAI has been extended to support this experiment, how the client was written and the way BonFIRE was used to create experiments. All this material is common to all experiments carried out. It also covers some background on OGSA-DAI which is relevant to the implementation.

The focus of these experiments is to learn more about factors which affect the speed and execution of a parallel hash join, with a view to identifying the conditions under which the join is most effective. The design of the experiment was intended to allow the effectiveness of the parallel hash join to be explored in a variety of scenarios. If the join proved particularly effective under identified conditions this could be used to modify the way in which DQP handles join operations to make for most effective use of resources.

4.1 Data Transfer

To transfer data between OGSA-DAI servers, a data sink is used. A data sink operates in a push mode and is able to push data to other servers. The data transmitted by a data sink is buffered locally until the data sink is ready to push the data to another server.

To receive the tuples the client uses a data source. The data source operates in a pull mode, data is buffered at server until the data is pulled from the server by a client.

4.2 Workflows

Within OGSA-DAI all activities are carried out as part of a workflow. This implementation of the parallel hash join uses the ‘PipelineWorkflow’ to control the workflow. The OGSA-DAI documentation [9] states, “A pipeline workflow consists of a set of chained activities that will be executed in parallel with data flowing between the activities”. This means that the workflow executes like a pipeline on a CPU, with each stage being equivalent to an activity.
The activities must be chained by the client using activity inputs and activity outputs. If an activity does not have all its mandatory input(s) and output(s) connected then an error will be thrown by the application.

For each server an instance of ‘PipelineWorkflow’ is created, this instance is then populated with activities by the client. Activities are equivalent to stages in the pipeline.

4.3 Requests

Once a pipeline has been constructed it is submitted to the server. Each pipeline is submitted individually to different servers. When a pipeline is submitted to the server it can be submitted in two modes, synchronous and asynchronous.

Because the client is communicating with multiple servers which execute requests in parallel, an asynchronous communication model must be used. This model requires the client to poll the server for the completion of the workflows.

The data sink and data source resources must be created in advance of a pipeline workflow being submitted. Activities in the pipeline reference the data sink and data source resources. Therefore the pipelines cannot be submitted until the data sink and data source are created.

4.4 OGSA-DAI Client Toolkit

OGSA-DAI provides a client toolkit. This acts as a proxy to the server and allows workflows to be constructed, submitted and their status monitored by the client. Each activity used by client toolkit comes in two parts, the client and the server activity.

The client is used locally to setup the inputs and outputs of the activity. The client class can also be used to provide literal values to the client rather than just connect inputs to the client.

4.5 Parallel Hash Join

The implementation of the parallel hash join relies on the existing activities available for the implementation of joins. The difference between a serial join and a parallel join is that it uses these activities in parallel. Given evidence provided by the pipeline model, we would expect a linear increase in performance for higher counts of tuple.

In order to see what performance can be gained from parallelising the joins, a serial join will be executed with the same parameters as a parallel join and results of the serial join compared to those of the parallel hash join.

4.6 Join Activity Implementation

The current join implementations available in OGSA-DAI for equality joins are a theta join and a pipelined join. These will be the joins that are considered here. The pipelined
The join runs two threads, each processing one of the data streams. The threads buffer the tuples they have received and compare them to the tuples of other threads as tuples are received.

The theta join implementation requires all of the data from one stream to be loaded into memory before comparisons between the first and second stream can begin. This blocks the pipeline until the entire first stream has been passed completely to the join activity and is available in memory.

This is a less than optimal solution as it increases the time to produce the first tuple and would require a large memory footprint before any processing can begin.

We will be using the pipelined join in the parallel hash join as it provides the best performance in a situation where network lag between nodes could occur. This is because it handles each data stream independently. Data can therefore be received independently and there is no need for the processing of data to be blocked whilst data is received from one stream in its entirety.

It is also closer to an adaptive join algorithm than the tuple theta join. This makes it the best suited category of algorithm for the parallel hash join. The pipelined join is also a better algorithm for handling the deadlock which occurred which is covered in the next section.

4.7 Existing problems

This section covers the existing problems which were encountered in OGSA-DAI that had to be overcome before the parallel hash join experiments could be conducted. The main problems addressed were the deadlock which we encountered and the broken ‘ThetaTupleJoin’ class.

4.7.1 Deadlock

When the parallel hash join was first tested a deadlock was detected. This was reported to the OGSA-DAI team and a bug ticket [11] created. The join activity has two tuple input streams. Between activities there is a 20 object buffer. When the buffer becomes full, no more objects can be passed to that activity and the pipeline becomes blocked. The deadlock is caused when the size of the byte array is 20, the number of blocks 60 and the number of tuples generated greater than 37.

The blocked pipeline occurred because of the hash split algorithm being unable to write any more tuples to the join. The join does not start processing tuples until it has received the metadata from each of the tuples input streams. This is illustrated in Figure 5.
When the hash split became blocked, it was not only unable to write any more tuples to the join, it could not write any more tuple to the TupleToByteArrays activity either. The tuples to byte array activity is responsible for combining the tuples into groups of bytes called chunks. If the number of tuples sent to the ‘bytes to array’ activity are less than the number of tuples that make up one byte array chunk, the byte array chunk will not be created unless a list end marker is encountered. The list end marker denotes the end of a list of objects.

Before the data sink can push data across the network, it requires a chunk to have been received. The data sink waits for a specified number of chunks to have been received.
before it starts sending blocks out across the network. This is unless non-blocking mode is enabled for the data sink in which case it sends whatever data it has stored after the last send request has been completed.

There were multiple ways to fix the deadlock within OGSA-DAI. The HashSplit, TupleToByteArrays, DeliverToDatasink and PipleinedJoin activities could all be modified. It would also have been possible to use a method that utilised a pull mechanism rather than a push mechanism to pass data across the network.

The hash split could be modified to incorporate buffering so that it never becomes blocked by the pipeline stages which follow. It could be multi threaded, with each thread containing a buffer. When the pipeline becomes free, the thread would then write the contents of its buffer to the pipeline.

By using this method none of the existing activities would have to be modified to create the parallel hash join and the changes would have been confined to one functional unit. However, the complexity of the hash split would have been increased and the memory footprint of the hash split would also have been increased.

Another alternative method for fixing the deadlock would have been to modify the TupleToByteArrays activity and change the data sink to a data source. The TupleToByteArrays activity would only create byte arrays that are one tuple in size. Using the existing pipeline for each tuple that was passed into the ‘tuples to bytes array’ activity, one would have been passed to the data sink activity. This on its own would not solve the problem and a data source would have to be written to allow the server to pull the data from the server. The data source would have buffered the byte arrays it received locally and stored them until one of the servers requested them.

In order for this approach to work the TupleToByteArrays would need to be modified and a new DataSource would have to be written. This would have required considerable development work. The solution would also have been more complicated than would have necessarily been required. There could have been problems that occurred with the performance of the system when using this approach as the combining of tuples into arrays or ‘chunking’ of data would not be used. However, this could be compensated for by using a data source which does not use the Simple Object Access Protocol (SOAP) and therefore runs with reduced overheads.

To fix the deadlock, the pipelined join was modified so that it included an additional stage between the receipt of the first tuple and the configuration. It now stores any rows it received before configuration can occur in an array list. Configuration can only occur when the metadata from both streams has been read. This is so that the tree data structure used to store tuples for comparison can be configured to store the tuples according to their primary key. This was the simplest approach to fixing the deadlock as the least amount of changes to the code needed to be made. This approach also allows changes to be made at the point where the deadlock was occurring and what may be perceived by some as the cause of the problem.
4.7.2 Join Client Toolkit Class

I identified that the original implementation of OGSA-DAI contained a broken TupleThetaJoin client class. This had to be fixed before experiments could be conducted using the OGSA-DAI client toolkit as an interface. To fix the TupleThetaJoin client toolkit class, the inputs and outputs had to be renamed to match those of the TupleThetaJoin activity.

Initially, the first experiment was going to be conducted using the TupleThetaJoin rather than the PipelinedJoin. However, due to the deadlock that occurred in OGSA-DAI this became unfeasible. Since the PipelinedJoinActivity currently has no client toolkit class, a way for the parallel hash join client to interface with the PipelinedJoinActivity was needed.

An entire new client toolkit class for the PipelinedJoinActivity class could be written. This would be time consuming but it would result in a cleaner interface to the PipelinedJoinActivity class.

Alternatively, the TupleThetaJoin client toolkit class could be used for the PipelineJoinActivity. To do this, the TupleThetaJoin client toolkit class would have the method `setActivityName` called with the option “uk.org.ogsadai.parallelhashjoin.PipelinedTupleJoin”. This would have meant that the TupleThetaJoin activity was no longer responsible for acting as a proxy between the client and the TupleThetaJoin but was now responsible for handling the control of the PipelinedTupleJoin activity.

The choice was made to use the `setActivityName` method to update the TupleThetaJoin client. This helped to save the amount of development time required. It also made the development of the parallel hash join test harness simpler.

4.8 Bespoke Activities

This section covers details of bespoke activities that are needed to carry out the experiments outlined in the experiments section. Bespoke activities were needed for the implementation of the algorithm and for the profiling of the code.

4.8.1 TupleTimer

To time how long an individual stage inside the pipeline takes a new activity is needed. Having a timer for each stage will allow the pipeline to be profiled. This will help with the analysis of an experiment and the results it produces. To profile the pipeline the time of the first tuple and last tuple to pass through a point in the pipeline is needed. This information could be combined with timings of other stages to determine how long the first tuple took to pass through an activity, and how long the last tuple took to be processed by an activity.

The TupleTimer activity is designed to time the points at which the first and last tuple pass through it. When a list of tuples is passed between activities in the pipeline, the list
of tuples begins with a ControlBlock.LIST_BEGIN object and ends with a ControlBlock.LIST_END object.

These objects tell the activity when to start and stop processing tuples. Inside the tuple timer, there is an object called the TupleListIterator which is a type of design pattern known as an iterator [12]. The TupleListIterator consumes the tuple pipeline inputs. As the Iterator iterates over the list of items, the tuple timer only needs the times of when the iterator began and when the iterator stopped processing tuples.

The times at which the first and last tuples passed through the tuple timer are recorded in milliseconds and passed to the client using a single output as a combined string.

The tuple timer has an optional output that reproduces the tuples it has been passed. An optional output allows the tuple timer to be inserted arbitrarily between activities that pass tuples to each other.

One of the problems with the TupleTimer that was encountered when running the experiment was the TupleTimer was not timing the time at which the first tuple arrived. The TupleTimer was timing the time at which the processIteration functions was first called. Whilst writing the code the assumption was made that the processIteration function would not be called until the first tuple had been passed to the activity. This assumption could have been incorrect, as will be seen later in the results that the Tuples appeared to arrive before they had been dispatched.

The effects of inserting instrumentation to profile code can have negative side effects. They can make the code execution quicker, or slower. The code could execute quicker with instrumentation as there is increased buffering between the activities available, and slower as there is more work to be done by the CPU for each tuple.

When the timings are returned from the TupleTimer they are returned as part of the request status. This happens after the query has finished executing and has a small amount of data overhead that should only have a small effect the overall execution timings.
4.8.2 RandomTupleGenerator

Figure 6: Tuple Generator UML

As part of the project a random tuple generator class was implemented. The random tuple generator activity is used to create tuples. By having an activity that produces tuples, the overheads that are associated with SQL database access are avoided. Reducing the variability of the SQL access times when performance testing will help to isolate the performance of the parallel hash join algorithm, hence making it easier to analyse.

The random tuple generator has the following inputs: tuple count, minimum key, maximum key, sorted keys, number of string columns, size of string columns, number of integer columns. In addition to these inputs it is also possible to specify duplicate keys that are to occur and how often they are to occur. The random tuple generator has only one output, the tuple stream.

The random tuple generator has a tuple count that can be controlled. This allows experiments to be performed which can have different numbers of tuples. The tuple count parameter is important when looking for trends in the time taken to perform an operation. Varying the number of tuples operated on also allows experiments to be performed with different levels of join hit rate.
It is possible to specify a minimum and maximum key number so that overlapping ranges of keys can be generated. This allows the number of join hits that occur to be controlled. A variable join hit count allows experiments to be conducted where there is less data transfer coming from the servers than they receive.

The number of and size of string and columns allows the size of the tuple to be controlled. It may be significant to the data transfer time if a larger tuple is used to perform the experiment. A smaller sized tuple may also effect data transfer times. Having the option to specify a tuple size will allow the cost of the data transfer in relation to tuple size to be experimented with.

Having an option to specify duplicate keys allows the workload to be weighted so that one server has a higher workload that the others. Having duplicate keys that can be specified also allows experiments to be conducted which explore the time taken when the join occurring performs a product join which is a more computationally expensive task.

The random tuple generator has separate classes responsible for the construction of the key and the tuple. These can be seen in Figure 6. The key generator provides a base class which any key generator can inherit from. This makes the activity easier to extend as a new key generator only needs to be added to the package to generate a new series of keys. Four different key generators were created for use by the RandomTupleGenerator: SortedKeyGenerator, RandomKeyGenerator, UniqueRandomKeyGenerator and SpecifiedKeyGenerator.

The RandomKeyGenerator class produced keys using the Java Random class. This was problematic as it results in duplicate keys being produced. The UniqueRandomKeyGenerator is similar to the RandomKeyGenerator however the keys produced are only produced once for each cycle. The UniqueRandomKeyGenerator operates by having a collection of keys. Each key generated is removed at random from a collection of keys and returned to the TupleBuilder.

The SortedKeyGenerator is responsible for generating keys in a sorted order. When the SortedKeyGenerator reaches the end of the range of keys that it is to produce values for it loops round and starts from the beginning of that range again.

The SpecifiedKeyGenerator is responsible for generating duplicate keys in a controlled manner. The user can specify a key and the frequency at which they would like that key to occur this key will then be produced a user specified fraction of the time. Keys can be generated by the specified key generator in a sorted or unsorted order.

4.9 HashSplit

The hash split activity has three inputs: number of buckets, key to hash the values of and the tuple stream. When hashing values it calls the Java hashCode function on the attribute of the tuple that the join will be performed across.
4.10 Testing

The activities created were tested using unit tests before they were used as part of the experiment. This allowed problems in the code to be isolated before the code was used as part of a larger system. Throughout testing extensive use was made of a MockPipe object. The MockPipe allows objects to be passed to an activity as if they are coming from a pipe. Using the MockPipe simplified the development of the tests greatly.

4.10.1 TupleTimer

The TupleTimer was tested using a unit test that passed in tuples and checked they matched the ones passed out. To pass the tuples to the TupleTimer a MockPipe was used. The mock pipe then verified that all tuples in the pipe had been consumed by the activity or that they had been placed in the mock output pipe.

4.10.2 RandomTupleGenerator Testing

When testing the RandomTupleGenerator activity the properties of the returned stream were tested rather than the values of tuples themselves. The advantage of this approach was that the implementation could be modified and the tests would not have to be updated to support the modifications. This approach was also beneficial because the data produced was random.

One approach to testing a class that produces random data would have been to use a mock [13] random number generator. A mock object is a class that conforms to an interface and can be used in place of an actual class. When the unit tests were performed they could have had a mock object used in the place of an actual random number generator. This would have separated any errors in the random number generator from errors in the class under test. However, as the random number generator used is part of the Java runtime environment (JRE) it is safe to assume that it functions correctly. This approach would also have allowed the random numbers generated to be controlled, by using a mock class that returns a predefined series of numbers. This would have been useful if the class was being tested by comparing the returned tuples to a hard coded list of tuples.

The approach that was actually taken was to test the properties of the sets of tuples returned. The whole set of tuples that were returned would be looked at rather than the individual tuples that were returned. So, for example, the number of counts of a particular key are tested rather than hard coding a list of tuples and checking that the list returned by the activity was matching. Tests that were performed were to check the distribution of the random keys generated, and to check the order when the keys were supposed to be sorted.

4.11 Client Design

The client design allowed multiple experiments to be performed. It was flexible enough to allow a number of different join scenarios to be implemented. This was important
because the experiment needed to be carried out with a variety of different join parameters.

Initial design of the client was performed under considerable time pressures and as a consequence the design of the client could have been more elegant. Nonetheless it was anticipated that this design would be sufficiently robust to perform initial experiments.

The client was designed to allow configuration options to be passed to the client using a Java properties file. This allows the data required to configure the experiment to be easily transported from an automated script to the client. Alternatives to this would have been a lengthy list of command line options which is not only inconvenient but also error prone. Alternatives to the use of a Java properties file would have been a different file format such as XML, JSON or YAML.

Although XML is better for passing data between applications which have support for XML it was rejected. XML was rejected because it had too much functionality and required writing larger files than Java file format, in order to accommodate all of the markup language used to represent configuration options. For the purposes at hand the same is true of YAML. The use of the JSON file format would have required additional code to have been written in order to handle the processing of the file whereas Java has built in support for the Java properties file format. This makes Java properties file the easiest to utilise.

Using XML or JSON could have made integrating the client into an automated testing framework simpler as many existing tools and frameworks are available that handle JSON and XML formats. If the client had been designed as a web service it would have been possible to communicate with it using a format such as JSON. The client could have received network requests to perform a parallel hash join with specific parameters encoded using JSON. Because Ajax web clients often communicate using JSON, choosing this format would have been an appropriate selection in the case where the client was written as a web service.

YAML has been designed with the expectation that the user of a system could manually edit any file written using it and so that its constructs can be readily mapped to data types in Java [4]. Hence data written in the YAML format could be handled by a program written in Java without a lot of code to handle the YAML format being required. As the configuration file that is passed to the program is generated by a script there is not a requirement for it to be editable by hand so this feature does not make YAML a good format for the client application. The ability to map data types from YAML to Java data types would have been a helpful attribute to have had when dealing with a list of data in a configuration file. However, the mapping between data types and the YAML format is only achieved by using a more complicated API.

Given additional time the experiment could have been made more automated. This would have simplified repeat tests and after the initial development time would have reduced experiment setup times and allowed more experiments to be conducted. This could have been done by having a web application control the client and BonFIRE.
Results could then be fed back to this web application installed in a relational data base. The relational data base could be used for producing graphs of experiment results and would have made querying and analysing the data produced by the experiment easier.

This would have been particularly true if more instrumentation was added to the client and hence more statistics produced. Greater analysis could have been performed and thus greater insight gleaned from the results of the experiments if more instrumentation were added to the experiments. Unfortunately, given time constraints this proved unfeasible.

The circumstances that have been explored are similar to those that would be encountered in an astronomy use case presented by this project’s supervisor. This astronomy use case was a scenario that motivated further exploration of the effectiveness of parallel hash join, hence it is appropriate that the experiment performed was based on it.
Chapter 5

BonFIRE Evaluation

This section gives an overview of how usable BonFIRE was, what advantages it brought to the experiment and what disadvantages resulted from its usage. One of the aims of this project was also to run experiments on the BonFIRE test infrastructure and to establish how useful it was for running these experiments. This information would then be fed back to the BonFIRE team.

5.1 Early Adoption of BonFIRE

The BonFIRE project was said to allow an emulated network environment to be constructed. It was anticipated that this would allow experiments on the use of parallel hash joins to be conducted in a stable network environment. Thus the conditions of the network environment could be controlled. Hence it made sense to become an early adopter of BonFIRE because of the potential advantages for this project identified elsewhere in this report.

It would have been useful to see more tutorials on BonFIRE available on the internet. These might have facilitated using the Virtual Wall’s technology to emulate network conditions. It could have also reduced the time taken to learn how to use BonFIRE infrastructure. Dependence on supervision in producing scripts using BonFIRE could have been reduced had the documentation been better. The ‘Restfully’ tutorials provided were helpful because they outlined clearly and in detail how to interact with BonFIRE, via a command line interface. Knowing how to use the command line interface assisted in debugging some of the problems encountered whilst using BonFIRE. The tutorials were helpful in producing the necessary scripts, developing a conceptual understanding of the technical implementation of BonFIRE, as well as knowing how to recognise when a site had gone down.

Since there were no on-line support materials available to guide the use of the BonFIRE portal to create experiments, support through tutorial supervision was essential. It would be helpful if a step by step guide were available on-line, to guide new adopters through the creation of a BonFIRE experiment.
5.2 Alternative Approaches

Two alternative approaches to experiment setup were available. The first would have been to manually setup experiments and create one experiment for the entire period under which the experiments were to be conducted. The second would have been to use the save images approach, saving the images created by BonFIRE. These disc images would then be loaded whenever an instance of a server was created for the lifetime of the experiment.

The first of these options would have occupied large unused resources and been expensive for the BonFIRE project, hence wasteful of resources. A further problem with this approach lies in the requirement to input manually the details of the experiment and the resources utilised, on every occasion the experiment is run. Given how cumbersome these processes are they pose genuine challenges for repeated experiments and replication of the work. The second option, save to disc, is an alternative approach that could have been employed had the high initial setup time of the experiments been anticipated.

As an early adopter of BonFIRE, problems could be expected with the technology. There is a feature in BonFIRE, called post install, that allows the user to specify a script which will be downloaded and executed when the machine is started. The purpose is to allow the machines to be configured for a particular experiment. However whilst work was undertaken on the project the post install function did not work. Additionally BonFIRE sites and services were unavailable on two separate occasions in the week in which a script was being developed for experiments using BonFIRE. It was subsequently unavailable for two days of the following week.

5.2.1 Use of the Virtual Wall

The scientific quality of the experiment would have been enhanced had it been possible to use the Virtual Wall to emulate a network environment. It would have been beneficial to have on-line tutorials available detailing how to setup and configure network emulation options when using the Virtual Wall. Unfortunately, the Virtual Wall was more complicated to use than anticipated. This problem would have been eased had there been documentation on BonFIRE that explained how to setup an emulated network environment using the Virtual Wall been readily available on-line. Therefore, it was not feasible to conduct my experiments using the Virtual Wall.

5.3 BonFIRE Configuration

Initially, I used ‘dropbox’ [14], an online file repository service based on the cloud, to host the required files. The virtual machine instances would then download files from ‘dropbox’. This would mean that there was not a need to upload the files to each server whenever the experiment was begun and that the files would be available to BonFIRE over a higher bandwidth connection. One file downloaded would be a script that would configure the network host ready for the experiment.
Unfortunately, ‘dropbox’ downloads began to fail and I could no longer use the ‘dropbox’ repository to download images from. This failure was probably caused by a limit that dropbox imposes on the number of downloads available to a user using a public URL.

This meant uploading the files required to setup BonFIRE which made the execution of the scripts take longer than they would have done if the files were available from a fast internet file host.

5.4 Availability

Bonfire was unavailable on multiple occasions. The availability of BonFIRE was particularly inconvenient given the window of time available for conducting the experiment. Particular problems were encountered using the INRIA site where it appeared that an upgrade had been performed on that site whilst the individual responsible for the upgrade was away. This meant that the site had to be reverted to a previous version of the software. Other problems that were encountered with availability were connections being refused for unknown reasons when trying to run the ruby script.

The BonFIRE ‘restfully’ interface also had a habit of denying authentication whilst an experiment was running. This caused the experiment to abort and for the experiment to be cleaned up, thus the results gained by a particular run became lost.

Another problem that was encountered when using BonFIRE was at the setup stage of an experiment. Whilst setting up an experiment BonFIRE would wait 15 mins as the machines were starting. This would only happen when the number of machines being requested was greater than three or four.

5.5 Parallel Setup Example

For the experiment that was performed, several machines needed to be setup with the same configuration. To setup these machines a SSH connection was established to the machines. The connection and setup were all automated using a ruby script based on the example scripts available. The setup of machines was done sequentially but it could have been done in parallel. It would be helpful to see examples of the setup occurring in parallel using the Net:SSH:Multi class of ruby or some other parallel setup mechanism. We turn now to the discussion of the experiments and the data they produced.
Chapter 6

Experiment

This experiment was originally concerned with the throughput at different levels of network performance and the ability of the join algorithm to cope with inconsistent network conditions. Unfortunately, due to complications with the Virtual Wall, the experiment will now have to rely on network conditions between BonFIRE sites being stable and representative of a real scenario.

In the book ‘The ThoughtWorks Anthology’, James Bull [16] says that software performance testing should be done by first finding the maximum throughput, then finding the response time at lower throughput levels. In this context, throughput is defined as the average rate at which a tuple is delivered to a client.

Specific user queries deal with the system’s ability to handle multiple queries effectively whilst the system is under stress. In this experiment we do not consider the effects that concurrent processing has on user queries and are only interested in throughput. The scalability of the implementation will also be explored and results used to test the validity of our earlier pipeline model.

In order to identify any relationships between the time taken to execute a parallel hash join and the number of servers utilised, the following experiment was undertaken.

6.1 Program Execution

To execute the experiment a Java program was created that setup a work flow on OGSA-DAI to run a parallel hash join. This programme is called the client and connects to OGSA-DAI instances running on tomcat servers. The servers are all running an instance of tomcat with OGSA-DAI deployed on them. The client includes instrumentation that allows the time from submission to completion of the workflow to be measured.

As the chosen experiment infrastructure was BonFIRE, a ruby script was created to handle configuration. This script was responsible for setting up the BonFIRE infrastructure ready to run the experiment. This script also had responsibility for starting more scripts on each host that would prepare them by deploying OGSA-DAI, starting OGSA-DAI or configuring the client.
Once the infrastructure was configured, the client was run inside a script that executed the client multiple times using an increasing number of servers. The timings that the client returned are stored in a file.

Each of the tuple generators were configured to produce 100,000 tuples. These would be joined to produce 100,000 tuples, and the keys were in the range of 0 to 100,000. The first tuple generator produced tuples in a sorted order. The second tuple generator produced tuples in a random order which were unique. Only the key columns were generated which may have effected the realistic nature of this scenario. However generating the key column only results in a decrease in the amount of data transfer that has to occur, and hence should produce a faster program execution time.

This programme was run on a virtual environment distributed across three geographic locations: the University of Edinburgh EPPC and sites in France (Informatics Research Centre) and Germany (Stuttgart University). The client was based in Germany, and one of the servers responsible for producing data was based in France. The remaining servers were based in Edinburgh. All three sites were linked using a VPN connection that allowed virtual machines to access the private IP ranges used at other sites.

The first run attempted used one to three servers. After looking at the results, it appeared there could be a trend for the execution time to decrease as the number of servers was increased. To test this trend, a second run using four to five servers was executed. This also appeared to show a trend. To further investigate this a further run using six to eight servers was run. This showed that there was not a trend and that the execution time was variable. There were some ambiguities in the results received. In order to verify the data a second run was executed utilising one to three servers.

### 6.2 Experiment Results

This section presents the data that was produced by the experiments. It outlines any of the data results that are of particular interest. The data results that have significance will be outlined and possible explanations for why these results have occurred will be proposed.

#### 6.2.1 First Run Experiment Results

The data that is presented in Figure 7 is for the overall execution times for a parallel hash join measured in seconds.
**Figure 7: Parallel Hash Join Timings**

**Figure 8: Repeat One Parallel Hash Join Timings**
Figure 7 shows the times for the parallel hash join times for between one and eight servers. The times shown are the average times over 4 runs, the error bars show the min and max values for a data point. Following best practice guidance, five runs were undertaken and the first of these were discarded because they included the time for OGSA-DAI and tomcat to start-up. The times were calculated by subtracting the time at which the last tuple was received by the client, from any server, from the time at which the client submitted a workflow. That is the execution time.

It will be argued that, overall, the data shows that no firm conclusions can be reached about consistent timings of a parallel hash join. It had been hoped that because BonFIRE provided a more stable network environment than the internet at large that more consistent timings would have been achieved.

The first sample, with one to three servers joining data, suggested that, despite some anomalies, there was evidence of a possible trend. This potential trend was that an increase in the number of servers led to a decrease in the execution time. To establish whether or not this trend was present, further samples were taken with four to five and six to eight servers joining data. This data failed to demonstrate that there was a trend present. In order to establish whether the anomalies could be consistent, an additional sample of join times was taken. The outcomes of this sample will be reported subsequent to a detailed consideration of the first samples.
The data presented in Appendix A shows that the fastest execution time for a single server was 475s and the slowest time was 512s. It can be noted that when only one server was used, the average execution time was 498s. However, when a second server was added, instead of the time decreasing, as predicted, it increased, with a range between 534s and 607s with an average execution time of 573s. Hence it can be concluded that the introduction of a second server always slows execution time in comparison with a single server. It is suspected that the data transfer costs are dominating the execution time.

When a third server was introduced, the execution times ranged between 523s to 585s with an average execution time of 560s. The latter execution time was slower than the average time for two servers, this demonstrated the high level of variation in the results.

After three servers are used there is a dip in execution times for four and five servers. However, after this dip there is an increase in execution time for six, seven and eight servers compared two four and five servers. The results were gathered by executing the experiment in three batches. The first batch was one to three, the second four to five servers and the final batch was six to eight servers. This was done as problems with BonFIRE prevented the experiment from being executed as one batch.

The results may be showing a sudden decrease in execution followed by increase in execution due to the usage of separate batches to generate them. This would suggest an unstable system, the results of which cannot be relied upon. The results of the batch that generated timings for four and five servers may be quicker than all the other batches due to favourable network condition. Alternatively it could be due to better virtual machine placement within the test infrastructure. This could occur when virtual machines are placed on different host nodes. Another possibility that could account for the improvement of execution time is more favourable access to resources such as disk, memory and CPU.

**6.2.2 Second Query Set Experiment Results**

After the first samples, another sample was taken to see if these results were reproducible. The sample was for one to three servers and produced different results from the previous sample. The results of the second sample can be seen in Figure 8. They showed that a reduced time for execution had occurred using one to three servers. This suggests that the results produced are not repeatable or do not have a high enough level of accuracy for any performance trends to be identified. The results of the first and second query batch can be seen in Figure 9. From this it can be gathered that the system does have a high enough level of stability for this experiment. It is not possible to trust that results produced would change given changes in experiment parameters rather than changes in the infrastructure such as those discussed earlier.
6.2.3 Setup Times

![Setup Time Graph](image)

**Figure 10: Run One Setup Times**

Analysing setup times is useful as it gives some indication of how much time is consumed with the setup overhead of the parallel hash join. If setup times are significant and increase with the number of servers, it will be possible to say that setup times are a part of what appears to be an increase in time when using a parallel hash join. Using the available data it should be possible to analyse setup times. Unfortunately, due to clock synchronisation problem this was not possible. The setup times are calculated by subtracting the time at which the first tuple is generated from the time at which the workflow is submitted. Because of clock synchronisation and possibly an error in the code, one shows a negative time for setup whilst the other shows a positive time, the results of which can be seen in Figure 10. This graph shows that the number of servers makes a slight difference to the setup times. This could be because of unanticipated instability in the bonfire framework. Alternatively this could be due to an error by the programmer in which the clock synchronisation problem is between sites at INRIA and Stuttgart University. The client was based at Stuttgart, whilst server 1 on Figure 10 was based at the INRIA site, and server 2 was based at the EPCC site in the UK.

An attempt was made to prove the existence of a difference in clock times, however, this failed. The BonFIRE servers are synchronised using NTP, and should all have the same time because of it.
6.2.4 First Tuple Time

The time at which the first tuple arrives at the client is often an important metric for a user as in a streamed environment, it dictates the time at which they can start processing data. To find the rate at which the first tuple arrived, the longest time that a server took will be subtracted from the submission time of the workflow.

For one server it can be seen that the quickest time was 21s and the slowest time 22s with an average time of 22s. As usual, the first time taken is discarded due to the lack of optimisation by the JRE and the start up time of OGSA-DAI.

![First Tuple Time](image)

**Figure 11: First and Second Query Batch First Tuple Arrival Time**

Figure 11 shows the arrival times of the first tuple when using one to three servers. These results are for the first and second runs of the experiment. The data shows comparable times for the parallel hash join to produce the first tuple. This means that there is some consistency in the results produced using the BonFIRE infrastructure.

It is possible to speculate that this indicates that the performance of the virtual machines could be consistent and the variation in results for the last tuple, discussed later, could be caused by network conditions. Use of the Virtual Wall would enable this speculation to be tested.

The time of 21s for the first tuple to be produced can be attributed to the nature of the data that is supplied to the join. One side of the data supplied to the join will come from the tuple generator that produced sorted tuple, whilst the other side of the data comes from the tuple generator that produces unsorted tuples. This would explain the low hit rate of matching tuples.

6.2.5 Cause Of Variability

The BonFIRE infrastructure was unable to produce reliable results at the time the experiments were conducted. However, if the BonFIRE infrastructure had produced
reliable results, the statistics could be used to determine workflow setup times. The
time of the first tuple and time of the last tuple are important since they can have an
impact on when the client can begin and finish processing data. These timings also
have implications for the theoretical model that was constructed of how a pipeline
would behave if run in parallel. However, all the evidence that has been gathered
suggests that the pipeline model discussed earlier appears to be too simple.

The BonFIRE platform was able to provide the infrastructure to run the software I had
developed. However, the results that my experiment produced were unstable and had
ambiguities within them. This suggests that although it is able to run the software and
provide an environment for testing that the software could scale, performance testing of
the software is currently beyond the scope of the BonFIRE project.

The BonFIRE platform uses virtual machines to provide its infrastructure. This could
be the reason that results produced were unstable. When running software on a virtual
machine there may be load balancing issues such as multiple users using the same
server. This could effect the amount of network bandwidth available and the I/O
response times for virtual disks. BonFIRE documentation [15] appears to state that
one virtual machine may be allocated one CPU and that this could vary on different
sites. Although some sites guarantee that one CPU will be allocated to each VM, this is
not true of all sites as the Wiki document states. However, it appears to be true of the
three sites selected for this experiment. When sharing the CPU resources between
virtual machines, the virtual machines cannot be expected to always react in the same
time as the load on one machine may effect the performance of another.

Another problem with using virtual machines for performance testing software is that
there is an unknown overhead produced by the host operating system. If the host
operating system is performing backups or preparing another virtual machine to run
then the load that this creates could reduce the performance of any virtual machines
running on that host.

The predictability of the manner in which virtual machines operate is also a problem
because they are sharing resources such as the network interface card and the memory.
By sharing the network interface card there is a limit placed on the response time for
the network. When one virtual machine downloads a large file this prevents another
virtual machine from downloading a large file at the same time.

Overall there appears to be a lot of variability in the infrastructure that makes it
unsuitable for performance testing this type of application. However if the Virtual Wall
had been used this situation may have been avoided. The Virtual Wall provides one
physical machine per virtual machine this would avoid any contention for resources. It
also provides an emulated network. Having a more stable and controlled environment
which the Virtual Wall provides, would have been a more suitable environment for
conducting this experiment.

Some other problems which were encountered when using the BonFIRE infrastructure
to time the experiment were clock synchronisation issues. This was due to the clocks on
different computers not being synchronised. This made it impractical to compare the
time that an event occurred on one machine to the time of an event on another. For two
query sets, the time for a workflow being submitted superseded the time at which the workflow was submitted. This can be seen in Figure 10.

The workflows setup times could increase with the number of servers added. This could have caused the increase in time that was seen. However, a trend could not be seen in times of the parallel hash join hence it is not possible to say if setups times would increase at a constant rate with an increase in the number of servers. It could be guessed that when processing 100,000 tuples, the workflow setup time would not be a large factor in the increase in time. Unfortunately, because of clock synchronisation problems it was not possible to accurately determine the setup time of the workflow.

By looking at the time that the first tuple took to pass from the tuple generator to the data source it is possible, theoretically, to determine the speed which a tuple could pass through the pipeline unhindered by the tuples ahead of it. This would be the fastest possible speed that a tuple could pass through the pipeline. This is important as it shows the peak rate of the pipeline. However, due to Java compiler optimisations that occur automatically whilst the program runs, this may not be the optimum. This is because the Java compiler can automatically optimise functions that are used more often than others.

By looking at how long the last tuple took to pass from the tuple generator to the data source, the time that a tuple takes to pass through the pipeline, when it is blocked, could be determined. If a theoretical model was used to model the pipeline, and each stage of the pipeline processed tuples using exactly the same amount of time, then by comparing the time that the last tuple took to pass through the pipeline to the time that the first tuple took to pass through the pipeline, the time which a tuple spends waiting to be transferred to the next stage of the pipeline could be determined. Alternatively, if it took less time it would be an indication of any Java automated optimisations that had occurred whilst the code was running.

If accurate statistics about the times which the parallel hash join took to run could have been obtained then they could have been used to test if the model produced in the theoretical section of this dissertation was or was not valid.

Although much of the instability in timings could be attributed to BonFIRE, it could also be attributed to Java. Because OGSA-DAI is a complicated program written in Java it is difficult to performance profile it without understanding how the Java compiler and the Java Runtime Environment (JRE) optimises code. If more time had been available to understand the way in which the Java compiler and JRE optimise code, then more conclusions could be drawn about their effects on the performance of OGSA-DAI.
Chapter 7

Conclusion

In this chapter I will discuss what can be concluded from the work undertaken in this dissertation. It will also cover the further work that would need to be performed for further analysis to be undertaken and the experiments that could produce this analysis.

7.1 Summary

Overall, a clear specification for the project was produced and the scope of the project was restricted to that which was relevant to the specification. Research was performed into existing work on parallel hash joins, adaptive joins and relational databases, which were all relevant to the scope of the dissertation. A deadlock was fixed in OGSA-DAI that occurred when joining data using a parallel hash join. Code that was relevant and allowed the experiments to be conducted was produced. This code had the potential to be used to conduct further experiments which explore the parallel hash join in more detail. The experiments were successfully run even though the results produced were not what were expected. This dissertation has allowed for a wide variety of future work to be conducted that could make the use of a parallel hash join worthwhile and explain its performance.

It can be concluded that the time taken to conduct a parallel hash join is greater than the time taken to conduct a serial parallel hash join. Although the explanation for this cannot be analysed in detail it appears that this is because of the overheads. More work would need to be done to analyse BonFIRE and find what the level of accuracy that it provides is. In order to be able to say what the overheads inside OGSA-DAI were that caused the problems, more instrumentation would need to be added to the experiment and it would need to be performed again.
7.2 Future Work

This section covers several ideas that could be used to help draw further conclusions about the effectiveness of parallel hash joins. It will also hope to address ways of quantifying effectiveness of the parallel hash join. It also covers ideas that could be used to test the reliability of the BonFIRE platform.

7.2.1 Additional Instrumentation

The first piece of further work that needs to be done is to conduct the experiment again with further instrumentation. When code or functionality is added to an application for testing it is called instrumentation. Care has to be taken when adding instrumentation to the pipeline. This is particularly true when doing performance testing. If the instrumentation is too data intensive, it could become a bottle neck in the processing. To test the implementation against the model that was created, each stage in the pipeline would have to be timed. There was not enough instrumentation used when the previous experiments were conducted to gain a detailed understanding of the times that the stages of the pipeline took.

There are two approaches that could be used to time the individual pipeline stages:

- A timer could be placed between each stage in the pipeline.
- To start the pipeline with one stage, have this stage timed, then a stage could be added and the entire pipeline timed. This could be repeated until the entire pipeline is complete. If the difference in execution time between a pipeline with n stages and n+1 stage were calculated, this would give the time of the n+1th stage.

The best approach for further analysis is the first option outlined above. The timer stage that could be inserted should execute faster than all other stages in the pipeline. This means that the results will not be modified substantially by its presence in the pipeline. It is also simpler to implement as less code will need to be written. If the first approach does not prove to be satisfactory then the second approach could be tried.

If the experiment were to be repeated, it could have a timer placed between each stage of the pipeline. The tuple timers would report the time that each stage took by measuring when the first and last tuples passed through them. By obtaining the times that each stage in the pipeline took, it would be possible to determine where the bottlenecks were. After the bottlenecks have been established, it would be possible to explain why the performance of the parallel hash join was less than was expected.

By adding more instrumentation it would also be the case that the timings taken could be calculated independently of any other node. This would prevent the problem that occurred with comparing the times taken on the client to the times taken on a server which were encountered when analysing the results of the experiments.

Currently BonFIRE cannot provide timings at the level of accuracy needed to determine if the use of a parallel hash join is beneficial with absolute certainty. Despite the potential problems and their possible explanations as outlined above, the
experiment failed to show an overall performance improvement from using a parallel hash join.

### 7.2.2 Virtual Wall Emulated Network

Once further instrumentation has been added, another experiment that explores the performance of the join activity in relation to network conditions could be performed. By using the Virtual Wall an analysis of the impact of network conditions on the parallel hash join could be performed. Network rates could begin high and then be reduced slowly to degrade the performance. The effect that this has on the parallel hash join could then be compared to the effect that this has on a standard join.

The delay for one data producer could be increased to see what impact this has on join rates. This would be particularly relevant because a multi-threaded adaptive join algorithm is being used to perform the joins between data streams. The variance in network conditions will show what performance benefits can be gained from the use of a multi-threaded adaptive join. It could be expected that in conditions where one network connection is slower than another that the multi-threaded join algorithm will offer some improvement to the speed at which a join occurs.

I would have assumed that in this scenario, delay and packet loss would have similar impacts on the results, as packet loss is only serving to increase the delay that occurs in delivery of packets. I would assume that changing the bandwidth will also increase the delay. The presumption I am making is that network transport times will be effected by bandwidth, delay and packet loss in the same way.

To conduct this experiment the number of servers used to execute the parallel join will be increased until a trend in time can be seen. If these results showed any trends then these could be used to analyse the parallel hash join performance further and it would be possible to establish if the network could account for the poor performance gained.

### 7.2.3 Weighting parallel hash joins

One option for future work is to experiment with using a weighted hash join. When a weighted hash join is used, the load between servers can be modified so that uneven division of work occurs. This would allow a more optimal distribution of work which would in turn reduce the time taken to execute a parallel hash join. The distribution of the load could be based on a number of factors.

One of these factors is the network delay between sites if network conditions allow large volumes of data to pass quickly between nodes. A parallel hash join could be weighted to allow processing to occur on nodes which are connected quickly to the source node. If the connection causes data to pass slowly between nodes, the parallel hash join could be weighted to perform more processing on the source node. Additional experiments would need to be conducted to establish what ratio of network bandwidth to tuples would be optimal.
The second experiment would look at the data transfer times between network nodes. In order to do this, the two variables that would be modified are the number of duplicate joins that occur and the number of joins that do not occur.

By increasing the number of duplicate joins that occur, the amount of data transferred between the joining node and the client will be increased. If the number of tuples that have no match is increased then the amount of data transferred between the join node and client decreases.

In order to change the number of joins that take place, the range of overlapping keys generated by the tuple generator can be modified. To change the number of duplicate keys generated by the tuple generator, the number of rows generated can be made larger than the range of key values generated.

7.2.4 Network Latency Exploration

It has already been noted that according to Tannenbaum [17], the majority of network transfer time can be attributed to latency rather than wire speed. It could be worth investigating how much of the time the parallel hash join takes can be attributed to latency that occurs in packaging and unpackaging data for transfer over the network. If this is found to be high, more effective methods of transferring data internally, between OGSA-DAI servers could be explored and in the case of high latency, its reduction could greatly improve the performance of the parallel hash join.

It may also be worth considering the size of data that has been chunked by OGSA-DAI for transmission over the network and experiments could be conducted to determine what the optimum size for data chunks might be. Subject to the limitations of BonFIRE identified earlier in this report, experiments on optimum size could be conducted using the BonFIRE infrastructure.

Variation in the size of chunks can be achieved by the use of a tuple generator to generate a large volume of tuples and subsequently varying the size of the chunks of these tuples by adjusting the parameters of OGSA-DAI to vary the size of chunks sent over the network.

The variations in network conditions are bandwidth, packet loss and delay. These are all issues that could be subjected to systematic investigation using BonFIRE.

If network transfer latency times are quick then it could be useful to have a smaller buffer as less time will be spent by the server buffering data before it can be sent across the network. This decreases the time taken by the network transfer stage of the pipeline. If the buffer is smaller than the rate at which data can be transferred then the pipeline would be blocked.

If network transfer latency times are slow then having a buffer that is too small for the bandwidth will waste time as the tuples will become queued in the pipeline waiting to be sent and processing will stop. When the latency times are slow it may be worth having larger chunks as data could be buffered which would speed up the processing of other parts of the OGSA-DAI server.
By using BonFIRE instances with different processing power and memory, it will be possible to test whether the bottlenecks in data transfer are caused by the network conditions or the time taken by the machine to package the data.

This will help to further work produced by Zhu [4], where he could not produce accurate statistics in relation to parallel hash joins because of the instability of performance of Java.

Another option for further experiments is to explore variation in the amount of data stored in a table. If a table contains a high volume of data and a large number of big tuples, then transferring this data from one site to another will be an expensive operation and it may be more cost effective to process this data locally and closer to the client. This could be established through experimentation.

7.3 The Consistency And Reliability of BonFIRE Results

To determine the possible causes of the inconsistent results a number of things need to be considered. The two key areas for consideration are the performance characteristics of Java and BonFIRE. It is important to know how consistent the performance that either of these platforms offer for the experiments conducted. In order to understand the way that the performance of BonFIRE behaves, a benchmarking suite could be tested or written.

Things that would need to be considered by the suite are how many virtual hosts are running on a particular server and if the performance of virtual servers are isolated from each other. It would also be worth considering if the performance of the virtual machines was isolated from the host. In order to answer these questions, more knowledge would be needed about BonFIRE design and mechanics and how BonFIRE is configured at different sites.

To test if the performance of the virtual hardware is consistent, a host could have one virtual server instantiated on it. A benchmarking suite which is able to test the memory access and CPU throughput could be run. An example benchmarking suite that could be used is the SPECjvm2008 [18]. This suite is designed to test memory and CPU usage. Multiple virtual machines could then be instantiated on the same server and the test suites repeated. This would reveal if the amount of load that a host had would be reflected in the performance of the virtual machines it was running.

A solution to check the performance of Java would be to use a Java based benchmarking suite on a BonFIRE server. The suite could be run on different sites when the sites are under different loads. Multiple BonFIRE machines could be started at a site. The experimenter could start with one server then gradually increment the number of servers used. Each server would then have a benchmarking suite run on them to see what their current performance is. If the performance varied between servers, this would should how reliable the BonFIRE platform is for testing.
7.3.1 Instance Size

Because virtual machines on the cloud are available as a number of different instances, it is possible that the size of instance that was chosen to run the experiment on was too small. If this was the case then the virtual machine would have been unable to cope with the amount of stress put on it. This could have caused slow memory access and disk access times, and these would have affected the performance of the OGSA-DAI server. This would have meant that the performance would have been below what could be expected of an OGSA-DAI server under normal conditions. It would be better to have a virtual machine with more server capacity than required. In this way the performance would not be bottlenecked by the virtual machine but instead would be bottlenecked by the implementation of the parallel hash join.
### Appendix A

## Test Data

### Query Batch 1

<table>
<thead>
<tr>
<th>Submit Time</th>
<th>Last Tuple Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Server 1</td>
</tr>
<tr>
<td>1312485852495</td>
<td>1312486341979</td>
</tr>
<tr>
<td>1312486345581</td>
<td>1312486846133</td>
</tr>
<tr>
<td>1312486848605</td>
<td>1312487233833</td>
</tr>
<tr>
<td>1312487325887</td>
<td>1312487837446</td>
</tr>
<tr>
<td>1312487845325</td>
<td>1312488341011</td>
</tr>
<tr>
<td></td>
<td>13124888361230</td>
</tr>
<tr>
<td></td>
<td>1312489567763</td>
</tr>
<tr>
<td></td>
<td>1312490151398</td>
</tr>
<tr>
<td></td>
<td>1312490909796</td>
</tr>
<tr>
<td></td>
<td>131249141850982</td>
</tr>
<tr>
<td></td>
<td>1312491708596</td>
</tr>
<tr>
<td></td>
<td>1312492429434</td>
</tr>
<tr>
<td></td>
<td>1312492429398</td>
</tr>
<tr>
<td></td>
<td>131249243017554</td>
</tr>
<tr>
<td></td>
<td>131249243018005</td>
</tr>
</tbody>
</table>

### Query Batch 2

<table>
<thead>
<tr>
<th>Submit Time</th>
<th>Last Tuple Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Server 1</td>
</tr>
<tr>
<td>1312539222283</td>
<td>1312539726075</td>
</tr>
<tr>
<td>1312539734354</td>
<td>1312540249727</td>
</tr>
<tr>
<td>1312540258274</td>
<td>1312540768542</td>
</tr>
<tr>
<td>1312540778019</td>
<td>1312541288690</td>
</tr>
<tr>
<td>1312541297966</td>
<td>1312541807108</td>
</tr>
<tr>
<td>1312541817577</td>
<td>1312542298467</td>
</tr>
<tr>
<td>1312542303011</td>
<td>1312542838984</td>
</tr>
<tr>
<td>1312542849609</td>
<td>1312543358152</td>
</tr>
<tr>
<td>1312543375586</td>
<td>1312543863647</td>
</tr>
<tr>
<td>1312543871982</td>
<td>1312544368259</td>
</tr>
</tbody>
</table>
## Query Batch 3

<table>
<thead>
<tr>
<th>Submit Time</th>
<th>Last Tuple Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Server 1</td>
</tr>
<tr>
<td>13125611039190</td>
<td>1312561645261</td>
</tr>
<tr>
<td>1312561644636</td>
<td>1312562210479</td>
</tr>
<tr>
<td>1312562223452</td>
<td>1312562774413</td>
</tr>
<tr>
<td>1312562787655</td>
<td>1312563328241</td>
</tr>
<tr>
<td>1312563341401</td>
<td>1312563891383</td>
</tr>
<tr>
<td>1312563891401</td>
<td>13125644448793</td>
</tr>
<tr>
<td>1312564442185</td>
<td>1312565500759</td>
</tr>
<tr>
<td>1312565000445</td>
<td>1312565548745</td>
</tr>
<tr>
<td>1312565554263</td>
<td>1312566099822</td>
</tr>
<tr>
<td>1312566113257</td>
<td>1312566652823</td>
</tr>
<tr>
<td>1312566661378</td>
<td>1312567206852</td>
</tr>
<tr>
<td>1312567217489</td>
<td>1312567767804</td>
</tr>
<tr>
<td>1312567782419</td>
<td>1312568333109</td>
</tr>
<tr>
<td>1312568448468</td>
<td>1312568899310</td>
</tr>
<tr>
<td>1312568910672</td>
<td>1312569456604</td>
</tr>
</tbody>
</table>

## Query Batch 1 Repeat 1

<table>
<thead>
<tr>
<th>Submit Time</th>
<th>Last Tuple Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Server 1</td>
</tr>
<tr>
<td>1312670342399</td>
<td>1312670366412</td>
</tr>
<tr>
<td>1312670777543</td>
<td>1312671200179</td>
</tr>
<tr>
<td>1312671207552</td>
<td>1312671628843</td>
</tr>
<tr>
<td>1312671631382</td>
<td>1312672054142</td>
</tr>
<tr>
<td>1312672056626</td>
<td>1312672478647</td>
</tr>
<tr>
<td>1312673446185</td>
<td>131267398509</td>
</tr>
<tr>
<td>1312673993513</td>
<td>1312674520709</td>
</tr>
<tr>
<td>1312674523441</td>
<td>1312675055180</td>
</tr>
<tr>
<td>1312675066449</td>
<td>1312675577090</td>
</tr>
<tr>
<td>13126755889353</td>
<td>1312676092855</td>
</tr>
<tr>
<td>13126772005730</td>
<td>1312677765929</td>
</tr>
<tr>
<td>1312677774488</td>
<td>1312678325048</td>
</tr>
<tr>
<td>1312678342448</td>
<td>1312678808077</td>
</tr>
<tr>
<td>1312678900740</td>
<td>1312679483832</td>
</tr>
<tr>
<td>1312679497120</td>
<td>1312680043022</td>
</tr>
</tbody>
</table>

47
Appendix B  

Code Structure

Tuple

To represent a tuple in OGSA-DAI there are two objects, an abstract Tuple interface and a concrete implementation of a tuple called SimpleTuple. A tuple can contain attributes of different types.

Metadata

Metadata is used to represent the types of the tuples so the client or server knows the type of the attribute. This is because the Tuple object does not contain any information about the types of the attributes.

A ColumnMetadata object is used to represent the individual attributes of a Tuple object. A TupleMetadata object is used to wrap a collection of column metadata objects. This is then wrapped by a MetadataWrapper, before it can be handled by other activities in the pipeline. Column and Tuple metadata contains links to information about the tables or data sources that they were retrieved from. Alternatively, this information may not be included.

Files

The server side classes are all in the package uk.org.ogsadai.parallelhashjoin. With the exception of the RandomTupleGenerator and RandomTupleGeneratorActivity class, all classes relating the RandomTupleGenerator reside in the uk.org.ogsada.parallelhashjoin.tuplegenerator package. The modified version of the pipeline join resides in the uk.org.ogsadai.parallelhashjoin.pipelinedjoin package.
References


