A BitTorrent-Based

Peer-to-Peer Database Server

Thesis by

John Colquhoun

School of Computing Science

In Partial Fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

Newcastle University

Newcastle-upon-Tyne, UK

2008

ACKNOWLEDGEMENTS

I would like to thank everyone who has offered me advice and support during my PhD especially my supervisor Paul Watson and my Thesis Committee, Pete Lee and Nigel Thomas.

ABSTRACT

Database systems have traditionally used a Client-Server architecture, where clients send queries to a database server. If the data proves popular, the server may become overloaded, leading to clients experiencing an increase in query response time.

In the domain of file-sharing, the problem of server overloading has been successfully addressed by the use of Peer-to-Peer (P2P) techniques in which users (peers) supply files – or pieces of files – to each other. This thesis will examine whether P2P techniques can be applied successfully in a database environment. It will introduce the Wigan Peer-to-Peer Database System, a P2P database system based on the popular BitTorrent file-sharing protocol.

The potential benefits of a P2P database system include performance and scalability; allowing peers to answer each others' queries will reduce the load on the database server and so could overcome the problem of a busy server becoming overloaded. Other potential benefits are fault tolerance and cost reduction.

The Wigan architecture is introduced in this thesis, firstly by describing the BitTorrent algorithms and then by discussing how these algorithms must be modified for use in a database system. Experiments carried out on a simulator of Wigan are analysed in order to determine factors which affect its performance. These allow the identification of scenarios where Wigan could outperform a traditional database server.

Further extensions to the Wigan architecture are discussed in this thesis, including possible means of handling data updates.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS II

ABSTRACT III

TABLE OF CONTENTS IV

LIST OF FIGURES VI

LIST OF TABLES VIII

1 INTRODUCTION 1

1.1 INTRODUCTION 1
1.2 AIM AND OBJECTIVES 2
1.3 THE DIFFERENCES IN REQUIREMENTS BETWEEN FILE-SHARING AND A P2P DATABASE 3
1.4 POTENTIAL BENEFITS OF A P2P DATABASE 4
1.5 MOTIVATING SCENARIOS FOR A P2P DATABASE 5
1.6 SUMMARY 7

2 RELATED WORK 8

2.1 INTRODUCTION 8
2.2 P2P COMPUTING 8
2.3 P2P FILE-SHARING 10
2.4 BITTORRENT 11
2.5 ANALYSES OF BITTORRENT 18
2.6 DYNAMIC CACHING 22
2.7 P2P DATABASE SYSTEMS 24
2.8 CONCLUSIONS 39

3 THE WIGAN ARCHITECTURE 40

3.1 INTRODUCTION 40
3.2 SHARING DATA 40
3.3 THE WIGAN COMPONENTS 42
3.3.1 The Seed 42
3.3.2 The Peers 46
3.3.3 The Tracker 46
3.4 DOWNLOADING AND UPLOADING 58
3.4.1 Downloading 58
3.4.2 Changing the Piece Structure 66
3.4.3 Uploading 72
3.4.4 An Empty Result Set 74
3.4.5 Examples of the Wigan System in Operation 74
3.4.6 The Choking Algorithm 77
3.4.7 Disconnecting 78
3.5 JOINS 78
3.6 THE EFFECT OF PEER FAILURE 86
3.7 SUMMARY 88

4 EVALUATION 89

4.1 INTRODUCTION 89
4.2 SIMULATING THE WIGAN SYSTEM 89
4.3 WORKLOAD 92
4.4 INVESTIGATING THE EFFECT OF THE NUMBER OF PIECES ON RESPONSE TIME 94
4.5 SINGLE-TABLE, SMALL QUERY EXPERIMENTS 99

4.6 MULTI-TABLE, SMALL QUERY EXPERIMENTS 111
4.7 CHOOSING A JOIN ALGORITHM 114
4.8 GREATER PARALLELISATION OF SEED DOWNLOAD 119
4.9 COMPLETELY RANDOM QUERIES 124
4.10 CONCURRENT PIECE REQUESTS 128
4.11 THE EFFECT OF CHANGING THE PIECE SIZE 134
4.12 REPLICATING THE SEED 139
4.13 EARLY RELEASE OF DATA 144
4.14 CACHE-WARMING 152
4.15 THE TRACKER 158
4.16 SUMMARY 159

5 CONCLUSIONS AND FUTURE WORK 161

5.1 INTRODUCTION 161
5.2 CONCLUSIONS 161
5.3 FUTURE WORK 163
5.3.1 Updates 163
5.3.2 Partial subsets 168
5.3.3 Dynamically choosing a join algorithm 172
5.3.4 Cost models and Optimisation 173
5.3.5 Indexing 175
5.3.6 Further Investigations Into Piece Size 175
5.3.7 Adapting the number of concurrent downloaders 176
5.3.8 Fault tolerance 177
5.3.9 Security 178
5.4 CONTRIBUTIONS OF THIS RESEARCH 180

REFERENCES 183

LIST OF FIGURES

FIGURE 1.1 – A CLIENT-SERVER DATABASE SYSTEM 1
FIGURE 1.2 – P2P DATA-SHARING 2
FIGURE 2.1 - THE LIFECYCLE OF A BITTORRENT PEER 15
FIGURE 2.2 – TWO PEERS UPLOADING AND DOWNLOADING FROM EACH OTHER 16
FIGURE 2.3 – THE BITTORRENT WORKLOAD MODEL 21
FIGURE 3.1 – A TABLE IS DIVIDED INTO PIECES 41
FIGURE 3.2 – A QUERY (WHITE CIRCLE) AND AN ADVERT (BLACK CIRCLE) WHICH DO NOT MATCH 47
FIGURE 3.3 – A QUERY (WHITE CIRCLE) WHICH IS A PARTIAL SUBSET OF AN ADVERT (BLACK CIRCLE) 47
FIGURE 3.4 – A QUERY (WHITE CIRCLE) WHICH IS A PROPER SUBSET OF AN ADVERT (BLACK CIRCLE) 48
FIGURE 3.5 – A QUERY AND ADVERT WHICH ARE IDENTICAL 49
FIGURE 3.6 – A DOWNLOADER'S MESSAGE TO THE TRACKER 50
FIGURE 3.7 – THE TRACKER'S DATABASE FOR COLUMNS INFORMATION, WITH ONLY THE SEED AND A SINGLE PEER ADVERTISING 51
FIGURE 3.8 – THE TRACKER'S DATABASE FOR CONDITIONS WITH ONLY THE SEED AND A SINGLE PEER ADVERTISING 51
FIGURE 3.9 – THE RESULT OF THE INITIAL SELECTION PROCESS 52
FIGURE 3.10 – THE TRACKER'S RETURN MESSAGE TO A DOWNLOADER 53
FIGURE 3.11 – THE TRACKER'S DATABASE FOR COLUMNS LATER IN THE DOWNLOAD PERIOD 54
FIGURE 3.12 – THE TRACKER'S DATABASE FOR CONDITIONS LATER IN THE DOWNLOAD PERIOD 54
FIGURE 3.13 – THE RESULT OF THE INITIAL SELECTION PROCESS 55
FIGURE 3.14 – CHOOSING 50 ADVERTS FROM THE SELECTION 57
FIGURE 3.15 – THE COMPLETION MESSAGE SENT BY PEERS TO THE TRACKER 57
FIGURE 3.16 – A DISCONNECTION MESSAGE 58
FIGURE 3.17 – A DOWNLOADER CONTACTS THE TRACKER AND RECEIVES A PEER LIST 59
FIGURE 3.18 – THE FORMAT OF HANDSHAKE REQUESTS AND RESPONSES 60
FIGURE 3.19 – A PIECE CONFIRMATION MESSAGE 60
FIGURE 3.20 – A PIECE REQUEST MESSAGE 61
FIGURE 3.21 – A MESSAGE CONTAINING A PIECE OF DATA 61
FIGURE 3.22 – A LACK OF INTEREST MESSAGE 62
FIGURE 3.23 – A CHOKE MESSAGE 62
FIGURE 3.24 – A WIGAN DOWNLOADER'S RESPONSE TO RECEIVING A CHOKE MESSAGE 63
FIGURE 3.25- A DOWNLOADER RECEIVES A NEW PEER LIST FROM THE TRACKER 65
FIGURE 3.26 - THE LIFECYCLE OF A BITTORRENT PEER 67
FIGURE 3.27 – THE LIFECYCLE OF A WIGAN PEER 72
FIGURE 3.28 – AN UPLOADER RECEIVES A PIECE REQUEST 73
FIGURE 3.29 – A SIMPLE QUERY REQUEST 75
FIGURE 3.30 – A DOWNLOADER CHOKED BY THE SEED AND TIMING OUT 76
FIGURE 3.31 – THE SEED UNCHOKES THE DOWNLOADER BEFORE THE TIMEOUT PERIOD ELAPSES 77
FIGURE 3.32 – CHOKE AND UNCHOKE MESSAGES 78
FIGURE 3.33 – THE UNCACHED JOIN ALGORITHM IN OPERATION 80
FIGURE 3.34 – THE CACHED JOIN ALGORITHM IN OPERATION 84
FIGURE 4.1 – THE TPC-H BENCHMARK DATABASE TAKEN FROM [74] 93
FIGURE 4.2 – THE INCREASE IN THE NUMBER OF PIECES RETURNED BY THE QUERIES 97
FIGURE 4.3 – THE INCREASE IN RESPONSE TIME FOR EACH QUERY 97
FIGURE 4.4 – THE NUMBER OF PIECES PLOTTED AGAINST RESPONSE TIME 98
FIGURE 4.5 – A TYPICAL BITTORRENT WORKLOAD MODEL 101
FIGURE 4.6 – THE CONSTANT WORKLOAD, A CONTINUAL STREAM OF QUERIES 102
FIGURE 4.7 – THE FLASHCROWD, INITIAL INCREASING INTEREST AND THEN A CONTINUAL STREAM OF QUERIES 103

FIGURE 4.8 – AVERAGE RESPONSE TIMES FOR THE TWO WORKLOADS 104
FIGURE 4.9 – THE PERCENTAGE OF PEERS THAT TIMED OUT AND HAD TO RE-CONTACT THE TRACKER 106
FIGURE 4.10 – AVERAGE RESPONSE TIME COMPARED TO START TIME (RANDOM QUERIES) 107
FIGURE 4.11 – AVERAGE RESPONSE TIME COMPARED TO START TIME (REPEATING QUERY) 108
FIGURE 4.12 – AVERAGE RESPONSE TIME COMPARED TO START TIME (REPEATING QUERY) LATER IN THE DOWNLOAD PERIOD 109
FIGURE 4.13 – THE NUMBER OF MESSAGES RECEIVED BY THE SEED DURING THE DOWNLOAD PERIOD 110
FIGURE 4.14 – AVERAGE RESPONSE TIMES FOR THE ALGORITHMS 113
FIGURE 4.15 – MOST PEERS SUBMIT A SINGLE-TABLE QUERY 115
FIGURE 4.16 – THE QUERY CONDITION ON THE SMALL TABLE 117
FIGURE 4.17 – THE AVERAGE NUMBER OF I/O'S 118
FIGURE 4.18 – AVERAGE RESPONSE TIMES FOR IDENTICAL QUERIES WITH A MORE POWERFUL SEED 121
FIGURE 4.19 – AVERAGE RESPONSE TIMES FOR MOSTLY DIFFERENT QUERIES WITH A MORE POWERFUL SEED 122
FIGURE 4.20 – AVERAGE RESPONSE TIMES COMPARED TO START TIMES 123
FIGURE 4.21 – AVERAGE RESPONSE TIMES FOR TOTALLY RANDOM QUERIES 126
FIGURE 4.22 – PERCENTAGE OF QUERIES ANSWERABLE BY THE SEED AND MULTIPLE PEERS 127
FIGURE 4.23 – THE AVERAGE RESPONSE TIME FOR A 20 PIECE QUERY 129
FIGURE 4.24 – THE AVERAGE RESPONSE TIME FOR A 40 PIECE QUERY 131
FIGURE 4.25 – A COMPARISON OF 20 AND 40 PIECE QUERY RESPONSE TIMES 131
FIGURE 4.26 – THE RELATIVE INCREASE BETWEEN THE ALGORITHMS 132
FIGURE 4.27 – CONCURRENT AND SEQUENTIAL PIECE REQUESTS FOR DIFFERENT SIZED QUERIES 133
FIGURE 4.28 – THE EFFECT OF CHANGING THE PIECE SIZE 136
FIGURE 4.29 – DETAILS OF THE RESPONSE TIMES FOR LARGER PIECES 137
FIGURE 4.30 – AVERAGE RESPONSE TIMES FOR REPEATING QUERIES WITH A LARGER PIECE SIZE OF 500 TUPLES 139
FIGURE 4.31 – AVERAGE RESPONSE TIMES FOR DIFFERING NUMBERS OF SEEDS 141
FIGURE 4.32 – 1/AVERAGE RESPONSE TIME COMPARED TO THE NUMBER OF SEEDS 142
FIGURE 4.33 – AVERAGE RESPONSE TIMES FOR 10 SEEDS 143
FIGURE 4.34 – AVERAGE RESPONSE TIMES FOR 25 SEEDS 143
FIGURE 4.35 – A DOWNLOADER CREATES A SINGLE PIECE FROM TWO HALF-EMPTY PIECES 145
FIGURE 4.36 – TWO DOWNLOADERS SEND CONCURRENT REQUESTS FOR DIFFERENT PIECES TO DIFFERENT UPLOADERS 146
FIGURE 4.37 – A DOWNLOADER RECEIVES PIECES OUT OF NUMERICAL ORDER 147
FIGURE 4.38 – AVERAGE RESPONSE TIMES FOR EARLY ADVERTISING OF DATA 151
FIGURE 4.39 – AVERAGE RESPONSE TIMES FOR THE INITIAL EXPERIMENTS WITH CACHE-WARMING PEERS. 153
FIGURE 4.40 – AVERAGE RESPONSE TIMES FOR THE WARM-CACHE EXPERIMENTS WITH FIVE QUERIES 155
FIGURE 4.41 – AVERAGE RESPONSE TIMES FOR THE WARM-CACHE EXPERIMENTS WITH FIVE QUERIES AND RANDOM QUERIES 157
FIGURE 5.1 – A QUERY (WHITE CIRCLE) WHICH IS A PARTIAL SUBSET OF AN ADVERT (BLACK CIRCLE) 168

LIST OF TABLES

TABLE 2.1 – SYSTEM PROPERTIES OF P2P DATABASES 36
TABLE 2.2 – ANALYSIS OF QUERY PROCESSING ALGORITHMS 38
TABLE 4.1 – SIMULATOR PROPERTIES 91
TABLE 4.2 – TPC-H TABLE SIZES 94
TABLE 4.3 – QUERIES AND THEIR SIZES 95
TABLE 4.4 – DIFFERING PIECE SIZES 135
TABLE 4.5 – THE PERCENTAGE OF PEERS RECEIVING RESULTS IN < 5 SECONDS 156

1 INTRODUCTION

1.1 INTRODUCTION

The modern relational database management system (RDBMS) [1] is based upon a Client-Server architecture; in which many clients submit queries to a database server as shown in Figure 1.1.

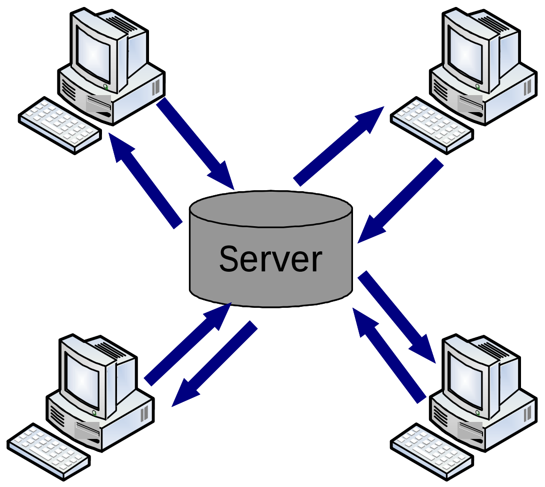


Figure 1.1 – A Client-Server database system

If the database server receives more queries than it is able to process at a particular time, clients submitting queries will experience an increase in the query response time and at some point, the performance may become unacceptable. The traditional solution to this problem has been to purchase a more powerful server, but the more powerful server will also have a finite capacity and there is the risk that the same problem will occur again at times of peak load.

In another domain – file-sharing – there was a similar problem. A high demand for some files results in file servers receiving more requests than they are able to process. This has been addressed by the use of peer-to-peer (P2P) techniques, which utilise the combined processing and networking power of individual clients to help reduce the load on the server. This has led to the emergence of many P2P file-sharing protocols including BitTorrent [2], Kazaa [3] and Gnutella [4]. Such file-sharing networks have been found to be robust and scalable and as a result have become very popular. In 2005, the web research company Cachelogic produced a survey indicating that approximately one third of all Internet traffic was generated by BitTorrent [5].

This thesis will investigate whether P2P techniques could operate successfully in a database environment in order to overcome the performance problem discussed above, resulting in more cost-effective scaling for Client-Server database systems. The core idea is to allow peers to answer (parts of) each others' queries as shown in Figure 1.2, instead of all queries being processed by the server as was shown in Figure 1.1.

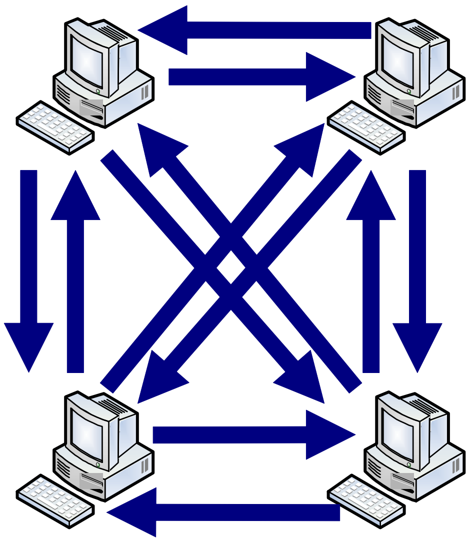


Figure 1.2 – P2P data-sharing

When this project began, BitTorrent was the most popular of the file-sharing protocols, as indicated above. It was well documented, with the protocol [6] and specification [7] readily available and had been the subject of performance modelling, as will be described in Section 2.5. It was therefore decided the base the work in this thesis on BitTorrent. Hence, this thesis will introduce the Wigan Peer-to-Peer Database System, a P2P database system based on BitTorrent (Chapter 2 will introduce BitTorrent and its algorithms in more detail).

1.2 AIM AND OBJECTIVES

The aim of this thesis is to design, implement and evaluate a P2P RDBMS, whose design is based on that of BitTorrent. This system should enable a database to be scaled over a network of peers and should produce an answer to each query, if this is possible. The aim results in two objectives.

The first is to design query processing algorithms for a P2P database system. This will involve analysing the algorithms used in BitTorrent and making changes where required in order to address the extra challenges presented by a DBMS when compared to file-sharing. These differences between file-sharing and a P2P database are discussed in more detail in the next section.

The second objective is to analyse the performance of the resulting P2P database. A comparison with a standard Client-Server database will also be made, in order to investigate whether, and in what scenarios, the P2P system can outperform a traditional DBMS.

1.3 THE DIFFERENCES IN REQUIREMENTS BETWEEN FILE-SHARING AND A P2P DATABASE

In order to begin to design a P2P database system using BitTorrent's techniques, the design of P2P file-sharing was examined and compared with what is needed to meet the requirements of a DBMS. BitTorrent is a file-sharing system which makes use of a central directory known as the Tracker. The Tracker keeps a log of which peers are holding parts of which files. It provides this information to peers wanting to download a file. Peers inject content (i.e. files) into the system. A special kind of peer, which holds a complete copy of a file, is called a seed. The seed places a file into the system and advertises that file at the Tracker. Initially, all other peers have to contact the seed if they want to download a copy of the file. When these other peers start to download the file, they also advertise the fact that they are doing so at the Tracker, and could then in turn be asked to provide parts of that file to their peers.

There are two prerequisites before a P2P database derived from BitTorrent could begin operation. A seed in BitTorrent holds a complete copy of a file as noted above, so there must be a logical equivalent in the P2P database system, to answer queries at the start of the download period. There will also need to be a Tracker which keeps a log of which peers have received the results of which queries.

However, there are significant differences between file-sharing and a P2P database system and these will now be described. These are due to the increased complexity of database servers when compared to file serving, and as will be seen in this thesis, have major implications for the design of the P2P RDBMS.

The only operation that needs to be supported by file-sharing is copying a file. However a database a collection of one or more tables that are queried. Further, it is rare for queries to select the entire database, or even an entire table; instead it is likely that queries will select only part of a table or will involve joins between two or more tables.

In a file-sharing system, binary data, typically encoding music or video, is stored. However, in a database, there may be a mix of data types, for example textual data and numerical data. A user may wish to perform operations on numeric fields for example arithmetic or aggregations such as "MAX." No similar concept exists in file-sharing.

Finally, database data may be updated. In a relational database, referential integrity rules mean that if data is updated in one table, the update might have to be cascaded across to another. Whilst not a major focus of this thesis, ideas for handling updates will be presented in Chapter 5.

These differences indicate that designing a P2P database derived from BitTorrent cannot be achieved by simply changing the items being shared from files to tables. Fundamental differences between file-sharing and a DBMS have major implications for the design of a P2P database system. However, the potential benefits are significant.

1.4 POTENTIAL BENEFITS OF A P2P DATABASE

Performance and Scalability: This chapter began with a motivating scenario for P2P databases – an attempt to reduce the response time by overcoming the problem of a busy server queuing query requests. If it were possible to send a query to another client instead of the server, this is reducing the load on the server. Some queries, which are not answerable by other peers, will still have to be routed to the server. However, if there are some popular queries, which are submitted regularly by clients, these can be answered by other peers. This spreads work around the system, with a set of peers contributing some effort towards the overall workload. This has the potential to be more scalable than a centralised DBMS in which only the server executes queries. In a system with a large number of peers, the combined processing power measured by disk space, CPU capability and amount of available memory could be high when compared to the server. As the number of peers increases, so does the extra processing power. This may mean that the system has the potential to handle increases in requests for a database.

If the database system has users in different geographic locations, the exploitation of locality could be used to improve query response times. If a peer was able to obtain query results from another peer geographically close to it, this is likely to be faster than having to contact a peer – or a central server – in a remote location because the network transfer time will be lower. The issue of locality in P2P networks has been examined by other researchers, for example in [8], but not for a P2P database. Note that locality will not be explored in this thesis.

Cost Reduction: Peers' ability to answer each other's queries could reduce the need for a powerful, expensive database server. The ability to route some queries to other peers will reduce the demand on the server, acting as the seed, leaving it free to answer queries which cannot be answered by other peers. If there are few such queries it may be possible to use a less powerful – and therefore less expensive – server.

Fault Tolerance: There could also be some potential benefit with regards to fault tolerance [9, 10]. Once multiple peers have obtained some data, they have in effect created dynamic copies of that data. This means there are more alternative ways to access data, should reliability problems occur with computers or networks. Chapter 3 will examine in detail how the failure of a peer is managed in the Wigan system.

1.5 MOTIVATING SCENARIOS FOR A P2P DATABASE

This chapter began with a motivating scenario for a P2P database. This will now be examined in more detail and two potential scenarios will be introduced.

The first of these scenarios concerns e-science databases. The increasing popularity of e-Science has led to requests for researchers to make their results widely available. A database with Internet access is the obvious means to achieve this. There are a small number of successful examples of such databases, for example the Sloan Digital Sky Server [11] presents its data using the SkyServer database [12]. However, not all researchers can afford to purchase, deploy, configure and manage powerful database servers. Therefore, the problems described at the start of this chapter, where a database receives more requests than it is able to process, could occur if a researcher 'publishes' data that proves popular. To prevent the research database becoming overloaded, researchers may decide not to publish their data, thus restricting the availability of data which could be of great interest to the research community.

A P2P database could provide a solution because it would allow other users, i.e. peers, to provide data and answer queries, therefore reducing the load on the server. This would hopefully permit and encourage more researchers to publish their results online for others to view and analyse. In addition, many e-science databases are often updated infrequently and this also adds to their suitability for conversion to a P2P system because updates in a P2P database are a challenge (updates are discussed in more detail in Chapter 5).

The second potential scenario concerns e-commerce applications which allow users to browse and purchase items online. These applications normally have a multi-tier architecture [13] to allow separate scaling of the application logic and the database servers. These database servers must be able to handle a very large number of simultaneous users; however recent experience has shown that the demands placed upon some database servers can become leading to poor performance [14]. Purchasing additional servers is a costly solution suffering from the limitations described above. Other solutions include database fragmentation [14] – known as "sharding" – or caching data in the main memory at the application tier. The problem with these solutions is that they need to be designed in an application-specific manner; hence any changes to the application can lead to changes in the way in which it interacts with the database. This could mean that the existing caching or fragmentation strategy no longer works. In addition, main memory caching suffers from the same problem as purchasing a more powerful server in that the increased capability may still be insufficient. Further holding a cache in the application layer means that additional cache management operations now have to take place there.

A P2P database in an e-commerce application would allow the middle-tier application servers to answer each other's queries. Unlike existing systems which cache data at each application server for that server's own use only, the P2P database would permit the servers to query each other and hence data cached by one server may be utilised by another requiring the same information. This would reduce the load on the database server and so could help to prevent the database server becoming a bottleneck.

1.6 SUMMARY

This Chapter has highlighted issues with Client-Server databases and introduced scenarios where a P2P database may be a viable alternative, given their success in the domain of file-sharing. This thesis will introduce the Wigan Peer-to-Peer Database System, a P2P database derived from the popular BitTorrent file-sharing protocol. The algorithms used in the Wigan system will be introduced, followed by a presentation of experiments conducted on a simulator to evaluate the system's performance.

The rest of this thesis is structured as follows. Chapter 2 will provide a review of background and related work. This will include an introduction to P2P computing with a description of BitTorrent and a review of previous research into the field of peer-to-peer databases. Chapter 3 will provide a detailed description of the Wigan architecture and Chapter 4 presents an evaluation of this architecture. Chapter 5 draws conclusions from the thesis and also makes suggestions for future work.

2 RELATED WORK

2.1 INTRODUCTION

Chapter 1 introduced the concept of a P2P database system. This chapter will examine previous contributions in the relevant areas, including a review of some existing P2P database systems.

2.2 P2P COMPUTING

The P2P computing model is designed to move away from the traditional Client-Server model, described in Chapter 1. The overall aim is that individual users – or peers – provide assistance to each other to help them carry out their overall tasks, be it file-sharing, data-sharing or other activities. P2P computing has been defined as: "Distributed systems consisting of interconnected nodes able to self organise into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the intermediation or support of a global centralized server or authority" [15].

There is no common terminology for classifying P2P system structures. Different papers use different terminology to define P2P network architectures and in particular, whether the P2P system has a central component or not. This thesis will use the terminology provided by Schollmeier [16] in his classification of P2P systems. Schollmeier defines "Pure" and "Hybrid" P2P networks. Pure P2P systems have no central components, instead all correspondence occurs between individual peers. Schollmeier refers to these peers as "Servents" because they perform the roles of both a server and a client. The alternative approach is a "Hybrid" P2P system which has some kind of central component and is discussed in more detail by Yang and Garcia-Molina [17]. Schollmeier notes that the difference between a hybrid P2P system and a client-server system is that clients do not share resources between themselves, unlike the peers in a hybrid P2P system. Bischofs et al

[18] also refer to a super-peer P2P system. In this approach, some peers have additional functionalities compared to regular peers, but are not central components.

In recent years, file-sharing has been the major application for P2P computing and this will be discussed in more detail in the next section. However, other uses for P2P architecture have been and are being examined by researchers. Bowen and Maurer [19] propose a P2P software development support system, which would replace a central code store or repository for software being developed by teams or individuals working in different locations. The LOCKSS peer-to-peer digital preservation system [20] is a P2P system designed to preserve access to journals and other sources of information available on the Internet. Articles are distributed in caches at different locations. LOCKSS is designed to prevent articles from being accidentally or maliciously damaged over time. A polling system is used to allow peers at the different locations to detect and repair such damage. Koloniari and Pitoura discuss ways in which P2P networks can be used to store and manage XML data [21]. The pMeasure system [22] sees a P2P network being used to provide a measurement structure for the Internet and is used to keep logs of varying statistics such as network traffic at a particular peer.

A further discussion of possible uses of P2P systems is provided by Roussopoulis et al in their paper "2 P2P or Not 2 P2P?" [23]. Different system characteristics and potential uses of P2P are examined, coupled with an analysis of which would be suitable for P2P and which would not. Some of these potential uses, such as auditing and routing are outside the scope of this thesis. One of the uses examined by Roussopoulis et al is data-sharing. The two items described under data-sharing are file-sharing and also censorship resistance. The latter includes systems such as Freenet [24] designed to allow users to publish items anonymously. Another potential use of P2P referred to in [23] is data dissemination, i.e. spreading content around a network quickly. Interestingly, BitTorrent is placed in this category, not under file-sharing. However, there is no mention in [23] of using P2P technology to provide a DBMS, by extending file-sharing algorithms or using other means. The characteristics of systems which would suggest a P2P solution are described as "limited budget, high relevance of the resource, high trust between nodes, a low rate of system change, and a low criticality of the solution" [23]. There are certainly some databases which could enter this category. If we consider a database which is used frequently but not updated frequently, is utilised by a single organisation (hence high trust between nodes) in a non-critical environment (e.g. not monitoring a safety-critical application), then a P2P system could be a practical solution.

2.3 P2P FILE-SHARING

A P2P file-sharing network is designed to enable users (peers) to obtain a file from other peers. This is in contrast to the Client-Server model of downloading a file, where a file is placed on a server and is obtained from that server by a user (client). P2P file-sharing networks can, like any other P2P network, be pure or hybrid. Gnutella [4] is an example of a pure system. An overview of the system is provided by Adar and Huberman in their paper "Free Riding on Gnutella" [25] which, although focussing on one aspect of Gnutella users' behaviour – free riding (obtaining files from others without providing anything in return) – also provides an introduction to the Gnutella system.

A user must install an implementation of the Gnutella protocol. There is no one 'official' Gnutella client program, instead there are many examples, some listed on [4]. Given that Gnutella is a pure P2P system, there are no central servers. Instead, a peer that wishes to join the network contacts one of several hosts, known to their Gnutella software. These hosts forward information about the joining peer, including its IP address, to existing peers in the network.

Once a peer has connected to the network, it must communicate with the other peers in order to obtain the file it is looking for. A peer may know of many other peers, which are known as its "neighbours." Peers can send keep-alive messages to each other and if a peer receives a keep-alive message, it also includes in its response the number of files it has shared. These keep-alive messages and the response to these are known as "Ping" and "Pong" messages respectively. Given that a pong message includes useful information about a peer, recipients of a pong message will forward this on to their neighbours, in order to help peers discover one another.

A peer that wishes to obtain a file broadcasts this request to its neighbours. This is known as a "Query" message. If a peer receives a query message for a file that it does not have, it can forward this query on to its neighbours. Alternatively, if a peer receives a query message for a file that it does have, it sends a "Response" message which contains all the relevant information concerning the location of the file. This peer also identifies itself by including a unique peer ID. Response messages return to the requesting peer by the exact same route taken by the query message. On receipt of a response message, the requesting peer will contact the peer that has the file and send an explicit request for that file.

There is no guarantee that any of the peers that initially receive a query message actually provide the file that the requester is looking for. The request message may have to be forwarded a number of times before it reaches a peer that has the file. To prevent query messages being forwarded indefinitely, they have a time-to-live, which is decremented every time the message is forwarded by a peer. If a peer receives a query message with a time-to-live of zero, it does not forward the query on to other peers. Also, it is possible a peer may receive the same query message multiple times from different neighbours. If this occurs, the peer will only forward the message once.

There are other P2P networks which have an increased level of centralisation. Kazaa [3] is an example of a super-peer system. Super-peers are used in Kazaa to provide an indexing service, i.e. store records of which peers possess which files. BitTorrent [2] is an example of a hybrid system, using a central component to assist in the file-sharing process. BitTorrent is the basis for the Wigan Peer-to-Peer Database System and will be introduced in more detail in the following section.

2.4 BITTORRENT

Chapter 1 explained that the Wigan Peer-to-Peer Database System introduced in this thesis is based on BitTorrent. This will now be described in more detail, along with an introduction to the key terminologies which will be used throughout this thesis. A complete description of the content and format of individual messages sent and received by BitTorrent peers is available in the BitTorrent Specification [7]. Unlike Gnutella, there is an 'official' BitTorrent client program, created by Bram Cohen, though nowadays many other implementations of the BitTorrent client exist, for example Azureus [26].

The process of receiving a file in BitTorrent is called "downloading" and the corresponding process of providing a file to other peers is called "uploading." Similarly, peers engaged in these activities are known as "uploaders" and "downloaders." Uploaders advertise the file (s) they have copies of through a central component called a "Tracker." The Tracker is the reason BitTorrent is a centralised P2P system and acts as a directory, keeping track of which peers are downloading and uploading which files. Any peer that is uploading a complete file is known as a "seed" in BitTorrent whilst any peer that is still in the process of downloading and therefore has a partial copy of the file is known as a "leecher" or a "leech." There must be at least one seed present to introduce a file into the system. A seed is the first peer to advertise at the Tracker and initially will be the only uploading peer; hence all downloaders will have to start obtaining the file from the seed. BitTorrent can be moderated by a human moderator to prevent malicious users advertising fake or corrupted files.

Large files in BitTorrent are split into pieces, normally 256KB in size. Each of these pieces is split in turn into sub-pieces, the idea being that sub-pieces are small enough (usually 16KB) to transfer quickly, but not so small that too much time is spent sending requests. The process of obtaining such a file will now be explained.

Any peer that wishes to download must begin by contacting the Tracker and announcing its interest in a particular file. The Tracker will provide a list of peers that already have some of the file or the entire file. The default size of the peer list is 50, however if fewer than 50 peers are advertising a particular file, the peer list will be shorter. If there are more than 50 peers advertising a particular file, then a selection of 50 is made at random. Not all uploaders will have the complete file. The Tracker will inform the downloader which of the peers has the complete file; hence the downloader will know how many pieces there are.

The first message that the downloader sends to an uploader is a handshake message. This must be transmitted before the downloader can ask an uploader for any pieces of the file. The uploader also informs the downloader how many pieces of the file it currently possesses. The downloader can use that information to decide in what order it will request the pieces. The first piece is normally chosen at random and the downloader can continue to request pieces at random, however [7] suggests using a rarest-first order. This allows a downloader to obtain some of the less-common pieces early, which allows these less-common pieces to spread further around the network. It was noted above that pieces of a file are split into sub-pieces. The downloader can request up to five sub-pieces concurrently from different uploaders. However, it does not request sub-pieces from different pieces concurrently. Once a downloader has received a complete piece, it is able to start uploading that piece to other downloaders that have not yet received that piece. Thus, a BitTorrent leecher may be downloading and uploading different pieces of a particular file at the same time. A peer normally uploads to no more than five downloaders at any one time. Peers also send keep-alive messages between each other at intervals. A peer may close a connection to another peer if no messages are exchanged for a period of time, usually two minutes [7], hence the keep-alive messages are used to keep a connection open. The BitTorrent Protocol Specification [6] notes that this timeout value could be set to a much lower value when data (i.e. a sub-piece) is expected. This would enable a downloading peer to realise if an uploader had failed and would prevent a downloader waiting indefinitely for a sub-piece to arrive from an uploader that had failed. If a peer does close a connection and then re-open it later, it would need to send a handshake again.

During the download process, peers re-contact the Tracker at intervals to provide an update on their progress. This progress report includes details of how much of the file the peer has downloaded, how much it has left to download and how much it has uploaded to other peers. The reporting interval is determined by Tracker and is included in the first peer list the downloader receives. The Tracker also provides another peer list, again containing randomly selected peers, to the downloader in its response. This may be of particular use for a peer which began downloading shortly after the file had been released by the seed and thus had few uploaders it could use at the time. Note that it is possible for a peer to re-contact the Tracker at more frequent intervals to ask for more peers, but repeatedly sending such requests is deemed to be 'impolite.'

It is possible to suspend and resume a download in BitTorrent. If a peer wishes to do this, it informs the Tracker that it is suspending and will then inform the Tracker again later, when it wishes to resume the download. If a peer does suspend a download, it can carry on from where it left off and does not have to go back to the beginning and start requesting the entire file again.

Towards the end of a download, when a downloading peer has received most, but not all of the pieces of a file, that peer can enter Endgame Mode. In Endgame Mode, a downloader sends out a request for the same sub-piece to all of its peers. Once the sub-piece arrives from a peer, the downloader then sends out a cancellation message to the remaining peers. Whilst making multiple requests for the same sub-piece may speed up the download process, it is not particularly efficient. Large numbers of additional messages are sent and in some cases, an uploader may have already despatched the sub-piece before the cancellation request arrives. This results in unwanted duplicate copies of the sub-piece being received by the downloader and is the reason that peers should not enter the Endgame Mode too early. There is no predefined point in the download process at which peers should enter Endgame Mode, this is left up to individual BitTorrent client programs. The BitTorrent Specification [7] offers some possible suggestions of when to enter Endgame Mode. One suggestion is to wait until all pieces have been requested – thus only requesting the sub-pieces of the final piece in Endgame Mode. Another alternative is to wait until the number of sub-pieces which have not been requested is lower than the number of sub-pieces which are in the process of being downloaded and also not to request more than 20 sub-pieces via Endgame Mode.

Once a peer has received a complete file, it does not automatically disconnect from the Tracker. It has already been noted that a peer is able to start uploading once it has received a single piece of the file. Once it has received the complete file, a peer will still contact the Tracker at the specified interval to announce that it is still uploading. The peer may send a specific message to say it wishes to disconnect from the Tracker, at which point the Tracker will remove the peer from its list of uploaders. If a peer fails for some reason, the Tracker will not receive the regular progress report from the failed peer and will remove it from its list of uploaders. The lifecycle of a BitTorrent peer can therefore be summarised as shown in Figure 2.1:

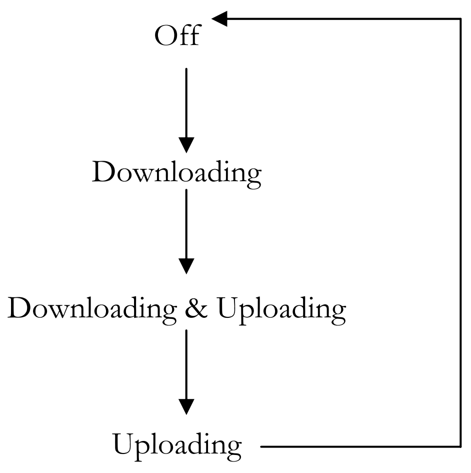


Figure 2.1 - The lifecycle of a BitTorrent Peer

However, there are some peers that will operate according to a slightly amended lifecycle and will download but perform no uploading at all. These peers are called "Free Riders" in BitTorrent and other P2P networks and were noted above when discussing [25]. Free riders are problematic because they consume resources but do not provide anything to other peers in return. BitTorrent's attempt to overcome this problem is to use a choking algorithm. "Choking" is the temporary refusal to upload a piece of a file to a particular downloader. This may be because an uploader does not want to receive too many connections from downloading peers. However, the main purpose of the choking algorithm is to ensure that those who do not provide much content into the system do not receive as much in return. This is known as a tit-for-tat protocol.

Every 10 seconds, a peer performs a choke. This involves replacing the slowest downloader with a faster downloader, using upload rate to measure speed, which will now be illustrated with an example.

Figure 2.2 shows two peers, Peer X and Peer Y, which are uploading from and downloading to each other. It can be assumed that both Peer X and Peer Y will also be uploading to and downloading from other peers as well.

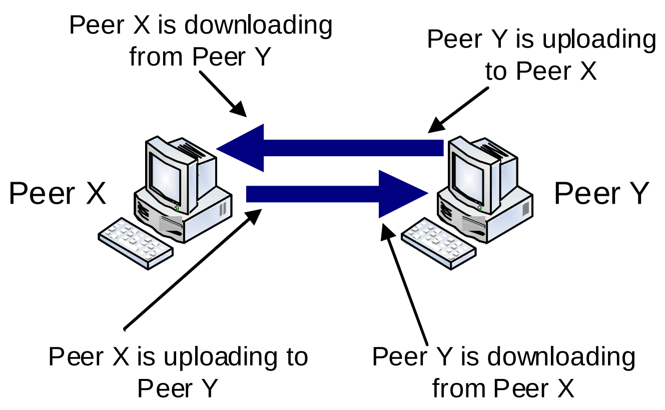

Peer X is downloading from Peer Y
Peer Y is uploading to Peer X
Peer X
Peer Y
Peer X is uploading to Peer Y
Peer Y is downloading from Peer X

Figure 2.2 – two peers uploading and downloading from each other

Peer X is receiving pieces of the file from Peer Y and measures the rate at which Y is sending those pieces, i.e. the upload rate of Y. If Peer Y was a very slow peer, this upload rate would be low; conversely if Peer Y was a fast peer, this upload rate would be high. Similarly, Peer Y measures the rate at which Peer X is sending other pieces of the file to it.

This choking process means that a peer which is both uploading and downloading will choke the peer that is providing it with the slowest upload rate. In this example, if Peer Y was providing pieces to Peer X at a slower rate than all other peers uploading to X, Peer X would choke Peer Y. The BitTorrent client software monitors how the amount of a file that it has received from each peer, to enable it to find the slowest uploader. The act of transmitting a sub-piece to a previously choked downloader is called "unchoking."

Every 30 seconds, each peer performs an activity called an "Optimistic Unchoke." This involves choking the slowest peer and unchoking a random peer from the list provided by the Tracker. This has two purposes. Firstly, it allows the peer to discover new peers which may be able to provide it with a faster upload rate. Secondly, it allows new downloaders, who have not received any pieces – and therefore cannot upload anything – to begin receiving pieces of the file. It has already been noted that, when performing a choke, peers will stop uploading to the peer that is providing them with the least amount of a file. A new downloader will not be uploading any part of the file, because it has nothing to provide so is likely to be choked by other peers in favour of others who have something to upload.

Consider two peers – Peer Y and Peer Z. Peer Z has just joined the download and has not received any of the file. Peer Y has received several pieces of the file.

Figure 2.2 showed how a peer measured the rate at which another peer was sending it pieces of the file. Clearly, Peer Y would view Peer Z as having an upload rate of zero because Peer Z is not sending Y any of the file. This is not because Peer Z is free riding, but because Z has no pieces of the file to upload. When Peer Y performs a choke, if it were already uploading to its maximum of five peers, it would not favour Peer Z because Z is not sending Y any of the file. With other peers adopting the same approach, there is a risk that Peer Z would never receive any pieces of the file. However, optimistic unchoking means that a new downloader, like Peer Z could be unchoked and can then start receiving pieces of a file. Peer Y may optimistically unchoke Peer Z and begin uploading to it. Once Peer Z has received a piece, it has something it can upload to others.

It is possible that a peer is choked by all of its uploaders. This peer is said to be "snubbed." If a peer has received no sub-pieces for one minute then it assumes it has been snubbed. A snubbed peer will optimistically unchoke all its peers in the hope that one of the peers will start downloading and will then reciprocate and begin uploading to the snubbed peer.

A seed cannot choke its downloaders based on their upload rate because a seed is not receiving any pieces of the file. Therefore, seeds use their peers' download rate to decide which peers to choke and unchoke.

Much of this activity remains invisible to the actual end user who is downloading a file. They will have found a .torrent file on the Internet, and been presented with a "Save As" dialog box as used in a standard client-server file download. The .torrent file contains only metadata – including the URL of the Tracker – and not any pieces of the file the user requires. BitTorrent will then automatically start the download process by contacting the Tracker. The user will be able to monitor progress including the upload and download rates offered to them by their peers and also has the choice of suspending the download and when to disconnect from the Tracker. However, the actual process of contacting the Tracker and other peers is all handled by the BitTorrent client software.

2.5 ANALYSES OF BITTORRENT

Various researchers have examined BitTorrent, its architecture and its performance. The original creator of BitTorrent, Bram Cohen, produced an article [27] introducing BitTorrent and explaining how the choking algorithm was designed to offer an incentive to users not to free ride. However, the issue of the choking algorithm as a means of avoiding free riding is critically analysed in [28] which presents experimental results showing that free riding peers can still download files quickly. An alternative algorithm is suggested in which downloaders keep a record of how much they have transferred to each peer and how much that peer has offered them in return. Wigan currently uses the BitTorrent choking algorithm, although an alternative approach like that described in [28] could be used if free riding peers became a problem.

Pouwelse has published some performance analyses of BitTorrent [29, 30] when used with the SuprNova website, a host for BitTorrent files [31]. This study involved taking measurements from the SuprNova website, the Trackers and the peers themselves. The aim was to measure the performance of BitTorrent and also collect statistics, for example how much each peer downloaded and uploaded and how many peers were connected to the Tracker. Dependability analysis was also obtained, noting the availability of peers and Trackers. Much of this analysis concerned SuprNova as much as it did BitTorrent. The overall conclusion drawn was that Tracker and web server downtime caused problems, but the integrity provided by the ability to moderate files meant a high standard of content. Like [28], [29] notes that BitTorrent's incentive mechanism could be amended, the suggestion being a means of rewarding peers that provide a file by offering them preferential treatment should those peers request another file.

Qiu and Srikant [32] developed a model of BitTorrent and then validated it using a discrete-event simulation of BitTorrent. Little details of the simulator are given, other than it was based on a Markov model introduced in [32]. A simulation of BitTorrent was also built for the research described in this thesis and was later adapted into a simulation of a P2P database. The Wigan simulator will be introduced in more detail in Chapter 4. Some experiments to validate the model were also performed by introducing a file into the BitTorrent network and monitoring the Tracker logs.

An analysis of five months' of monitoring BitTorrent is presented by Izal et al in [33] which unlike [32], does not attempt to model or simulate BitTorrent; but instead presents actual measurements that are used to assess the performance of BitTorrent as a means of distributing files. Interestingly, the measurements indicated that for the file they were observing – a 1.77GB Linux distribution – the choking algorithm was a suitable incentive to prevent free riding. In addition to analysing performance factors such as the number of downloaders obtaining a file and the response time achieved by those downloaders, other statistics concerning the geographical location of peers are also examined. The overall conclusion reached is that BitTorrent is an effective means of distributing large files. This thesis will examine whether a similar approach would work for database queries.

Legout, Urvoy-Keller and Michiardi [34] investigate the performance of the choking algorithm, the process of requesting pieces in rarest-first order and also the protocol overhead. The main conclusions are that both choking and the rarest-first algorithm lead to an efficient means of content distribution and that the protocol overhead – sending and receiving various messages, such as handshakes – is very low. Interestingly, it is argued that the choking algorithm is fair, a point contested by some of the previous analyses discussed in this section, because it does perform as expected. Uploaders providing a peer with the best upload rate are kept unchoked, whilst the optimistic unchoke algorithm does allow new peers the opportunity to join the download.

Another performance analysis was performed by Bharambe, Herley and Padmanabhan [35], which like [33] concludes that BitTorrent can be robust and scalable. However, unlike Izal et al's study, the choking algorithm is deemed insufficient to prevent unfairness occurring in the system, and alternatives are suggested. Unfairness does not just include free riders, it is defined as also including peers which do upload some pieces of a file, but download considerably more. One of the techniques used in [35] is simulation as used in both this thesis and in [32]. Bharambe, Herley and Padmanabhan note that some of their results differ from those obtained by Pouwelse and offer some explanations for this.

Gkantsidis and Rodriguez [36] propose an alternative to BitTorrent using an approach called network coding to propagate a file around a P2P network which could reduce the overall download time. A simulator was used which was later developed into a prototype system, Avalanche [37]. This in turn led to the Microsoft Secure Content Distribution system, which was under trial [38] at the time of writing.

A model of BitTorrent as a graph of peers is produced in [39]. The model tracked individual pieces of a file and was used to evaluate possible routing algorithms, including that actually used by BitTorrent. The performance was analysed by counting the number of time steps taken for a peer to download a certain number of pieces. It was discovered that BitTorrent's policy of only uploading to a small number of peers could help data spread faster round the network. In addition to analysing BitTorrent, [39] proposes the construction of a network graph which could result in a lower response time, by having more peers acting as seeds. One suggestion made is that sharing streamed data might be possible on a BitTorrent-like network.

Some of these analyses [30, 33] presented the workload model of BitTorrent in a form which can be approximated as shown in Figure 2.3:

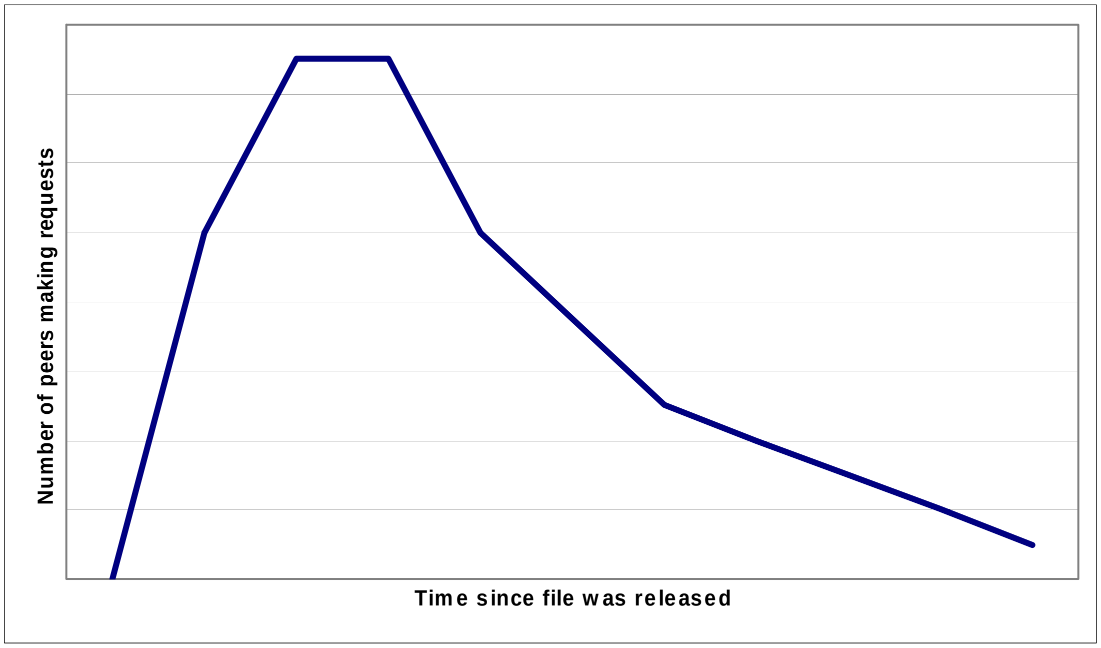


Figure 2.3 – The BitTorrent workload model

The initial state of rising interest is known as the "Flashcrowd" and occurs after the file is released. This is when a new file is generating lots of interest and large numbers of people wish to download it. There is then a period when the interest is reasonably steady, i.e. a regular stream of peers is joining the download, before a long period of declining interest. This will be due to the popularity of the file – [33] for example was monitoring the release of a new Linux distribution, hence initially this would be popular with users wanting to upgrade, but over time this could be replaced by a newer distribution and thus interest wanes. Similarly, the popularity of music files, as measured by the charts also follows this trend.

One important point to note is that the majority of these analyses contain descriptions of BitTorrent and some of these descriptions conflict with each other and with the protocol [6] and specification [7]. In particular, [34] introduces a newer version of the choking algorithm used by seeds to prevent free riding peers consuming a large amount of resources. The simulator introduced in Chapter 4 uses the choking algorithm described in the protocol and specification.

In conclusion, many of the criticisms in the literature relate to the choking mechanism as a means of preventing peers either free riding or uploading a fraction of the amount they download. The main aim of this thesis is to investigate whether a database modelled on BitTorrent would be effective and hence the incentives are currently not a major concern. It could, for example, be the case, that all peers using a Wigan database belong to the same organisation and could therefore be encouraged to share data with each other. Whilst some improvements and extensions are presented in some of the papers [36, 39], many of these analyses praise BitTorrent's robustness and scalability as a content-distribution network. This thesis will investigate whether similar algorithms would be as effective for database access as BitTorrent is for sharing files.

2.6 DYNAMIC CACHING

Although Wigan is a P2P database system, previous research in distributed Client-Server database systems has examined the possibility of caching results nearer to the end user than the database server.

The IBM DBCache system [40] is a mechanism for providing a local cache for some remote database. For all queries, two plans are constructed, one using the local cache and one using the original database. If the query cannot be answered locally, the latter is used and the data is also loaded into the local cache because the cache has been unable to satisfy the user's query. An example of DBCache in operation is presented in [41], which is an implementation of DBCache using the DB2 database server and is designed for e-commerce applications. DBCache is not designed for P2P database systems, but for Client-Server systems, as noted above. However, the ideas have some relation to the concept of Wigan, where the seed could be viewed as the remote, original, database and another peer which downloads data could be viewed as the local cache. If a downloader obtains data from the seed, this downloader is effectively receiving the data from the original database and making it available to other peers which may be geographically closer to it than the seed. However, there is a difference in the way in which the cache is managed. The actual spread of data across the Wigan network would depend purely on the data which the downloaders download; in the same way that in BitTorrent, the extent that a file spreads across the network depends on the number of downloaders. In particular, there is no two-phased query plan produced in Wigan. All queries are routed directly to the Tracker which provides a list of possible options. It is up to the downloader to decide from which peer they actually obtain the data. In BitTorrent, the location of the possible peers is not taken into account at any time.

Microsoft's MTCache [42] is a similar system to DBCache in that data from some remote database is stored in a local cache. However, in MTCache, a human administrator initially decides what is to be cached locally and runs a script to perform this caching. In addition, the cost of executing a query is used to determine where a query is executed. Therefore, if for some reason it would be more efficient to execute a query remotely than locally, the query would be executed remotely. Also, it is possible that some parts of a query are executed locally and other parts are executed remotely. This also bears resemblance to Wigan where different parts of a query may be received from different peers. However, like DBCache, this is a specific cache management system for Client-Server systems. The issue of updates in MTCache has been considered [43] by allowing users to specify how out of date their data can be, for example some users might be happy to accept data that is less than 24 hours old.

Another dynamic caching approach is bypass yield caching, described in [44]. This approach is specifically designed for scientific databases which could have a high workload. The aim of bypass yield caching is to reduce the overall network load on such databases. Local caches are placed near to the clients (i.e. the query submitters) and each cache acts independently. Ideas from the field of economics are used to determine whether to use a local cache or not, the cost being in terms of network traffic. If a request is made for data which is not in a local cache, a formula is used to determine, economically, if it is worth loading this data into the cache or not. The sample implementation discussed in [44], uses an astronomy database. BitTorrent and Wigan do not use economic formulae. Reducing overall network traffic is not a particular goal of Wigan.

A discussion of caching in an e-Business concept is provided by Mohan, one of the DBCache researchers, in [13]. Mohan provides a survey of caching technologies and concepts rather than a comparison between specific caching systems. Various forms of caches are discussed, including web caching in addition to database caching. Cache design issues, such as whether to cache entire tables or subsets and how to refresh the cache, are discussed along with a mention of caching projects including DBCache.

The Distributed Cache Table [45] was designed for P2P file-sharing systems to help those searching for files to find suitable peers. In [45], the running example is a system for sharing text files. Caches store digests of each document – containing key words about that document's contents – and interestingly, query subsumption is used. This means that the query results do not have to match the cache contents exactly, providing the query is a proper subset of the cache contents. However, any queries which cannot be answered by the caches are broadcast throughout the peers so that the results can be found. Initially all queries are broadcast until peers begin to populate their local cache. A mathematical formula is used to calculate the 'profit' of storing a query in the local cache, which takes into account factors including the size of the query results and the number of broadcasts made for that query. The latter statistic is an indicator of the query's popularity.

The use of subsumption is similar to the way in which the Wigan Tracker will match queries to adverts as will be discussed in Chapter 3. However, the distributed cache table was designed for a P2P file-sharing system and not a database. In addition, the distributed cache table assumes a pure P2P system architecture with no equivalent of the Tracker. The Tracker, with its ability to match queries to adverts ensures there is no need for broadcasting in the Wigan system.

2.7 P2P DATABASE SYSTEMS

Earlier in this chapter, it was noted that P2P computing had been considered for other uses in addition to file-sharing. Over the years, the potential of P2P computing has attracted the interest of the database community and this section will introduce some previous research into P2P database systems.

The Local Relational Model (LRM) [46, 47] is designed to help P2P databases co-operate with each other. However, unlike Wigan, the LRM views the database system as a collection of local relational databases with no global schema. The LRM is concerned with importing data between these local databases if required, the example given in [46] is of a hospital's database importing data from a GP's database. Relational algebra is used to present the LRM as a formal model. It is suggested in [46] some directory service may be introduced to help peers find each other and advertise their data. This would be similar to BitTorrent – and therefore Wigan – where the Tracker provides a directory of suitable peers. However, there is one major difference. The aim of this project is to have a single P2P database, managed by one organisation with a global schema used by all peers. The LRM is designed to merge a collection of disparate local databases, each of which is managed by a separate organisation, e.g. the hospital and the GP's surgery.

A characterisation of P2P databases is provided by Franconi et al in [48] which introduces a formal model, again using relational algebra. Like the LRM, it is assumed that a P2P database is a collection of separate databases, managed by different organisations, with no global schema. If a node receives a query that it cannot answer, it may consult neighbour nodes to help obtain the answer. The concept of how a peer should behave if it receives a query it cannot answer should not occur in the proposed Wigan system. This is because all query submitters would receive a list of peers from the Tracker which can definitely answer the query. The issue of updates in P2P database systems is considered by the same research group in a separate paper [49] which presents an outline algorithm for data updates.

The update problem is also considered by Vecchio and Son in [50]. Like Wigan, this is based on a file-sharing system, however whereas Wigan's basis is BitTorrent, [50] is based upon Gnutella. A voting style algorithm is presented in [50], which allows peers to agree on the correct data should inconsistencies arise, and also discuss the levels of data replication needed for this algorithm to work. The query processing aspect of databases is not discussed. Updates in Wigan would have to involve both the Tracker, because this is the place where records are kept, and the seed because this is the peer that originally placed the data. Other peers should not need to participate in a vote. The issue of updates in Wigan will be discussed in more detail in Chapter 5.

Another formal approach to P2P databases is provided by Majkic [51] which provides a formal semantics for an unstructured P2P database system. Again, it is assumed that there is no global schema of any kind, unlike the Wigan system proposed in this thesis.

P2P database interaction is studied by Giunchiglia and Zaihrayeu in [52], which also assumes there is no global schema and builds on the LRM described above [46, 47]. Like some of the other systems described in this section, it is possible for the peers to join together in interest groups. This is a concept which does not arise in either BitTorrent or the Wigan system and involves groups of peers with some common interest working together in some way. [52] notes that one node will need to manage each interest group, which involves holding metadata about the group. If a peer receives a query it cannot answer with its local database, it has to contact other peers and forward the query on to these peers. The peer may have to find the generic query topic and then look up its list of peers to find other peers that can help. This process is recursive; the peers that receive the forwarded query may in turn forward the query to other peers. The results are all returned to the peer that received the query request, which reconciles the results and despatches them to the query submitter. Note that unlike Wigan, there is no central Tracker. Peers keep a record of their acquaintances, with respect to individual queries. This means that the peers construct graphs of how a query is evaluated. In Wigan as noted earlier, a peer should never receive a query that it cannot answer with the data that it possesses. This also means that a peer does not need to forward a query on to another peer. Wigan does not currently have peers joining interest groups although this may be a possible extension to the system.

Aberer et al [53] present an indexing system that could be used in either a P2P database or a P2P data-retrieval system, which uses keyword searching. However, Aberer et al concentrate on describing their indexing algorithm and do not outline or recommend a query processing system which could use their algorithm. Unlike Wigan, [53] is designed for structured P2P systems and is built on the P-Grid [54] P2P architecture developed by the authors' research group. The algorithm presented in [53] is also completely decentralised, unlike the Wigan system. Currently, Wigan does not have an indexing system, but the ideas presented here could be applied to Wigan. Indexing is discussed further in Chapter 5.

The Query Difference Operator [55] was developed to find a way of determining which part of a user's query over a set of separate information resources cannot be answered. These resources are referred to as advertisements. The query difference operator was not intended for P2P databases; instead the information resources could be on the Internet or a corporate intranet, the example given in [55] is for an online film database. Relational algebra is used to present the formulae for the query difference operator. It is assumed, like Wigan, that there is a global schema that all local databases conform to which takes the form of a single universal relation that can be constructed by combining the data from all of the advertisements. However, the global schema in [55] is obtained from many separate data resources, unlike Wigan where there is a single database. The query difference operator is concerned with integrating answers from the different sources using this universal relation, and builds on [56] which examines how to check if query results are complete. Wigan does not need to deal with the completeness issue because there would always be a peer which can resolve each query (the seed in the last instance). However, the concepts involved in the query difference operator could be used to provide an alternative to contacting the seed in the case that no other peer can resolve a particular query. This idea will be discussed further in Chapter 5.

The relationship between databases and P2P computing is explored in [57], which concentrates particularly on the problem of where to initially position the data in the network and also introduces the Piazza P2P database system, explored in more detail in [58]. Piazza peers each have their own separate relational databases and schema mappings are required to translate data between different database schemas, held by the different peers. Interestingly, [58] notes that the indexing system in Piazza, designed to help query submitters find peers with the relevant data, was implemented using a centralised architecture, unlike many file-sharing systems.

Individual Piazza peers can work collectively in clusters, known as spheres of co-operation, to pool their data resources. If a query is not answerable by a peer or its neighbours, the query must be routed to the data origin, which is the peer that placed the data in the network and is equivalent to the seed in Wigan. Piazza peers broadcast such unanswerable queries amongst their sphere of co-operation in case other nodes in the group also have unanswerable queries. Different nodes within the sphere of co-operation then request different parts of these unanswerable queries, using a cost model in order to find the most efficient means of requesting the data. The concept of routing a query to the peer which placed the data does exist in Wigan, because downloaders will request data from the seed if there is no other peer that can provide the data. However, if there is a downloader's query which can only be answered by the seed, the Tracker informs the downloader that the seed is the only possible source of the data and the downloader requests the data from the seed itself. Wigan peers do not broadcast any messages and as noted above, do not form spheres of co-operation.

The BioWired system [59] is a P2P database specifically designed for e-Science applications. A prototype has been implemented using the JXTA protocols [60]. The creators of BioWired have specifically decided to focus on the area of biodiversity, hence the data model presented in [59] is only concerned with this area. BioWired splits the data into various domains, each representing a particular area of biodiversity. Domains can be related to each other; the example given in [59] is that in the case of a domain involving species and another domain involving geographic locations, the domains are related because a species is found at some geographic location. Unlike Wigan, searches over the adverts in BioWired are performed according to domains. Each domain has a data source processor (DSP). A user who wants to make their data available must start a DSP service on their local machine. Queries in BioWired are written in the BioWired Query Language, BWQL. When a DSP receives a BWQL query, a connection is opened to the data source. Interestingly, the BWQL query is converted to SQL during query processing. It is noted in [59] that the BWQL language only provides simple data retrieval and that a future extension is to enable BioWired to provide distributed query processing facilities. BioWired does assume a global schema, with each peer maintaining a local database, but does not guarantee primary key uniqueness or referential integrity. BioWired is similar to Wigan in certain aspects. Although not the same as a Tracker, JXTA rendezvous nodes are places where nodes can advertise their data. These advertisements are how a peer discovers other peers, known as its acquaintances. However, it is noted in [59] that if none of a peer's acquaintances can answer a query, the peer does not get an answer in the current BioWired system. A planned extension is to modify BioWired so that if an acquaintance that cannot answer the query it will check its own list of acquaintances to see if any of them can. The Wigan approach is to use the seed and Tracker to deal with this problem. The seed has the answer to all queries, so there would never be a query that could not be answered by any peer. There is only one Tracker and this monitors the whole system, so it will know exactly which peers have which query results. Thus, there would never be an unanswerable query in the Wigan system.

The Unified P2P Database Framework [61] is designed to provide information about a Data Grid. It broadcasts messages and does not guarantee complete answers, each query having a specific time to live. Anybody that wishes to submit a query contacts an agent node which searches its local database as well as forwarding the query on to its neighbours. There are various strategies used in the Unified P2P Database Framework for returning data to the query submitter. One strategy sees data being returned to the agent node which then compiles the results and sends them back to the query submitter. Alternatively, data can be sent directly to the query submitter, bypassing the agent node. The Unified P2P Database Framework's query processor is organised in the same way as a standard query processor, in a Client-Server database. Therefore, if an agent node has no neighbours, the system behaves like a normal distributed database. The agent node is similar to the Wigan Tracker in that it acts as the link from the query submitter to the rest of the system and in that it can find other peers to assist with the query. However, the Tracker does not have a local database to search itself, nor does it attempt to start executing the query on the submitter's behalf. In addition, the Wigan Tracker would keep a log of the query result size, in the same way in which the BitTorrent Tracker keeps a note of the number of pieces. This would enable a Wigan peer to realise when it had received the complete results of a query. BitTorrent only sends out multiple requests for the same piece when in Endgame mode. Wigan would do the same. Endgame mode is not compulsory, so if necessary could be avoided if bandwidth was a major issue.

The PIER [62, 63] Internet query engine introduces database query processing techniques into a non-database scenario. PIER was designed for various applications, including file-sharing and network monitoring (both of which were among the potential uses for P2P noted by Roussopoulis et al [23]) and can send out continual queries with timeouts.

One example of PIER in operation is the Information Plane project [64], which uses PIER to monitor a PlanetLab system. For file-sharing applications, PIER was tested with Gnutella. PIER uses distributed hashtables (DHTs) to provide its overlay network. The architecture of PIER is described fully in [63].

PIER queries are written in its own dataflow language, UFL. Query plans in UFL are sent by a user to any node. This PIER node then becomes a proxy for that user. The proxy parses the UFL into Java and sends the query plan to nodes which can answer the query. A distributed indexing system, using DHTs as noted above, is used to keep track of which nodes can resolve a query. The results are returned to the proxy which forwards them on to the user. There are the similarities between the proxy and the Wigan Tracker, in that the Tracker is also the gateway to the system. However, a query submitter in Wigan must contact the Tracker and not any random peer. The Tracker does not execute the queries, nor does it collate the results.

SQPEER Middleware [65] is a query processor designed to operate in semantic overlay networks. It uses the semantic web RDF/S technology to route queries whilst peers use RDF/S schemas to advertise the data they hold. Note that each peer has their own individual RDF/S schema for their database and there is no global schema. SQPEER establishes what it calls communication channels while executing a query. Each peer that submits a query creates a new communication channel and sends the query plan to its peers. If the query submitter does not know how to obtain some parts of the query, these parts of the plan are left blank, for the peers that receive the plan to complete. There is some similarity here with Wigan in that the peers advertise their data. However, in Wigan this is done through the Tracker. The Tracker provides a list of peers, so the query submitter will know which peers can provide the data they require and thus do not need to send partially blank plans around the system. An interesting feature of SQPEER is that it can operate in different P2P overlay network structures, [65] shows examples of SQPEER running in pure (referred to as "ad-hoc" networks in [65]) and hybrid networks.

The PeerDB system [66, 67] is a distributed data management system, built on the BestPeer P2P system [68] developed by the same research group. It does not have a global schema and instead helps a user find potentially relevant data by using an initial keyword search. The user selects some relations from the list returned after the search is complete. The reasoning behind this is to enable different terminologies to be used in different databases; however, there may be occasions where the keyword search returns relations which contain irrelevant data, not required by the user. This is similar to an Internet search, where the search engine may return irrelevant results. There is a similarity with Wigan in that the user is presented with a selection of data sources to choose from. However, the problem of unsuitable options being returned will not arise in Wigan because the Tracker will not inform a downloading peer of unsuitable uploaders. There is no single, central Tracker in PeerDB; instead an agent is created to handle each query. All query processing work in PeerDB is handled by agents, a concept which does not exist in Wigan where the peers submit their own queries.

Mutant Query Plans [69] provide a distributed catalogue for a P2P file-sharing system. Data is split into interest areas; and components called base servers keep records of data in a particular interest area. Other types of servers – index servers, meta-index servers and category servers – keep records of base servers and overlapping interest areas. Each peer publishes the data they possess, encoded in XML. A mutant query plan is a query plan, also encoded in XML, which is passed between servers. If a server possesses data which matches the query, it can replace the relevant part of the query plan with that data. Alternatively, if a server knows of another peer which holds the data that the query submitter requires, the server will add the URL of that location into the plan. This approach is similar to Wigan in that peers advertise the data they have through a centralised component. However, in Wigan, the Tracker does not actually provide any peer with data; instead it just provides the details of peers which have the data. The query submitter can then choose which peers to query for data. There is no query plan handed around the system, as noted above. Also, Wigan does not have a Tracker containing details of other Trackers, although this could be added as a possible future extension. Wigan is a database, not a file-sharing system and as has been noted, is not divided into areas of interest.

The query trading algorithm [70] is designed for distributed networks of autonomous database systems. This idea combines query optimisation techniques with economic ideas.

Queries are viewed as commodities hence those users submitting queries are thought of as the buyers and those providing the answers to the queries are the sellers. The cost is given as the overall execution time, though it is noted in [70] that other measurements of cost could be used. Using one of a choice of negotiating protocols, including auctions, the buyers ask the sellers for an estimate and then select the most cost-effective option. Interestingly, a seller does not need to hold the full query. It may have only part of the query, in which case it gives the buyer the cost estimate for the part that it possesses. The buyer may then obtain different parts of the query from different peers if it wishes. The process is iterative, after a buyer receives the offers and constructs a query plan; the algorithm is re-run to see if there is an even better query plan.

The query trading algorithm is not concerned with query execution; instead it solely concentrates on the optimisation issue, i.e. how to execute the query in the most efficient manner with the given resources. The result of this algorithm would be the query plan which is executed. The buyers would inform the successful sellers who would then start executing the queries. Currently, Wigan is not concerned with query optimisation, only with query processing; however, there is a similarity in that advertisements are used. The advertising process used in the query trading algorithm is similar to that which occurs at the Tracker, where peers advertise what parts of the query they can answer and these adverts are then used to determine how the query is executed. However, Wigan has the separate Tracker entity which does not exist in the query trading algorithm. A Wigan peer submitting a query does not announce this fact to other peers; instead it contacts the Tracker which examines the adverts. The advertisements planned for Wigan are more proactive in that peers would continue to advertise at the Tracker even if there is nobody requesting the results of that query at the present time. This is the analogous to BitTorrent where peers must explicitly disconnect from the Tracker once they have completed their download. Wigan does not currently include query optimisation or cost models, however, there is the facility for cost-based selection to be introduced. A scheme similar to that described in [70] could be used to implement cost-based selection where each peer would advertise a cost estimate at the Tracker as well as the query. When the Tracker returned the details of relevant advertisements, the cost estimate would be included. A peer could then use this cost estimate to decide from where to obtain the query results.

The Mapster P2P database system [71] is designed to allow a user to share their database with other peers. Like some of the other works described in this section, there is no global schema. Data is split into domains, with each domain containing data about some particular area of interest. Each domain is managed by super-peers. One super-peer is deemed to be in overall charge, the other super-peers are there to act as backup should the super-peer in charge leave or fail. This super-peer in charge is known as the virtual super-peer (VSP) as it is the only peer visible from outside the domain and it maintains what the authors call a mediated schema – a combination of the various schemas of the peers' databases in that domain. Any new peer that wishes to join, logs into a Mapster network to receive a username and can then listen for advertisements about various domains. Once a Mapster peer has selected a domain, it contacts the VSP, sending its database schema. The virtual super-peer then updates the mediated schema to include the new database. Each peer is either allocated a parent super-peer by the VSP or is chosen to be a super-peer. Alternatively instead of joining an existing domain, a peer can start their own, new, domain and become a virtual super-peer. If a peer chooses this option, it must start to advertise this domain.

Mapster queries are written in SQL, as in Wigan, and are sent by a peer to their parent super-peer which examines the mediated schema to find peers which can resolve the query. For each of these possible peers, the parent super-peer constructs a query and sends it to the peers. The results are returned directly to the query submitter and do not go via the super-peer.

Mapster uses advertising and peers with additional powers in a different way to Wigan. There is no concept of domains in Wigan with all adverts being placed by individual peers. Wigan adverts are stored at the Tracker and are not circulated elsewhere around the system. There is no need for the Tracker to perform any kind of schema mappings because there is a global schema known to all peers. The concept of super-peers differs from Wigan where all peers perform identical functionality apart from the Tracker. The Tracker does not perform the functionality of a Mapster super-peer because it only suggests possible peers to the query submitter and does not construct or execute queries on the submitter's behalf. In addition, a Wigan peer does not need to log in anywhere to obtain a username before it can listen for advertisements. It needs to contact the Tracker, where it is able to access all the relevant advertisements.

The DÍGAME architecture [72, 73] allows peers to make their local databases available to other peers using a subscription service. Subscribers request particular data from a peer, which responds by sending the data. The subscriber then has a replica of the database on their local machine and routes any queries they have to this replica and not the original database. The database can only be updated by the originating peer which sends out a new version of the database to the subscribers when an update occurs. A piece of middleware, known as the wrapper component, manages the schema integration between the original database and the replicas held by the subscribers. A major difference is that in Wigan, there is one original database at the seed. Downloading peers submit queries on this database and the results of these queries are despatched. The downloaders do not receive a full replica copy of the database unless they specifically request it. DÍGAME has no central component managing any part of the process. Each DÍGAME peer manages their own list of subscribers.

In conclusion, there are several existing P2P database systems, which are summarised in the tables that follow. Table 2.1 shows a comparison of the systems and their properties. The "Summary" column classifies the system as follows:

- P2P Database – a working P2P database, data management or replication system which has been built and/or simulated.

- Formal model – a system whose architecture and semantics are defined formally, e.g. with relational algebra, but which has not yet been built and/or simulated.

- P2P database component – not a complete P2P database system, but instead a component of one. This will be followed by a description of that component.

The "Global Schema?" column indicates whether or not the system is designed with any kind of global schema, known to all peers. Similarly, the "Scales a Single Database?" column indicates whether the system views a P2P database as a single database which can be distributed in a P2P manner, or as a collection of heterogeneous databases maintained by different organisations. The final column in Table 2.1 indicates if the system has any kind of centralised component.

Table 2.1 – System Properties of P2P databases

System	Summary	Global Schema?	Scales a Single Database?	Central Component?
LRM [46, 47]	Formal Model	No	No	No
Franconi et al [48]	Formal Model	No	No	No
Vecchio & Son [50]	P2P Database Component – an algorithm for handling data updates	No		No
Majkic [51]	Formal Model	No	No	No
Giunchiglia & Zaihrayeu [52]	Formal Model	No	No	Group Manager
Aberer et al [53]	P2P Database Component – an indexing system for P2P databases	No	No	No
Query Difference [55]	Formal model	Yes	No	No
Piazza [57, 58]	P2P Database	No	No	No
BioWired [59]	P2P Database	Yes	No	No
Unified P2P DB Framework [61]	P2P Database (under construction)	No	No	No
PIER [62, 63]	P2P Database (though designed for non-database applications)	No	No	No
SQPEER Middleware [65]	P2P Database Component – a query processor/router for P2P data management systems	No	No	No
PeerDB [66, 67]	P2P Database	No	No	No
Mutant Query Plans [69]	P2P Database Component – a query processor for P2P data management systems	No	No	Yes
Query Trading [70]	P2P Database Component – an algorithm for query optimisation	No	No	No
Mapster [71]	P2P Database	No	No	Super-peers
DÍGAME [72, 73]	P2P Database	No	No	No
Wigan	P2P Database	Yes	Yes	Yes

Given the importance of query processing to this research, Table 2.2 provides a comparison of the query processing algorithms used in the P2P database systems (where applicable) and compares these with Wigan. Various characteristics are examined and these shall be summarised here.

The first item is the existence of interest groups. This section has shown that there are some systems where groups of peers work together in groups. These groups could be either members of the same organisation or peers with a common interest about data in a particular area.

Also noted in Table 2.2 are any issues affecting query execution, for example in some systems, peers may receive queries they are unable to answer and have to try and obtain the answers from other peers if possible. If this column is left blank, there are no such issues to note.

Finally, the existence of broadcasting is noted. Broadcasting occurs when a peer sends a message to all the peers it is aware of. Whilst broadcasting is a way of ensuring that a message reaches as many peers as possible, it has the adverse affect of flooding the network with messages, sometimes more messages than are actually needed. An example is BitTorrent's Endgame Mode.

Table 2.2 – Analysis of query processing algorithms

System	Interest groups?	Execution issues	Broadcasts?
LRM [46, 47]	No		No
Franconi et al [48]	No	Peers may receive queries they can't answer	No
Vecchio & Son [50]	No	Not concerned with query processing	No
Majkic [51]	No	Peers may receive queries they can't answer	No
Giunchiglia & Zaihrayeu [52]	Yes	Peers may receive queries they can't answer	No
Aberer et al [53]	No	Indexing system only	No
Query Difference [55]	No		No
Piazza [57, 58]	Spheres of co-operation		Sometimes
BioWired [59]	Yes	Complete results not guaranteed. Primary key uniqueness not guaranteed	No
Unified P2P Database Framework [61]	No	Complete results not guaranteed	Yes
PIER [62, 63]	No	Queries may have a timeout value	Yes
SQPEER Middleware [65]	No	Peers may not know how to answer part of a query	No
PeerDB [66, 67]	Yes	Keyword searching may return irrelevant results	No
Mutant Query Plans [69]	Yes	Peers may not know how to answer part of a query	No
Query Trading [70]	No	Not concerned with query processing	No
Mapster [71]	Yes		No
DÍGAME [72, 73]	No		No
Wigan	No		Endgame only

2.8 CONCLUSIONS

This chapter has introduced some of the background concepts behind the Wigan P2P database system presented in this Thesis. Section 2.2 introduced differing types of P2P network while Sections 2.3 and 2.4 presented an example of a pure system (Gnutella) and a hybrid system (BitTorrent). BitTorrent was selected as the basis of the Wigan Peer-to-Peer Database System because, as noted in Section 1.1, it has been well-documented and also subject to performance modelling, described in Section 2.5. In addition, the hybrid architecture of BitTorrent could make it easier for a database administrator to control the data. In particular, if an administrator updates the database, the records held by the Tracker should inform the administrator of which peers have obtained the data that has been updated. This information could be used to inform peers that their data has changed. In a pure system like Gnutella, it may be much harder for an administrator to determine which peers have obtained a particular dataset, given that each peer maintains their own separate indices of the data (or files in the case of Gnutella) held by each peer.

Also in this chapter, some existing P2P database systems – including operational examples and outline ideas, models or algorithms – have been analysed and compared to the proposed architecture of the BitTorrent-based Wigan system. This shows Wigan is the only P2P database system which distributes a single RDBMS in a P2P manner thus scaling the workload over a number of peers and providing complete and correct results to all queries submitted. It is also the first P2P RDBMS which specifically uses techniques from a file-sharing protocol to enable the sharing of database data. Chapter 3 will introduce the Wigan architecture in detail and will include a description of how the BitTorrent algorithms presented in this chapter have been amended for a P2P database.

3 THE WIGAN ARCHITECTURE

3.1 INTRODUCTION

This chapter introduces the architecture of the Wigan Peer-to-Peer Database system and includes a description of the key components. BitTorrent was used as the starting point for the design and hence the key terminologies – Tracker, seed, uploading, downloading, etc. – have the same overall meaning in Wigan as in BitTorrent, but the necessary changes required to support database queries are described. Note that the terms "client" and "server" will only be used in this chapter – and in the remainder of this thesis – when discussing the traditional Client-Server architecture introduced in Section 1.1.

3.2 SHARING DATA

Chapter 2, Section 2.4, explained that all files in BitTorrent were downloaded in pieces. These pieces serve two functions. As noted in Chapter 2, they provide peers with a portion of the file to upload to their peers and they also act as a means of parallelising the download process. It is more efficient to obtain a large file, such as a Linux distribution, in parallel pieces from multiple peers than it would be to attempt to request the whole file in one go. In addition, dividing the file into specifically numbered pieces provides a means of recording exactly which peers possess which parts of a file.

Given that Wigan is derived from BitTorrent, the database tables are also split into pieces. The division into pieces was done horizontally, i.e. each piece contains a fixed number of tuples. For a piece size N, the first N tuples in the table would be in the first piece, the second N tuples would be in the second piece, etc. This is shown in Figure 3.1, where the tuples are shown within a table and a horizontal line indicates a division between pieces. The number of pieces in a particular table clearly depends on the number of rows within that table. If the number of tuples in the table does not divide exactly by the piece size, N, the final piece contains less than N tuples.

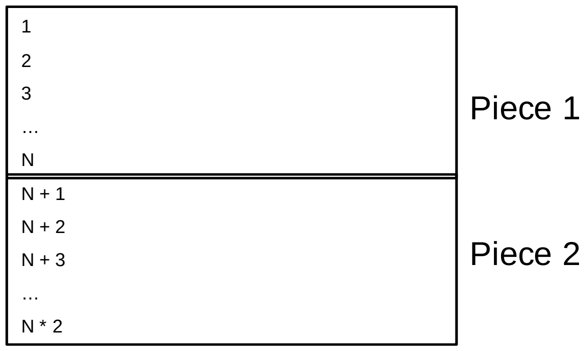


Figure 3.1 – A table is divided into pieces

This division into tuples permits a simple tuple-to-piece mapping system to be implemented where, for each query, every tuple can be uniquely identified with a piece. For example, in Figure 3.1, tuple number three is in piece number one.

Figure 3.1 illustrates how a database table is divided into pieces. The results of a query can also be divided into pieces in an identical manner. Query results are, like database tables, a set of tuples and therefore, these are also horizontally partitioned. Calculating the tuple-to-piece mappings for query results will be discussed in more detail later in this chapter, in Section 3.4.2.

In Wigan, no tuples are split over pieces. Uploading (discussed in Section 3.4.3) would be much more complicated if a downloader submitted a query involving an SQL SELECT for multiple columns, some of which were on one piece and some of which were on the next piece. When the downloader requested the first piece, the uploader would have to return only part of the tuple and similarly, when the downloader requested the second piece, the uploader would have to return the remaining columns.

As in BitTorrent, pieces in Wigan allow parallelisation of downloads and prevents the need to transfer an entire large table, or a large set of query results, across a network in one large packet from one peer. A slow or failed uploader is therefore less likely to have an impact.

The initial simulated version of Wigan, which will be discussed in detail in Section 4.2, was tested with a piece size of 50 tuples per piece to investigate whether the implementation of the algorithm was successful. Experiments presented in Chapter 4 will investigate the effect of changing the piece size between the extremes of one tuple per piece and one table per piece in different simulation runs.

3.3 THE WIGAN COMPONENTS

3.3.1 THE SEED

An important part of the Wigan architecture is the seed. A BitTorrent seed, introduced in Chapter 2, is a special kind of peer that possesses a complete copy of a particular file. In BitTorrent, a seed must place the first advertisement at the Tracker to enable other peers to begin downloading. There must be a similar concept in Wigan, as noted in Chapter 1.

There are various design options for a seed in the Wigan system. Firstly, it would be possible to take the BitTorrent approach and have a single seed storing an entire copy of the database. A second alternative would be to partition the database by tables so that each table is stored on a separate seed. A third alternative is to use horizontal or vertical partitioning so that there are multiple seeds each storing blocks of data from several different tables. These options will now be discussed.

The single-seed approach has some advantages. Such a seed could answer any query on the database, regardless of the queries that other peers were advertising. It would also allow downloaders to easily submit single multi-table queries. If such a seed received a multi-table query, it would have all tuples in all the tables involved in the join. The seed would simply execute the query and return the joined rows to the downloader. Although the initial version of the Wigan system will examine simple single and multi-table queries, future developments could involve more complex SQL with nested subqueries. If a subquery contained a join, the model of a single seed with the entire database would be able to support this easily because the entire subquery could also be executed by the seed, even if the subquery was over a different table to the outer query or the subquery contained a join. Another area of future work concerns data updates. Although there are several complex issues surrounding updates, discussed in Chapter 5, as will be seen a single seed has the advantage that updates can initially occur at a single seed and do not have to be co-ordinated amongst several seeds.

However, a single seed could be a bottleneck in terms of performance if there were many peers submitting queries which could not be answered elsewhere. These queries would all be routed to the seed which may then receive more queries than it is able to process. Failure of the seed could also render some queries unanswerable. The exact proportion of unanswerable queries would depend on the quantity of other uploaders and the queries they were advertising. However, if there were no other uploaders available, the whole system would be unavailable until the problems were solved by the database administrator.

The second design option noted above is to partition the database by tables. For example, each seed could hold a single table. If the database contained a mix of small and large tables, some of the small tables could be combined on a single seed if required. In this design, assuming the database has more than one table, there would be multiple seeds. Note that this design approach assumes that all seeds are provided by the database owner. If users were able to volunteer their computers as seeds, care must be taken when allocating tables to seeds to prevent very large or very popular tables being placed on a less powerful computer which may be unable to handle all of the queries it receives.

Having multiple seeds removes the bottleneck of a single seed because, if there were a large number of queries on different tables, these would be answered by the separate seeds. If a seed failed and no other uploaders were available – except for the other seeds – then only the table held by the failed seed would be unavailable to new downloaders.

There are, however, some disadvantages to this approach. Multi-table queries cannot be answered in such a straightforward manner. A downloader would either have to submit a collection of single-table queries and then join the results locally, or distributed query processing (DQP) techniques would have to be employed by the Wigan system to handle the join operation. There are various means of achieving this, for example the downloading peer could create and execute a query plan instructing the seed holding the copy of the first table to select the relevant rows from its table and send them to the seed holding the copy of the second which will select the relevant rows from its table, join the results to those from the first table and return the final answer to the downloader. Nested subqueries would also be problematic. If a seed received a query which included a nested subquery on another table, the seed would not be able to answer this query, so again more complex DQP techniques would be required. This problem would be exacerbated if the nested subquery contained a join.

There may also be problems handling data updates. For example, consider a table, T1, which is being updated by the administrator. Foreign keys in the database schema mean that updating data in T1 also requires an update to another table, T2 to maintain referential integrity. T2 would be stored on a different seed. If a user was able to execute a query on T2 after the update on T1, but before the update on T2, they could access data in T2 that was about to be altered or even deleted. Further inconsistencies could arise if one of the seeds failed and did not perform the update. To prevent these kinds of problems occurring, distributed transactions must be used. These are, by their nature, more complex than standard transactions and are not supported by all DBMS software.

The third approach to designing the seed would be to use horizontal or vertical partitioning. In horizontal partitioning, each table is divided into blocks horizontally, thus for a block size B, the first B tuples are included in the first block, the second B tuples are in the second block, etc. In vertical partitioning, each table is divided according to columns, thus for a block size B, the first B columns would be in the first block, the second B columns in the second block, etc. Whichever approach is used, a seed would hold one block of each table. Like the previous approach, this would require multiple seeds. The first seed would therefore hold the first block of each table, the second seed would hold the second block of each table, etc. A seed will only hold a complete table if that table is small enough to fit in one block.

This approach further parallelises the database and in particular would be more robust to a seed failing. If this occurs, only part of a table would be unavailable, unless there was a table on the failed seed that was small enough to fit on one block. If part of a table was unavailable, downloaders could obtain the other blocks from the other seeds whilst the failed seed was offline. If the seed was only offline for a short time, it may be available again by the time the downloaders have obtained the other blocks from the other seeds and thus there would be little or no increase in the query response time.

Using this approach would alter the basic process of receiving a single-table query. Downloaders would have to request different blocks from the different seeds. This should not be a problem because the Tracker could have a list of which seeds have that particular table and the blocks should have identifiers, so a peer is able to plan how to obtain each block.

However, this approach also has its disadvantages. Like the use of one seed per table, submitting multi-table queries would be problematic. DQP techniques would have to be used, however in this design option it would be more complicated because each table is divided between a number of seeds. Data on one block of the first table may join with data from any block on the second table (which will also be stored on a collection of different seeds), so the results from all tables must be combined and joined somewhere. This would need a very careful query plan to prevent this join occurring on a particularly slow or heavily loaded peer and also to prevent very large amounts of data being transferred between peers. Subqueries would further complicate the query plan.

Updates would again involve distributed transactions, but would be more complex than the previous design option because the block structure of the tables may change. If the table was horizontally partitioned, this would occur if rows were added or deleted in the middle of a table. If the table was vertically partitioned, this would occur if new columns were added or existing columns deleted. Updates may increase or decrease the total number of blocks. If the block structure changes, the data must be re-partitioned amongst the seeds. If an update increased the number of blocks, a seed must be allocated to store each new block. The whole re-partitioning process may take some time to complete, especially if one of the seeds fails during the update.

It is clear that each of the three options discussed above have their own advantages and disadvantages. It has been decided for this thesis to use the first approach, i.e. a single seed holding the entire database. Whilst this may be a performance bottleneck in some circumstances, this model permits multi-table queries to be executed in a relatively simple manner when compared with the other approaches. The ability to handle multi-table queries is important, given that joins are a common operation in database systems and thus will be investigated in this thesis. The single seed would also easily permit future extensions to investigate data updates and nested subqueries.

3.3.2 THE PEERS

The most numerous components in Wigan are the peers. These do not have to be a specialised database server, as noted in Chapter 1 in most cases we expect peers to be standard PCs running database client applications. Apart from the special example of a seed, described above, there are no maximum or minimum numbers of peers which must be present in the Wigan system. Each peer in the system, including the seed, is allocated a unique identifier so that other peers can communicate with it. The peers do not need to be in the same geographic location nor do they need to be on their own private Wigan intranet. Like BitTorrent, there is no assumption made about the amount of time the peers spend connected to the system – a peer may decide to disconnect at any time. This does unfortunately mean that free riding peers are still a possibility. Free riders can be problematic in P2P systems because they consume resources without contributing anything in return as described in Section 2.4. This decreases performance and so if the seed and peers belong to a single organisation, this sort of behaviour could be discouraged.

3.3.3 THE TRACKER

The Wigan Tracker performs the same basic functionality as its namesake in BitTorrent in that it provides the downloading peers with a list of possible uploaders for the query they are requesting. In this thesis, the term "advert" will be used to describe the files (BitTorrent) or the database queries (Wigan) that peers are advertising through the Tracker. However, due to the differences between databases and files, the Wigan Tracker has more functionality. The BitTorrent Tracker simply has to look for those peers which have a full or partial copy of a file, pick up to 50 at random and return a list of these peers to the downloader as described in Section 2.4. Any uploader with a piece of the file will suffice. With regard to matching the adverts to requests, there are only two possibilities – either they match or they do not.

In contrast, in Wigan, the conditions in the "WHERE" clause of a query mean that it is possible that one query over a particular table cannot be resolved by an advert for another query over the same table. The Tracker must filter out these unsuitable adverts and return only those that can resolve the downloader's query. There are four possible relationships between the query and the advert which will now be described.

The first case is that the advert and the query do not match each other at all as shown in Figure 3.2 below.

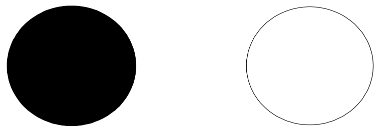


Figure 3.2 – A query (white circle) and an advert (black circle) which do not match

In SQL, an example would be:

Query: "`SELECT item FROM parts WHERE cost <= 10`"

Advert: "`SELECT item FROM parts WHERE cost >= 50`"

These adverts are of no use to the downloader and thus the Tracker must not inform the downloader about such adverts.

The second case is that the query is a partial subset of the advert as shown in Figure 3.3 below.

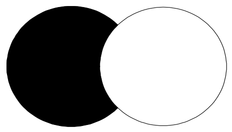


Figure 3.3 – A query (white circle) which is a partial subset of an advert (black circle)

In SQL, an example would be:

Query: "`SELECT item FROM parts WHERE cost <= 10`"

Advert: "`SELECT item FROM parts WHERE cost <=5`"

Such an advert on its own cannot resolve the downloader's query. In the example SQL above, the advert does not provide any information about parts which cost between £5.01 and £10. This information would have to be obtained from another advert. The current Wigan Tracker would not return adverts like this which could not resolve the query completely. However, Chapter 5 discusses ways in which a collection of partial subsets can be returned so that the downloader's query can still be resolved by obtaining different parts of the query from different adverts.

The third case is that the query is a proper subset of the advert as shown in Figure 3.4 below.

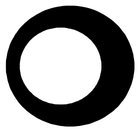


Figure 3.4 – A query (white circle) which is a proper subset of an advert (black circle)

In SQL, an example would be:

Query: "`SELECT item FROM parts WHERE cost <= 10`"

Advert: "`SELECT item FROM parts WHERE cost <= 15`"

Although the query and advert are not identical, the advert can resolve the downloader's query. The seed is a special example of this case. The seed has the entire database hence any query executed on the database is a subset of the seed's data. It is perfectly acceptable for the Tracker to return such adverts, though an exact match is preferable if possible, for reasons that will be described below.

The final case is that the query and the advert are identical as shown in Figure 3.5.

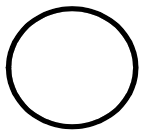


Figure 3.5 – A query and advert which are identical

In this case, there is an uploading peer advertising exactly the same query as is required by the downloader. An exact match is preferable to the previous example of a proper subset, shown in Figure 3.4, for both the uploader and the downloader. The uploading peer will have to perform fewer I/O's and no filtering. If the query is a proper subset of the advert, there is likely to be some data in the advert which the downloader does not require. The uploader, however, has to scan all of the data it holds to see if it should be returned to the downloader hence some data will be scanned but not selected. A particularly bad case of this occurs when a query which has a small result set is executed at the seed. An entire table may be scanned for only a few results to be obtained. Indexing may help overcome this problem and this will be discussed further in Chapter 5. However, with an exact match, this issue will not happen as all data scanned is returned as part of the result set. From the downloader's point of view, it is likely that an exact match will also take less time to be returned. This will be discussed in more detail later in this chapter. Chapter 4 will introduce experiments indicating the relationship between the number of pieces being sent and the overall query response time.

Another major difference between the BitTorrent and Wigan Trackers is that, instead of receiving a request for a particular file, a Wigan downloader sends a database query. In the simulation of Wigan introduced in Chapter 4 and in the examples discussed in this thesis these queries are expressed in SQL. Although some of the existing P2P database systems, for example BioWired [59] and PIER [62, 63], have their own query processing languages, SQL provides all of the required query processing expressions and also offers scope for more complex queries, for example nested subqueries. It is also a common, standardised language known to many developers and used in many applications. This may allow future versions of

Wigan to integrate with existing database applications. Note that it should also be possible to implement the Wigan architecture using other existing query languages if required.

A downloader's message to the Tracker takes the form shown in Figure 3.6.

Message Type = InitialDownloader Request	Sender	Query ID

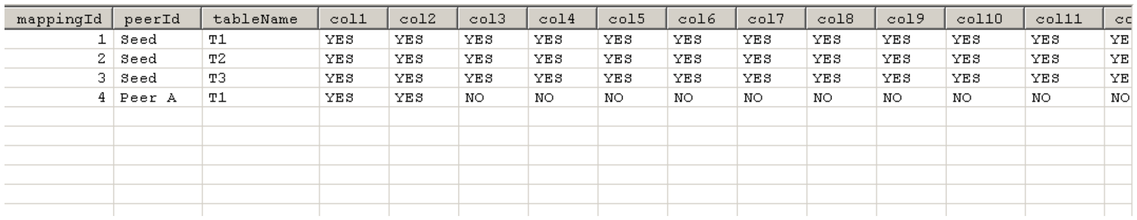
Figure 3.6 – a downloader's message to the Tracker

The message has three components. The first of these is the message type. The sender is is the ID of the downloading peer. The final component is the SQL query that the downloader wishes to execute.

The Tracker holds all the uploaders' advertisements, in a database using a canonical form representing the table names, *columns* from the tables and *conditions* on these columns (the elements contained within a "WHERE" clause in SQL) for each query. This allows for SQL queries written in any order, for example in the Tracker's canonical form, the query "`SELECT name, address, phone FROM person`" will be identical to the query "`SELECT name, phone, address FROM person`".

When the Tracker receives a downloader's query, it must examine the adverts in its database and compare these to the query in order to find relevant adverts. The Tracker's database contains two tables, one of which contains data about the *columns* being advertised and the other containing details of the *conditions* on these *columns*. Two examples of this matching process will now be described.

Consider a database containing three tables, T1, T2 and T3. The following example illustrates a scenario where a peer, Peer A has just completed the download of the query "SELECT x, y FROM T1 WHERE y < 100". Peer A is the first peer to finish downloading a query; hence the Tracker contains details of only the seed's advert and Peer A's advert at this point. The process of contacting the Tracker on completion of a query, in order to advertise the results will be discussed in more detail later in this section. Figure 3.7 shows the resulting Tracker database table for *Columns*.

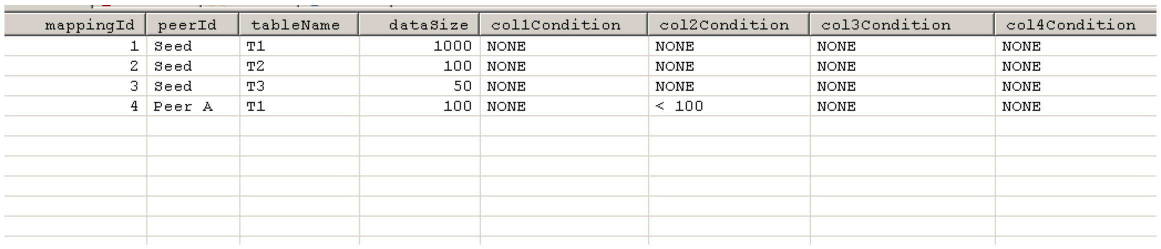


mappingId	peerId	tableName	col1	col2	col3	col4	col5	col6	col7	col8	col9	col10	col11	cc
1	Seed	T1	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YE
2	Seed	T2	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YE
3	Seed	T3	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YE
4	Peer A	T1	YES	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO

Figure 3.7 – the Tracker's database for columns information, with only the seed and a single peer advertising

The first column is a mapping ID and is used by the Tracker to join rows from the *Columns* and *Conditions* tables when matching queries to adverts. The second column shows the ID of the peer placing the advertisement and the third column shows the table the advertisement covers. The remaining columns show, for each *column* in the table being advertised, if that advert contains those *columns* or not. For example, Peer A is shown as having only two *columns* from the table T1, but the seed has all *columns* from all tables. If a peer had performed an aggregation on a *column*, for example "MAX", this would be shown instead of "YES". An example of this will be shown later in this section.

Figure 3.8 shows the Tracker database table for *Conditions*.



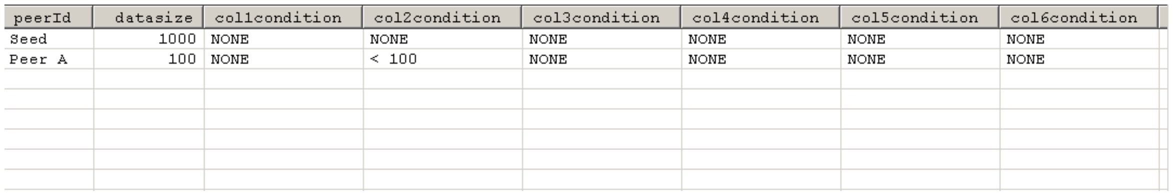
mappingId	peerId	tableName	dataSize	col1Condition	col2Condition	col3Condition	col4Condition
1	Seed	T1	1000	NONE	NONE	NONE	NONE
2	Seed	T2	100	NONE	NONE	NONE	NONE
3	Seed	T3	50	NONE	NONE	NONE	NONE
4	Peer A	T1	100	NONE	< 100	NONE	NONE

Figure 3.8 – the Tracker's database for conditions with only the seed and a single peer advertising

The first column is a mapping ID which corresponds to the mapping ID in the *Columns* table. The peer ID and the table name are the next two columns. The fourth column shows the number of tuples received by the uploading peer. The remaining columns show *conditions* on *columns* in the table being advertised. For example, Peer A has no *condition* on the first *column* (x), but there is a *condition* on the second *column*, because the "WHERE" clause of Peer

A's query contains "y < 100". The seed has no *conditions* on any *columns*, because it holds all data from all tables.

Now consider a new downloader, Peer B, which submits the query
"`SELECT x, y FROM T1 WHERE y < 10`" to the Tracker. This query is a proper subset of Peer A's. The Tracker initially retrieves all peers which have the correct *columns* from the relevant table, T1 in this example. This retrieval process will return both the seed and Peer A as each possesses data from the first two *columns*, as shown in Figure 3.9.



peerId	datasize	col1condition	col2condition	col3condition	col4condition	col5condition	col6condition
Seed	1000	NONE	NONE	NONE	NONE	NONE	NONE
Peer A	100	NONE	< 100	NONE	NONE	NONE	NONE

Figure 3.9 – the result of the initial selection process

The Tracker then checks the *conditions* on the relevant *columns* for these advertisements. For the first *column* (x), the seed has no *conditions* and neither does Peer B's query. For the second *column* (y), the seed also has no *conditions* and Peer B's query has a *condition* (y < 10). The seed's data will include items where y < 10, which is acceptable. The Tracker must also ensure there are no *conditions* on other *columns* which will restrict the result set, for example an advert for
"`SELECT x, y FROM T1 WHERE x < 5 AND z < 10`" could not resolve the query
"`SELECT x, y FROM T1 WHERE x < 5`"
because the *condition* "z < 10" further restricts the result set and will not necessarily return all items where x < 5. There are no such additional *conditions* here however, thus the seed's advert is accepted.

The Tracker then examines Peer A's advertisement. Like the seed, there is no *condition* on the first *column*. However, there is a *condition* (< 100) on the second *column*. These *conditions* are checked against each other. Given that anything that is less than 10 will be included in a results set containing data less than 100 and Peer A has no other *columns* which could affect the result set, this advert is also accepted. Once both relevant adverts have been checked, the selection process terminates. In this example, the Tracker has found two adverts that could satisfy the downloader's query. Note that both of these adverts are for different queries, neither of which was that requested by the downloader.

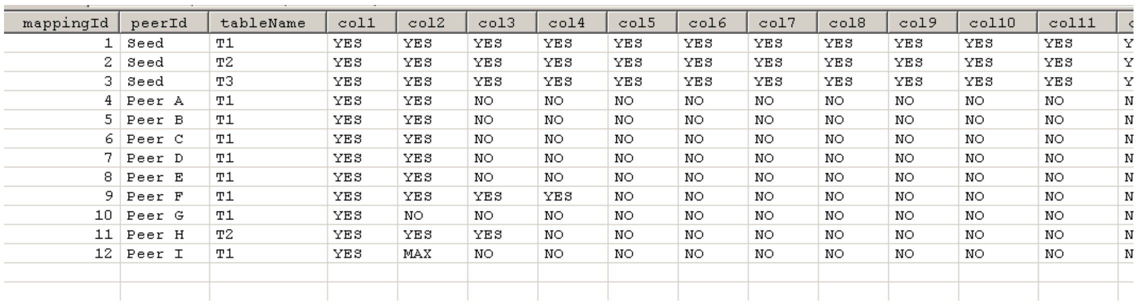
A downloader should not be simultaneously receiving results from two uploaders advertising different queries for reasons which will be explained later in this chapter. Therefore, to prevent this occurring, the Tracker groups the adverts by query. For each group, it lists the number of tuples – and also the number of pieces – in addition to the ids of the peers with that query. These are ordered based on a priority system, which favours adverts matching the downloader's query for the reasons explained earlier. If, as in this example, none of the adverts match the downloader's query, the highest priority is given to the closest to an exact match. The closest is given in terms of the number of tuples in the query results set. In this example, Peer A has 100 tuples but the seed has 1,000 and hence Peer A is the closest to an exact match. Note that if there are multiple adverts that could answer the query, then the seed will always have the lowest priority as there is another advert which will have a smaller results set to examine. For this example, if we assume 50 tuples per piece, the groups will be ((100, 2, Peer A), (1000, 20, Seed)). The Tracker then returns the groups to the downloader. Such messages take the format shown in Figure 3.10.

Message Type = QueryGroups	Query	Query Groups

Figure 3.10 – the Tracker's return message to a downloader

The downloader's query is also included in the return message from the Tracker. This is because a downloader may be submitting multiple queries and must be able to relate the adverts returned by the Tracker to a particular query. Note that the message sender does not need to be specified because this message could only have originated from the Tracker.

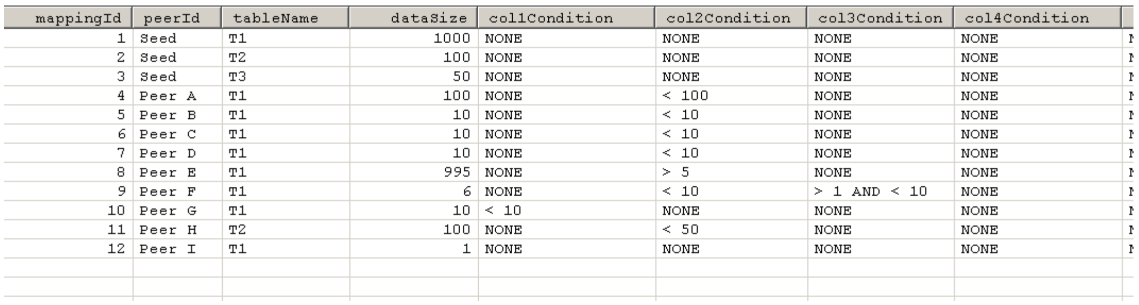
The example described above was fairly straightforward because there were only two adverts, both of which could satisfy the downloader's query. A more complex example will now be described. Consider the same database used in the previous example. Over time, more peers have downloaded queries and are advertising at the Tracker, as shown in Figure 3.11 (*columns*) and Figure 3.12 (*conditions*).



mappingId	peerId	tableName	col1	col2	col3	col4	col5	col6	col7	col8	col9	col10	col11	c
1	Seed	T1	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	Y
2	Seed	T2	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	Y
3	Seed	T3	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	Y
4	Peer A	T1	YES	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	N
5	Peer B	T1	YES	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	N
6	Peer C	T1	YES	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	N
7	Peer D	T1	YES	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	N
8	Peer E	T1	YES	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	N
9	Peer F	T1	YES	YES	YES	YES	NO	NO	NO	NO	NO	NO	NO	N
10	Peer G	T1	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	N
11	Peer H	T2	YES	YES	YES	NO	NO	NO	NO	NO	NO	NO	NO	N
12	Peer I	T1	YES	MAX	NO	NO	NO	NO	NO	NO	NO	NO	NO	N

Figure 3.11 – the Tracker's database for columns later in the download period

Note that the final advert, placed by Peer I in Figure 3.11 is for a query which includes an aggregation ("MAX") on column 2.

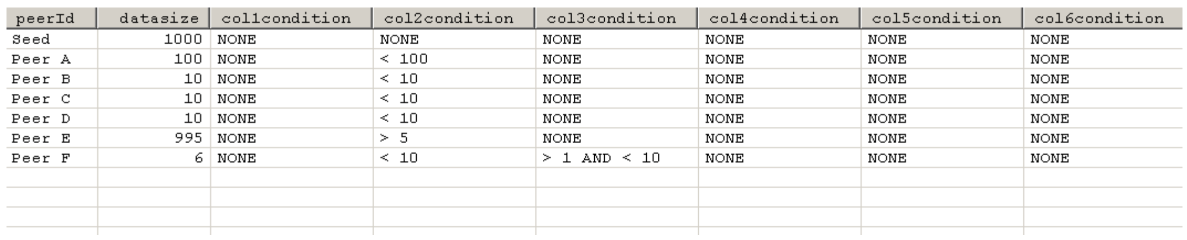


mappingId	peerId	tableName	dataSize	col1Condition	col2Condition	col3Condition	col4Condition	
1	Seed	T1	1000	NONE	NONE	NONE	NONE	N
2	Seed	T2	100	NONE	NONE	NONE	NONE	N
3	Seed	T3	50	NONE	NONE	NONE	NONE	N
4	Peer A	T1	100	NONE	< 100	NONE	NONE	N
5	Peer B	T1	10	NONE	< 10	NONE	NONE	N
6	Peer C	T1	10	NONE	< 10	NONE	NONE	N
7	Peer D	T1	10	NONE	< 10	NONE	NONE	N
8	Peer E	T1	995	NONE	> 5	NONE	NONE	N
9	Peer F	T1	6	NONE	< 10	> 1 AND < 10	NONE	N
10	Peer G	T1	10	< 10	NONE	NONE	NONE	N
11	Peer H	T2	100	NONE	< 50	NONE	NONE	N
12	Peer I	T1	1	NONE	NONE	NONE	NONE	N

Figure 3.12 – the Tracker's database for conditions later in the download period

Note that the advert placed by Peer F (mapping ID 9) shows an example of a query where multiple *conditions* occur on the same column, in this case "> 1 AND < 10" in column 3.

A new downloader, Peer J, submits "`SELECT x, y FROM T1 WHERE y < 10`" to the Tracker. Once again, the Tracker will retrieve all peers with the correct *columns* from the relevant table. The results of this query are shown in Figure 3.13.



peerId	datasize	col1condition	col2condition	col3condition	col4condition	col5condition	col6condition
Seed	1000	NONE	NONE	NONE	NONE	NONE	NONE
Peer A	100	NONE	< 100	NONE	NONE	NONE	NONE
Peer B	10	NONE	< 10	NONE	NONE	NONE	NONE
Peer C	10	NONE	< 10	NONE	NONE	NONE	NONE
Peer D	10	NONE	< 10	NONE	NONE	NONE	NONE
Peer E	995	NONE	> 5	NONE	NONE	NONE	NONE
Peer F	6	NONE	< 10	> 1 AND < 10	NONE	NONE	NONE

Figure 3.13 – the result of the initial selection process

It can be seen that not all of the advertisements shown in Figure 3.11 and Figure 3.12 have been included. Peer G's advert was rejected because it has only one of the required *columns*, x. Peer H's advert was for a query on one of the other tables. Finally, the advert placed by Peer I was rejected because it contained an aggregation ("MAX") on the second *column*, y, and not the complete column data.

The Tracker will now examine the *conditions* in each of the advertisements to see if they can be used to satisfy the downloader's query. Again, the adverts placed by the seed and Peer A can be accepted, as discussed in the previous example.

The next three adverts, placed by Peers B, C and D can be accepted quickly because they contain identical *conditions* to the downloader's query – nothing on the first *column* and "< 10" on the second.

Peer E's advertisement is checked next. The *conditions* on the first *column* are acceptable (neither the query nor the advert has any), but those on the second *column* are not. The downloader is interested in items where y < 10, but Peer E is advertising data on items where y > 5, thus Peer E's advert is rejected.

The *conditions* on the first two *columns* in Peer F's advert are the same as those in Peer J's query. However, Peer F has a condition on another *column* ("> 1 AND < 10" on *column* 3) which means that this advert will not be able to answer the query. There may be some tuples which match Peer J's query that are not included in Peer F's results and hence this advert is rejected.

At this point, all the relevant adverts have been checked with those placed by the seed and Peers A, B, C and D being accepted. These adverts will now be placed in groups and returned to the downloader. The grouping process is the same as described in the previous example, however this time there are more groups because there are more uploaders with different queries. Assuming 50 tuples per piece again, the groups returned to Peer J are ((10, 1, Peer B, Peer C, Peer D), (100, 2, Peer A), (1000, 20, Seed)). It can be seen that the first group contains multiple uploaders. These peers are all advertising the same query and hence are all included in the one group.

Later in the download period, there may be a very large number of peers able to answer a new downloader's query. In BitTorrent, the Tracker selects 50 random peers' adverts to return if there are more than 50 suitable peers available, as described in Chapter 2. The Wigan Tracker follows a similar approach, however, the selection of peers is based on the query groups described above. The selection is chosen on the closeness of the advert to the downloader's query. Returning to the first example from this section, if there were another 49 adverts identical to Peer A's in addition to the seed's advert, there would be a choice – either return the seed's advert and 49 of the others, or return Peer A's advert and the other 49 identical adverts. Given that Peer A's advert is closer to the downloader's query than the seed's, this latter approach is taken.

Now, consider a scenario later in the download period where there are not yet 50 adverts for the downloader's query, but more than 50 other adverts which can satisfy the query. The Tracker examines the groups, starting with that which is furthest away from being an exact match. This is measured by the number of tuples. If this entire group can be removed from the set of adverts found, still leaving 50 or more adverts, then it is. If it cannot, then adverts are removed at random until only 50 adverts remain in total. Note that adverts are only removed from this particular result set and are not deleted from the Tracker's list of advertisements. If there are 40 adverts for some query Q, 15 adverts for a query, Q1, of which Q is a subset and the seed, the Tracker will return the 40 adverts for Q and a random selection of 10 adverts for Q1. This process is illustrated in Figure 3.14.

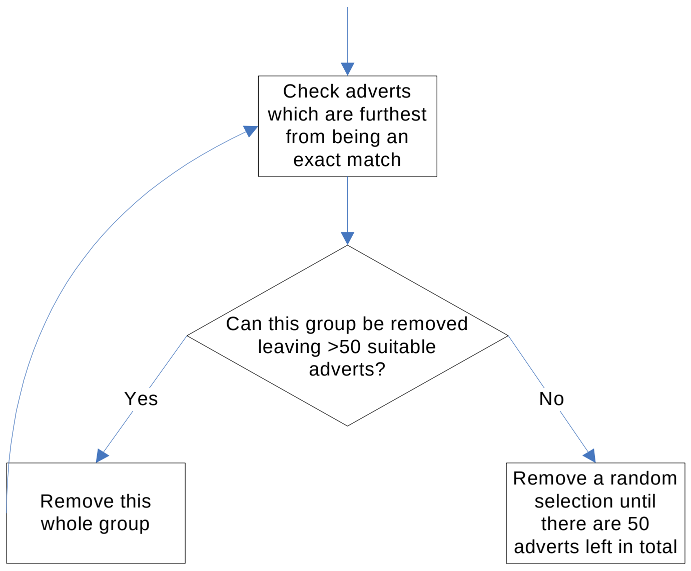

Check adverts which are furthest from being an exact match
Can this group be removed leaving >50 suitable adverts?
Yes
No
Remove this whole group
Remove a random selection until there are 50 adverts left in total

Figure 3.14 – Choosing 50 adverts from the selection.

In addition to providing adverts to downloaders, the Tracker performs other advertisement management functions. When a peer completes its download and then wishes to start advertising a query, it sends this query to the Tracker. The message sent by the downloader to the Tracker is shown in Figure 3.15.

Message Type = AdvertiseQuery	Sender	Query	Tuples Received

Figure 3.15 – the completion message sent by peers to the Tracker

The sender is a vital part of this message, so the Tracker can refer future downloaders to the correct peer. The query must also be included so that the Tracker is aware of what the peer wishes to advertise. The peer also includes the number of tuples received in the message. This is stored in the Tracker's database, as seen in the examples above, and is used when sorting the adverts into query groups. Once the query has been parsed into the Tracker's canonical form, it is stored in the Tracker's database and is available for use by future downloaders.

Similarly, the Tracker also receives messages from uploaders that wish to disconnect from the system. An example of such a message is shown in Figure 3.16.

Message Type = Disconnect	Sender

Figure 3.16 – a disconnection message

The disconnection message is very straightforward. It simply includes the message type, so that the Tracker can distinguish that a peer is going to disconnect and the ID of the peer that is disconnecting. On receipt of a disconnection message, the Tracker will delete all advertisements placed by the sender. This prevents a downloading peer being informed of adverts that are no longer available.

It is important to note here that the Tracker is only responsible for managing adverts and matching them to queries. The Wigan Tracker does not contain any query results itself. Also, as in BitTorrent, the Tracker does not control the download process; this is left to the downloading peer. Whilst it is logical for peers to begin downloading from the closest possible group to an exact match, i.e. the first group in the list provided by the Tracker and thus the reason why the Tracker sorts the groups in the manner it does, the Tracker does not force peers to adhere to this policy. It does not prevent peers using other metrics (some possibilities of which are discussed in Chapter 5) to decide which group of peers to contact first, nor does it ever attempt to execute queries on the downloaders' behalf.

3.4 DOWNLOADING AND UPLOADING

The process of downloading will now be described, followed by a description of the uploading process.

3.4.1 DOWNLOADING

A new downloader that wishes to submit a query must contact the Tracker, sending a message of the format shown previously in Figure 3.6. The Tracker, using the processes described in Section 3.3.3, will find some suitable adverts, group them as illustrated in Section 3.3.3 and send the downloader a message as illustrated earlier in Figure 3.10. On receipt of this message, the downloader must extract the information and pick a query group. It is logical, for reasons described in Section 3.3.3, to start with those queries which exactly match the one it is searching for if this is possible or if it is not, to start with the closest to an exact match. This overall process is shown in Figure 3.17.

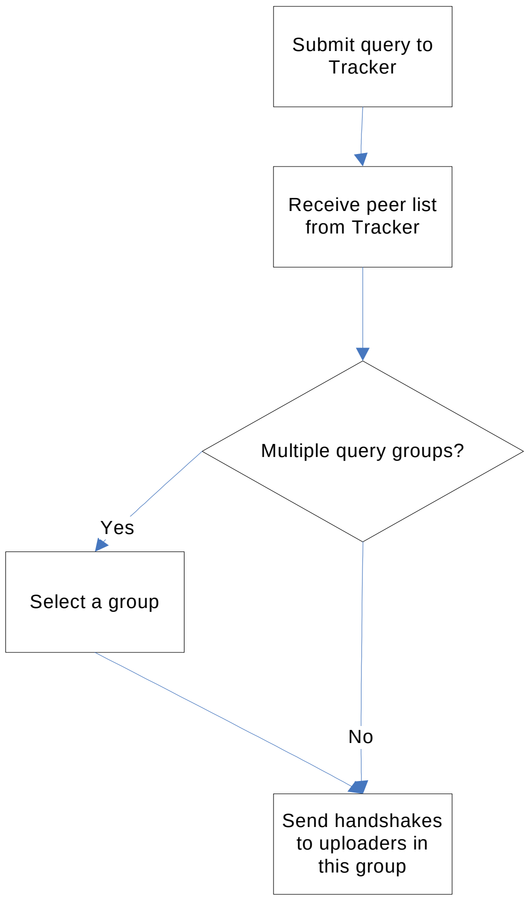

Submit query to Tracker
Receive peer list from Tracker
Multiple query groups?
Yes
Select a group
No
Send handshakes to uploaders in this group

Figure 3.17 – A downloader contacts the Tracker and receives a peer list

The initial contact protocol which the downloader follows is very similar to BitTorrent. The downloader despatches a handshake message to the uploaders, the one difference between Wigan and BitTorrent is that the Wigan downloader sends the message only to those uploaders in the group it has chosen. This is because a downloader only uses one uploader group at a time so it has no need to handshake uploaders in other groups for the time being. The uploaders receive this handshake and send a confirmation message in reply. Both handshake requests and responses take the format shown in Figure 3.18.

Message Type = HandshakeRequest/ HandshakeResponse	Sender	Query

Figure 3.18 – The format of handshake requests and responses

A query ID (or possibly the actual query) is included because the downloader may be working with multiple queries simultaneously and needs to know which query the handshake response refers to.

The downloader then verifies that each uploader has the same number of pieces that the Tracker stated. A piece number verification message also takes the format shown in Figure 3.18, however the message type will be 'PieceVerification'. This message must contain some details of the query or table because the uploader may be advertising multiple queries and needs to know which one the verification request refers to. Note that the downloader sends out a single verification message for the whole query, it does not send out a separate verification message for each individual piece. This is because the uploader has a record of the number of pieces it has and can simply return that number. Separate verification messages for each piece would take a long time if there was a large number of pieces to verify. The corresponding reply, a piece confirmation message, is shown in Figure 3.19.

Message Type = PieceInformation	Sender	Query	Number of pieces

Figure 3.19 – A piece confirmation message

This message must also include some details of the query in case the downloader has submitted multiple query requests. The additional field is the piece confirmation – the number of pieces the downloader can expect to receive.

In BitTorrent, the downloader must choose the order in which it will request pieces, but in Wigan, the pieces can be requested in sequential order because no uploading peer may possess partial results, received out of numerical order, for reasons that will be discussed later in this Chapter in Section 3.4.2. Once the downloader has received the confirmation messages from every potential uploader, the downloader sends an uploader a request for the first piece, including in its request the piece number, the SQL query and a query ID. This format is shown in Figure 3.20. If there are several possible uploaders, one is selected at random.

Message Type = PieceRequest	Sender	Query	Piece number	Query ID

Figure 3.20 – A piece request message

There are two possible responses from the uploading peer. It may choose to choke the downloader, or it may send back all the rows from the first piece that match the query. The former response will be examined in more detail later in this section. If the uploader does return the data, the message it sends contains the query, piece number, the same query ID that was sent in the request, plus all of the tuples in the piece that match the query. This is shown in Figure 3.21.

Message Type = ResultTuples	Sender	Query	Piece number	Query ID
Any tuples in this piece which match the conditions of the query				

Figure 3.21 – A message containing a piece of data

The inclusion of the query ID is for verification, so the downloader is 100% certain to which request the response is related and allows downloaders to be receiving results of multiple queries at the same time. Once the downloader has stored any tuples it has received, it will send a request for the second piece, again picking an uploader at random if there is more than one. This process continues until the downloader receives the all of the pieces. The downloader knows when this point occurs because the Tracker has informed it of the total number of pieces, as described in Section 3.3.3. The downloader will then contact all the uploaders it has sent handshakes to and inform them it no longer wishes to receive data from them. A lack of interest message is shown in Figure 3.22.

Message Type = NoInterest	Sender

Figure 3.22 – a lack of interest message

At this point, a free-riding downloader, as described in Chapter 2, will quickly disconnect. Other downloaders would now however become uploaders and would contact the Tracker as described in Figure 3.15 in Section 3.3.3. The process of uploading will be described in Section 3.4.3.

Alternatively, a downloader will receive a choke message if the uploader is too busy to handle a new downloader. A choke is shown in INSERT REF and contains only the message type and the ID of the peer that is choking the downloader.

Message Type = Choke	Sender

Figure 3.23 – a Choke message

On receipt of the choke message, the downloader must make a decision about the course of action to take next. This decision process is shown in Figure 3.24:

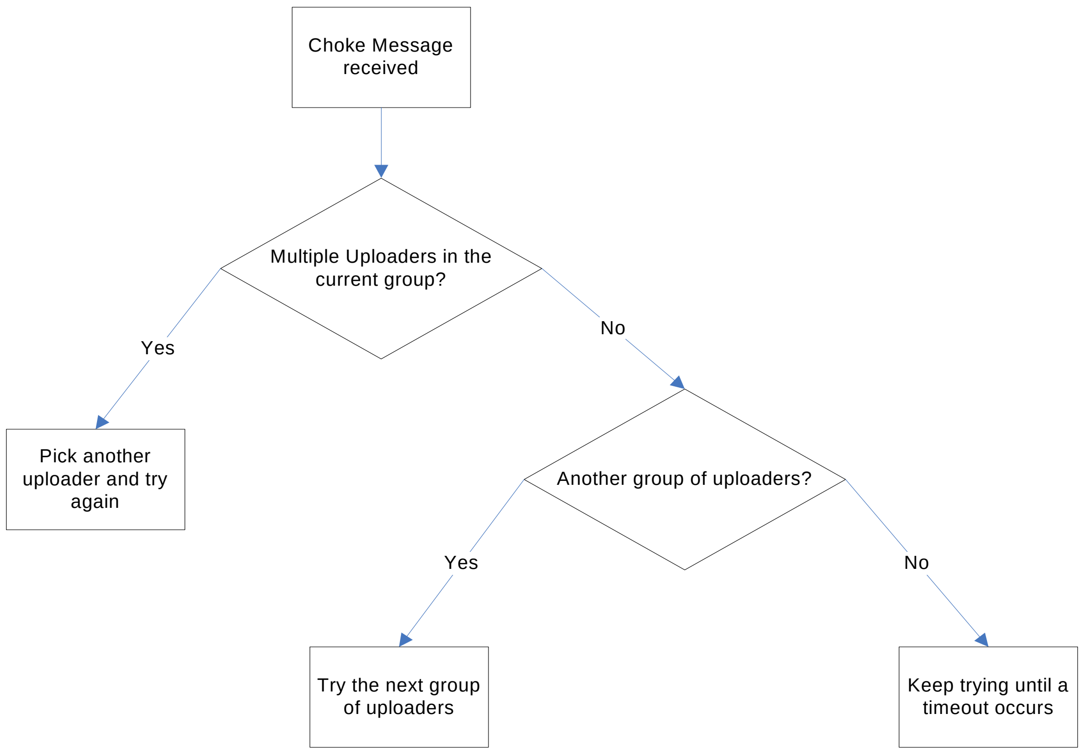

Choke Message received
Multiple Uploaders in the current group?
Yes
No
Pick another uploader and try again
Another group of uploaders?
Yes
No
Try the next group of uploaders
Keep trying until a timeout occurs

Figure 3.24 – A Wigan downloader's response to receiving a choke message.

If there are multiple uploading peers to choose from in the current query group, the downloader will select another uploader at random and send it the same piece request. Alternatively, if the peer list sent by the Tracker is split into multiple query groups and the current group includes only the choking uploader, the downloader will try one of the other groups and contact a randomly chosen uploader in that group. If the downloader only knows about the seed then it has no option but to try again and send the seed another request. This might have to continue for some time until the downloader gives up and contacts the Tracker again – discussed in more detail in the following paragraph – or the seed is able to answer the query. This can happen if one of the other downloaders being served by the seed has completed and thus the seed has a vacant slot when the downloader's request arrives.

In BitTorrent and in Wigan, peers contact the Tracker to provide an update on their progress at intervals and to ask for more peers if needed. In BitTorrent, this value is decided by a Tracker. The default time for this in Wigan is currently every 700 seconds. There are two important factors that a timeout value should have. It should not so long that a peer which is being snubbed, i.e. choked by every uploader that it knows about, is kept waiting for a very long time before contacting the Tracker again to ask for more peers. However, a timeout interval should also be long enough to allow a table of several thousand tuples to be despatched in its entirety without unnecessarily timing out and asking for additional peers. Furthermore, it should also be long enough to give requests for further peers a chance to be met. Consider a peer which has submitted a query request shortly after the data was released and has been choked by the seed, which is the only uploader. If the downloader times out instantly and requests more peers, it is likely that there will still only be the seed uploading. Allowing more time before timing out gives other peers a chance to start uploading and hence when the downloader receives a new peer list from the Tracker, it is more likely that there will be additional uploaders apart from the seed. The value of 700 seconds meets these characteristics. Note that the timeout value could be amended for different database systems, depending on their table structure. If all of the tables were very small – and thus could be transferred very quickly – the timeout interval could be set to a much lower value of one or two minutes.

If the download completes before the timeout interval expires, the downloader will never need to ask the Tracker for more peers because it has been able to receive all the results using the initial set of peers provided by the Tracker.

Requests to the Tracker for more peers take the same format as the initial request, shown in Figure 3.6 in Section 3.3.3, except that the message type will have a different value (AdditionalPeers), to enable the Tracker to distinguish between an initial request and a timeout.

The result of any requests to the Tracker for more peers is identical to the initial request for peers, shown in Figure 3.10 in Section 3.3.3, though again the message type ID will differ so that the downloader can distinguish this is a revised peer list. The downloader can then take one of three actions, depending on the contents of the new peer list. This is shown in Figure 3.25:

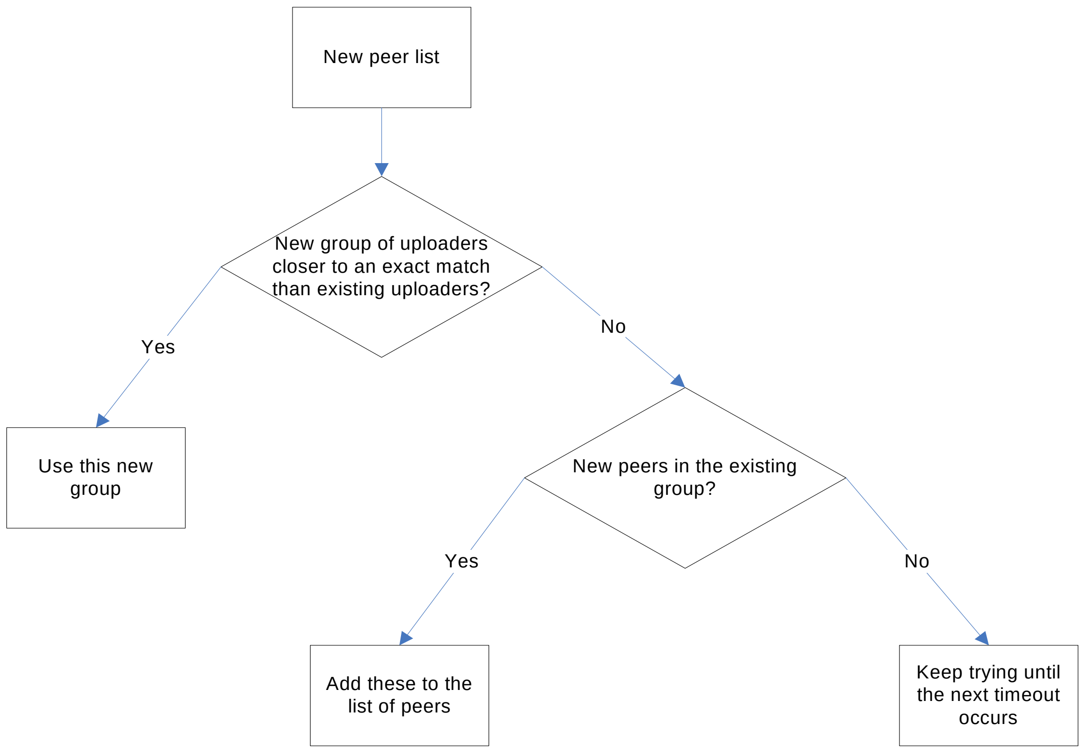

New peer list
New group of uploaders closer to an exact match than existing uploaders?
Yes
No
Use this new group
New peers in the existing group?
Yes
No
Add these to the list of peers
Keep trying until the next timeout occurs

Figure 3.25- A downloader receives a new peer list from the Tracker

The list sent by the Tracker might be identical to the set of peers that the downloader currently possesses. This could happen, for example, if the downloader is submitting a query that no other peer has ever submitted so only the seed can resolve the query. In this case, there is little else the downloader can do, so it continues trying to obtain the query results from the peers on the list. Alternatively, the list might contain the same query groupings as the current list, but with some additional peers. This is possible if more peers have started advertising in the time since the downloader last contacted the Tracker. In this case, the downloader will go through the list, send handshake messages to those uploaders that were not included in the previous list and verify they possess all the pieces. The downloader can then ask these peers for a piece of the query as and when it needs to. Note that the downloader will not send out handshakes again to peers that the Tracker had previously informed them about because this would be an unnecessary repeated communication. Finally, the list might contain a new query group which is closer to an exact match. This could happen if there was only the seed available when the downloader first contacted the Tracker and then before the downloader times out, other peers have downloaded and then started advertising the downloader's query. If this occurs, the downloader will delete any data it has received so far because the new group will have a different piece structure (see below). The downloader will then inform the previous set of uploaders that it is no longer interested in downloading from them, sending a lack of interest message as described earlier in this section, and restart the download process using the new group of uploaders.

3.4.2 CHANGING THE PIECE STRUCTURE

Unlike BitTorrent, a peer may have to amend the piece structure in Wigan, on completion of its download, before contacting the Tracker to begin advertising. This process will now be described. A BitTorrent peer can start uploading as soon as it has received a single piece of the file. This is because BitTorrent downloaders can only receive pieces of a file from uploaders advertising the same file. The lifecycle of a BitTorrent peer has already been introduced in Section 2.4 of Chapter 2 and is summarised again in Figure 3.26:

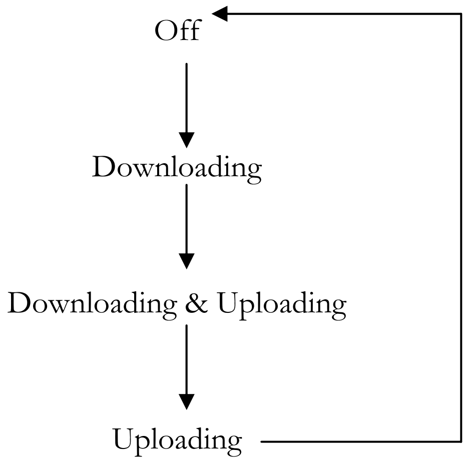


Figure 3.26 - The lifecycle of a BitTorrent Peer

However, as discussed in Section 3.3.3, in a database such as Wigan queries may be resolved by adverts for other queries, providing the query is a subset of that advert. If a downloader receives its results from an uploader advertising a different query, the downloader will amend the piece structure before it begins to upload.

For example, consider a university department's database, which has a Student table containing details of all students studying in the department. This table is split into 20 pieces. Initially, there is one seed containing the whole database and therefore a complete copy of the Student table. A new downloader requests the following query:

```
SELECT * FROM student WHERE tutor = ‘Professor Lee’
```

During the download, the downloader will send 20 requests to the seed, despatching messages of the format shown earlier in Figure 3.20, one for each piece. For each request, the seed will send data from that piece, in the message format shown earlier in Figure 3.21, containing details of all students whose tutor is Professor Lee. Let us assume that there are 20 such students. There is no guarantee of how these 20 students' details are distributed across the pieces. They may all be stored in one piece, in which case the downloader will receive 19 empty responses. This happens because the downloader has to request data from each piece. It cannot predict in advance which pieces will contain data matching the query. If the downloader only requested data from a selection of pieces, there is a risk that it will not receive all of the data. At the other extreme, each piece may contain one tuple of data which matches the query, in which case there will be 20 responses each containing only one tuple. Let us suppose, the seed has 13 pieces containing some data which matches the downloader's query and seven other pieces whose data does not match the downloader's query. The downloader will receive 20 responses, seven empty and 13 containing a small number of tuples.

When the downloader makes this data available to other peers, it would not make sense from an efficiency point of view to have 20 pieces again. Instead, the 20 tuples which match the query could all be grouped together in one piece. This prevents the downloader advertising seven empty pieces and 13 pieces that contain only a small amount of data, which would be inefficient.

Note that if the query results were too numerous to fit into a single piece, the downloader would combine as many tuples as possible into one piece and then keep repeating this process until there were no tuples left. For example, if the downloader filled a piece and there were ten tuples left over, these ten tuples would go on a second piece. This is one feature of Wigan that does not occur in BitTorrent – pieces in Wigan do not have to be full of data. In BitTorrent, an empty or partially empty piece would not make sense because a downloader would not want a partial file. However, in Wigan, it is quite possible that the size of some query results is not large enough to fully occupy a complete piece. An example would be a query containing an aggregate function, such as "MAX", which would return only a single tuple.

It is important, therefore, that tuples are mapped correctly into pieces. When the seed creates the database, each tuple in each table is given an ID. This ID can be determined by the database administrator during database setup. If the database already has a separate key field, like TPC-H [74], this can also double as the tuple ID. The seed contains a metadata table of tuple to piece mappings using the tuple ID and the piece number. Note that if the data is retrieved by ID, this metadata table only needs to store the piece boundaries, e.g. piece one contains tuples 1-50. When the seed receives a query request, it will add the tuple ID for each tuple in the result set. Other peers will also store this mapping for the data they have received and will again include the tuple ID in data they despatch to downloading peers.

The piece structure does not always have to change. If a peer received its results from uploaders advertising the exact same query it requested, there would be no need to change the piece structure because the data would already be arranged on the smallest possible number of pieces. An example would be a second downloader contacting the first peer to download the query and requesting the same query:

```
SELECT * FROM student WHERE tutor = ‘Professor Lee’
```

Indeed, the empty responses highlight the reason why a downloader can benefit from receiving data from an uploader which already has the results of the exact same query. Consider the case described earlier in this chapter where a query with a small result set is executed at the seed. The seed will have to check every piece of the table and therefore the downloader may receive many responses indicating that there is no data in a particular piece which matches the query. If, however, the downloader is receiving data from a peer with exactly the same query that it requires, this will not occur. All of the pieces the downloader despatches will contain data.

There is another issue which must be resolved whilst creating the new piece structure. If there are several uploaders advertising the same query, it is vital that each piece at one uploader is identical to the corresponding piece at the others. For example, piece number one at an uploader must contain exactly the same tuples as piece number one at all the other uploaders advertising that query. If the pieces were not the same, there is a risk that a downloader receiving different pieces from different uploaders may receive multiple copies of some tuples and no copies of other tuples. The downloader could detect multiple copies of the same tuple if it encountered the same tuple ID twice in the results it received, but could not detect if it were missing any tuples from the result set. It is aware of how many tuples the uploader has in total, due to the information supplied by the Tracker, but neither the downloader nor the Tracker can predict how many of the uploader’s tuples match the conditions of the downloader's query. Therefore, it is vital that all uploaders advertising the same query create the same piece structure.

There is the possibility that, if two downloaders are receiving the query concurrently, each downloader will pack the tuples into pieces in their own, different manner. There are several possible means of overcoming this problem. Firstly, the Tracker could queue requests for the same query, if it knew that another peer was already executing that query. The first peer would then choose the new piece structure for the query. Once it completed the query, the queued peer requests could then be directed to it. However, this could be problematic if the query was very popular. All the requests queued at the Tracker would go to the first peer that had received the query which could then be swamped by requests. Indeed, if the query were exceptionally popular, the result would be the seed answering only a few queries and the first peer to advertise the results receiving more queries than it could handle. Given that the seed is likely to be a more powerful computer than an individual peer; this is not an ideal scenario. Therefore, Wigan uses an alternative algorithm. Once an uploader receives all the query results, it orders the tuples based on the tuple ID and then creates the new piece structure. This means that it does not matter how many downloaders are receiving the query concurrently, the order of the tuples within the pieces will always be identical.

Whilst all adverts for one particular query will have the same piece structure, there is no guarantee that two different queries on the same database or even on the same table or tables will have the same piece structure because different queries could range in size from one tuple up to every tuple in the table. This will mean that a downloader can only receive data at any one time from uploaders which all have the same query. Consider a downloader that wants this query:

```
SELECT * FROM student WHERE course = ‘G402’ AND stage = 3;
```

Consider three possible uploaders – the seed and two other uploaders, one, U1, which has already downloaded the same query requested by the downloader and another, U2, which has the results of this query:

```
SELECT * FROM student WHERE course = ‘G402’;
```

All three of these peers could answer the query. However, each of these adverts could have a different number of pieces. Each of the three peers will have their own piece number one which may not necessarily be the same as piece number one at either of the other uploaders. The seed may be the only one to have a piece numbered 20. A tuple in piece 15 at the seed may be in piece 2 at uploader U1 and in piece 3 at uploader U2. The seed and possibly peer U2 will contain some tuples that do not match the downloader's query. Attempting to download data from these three mixed sources could result in duplicate copies of data being received or some data being omitted. In certain circumstances, it could even lead to a peer receiving no data at all. Consider a downloader which is downloading the results of a query, Q from two uploaders, U1 and U2 advertising queries Q1 and Q2 respectively. There are only 10 tuples which match Q. The results of Q1 and Q2 can both be divided into three pieces. Let us assume that the tuples matching the query are on the pieces 1 and 2 at U1 and on pieces 3 and 4 at U2.

If the downloader requests the first two pieces from U2 and the second two pieces from U1, the downloader receives no tuples at all. It has received responses for all four pieces and has no indication that these are the incorrect results. The Tracker will have stated that U1 and U2 have four pieces each, but because they are advertising different queries, the Tracker cannot state how much data held by these peers matches the downloader's query. The downloader will then contact the Tracker to announce that the query Q has an empty results set (note that empty results sets will be discussed in more detail in Section 3.4.4). The Tracker could then pass this incorrect information on to other peers and hence the incorrect results may spread around the Wigan network.

Therefore, the set of uploaders that a downloader is using at any moment in time must all be advertising the same query. Note that this query may or may not be the same as that requested by the downloader, the key issue is that all the uploaders being utilised have the same query as each other.

This requirement for a downloader to create a new result with a new piece structure introduces another difference between Wigan and BitTorrent. In BitTorrent, a downloader always knows that, when receiving a file, the number of pieces that it will eventually advertise is the same as the number of pieces that it will receive. In Wigan, as discussed, this assumption cannot be made. Therefore, a downloader is unable to start advertising the results of a query until it has received all of these query results. If we refer back to the earlier example of a university department's database, any peer downloading the results of a query on the Student table from the seed will receive 20 pieces. However, it is unable to say how many pieces the query results could fit into until it receives responses for all 20 pieces from the seed. The lifecycle of a Wigan peer therefore does not contain a phase during which the peer is simultaneously uploading and downloading the same query. This revised lifecycle is shown in Figure 3.27:

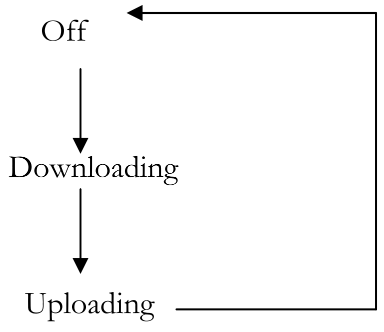


Figure 3.27 – The lifecycle of a Wigan peer

Once a peer has created a new piece structure, it contacts the Tracker and waits in preparation for a downloader to send it a handshake message.

3.4.3 UPLOADING

If an uploader receives a handshake message, it sends a confirmation of receipt back to the sender, as discussed in Section 3.4.1. The uploader is then likely to receive a piece check message from the same downloader and responds to this by confirming the number of pieces that it has, as shown in Section 3.4.1, Figure 3.19. Note that because downloaders select a random uploader for each piece, there is no guarantee that a downloader which sends a handshake and a piece check message to an uploader will request every piece of the query from that uploader. The uploader's handling of an incoming piece request (Section 3.4.1, Figure 3.20) is shown in Figure 3.28.

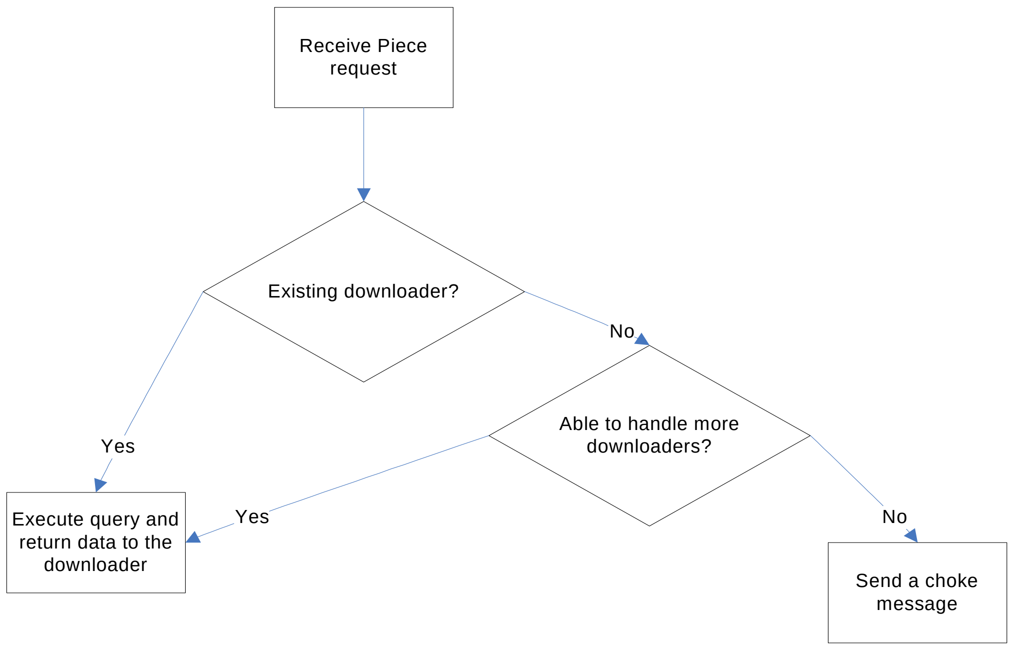

Receive Piece request
Existing downloader?
No
Able to handle more downloaders?
Yes
Execute query and return data to the downloader
Yes
No
Send a choke message

Figure 3.28 – An uploader receives a piece request

When a request arrives, the uploader will check that it can accommodate the downloader, i.e. that it will not have too many peers downloading from it if it accepts the request. An uploader can serve up to five downloaders at any one time, as in BitTorrent. If the uploader already has five downloaders, it will only respond to requests from those five downloaders. Any other downloaders will be sent a choke message, as illustrated earlier in Section 3.4.1, Figure 3.22. If, however, there is room for a new downloader or the downloader is one of the five currently connected, the uploader will extract the SQL query and the piece number from the request and obtain from its local database, all tuples on that piece whichs match the query. These tuples are then despatched to the downloader as illustrated in Section 3.4.1, Figure 3.21. If there are no tuples in that piece which match the query, the uploader still responds by returning the query, piece number and query ID. Sending a response like this instead of simply not replying informs the downloader that there were no tuples in this piece matching the query and prevents the downloader from assuming that the response has gone missing because of some technical problem.

An uploader may receive a message from a downloader expressing a lack of interest if that downloader no longer wishes to receive data from that uploader, as illustrated in Section 3.4.1, Figure 3.22. This will happen if the downloader is changing to a new query group or if the downloader has completed its download. On receipt of such a message, the uploader will delete this downloader from the list. The uploader is therefore able to accept a new request from another downloader.

3.4.4 AN EMPTY RESULT SET

A special case of downloading and uploading occurs when a query has an empty result set, i.e. no tuples match the conditions in the query. If no other peer has advertised this query, the downloading peer will send the query request to the seed. For every piece, the seed will return an empty result set as described above. Once the downloader has received a response for every piece, it will inform the Tracker that it has submitted the query but there were no rows in the result set. The Tracker will parse the query into its canonical form as usual, the only difference being the number of pieces and tuples will each be set to 0. If another downloader wanted to run the same query, the Tracker would return a result as normal, with the number of pieces and tuples shown as 0. The downloader then does not need to send out any handshakes or make contact with any other peers. It simply has to inform the Tracker that it expressed an interest in the query but there was no data. The downloader is also then likely to send a disconnection message to the Tracker as it has nothing to upload.

3.4.5 EXAMPLES OF THE WIGAN SYSTEM IN OPERATION

Sections 3.4.1, 3.4.2, 3.4.3 and 3.4.4 have described how the Wigan components, introduced in Section 3.3, interact with each other during the downloading and uploading processes. This section will illustrate this further with some worked examples. The simplest scenario is a downloader requesting a single-piece query from an uploader which already has the results of that query. This is illustrated in Figure 3.29.

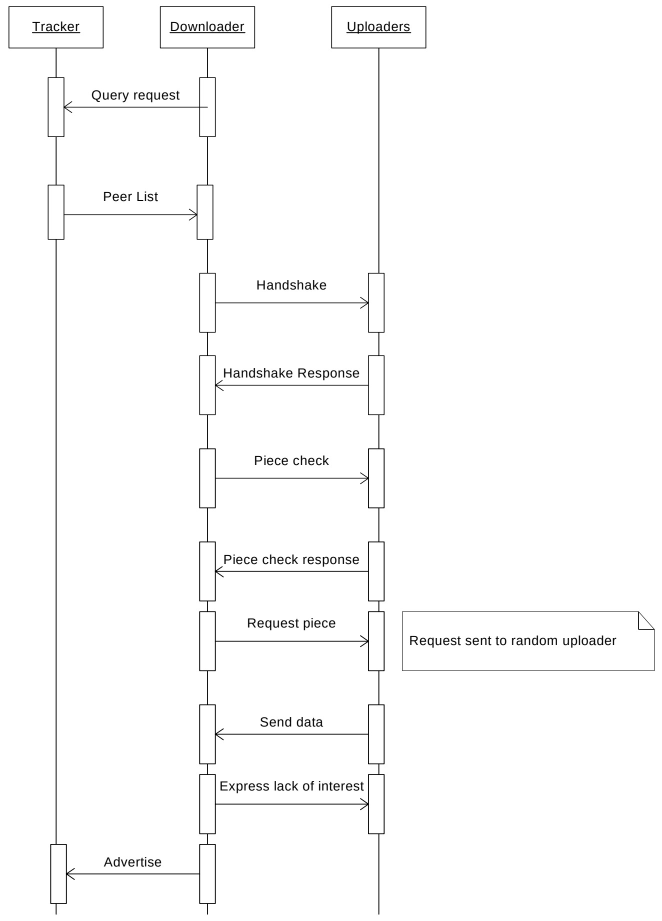


Figure 3.29 – a simple query request

In this scenario, the downloader contacts the Tracker, receives a peer list, sends the handshake and piece check messages (receiving the appropriate responses in each case) and then requests the data. On receipt of the data, the downloader expresses a lack of interest to the uploader. There is no need to change the piece structure (the uploader and downloader have the same queries), so the downloader begins to advertise at the Tracker.

In order to simplify the following diagrams, the handshake requests, handshake responses, piece checks and piece check responses have been grouped together under the title "Initial Contact."

For a multi-piece query, the piece requests are simply repeated for each piece. Another scenario, illustrated in Figure 3.30, shows the seed initially being unable to respond to a downloader's request because it is too busy with other downloaders. In this example, the timeout interval elapses and the downloader contacts the Tracker again. It receives a new peer list and obtains the data from these peers.

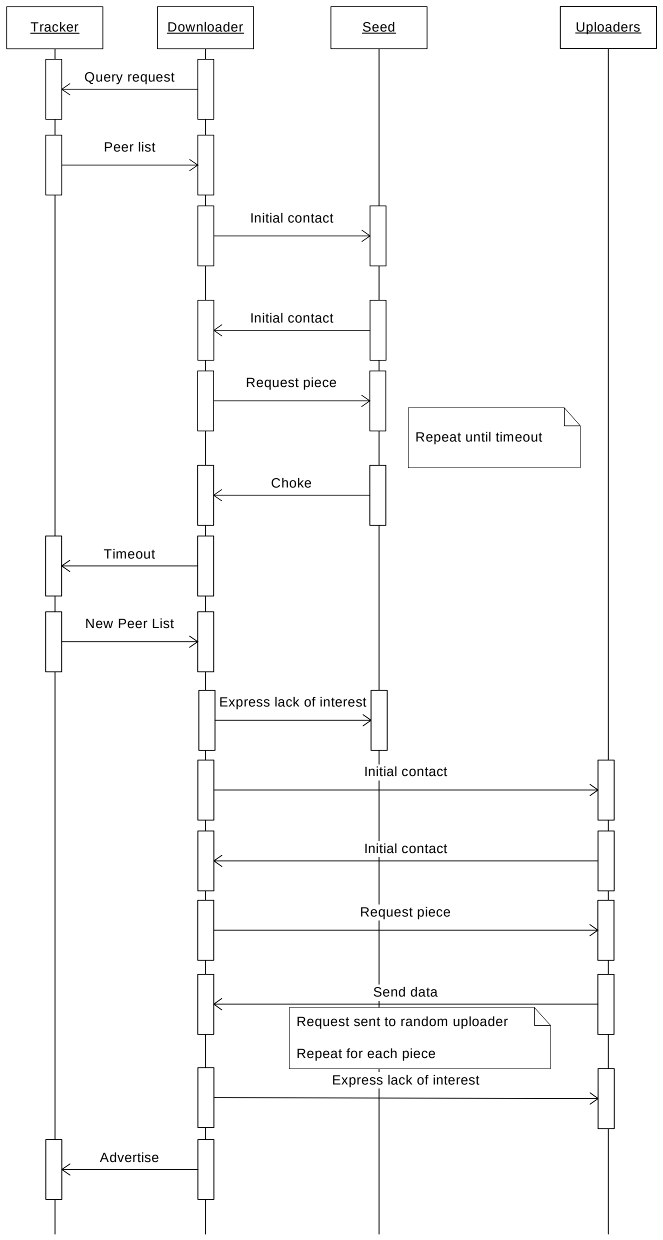


Figure 3.30 – a downloader choked by the seed and timing out

An alternative scenario to that presented in Figure 3.30 would be that the seed was able to accommodate the downloader before the timeout period had elapsed. This is presented in

Figure 3.31 and could happen if another downloader had completed its download and had sent the seed a lack of interest message.

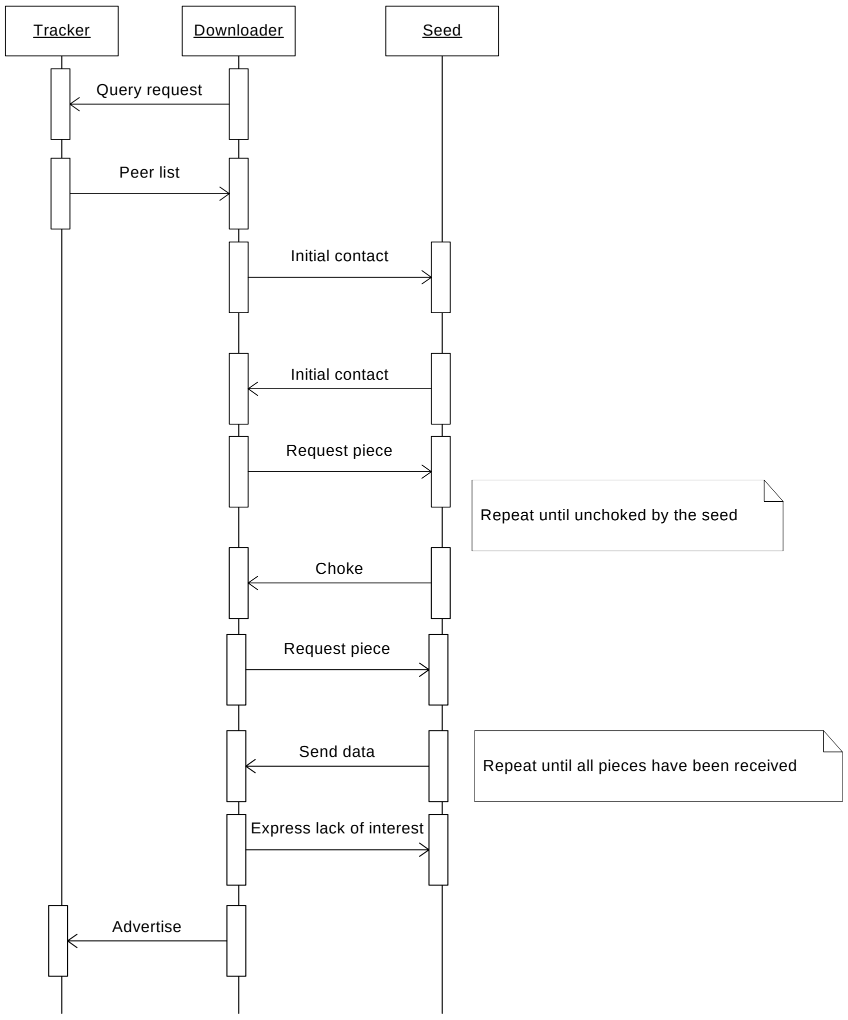


Figure 3.31 – the seed unchokes the downloader before the timeout period elapses

3.4.6 THE CHOKING ALGORITHM

Chapter 2 described BitTorrent's choking algorithm which results in peers generally connecting to others with a similar download and upload rate. The choking algorithm employed in Wigan is identical to that employed in BitTorrent and discussed in Section 2.4 of Chapter 2. It was noted in Section 2.4 that any peer which was uploading but not downloading uses the download rate of a peer to decide who to choke. This is always the case in Wigan because, as previously described, peers do not upload and download pieces of the same query simultaneously. Wigan uploaders perform a choke every 10 seconds and an optimistic unchoke every 30 seconds, as does BitTorrent. The format of a choke message was shown in Figure 3.23 and an unchoke message is shown in Figure 3.32.

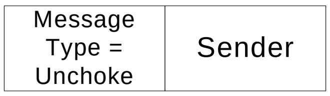


Figure 3.32 – choke and unchoke messages

The message simply contains the message type ID and the message sender. Choke and unchoke messages will have different message type IDs so that a peer is aware of whether it is being choked or unchoked by the message sender.

3.4.7 DISCONNECTING

In BitTorrent, a peer is not automatically disconnected from the Tracker on completion of its download, but instead continues to upload until a disconnection message is sent to the Tracker, as discussed in Section 2.4. The same process must be followed in Wigan. As Wigan peers do not download and upload the same query at the same time, if every peer was a free rider, there would only ever be the seed available to answer queries. Therefore, an uploader must send a message to the Tracker if it wishes to disconnect. This disconnection can happen after a random amount of time because, as with BitTorrent, there is no set amount of time that a peer must be uploading.

3.5 JOINS

The Wigan system is able to accommodate queries over multiple tables. There are various algorithms which can be used to execute such a query in the Wigan system.

In one option, a downloader requests queries on the different tables to be joined at the same time by simply sending multiple requests to the Tracker, one for each query. The downloader will receive separate peer lists for each query and will download each query independently. It is quite possible that the queries will take differing amounts of time to download, thus a peer may be advertising and uploading one query whilst still downloading the results of another. The downloading peer will perform a local join once it has received results of multiple queries against single tables. This process will be referred to as the *uncached join algorithm* as the join is purely local and the results of the joined query are not advertised at the Tracker. The algorithm is shown in Figure 3.33 where a peer is using the algorithm to obtain the result of this query from an airline's database

```
SELECT departuretime FROM routes, departures
WHERE routes.flightnumber = departures.flightnumber
AND start = ‘Newcastle’ AND destination = ‘Heathrow’
AND day = ‘Friday’ OR day = ‘Saturday’.
```

Using the uncached join algorithm, this query would be split into two queries:

1) `SELECT flightnumber FROM routes WHERE start = ‘Newcastle’ AND destination = ‘Heathrow’.`

2) `SELECT flightnumber, departuretime FROM departures WHERE day = ‘Friday’ OR day = ‘Saturday’.`

Once these are complete, the results of 1) and 2) are locally joined by the downloading peer.

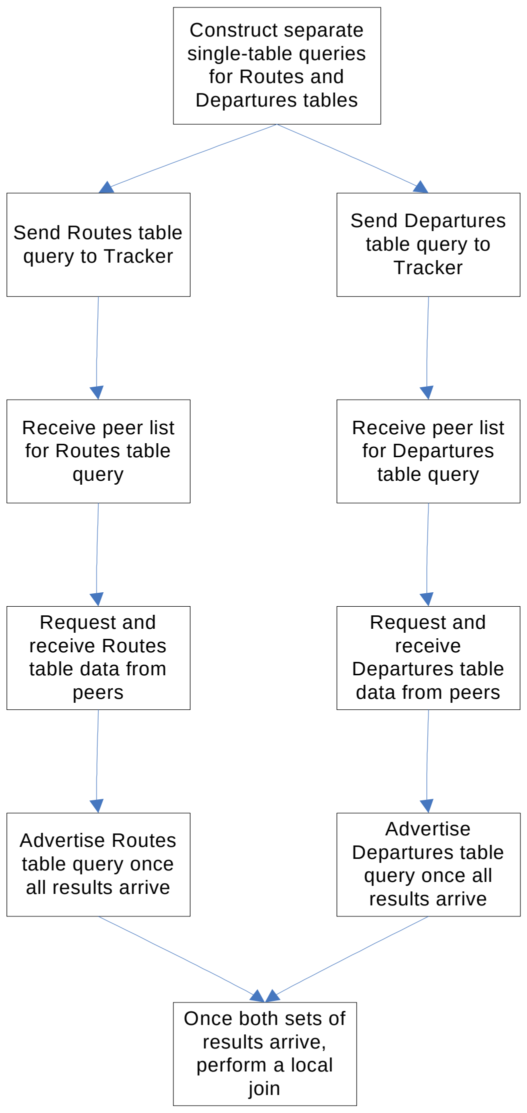

Construct separate single-table queries for Routes and Departures tables
Send Routes table query to Tracker
Send Departures table query to Tracker
Receive peer list for Routes table query
Receive peer list for Departures table query
Request and receive Routes table data from peers
Request and receive Departures table data from peers
Advertise Routes table query once all results arrive
Advertise Departures table query once all results arrive
Once both sets of results arrive, perform a local join

Figure 3.33 – The uncached join algorithm in operation

An advantage of this algorithm is that it simplifies the Tracker, whose processes were described earlier in Section 3.3.3. The Tracker does not have to look for peers with multiple tables and it does not need to check if the peers have all of the tables, only the correct columns and rows for the single table. In this algorithm, peers with single table queries are also able to assist peers wanting multi-table queries. Similarly, a peer using the uncached join algorithm that has received results for one table can be uploading these results to others whilst it waits for the results for the other table or tables to arrive.

However, there are also some disadvantages. Firstly, there are at least two sets of results to receive. This could cause a variety of problems because a peer sends at least double the number of messages to receive the query results. Section 3.3.1 noted that one problem with a single seed is that it could become a bottleneck if there are a large number of downloaders submitting query requests, particularly at the start of the download period when only the seed was available to answer queries. In the uncached join algorithm, downloaders submit multiple queries and thus if there were a large number of downloaders using the uncached join, the seed would receive even more requests. This problem would be exacerbated if there were many downloaders' queries that involved joins over large numbers of tables.

Another potential problem is that a downloader may find it has requested data it does not actually need. Consider this query, also from an airline's database:

```
SELECT departuretime FROM routes, departures

WHERE routes.flightnumber = departures.flightnumber

AND start = 'Newcastle' AND destination = 'Paris'

AND day = 'Sunday'.
```

Using the uncached join algorithm, this query would be split into two queries:

```
1) SELECT flightnumber FROM routes WHERE start = 'Newcastle'

AND destination = 'Paris'.

2) SELECT flightnumber, departuretime FROM departures

WHERE day = 'Sunday'.
```

There may not be many differing flight codes for the Newcastle to Paris route in which case the first query would not return a large number of tuples. However, there may be many departures leaving on Sunday so the second query could return a large number of tuples. If there were no flights from Newcastle to Paris on Sunday then the user would have downloaded two queries and potentially a large number of tuples only to find there is no data matching their query.

Alternatively, the system may allow the downloader to submit a single multi-table query. It has already been noted in Section 3.3.3 that the Tracker is able to accommodate such queries. This algorithm shall be referred to as the *cached join algorithm* because the results will be advertised at the Tracker. In this algorithm, the downloader receives from the Tracker a list of uploading peers which have all of the tables, columns and rows in the downloader's query along with the number of tuples and pieces. The downloader can then query these uploaders in the usual way. From a downloading point of view, there is no difference to downloading a single table query. The downloader continues to request pieces until it has received them all, creates a new piece structure if needed and then starts advertising. However, from the uploader's perspective, there could be a significant difference in receiving a request for a multi-table query. Consider a Wigan database system which contains only the seed and the Tracker. A new downloader, which shall be referred to as Peer X, connects to the system and submits this multi-table query, which shall be referred to as Q1:

```
SELECT departuretime FROM routes, departures

WHERE routes.flightnumber = departures.flightnumber

AND start = 'Newcastle' AND destination = 'Paris'
```

The seed has both of the required tables, so the Tracker's response will contain a single advert – that of the seed. The number of pieces noted by the Tracker will be the number of pieces in the largest table involved in the query because all of these pieces must be checked to ensure the tuples can join with those on the other table or tables. For each piece request, N, the seed must work out how to pick out the N^{th} piece of the results set. In this example, Peer X will send the seed the first piece request. On receipt of this request, the seed will run the query. This will involve joining all rows on the first piece of the largest table with rows on the other table. The result set is returned to Peer X. Each time a piece request arrives from Peer X and there is data on the requested piece of the largest table which joins with data on the other table, the seed must perform a join. Once Peer X has received all of the results, it can create a new piece structure. X will have received a certain number of tuples, so as described earlier, it will fit these onto the minimum possible number of pieces. Once this has been completed, X will contact the Tracker and begin advertising. Now, consider a second downloader, Peer Y which also wishes to execute the same query, Q1. The Tracker will inform Y that there are two possible sources for the query – Peer X which has the same query results and the seed. Peer Y chooses the exact match and requests the first piece from X. Peer X has only the results of the joined query and therefore does not have to perform a join. Peer X uses its tuple to piece mapping to determine which tuples are on piece 1 and returns these tuples to Peer Y. This process will continue until Y has received all of the results. Now, consider a third downloader, Peer Z which wishes to execute another multi-table query which shall be referred to as Q2:

```
SELECT departuretime FROM routes, departures

WHERE routes.flightnumber = departures.flightnumber

AND start = ‘Newcastle’ AND destination = ‘Paris’

AND day = ‘Sunday’
```

A list of all flights from Newcastle to Paris will include those flights on a Sunday (Q2 is a proper subset of Q1) so there are three possible uploaders – X, Y and the seed. Peer Z opts to use the uploader group which is nearest to an exact match, in this case X and Y. Z has to choose one of these uploaders at random and decides to send the first piece request to X. On receipt of the request, X will still not have to perform any join. It will examine the first piece and return only those tuples which contain details of flights on Sunday. Z can request pieces from both X and Y, neither of which will have to perform any join. This example is shown in Figure 3.34.

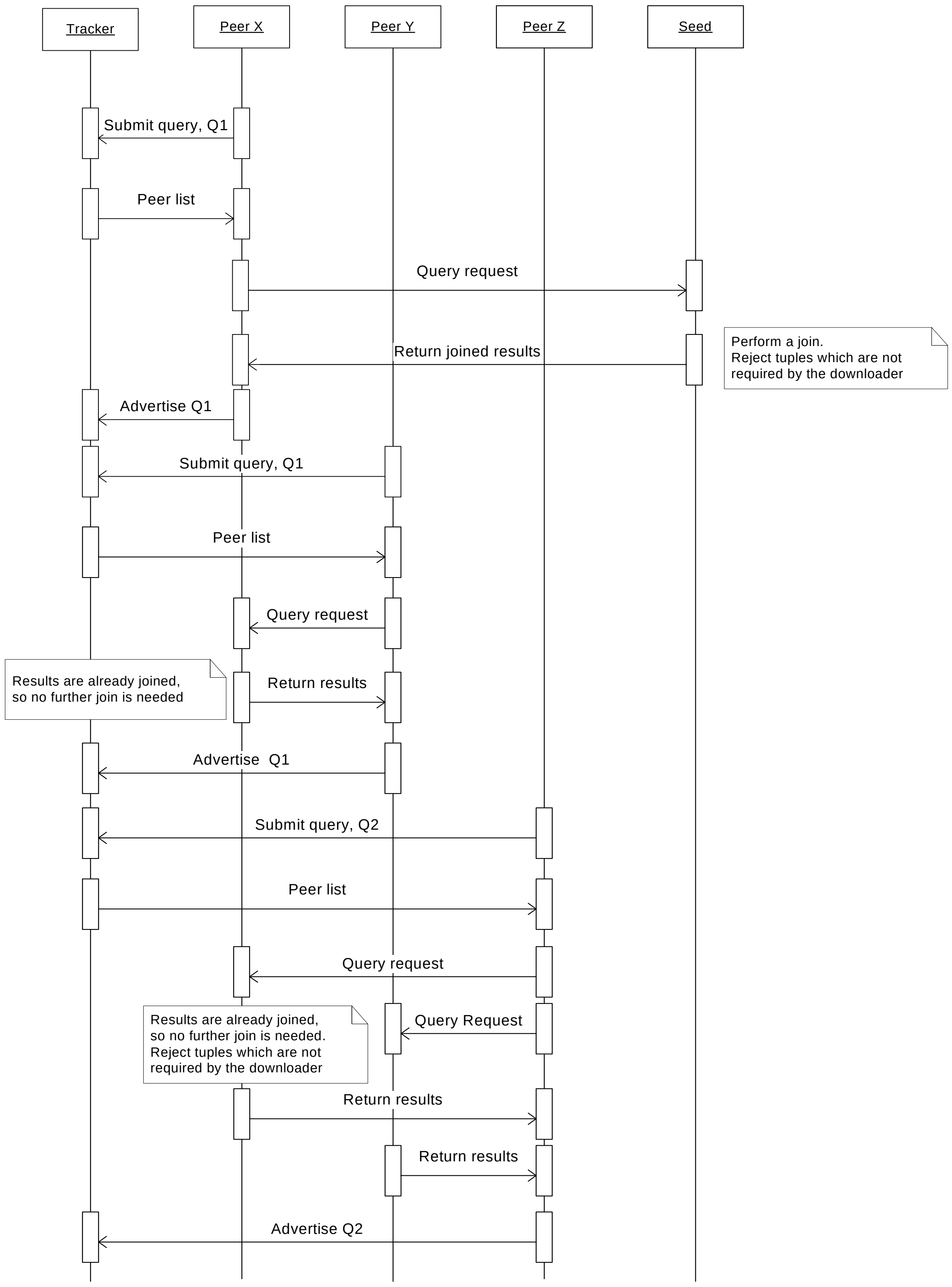


Figure 3.34 – The cached join algorithm in operation

This same process will follow for all multi-table queries in the cached join algorithm. If the uploader has a collection of separate tables, like the seed, then it will have to perform a join. If the uploader has the results of a multi-table query, it will not have to perform a join. Returning to the recent example, assume that Q1 was a very popular query which was requested many times by different peers. Once the first advert for this query was placed at the Tracker – by Peer X in this example – no other peer would ever need to perform a join during the download of the query results. The seed performed the initial join and thus all subsequent peers requesting this query have received just the results. Returning to the scenario discussed in Chapter 1 where the seed was a database server and the other peers were users' computers, this algorithm could be efficient. The more powerful server performs a join and then the users share the results amongst themselves.

The Tracker always informs downloaders of the total number of pieces, as stated in Section 3.3.3, so that the downloader is aware of how many requests to send. If a peer is downloading from an uploader that already possesses the joined results – as Peer Y was doing in Figure 3.34 – the total number of pieces will be the total number of pieces that the joined results fit on to. However, if a peer is sending a multi-table query to the seed – as Peer X was doing in Figure 3.34 – the total number of pieces in the joined results is not known because the seed is not using joined results, but instead is performing a join. For a standard inner join, the number of tuples will never be greater than the number of tuples in the largest table involved in the query. For this reason, if downloading from the seed, the Tracker shows the total number of pieces as being the number of pieces that the largest table is divided into.

However, it is possible that a user may submit a multi-table query which involves a cross join, which will produce the Cartesian product of the tables involved in the query. This will mean there are more tuples in the result set than in any of the tables involved in the query. If a user submitted such a query, the Tracker would still show the number of pieces as being the number of pieces that the largest table is divided into. However, these pieces must contain a higher number of tuples than normal.

An alternative design for the uncached join algorithm would be, if downloading from the seed, for the Tracker to always show the total number of pieces as being the number of pieces in the Cartesian product of the tables involved in the query. However, this will result in a large number of empty piece responses for all queries involving a standard inner join. Given that the cross join is not a common operation, especially when compared with the inner join, this option was rejected, to prevent downloaders having to request large numbers of empty pieces.

In conclusion, this section has outlined two potential join algorithms, each of which has their own advantages and disadvantages. This ability to have different join algorithms reinforces the discussion in Section 3.3.1 concerning the architecture of the seed. If there was not a single seed, the cached join algorithm could not be implemented in a simple manner. Both join algorithms presented in this section have been implemented and a discussion of their behaviour can be found in Chapter 4, Sections 4.6 and 4.7.

3.6 THE EFFECT OF PEER FAILURE

In a Client-Server database system, the worst case scenario is a failure at the server. This would mean that all queries would be completely unanswerable until the server's backup mechanisms were activated. In Wigan, there are a small number of cases where the same worst case scenario would apply, but there are also cases where the failure of a peer does not mean failure of the system as a whole.

Failure of the Tracker will have a major impact on the system. New downloaders would be unable to join the system by submitting a query and existing downloaders being choked by their uploaders would be unable to obtain more peers. Also, any peer that completes its download would be unable to place an advertisement meaning potentially useful adverts are withheld. However, the entire system will not fail. Existing downloaders can continue to receive data from their uploaders whilst the Tracker is offline.

Failure of the seed is one scenario which could cause the system to fail. If the seed fails before another peer has advertised a query, all queries become unanswerable and downloaders must wait until the seed's backup system is activated. If the seed fails after another peer or peers have started uploading, queries which are identical to or proper subsets of those being advertised by the other peers can still be answered.

Failure of another uploading peer should not lead to failure of the system. Any downloader which attempts to handshake the failed peer will not get a response and hence will not ask the failed peer for any data. It is possible an uploading peer fails whilst it is dispatching data. In this case, the downloader will not receive a response to a piece request. The worst case scenario is a delay until the timeout period expires, at which point the downloader is due to contact the Tracker to ask for more peers anyway. The downloader will then contact the Tracker and receive another list of peers that it can obtain data from. There will always be at least one other peer because the seed has not failed. BitTorrent permits peers to send keep-alive messages at intervals decided by the peers themselves, so if a downloading peer was particularly worried about an uploader failing, it could send more regular keep-alive messages to its uploaders and then avoid sending requests to any that do not respond. Note that an uploader failing may lead to increased query response times. If a downloader was only aware of one uploader and the seed, it would have to try and obtain data from the seed instead which may be busy. However, the data will still arrive eventually.

Peers contact the Tracker at intervals as already noted earlier in this chapter. If the Tracker does not hear from an uploading peer for some time period, it can delete that uploader from its list of uploading peers. This will prevent future downloaders being informed of an uploader that is no longer available.

Failure of a downloading peer corresponds to the failure of a client in a standard Client-Server database system. The peer that has failed will naturally be unable to receive any messages including data. However, there will be little impact on the overall system; indeed this failure may even go unnoticed by the majority of users.

In the initial implementation introduced in Chapter 4, there were no peers programmed to fail or to transmit erroneous messages, for example claiming to have 15 pieces when they had only 14 or falsely telling the Tracker they had completed their download. Chapter 5 will introduce suggestions for improving the level of fault tolerance and will examine both the issue of how to avoid total system failure and also how to deal with peers sending erroneous messages.

3.7 SUMMARY

This chapter has introduced the architecture of the Wigan Peer-to-Peer Database System, describing the components and the process of downloading and uploading. The Wigan system is modelled on BitTorrent hence the components and terminology are identical to that used in the file-sharing network. The choking algorithm in Wigan is also identical to that in BitTorrent. However, the concept of sharing data is different to sharing files hence there have been some necessary alterations to some of the BitTorrent algorithms to enable the Wigan system to operate correctly. The data is still structured in pieces, however not all of these pieces may be full of data. The order in which pieces are received is now different and a peer may not start uploading until its download has completed. Also, a peer may receive data from peers advertising a different query to the one it is requesting and the piece structure may change between the downloading and the uploading phases. The Wigan Tracker has to perform more checks to ensure the tables, rows and columns of the adverts are compatible with the query and must group adverts by query before returning a list to a downloading peer.

Chapter 4 will introduce a simulator of the system described here, together with experiments and analysis to analyse the performance of Wigan.

4 EVALUATION

4.1 INTRODUCTION

Chapter 3 introduced the architecture of the Wigan Peer-to-Peer Database System. This chapter describes a simulation of the Wigan system, followed by an analysis of its behaviour in a set of experiments.

4.2 SIMULATING THE WIGAN SYSTEM

In order to evaluate the Wigan system, a simulator was created. The aim was to allow a variety of scenarios to be explored; the simulator was used to conduct experiments which would be difficult to produce using a real P2P system, for example a database with thousands of peers, or a very heavily loaded database. Parameters, such as the number of peers or the number of queries per second submitted by the peers, can be varied easily. Collecting statistics, such as the average response time and the number of advertisements in the system is also much more straightforward. Design decisions can be evaluated using different versions of the simulator to test multiple algorithms. In addition, the simulator also allowed the testing of all the algorithms discussed in Chapter 3 to ensure they were working correctly and helped identify any potential faults. Simulators have been used successfully by other researchers examining P2P computing [18, 22, 36, 75], in some cases to simulate BitTorrent [32, 35]. Wigan was simulated using the SimJava tool [76] developed at Edinburgh University.

Wigan is based on BitTorrent, and thus it was decided to initially construct a simulation of the file-sharing algorithm. The advantage of this was that the algorithms and processes could be easily obtained from the BitTorrent protocol [6] and the BitTorrent specification [7]. The BitTorrent source code was also available for download. In addition, as described in Chapter 2, evaluations of BitTorrent have been undertaken previously by other researchers, so providing a way to calibrate the simulator. Therefore, on completion of the BitTorrent simulator, it was calibrated against the study by Izal et al [33].

Having successfully completed this stage, the simulator evolved in stages from a simulation of BitTorrent to a simulation of the Wigan P2P database system, described in Chapter 3.

The initial version of the Wigan simulator used a small MySQL [77] database for testing purposes, as noted in Chapter 3. This database initially consisted of one table which contained all of the data stored by the seed and the peers. Having only one database (rather than one per peer) made the simulator easier to configure and allowed it to scale up to thousands of peers. In addition, there were tables containing metadata for example row to piece mappings (described in Chapter 3) and the Tracker data (Section 3.3.3). Various queries were tested ranging from those returning a single tuple through to queries returning almost the entire table. Some queries were proper or partial subsets of other queries to ensure that the Tracker returned adverts which were proper subsets but not those which were partial subsets, as described in Section 3.3.3.

The simulator was then extended to accommodate multi-table queries and, to allow testing, the sample database was duly extended with the addition of some new tables. Both of the join algorithms described in Chapter 3, Section 3.5 were implemented. Initially two-table joins were implemented and tested, before the simulator was extended to handle joins involving any number of tables. Further experiments were performed, in which peers submitted a mixture of single and multi-table queries to ensure that suitable advertisements were returned to downloading peers.

The final modifications to the simulator permitted queries containing aggregations or arithmetic operations on columns. This enabled more complex queries to be evaluated.

Note that in all of these simulator variants, query execution itself was not simulated. Each peer that received a query contacted the underlying MySQL database via a Java Database Connectivity (JDBC) [78] connection and executed a real query. This approach removed the need to build a simulator for query compilation, optimisation and execution and reflected the fact that Wigan was designed to operate over existing database servers deployed on the peers and seeds.

Another important issue concerned the time taken for a message sent by one peer to arrive at its destination. The behaviour of BitTorrent was used as a guide. Each peer had a randomly chosen upload rate, to reflect the fact that in reality, a Wigan network may consist of many peers of varying upload rates. The message arrival time was calculated using this upload rate and the message size (based on BitTorrent as noted above). There was also a small constant value of network latency added onto the arrival time of all messages.

However, the arrival time of some message types was slightly more complex to calculate. For peer lists from the Tracker and data response messages from uploaders, the query execution time had to be added to the response time. This execution time was not simulated, as noted above; the actual execution time taken by the underlying database was used. Also, data response messages would be of varying sizes depending on the number of tuples contained within the message; hence the number of tuples was also included in the calculation of the message size.

The simulator can be summarised as shown in Table 4.1.

Table 4.1 – Simulator properties

Property	**Value**
Tuples/piece	50 unless stated otherwise
Endgame Mode	Not used
Maximum number of concurrent uploaders	5
Maximum number of concurrent downloaders	5 unless stated otherwise
Number of seeds at start of download period	1 unless stated otherwise
Number of downloading peers	999 unless stated otherwise
Number of free riding peers	0

4.3 WORKLOAD

It has already been noted in the previous section that the initial version of Wigan was tested using a sample MySQL database consisting of a small number of tables. However, to enable more thorough testing and evaluation, a more complex database was required with several different tables offering the choice of many different queries. It was decided that the evaluation would utilise a database workload that reflected a realistic database scenario. The TPC-H benchmark [74] was selected. It is a Transaction Processing Council benchmark.

The TPC-H benchmark did have some potential drawbacks. It included a set of update queries which could not yet be executed in the Wigan system and also many of the sample queries involved complicated subqueries which the Wigan simulator was unable to handle. Even if it was not possible to use all of the sample queries and updates provided with TPC-H, it was possible to construct several other meaningful sample queries on the dataset provided. The TPC-H database was available to download to run on a local machine but could not run on MySQL because this was not one of the database systems supported by the benchmark, therefore, the database and all metadata held by the peers was placed into a SQL Server [79] database instead. The existing Tracker continued to use a MySQL database.

The TPC-H Benchmark database is a sample relational database from a manufacturing company. It has eight tables in total – Region, Nation, Customer, Supplier, Part, Partsupp (a mapping from Part to Supplier), Lineitem (a mapping from orders to parts on the production lines) and Orders. TPC-H is based around a relational database and the relationship between the tables is shown in Figure 4.1, which is taken from the TPC-H Benchmark definition [74].

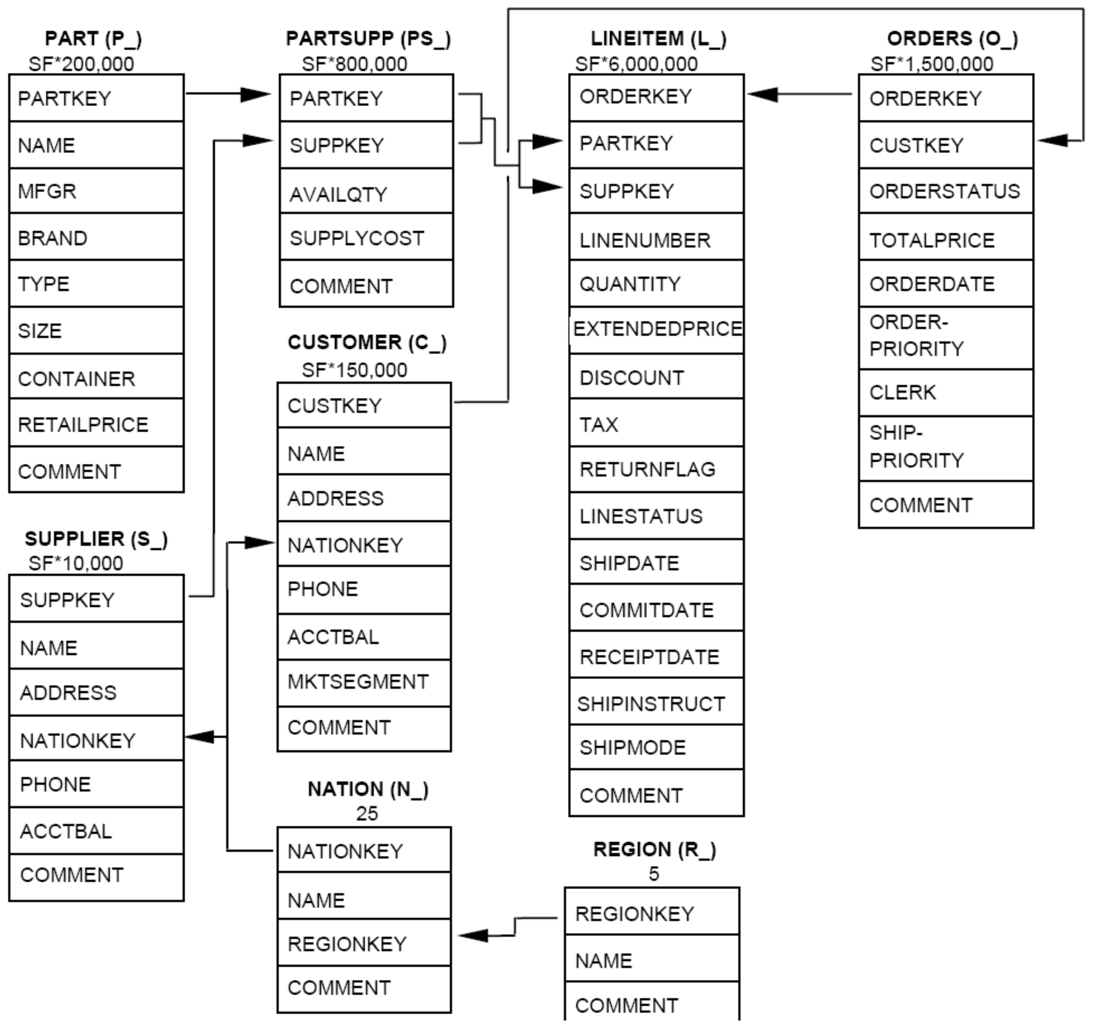


Figure 4.1 – The TPC-H Benchmark database taken from [74]

The arrows point in the direction of one-to-many relationships between the tables. The numbers above each table refer to the cardinality, in some cases multiplied by the Scale Factor. In the experiments presented in this chapter, the Scale Factor was set to at 1. This still allows evaluation of Wigan over a large database with thousands or even millions of tuples.

The table sizes used in the experiments are shown in Table 4.2.

Table 4.2 – TPC-H table sizes

Table Name	Size (tuples)
Region	5
Nation	25
Customer	150,000
Supplier	10,000
Part	200,000
Partsupp	800,000
Lineitem	6,001,215
Orders	1,500,000

4.4 INVESTIGATING THE EFFECT OF THE NUMBER OF PIECES ON RESPONSE TIME

Tables in Wigan are divided into pieces, as described in Chapter 3. This initial set of experiments will examine the effect that the number of pieces can have on the response time of a query. The aim was to investigate how the response time increased as the number of pieces increased.

During these experiments, 12 sample queries of varying sizes were executed on the Lineitem table, the largest table in the TPC-H benchmark database containing over 6,001,200 tuples. The queries were of increasing size, the first returning only one tuple and the twelfth returning the entire table. These queries are shown in Table 4.3.

Table 4.3 – Queries and their sizes

Query	Number of Pieces returned
`select count(l_orderkey) from lineitem where l_linenumber = 1 and l_partkey = 1`	1
`select l_orderkey from lineitem where l_linenumber = 1 and l_partkey < 10`	2
`select l_orderkey from lineitem where l_partkey < 10`	6
`select l_orderkey from lineitem where l_linenumber = 1 and l_partkey < 100`	15
`select l_orderkey from lineitem where l_partkey < 100`	59
`select l_orderkey from lineitem where l_partkey < 500`	297
`select l_orderkey from lineitem where l_partkey < 900`	535
`select l_orderkey from lineitem where l_partkey < 2000`	1,196
`select l_orderkey from lineitem where l_partkey < 2400`	2,400
`select l_orderkey from lineitem where l_partkey < 12000`	7,196
`select l_orderkey from lineitem where l_linenumber = 1`	30,000
`select l_orderkey from lineitem`	120,025

In these experiments, the Wigan system was configured with one downloader submitting a query, one uploader advertising the same query, no seed and the Tracker. The reason for having only one uploader advertising the downloader's query, and no seed, was to prevent there being any empty responses. Chapter 3, Section 3.4, explained how, to ensure complete results, all pieces must be requested from the seed. For those pieces which do not contain data matching the downloader's query, an empty response is returned by the seed. Such empty responses would distort the results of this experiment because pieces containing data would take only marginally longer to arrive than those that do not and the seed would still have to return the same number of piece responses for each query. In addition, using just a single uploader and downloader prevented the choking algorithm having an impact on the response time. Chapter 2, Section 2.4, introduced BitTorrent's choking algorithm and Chapter 3, Section 3.4.6, explained how this worked in Wigan. If there were multiple uploaders and downloaders, uploaders could be choking some downloaders to unchoke others, thus distorting the response time. Each experiment was repeated five times and an average taken. The increase in the number of pieces, as the query results sets increase in size, is illustrated in Figure 4.2 and the corresponding increase in response times is shown in Figure 4.3.

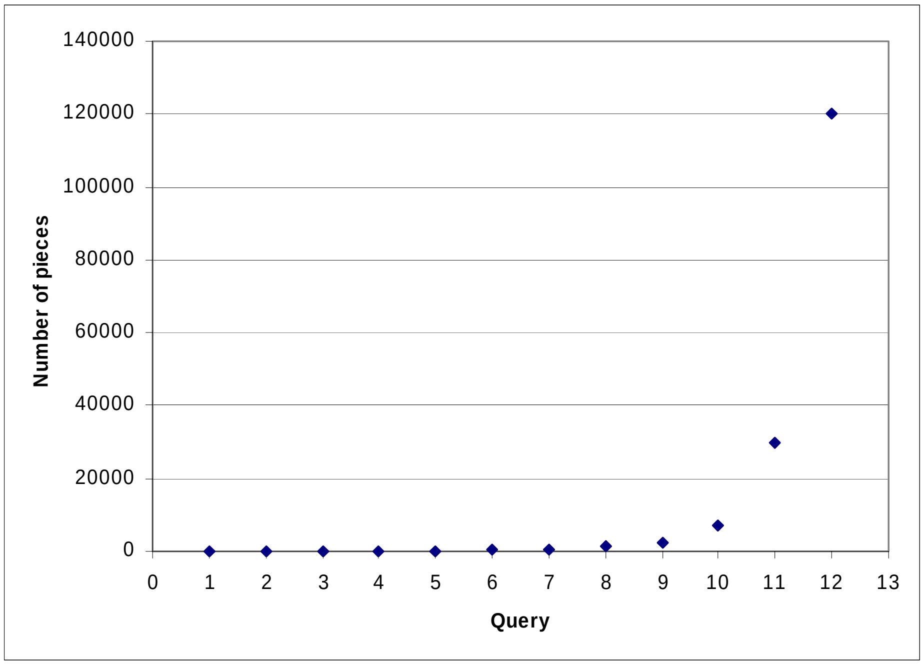


Figure 4.2 – The increase in the number of pieces returned by the queries

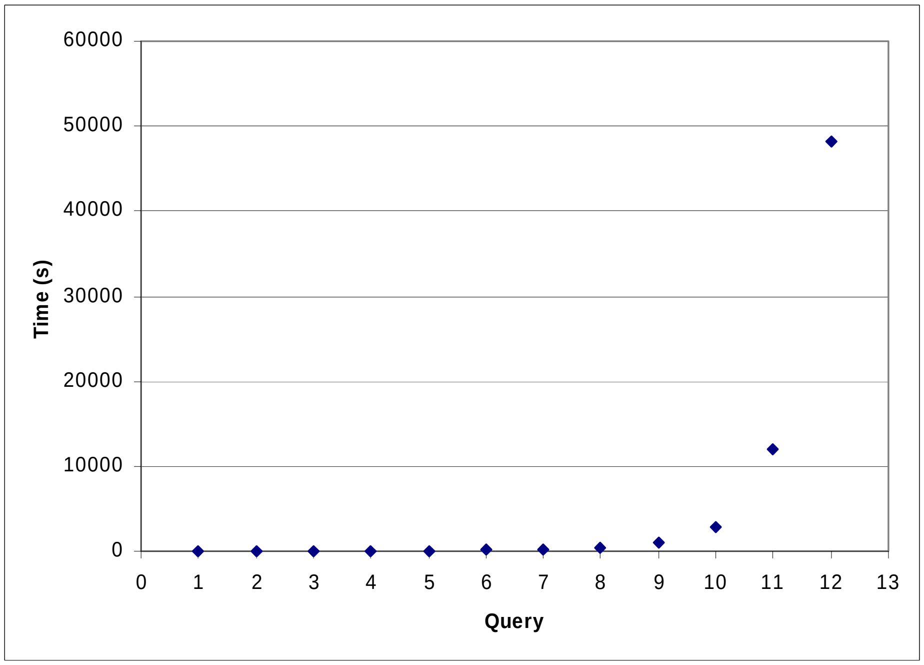


Figure 4.3 – The increase in response time for each query

The shape and the slope of the curves in Figure 4.2 and Figure 4.3 are very similar, for example when the number of pieces quadruples from query 11 to query 12, the response time also quadruples. Figure 4.2 and Figure 4.3 suggest that as the number of pieces increases, the response time increases by a similar proportion. To verify this, a scatter graph comparing the number of pieces and the query response time was produced; and the points were connected with a line. This graph is shown as Figure 4.4 and does indicate a linear relationship between the number of pieces and the response time.

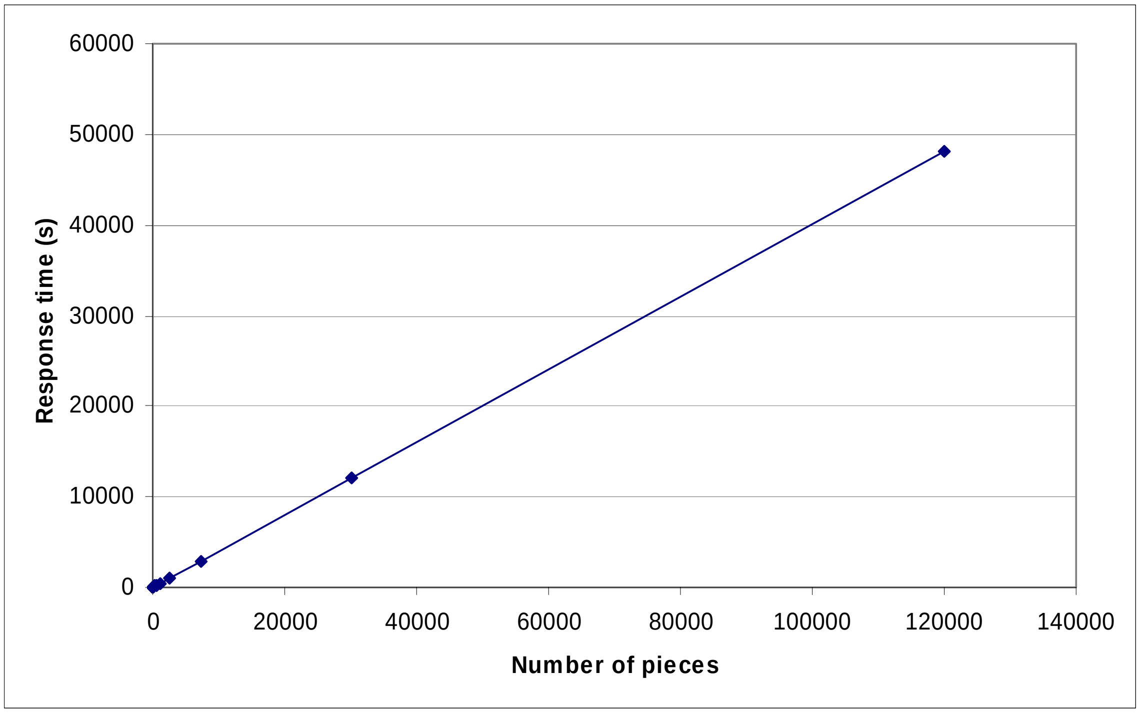


Figure 4.4 – The number of pieces plotted against response time

The graph takes this form because the experiment shows a single downloader receiving data from a single uploader advertising exactly the query that is being requested. Having a single uploader and downloader means that the uploader does not use the choking algorithm to decide which downloader to serve, and hence the downloader is not kept waiting whilst the uploader is busy serving other peers. As the uploader is advertising the same query that the downloader is requesting, the uploader is not spending time sending any empty or partially empty pieces back to the downloader. Therefore, when the number of pieces doubles, the response time also doubles because the downloader has twice as many requests to send.

Note that the times shown here are the minimum possible response times because, as noted above, choking may have an impact on the results. This could be of particular significance in the case where there are many downloaders joining at the start of the download period and the seed is unable to respond to all requests.

These experiments indicate that Wigan offers faster response times to queries which contain a small number of pieces and therefore a small number of tuples. Note that this is not dependant on the overall size of the table. In these experiments, the table used was very large, and yet queries consisting of a small number of pieces arrived in a few seconds. This is because the uploader only possesses the results of one particular query and not the entire table. Consequently, the uploader has only to scan its query result set and not a table of over six million tuples.

The smallest possible query is one whose result set contains only one tuple. The response time for such a query was approximately 2.6 seconds. This represents the initial overhead of contacting the Tracker (approx 1.4 seconds), selecting a peer and establishing contact (approx 0.8 seconds) and then requesting and receiving a single piece (approx 0.4 seconds). Note that all of these figures include the time taken to send all relevant messages, for example the 1.4 seconds to contact the Tracker includes the time take for the request and response messages to travel across the network. Also note that any query containing more than one tuple, but whose results could still fit on a single piece could also arrive in approximately 2.6 seconds. This may seem somewhat slow compared to a Client-Server system, which could return the result in a few hundred milliseconds as measured by the simulator. However, later in this chapter, a comparison will be made between a busy Wigan system and a busy Client-Server system.

4.5 SINGLE-TABLE, SMALL QUERY EXPERIMENTS

These experiments evaluate Wigan's performance with small queries over a single table and in particular, examine the effect of multiple peers submitting identical queries. These experiments used the TPC-H Supplier table, which contained 10,000 tuples. This offered the ability to produce a good range of queries which would not overlap with each other. It is important to analyse how Wigan behaves with queries that match each other and with those that do not – and hence must be sent to the seed – and a very small table would not offer this opportunity.

The 999 downloaders all requested 10 tuples from the Supplier table. To examine the effect of matching queries, a proportion of the peers requested the same 10 tuples as other peers. Each tuple in the table has a unique integer ID from 1-10,000, which enables downloaders to select specific tuples. Peers requesting the same query all requested tuples 1-10.

Those peers that did not request the first 10 tuples picked a starting tuple at random and selected that tuple and the following nine, again using the tuple ID to distinguish between tuples. The random starting tuple was uniformly distributed between two and 9,990. Starting at two meant that a peer picking a random starting tuple did not select the first 10 tuples, whilst 9,990 was the final starting point from which it would be possible to select 10 tuples. The probability that a peer would select the first 10 tuples was varied; the values tested were 100%, 75%, 50%, 25% and 0%. When the probability was 100% all peers were submitting the same query; and when the probability was 0%, all peers submitted a random query. Note that it was still possible for two random queries to match with each other, if multiple peers happened to select the same starting tuple, although the chances of this would be low. Hence in the graphs shown later in this chapter, 0% is referred to as "Mostly Different."

Another parameter of importance is the number of queries per second submitted by the downloaders as time progresses. Chapter 2, Section 2.4, explained how, in BitTorrent, there was an initial state of increasing interest, known as the flashcrowd, then a period during which there was a steady stream of downloaders, followed by a period in which interest waned until no new peers requested a copy of a file. This is shown again in Figure 4.5.

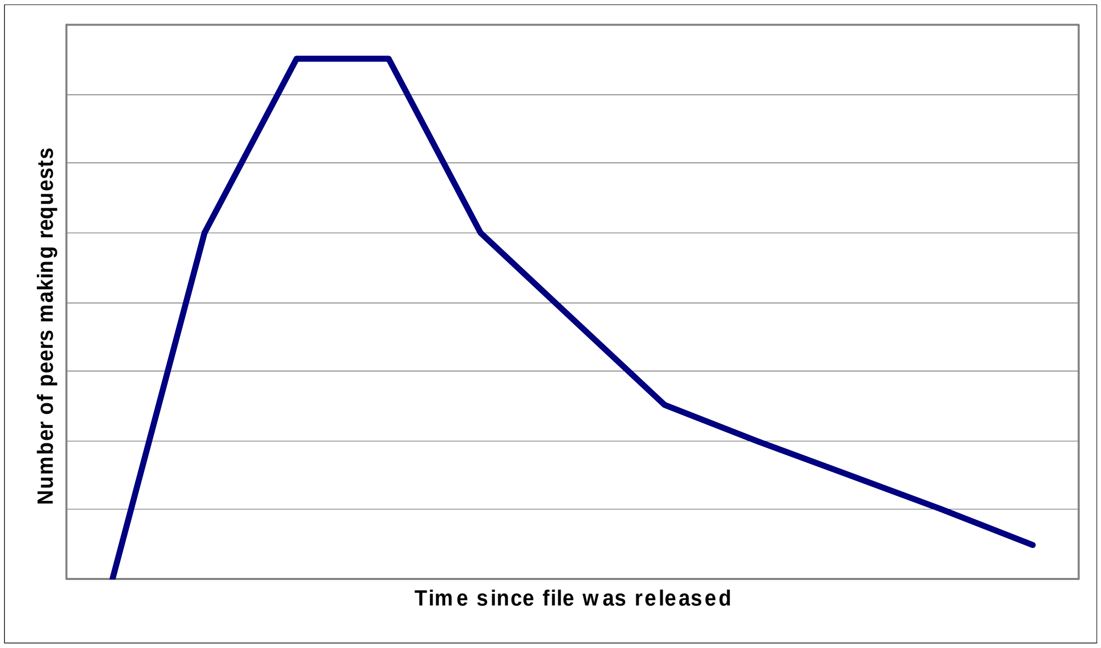


Figure 4.5 – A typical BitTorrent workload model

In Wigan, it is quite possible that there will be continued interest in some data. The final stage, of declining interest, might not occur. In a company database, as represented by the TPC-H benchmark, interest in the data is likely to be continual. In a very busy system, the workload could be like that shown in Figure 4.6.

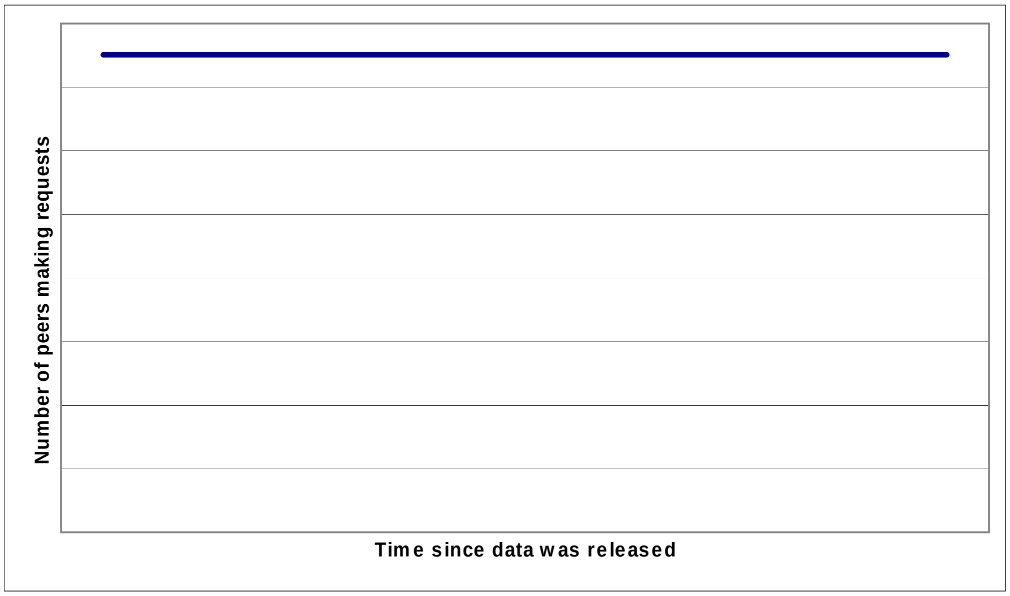


Figure 4.6 – The constant workload, a continual stream of queries

In this workload, there is a steady stream of queries during the download period. However, it is also possible that the system may not reach this steady stream of queries instantaneously. The BitTorrent-style flashcrowd is still a possibility. This workload model is shown in Figure 4.7.

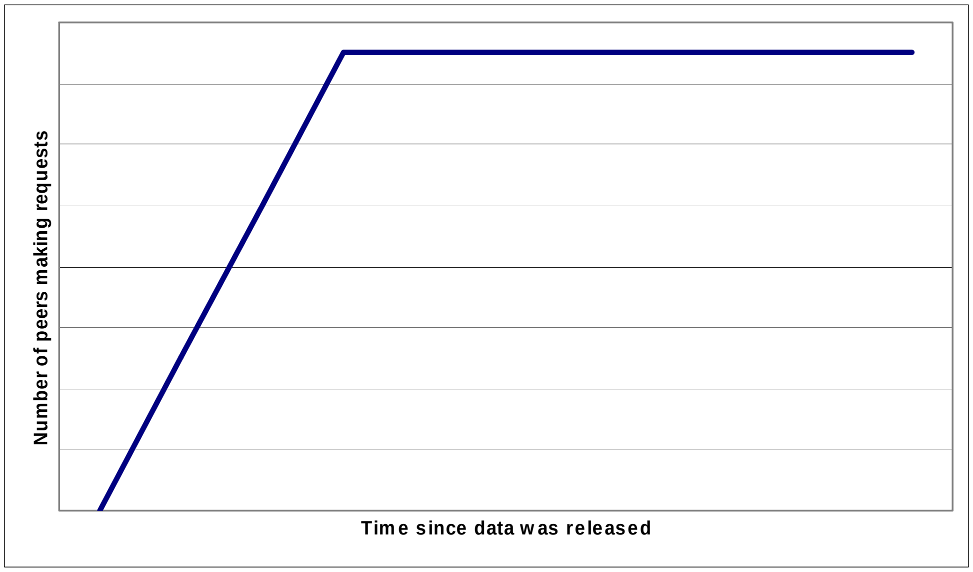


Figure 4.7 – The flashcrowd, initial increasing interest and then a continual stream of queries

In this workload, the number of queries per second grows until it eventually reaches a steady stream. This corresponds to the initial increase in interest in a file in BitTorrent. Possible examples of this scenario in a database system include a database where new data is released at the start of a working day, with interest growing as more users log in; or a database system with peers in different geographical locations, where peers in one time zone might start obtaining data before others. Both workloads illustrated in Figure 4.6 and in Figure 4.7 are possible in a real world scenario, hence in this section the same experiments will be repeated with both workloads. The former workload shall be referred to as "Constant" and the latter as the "Flashcrowd".

In the constant workload, each of the 999 ordinary peers, i.e. not the seed, submitted a single query. Queries were submitted at the rate of two per second. In the flashcrowd workload, each of the 999 ordinary peers again submitted a single query. However, for each of the first three minutes of the download period, the number of queries/minute increased by 10 compared to the previous minute. After three minutes, all queries arrived at a steady rate, again of two per second. Three minutes was selected as the starting point for the steady state because it was not so near the start time that the period of increasing interest was too short, but long enough to give a sustained steady stream of queries from the 999 peers.

For each of the frequencies of repeated queries described above, five experiments were performed, to achieve a 95% confidence interval, and an average was taken. For each run, the average response time was taken and the average of these five average response times is shown in Figure 4.8.

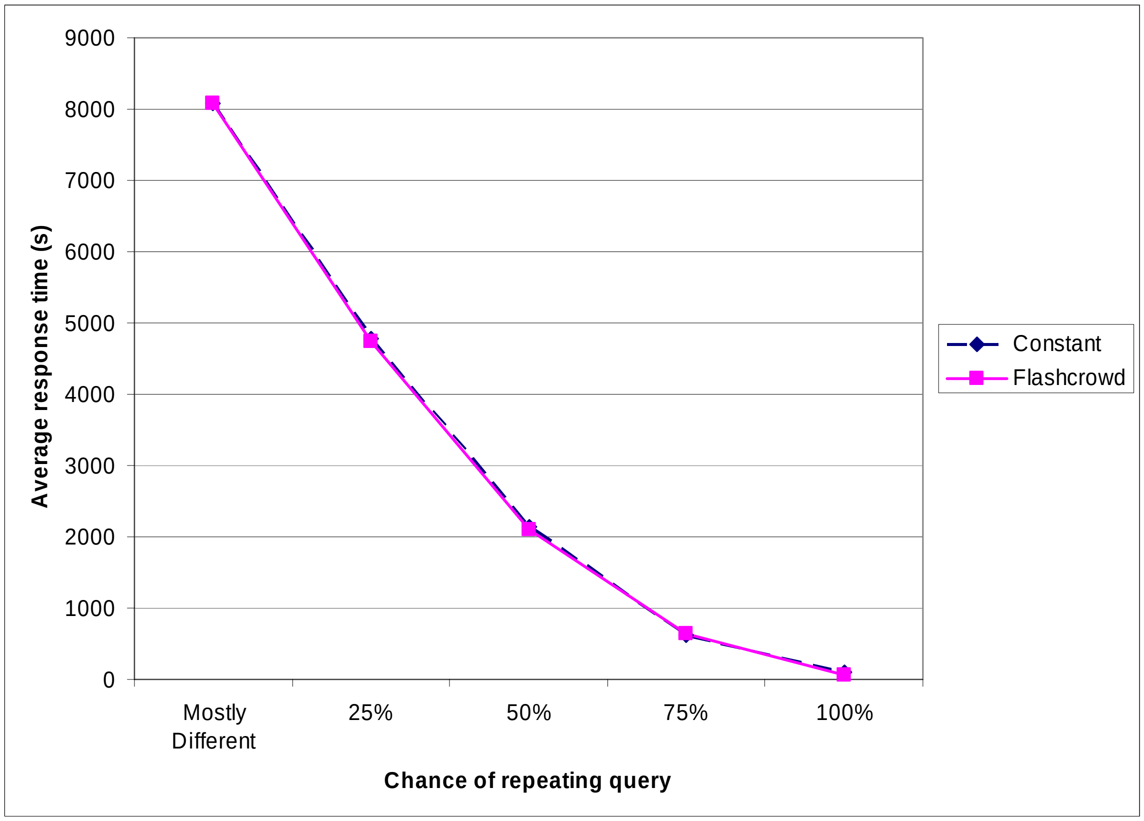


Figure 4.8 – Average response times for the two workloads

The first observation to note from Figure 4.8 is that the average response time falls as the probability of a repeating query increases. In the worst-case scenario, there is only a 1 in 9,989 chance of a query matching with another; hence almost all queries are being routed to the seed. This means the seed will have up to 999 peers attempting to submit queries during the download period. Chapter 3 noted that a single peer could serve up to five downloaders at once, as in BitTorrent. This therefore, means that the seed receives many more queries than it can handle, hence some peers have to wait for a long time before they can receive their results. With 100% chance of a repeating query, all queries are the same; however the average response time is just under two minutes for the constant workload and just over a minute for the flashcrowd. This is because of the opening part of the download period when each of the peers that have submitted a query is waiting for the seed to answer it. Once one peer starts to advertise the first 10 tuples from the Supplier table, the next request for that query will be sent to that peer. However, until that first advertisement is placed at the Tracker, all queries must go to the seed. It will take the seed some time to scan through the 200 pieces of the Supplier table, during which there will be many more peers joining the system. Each of these will have to wait for the seed to have a free slot for a new downloader, which increases the overall average response time. The seed will have a free slot for a new downloader when another downloader receives all the results and stops downloading. In addition, the seed may choke one downloader and unchoke another. This is another scenario where a downloader may start to receive results from the seed. The choked downloader will have received some but not all of its results. There may be several such downloaders, receiving a few pieces, then choked for some time period then unchoked again to receive more pieces. This will be discussed in more detail later in this section, in Figure 4.13.

Examining the figures in more detail, when all the queries were the same, on average over 83% of peers in the constant workload and over 87.5% of peers in the flashcrowd workload received their data in less than 10 seconds.

For the majority of the time, the flashcrowd outperforms the constant workload. This is again due to the beginning of the download period, when all peers are querying the seed. In the constant workload, the system suddenly starts receiving a fast stream of queries which results in the seed receiving more queries that it can process. However, in the flashcrowd, this continual stream takes some time to build up and, during this time, more peers are able to complete their download before the seed becomes overloaded. During this time – the upward sloping line in Figure 4.7 – there is less competition for a slot at the seed and hence more peers complete.

During these experiments, various statistics were collected, including the number of downloaders which timed out at least once and had to re-contact the Tracker after failing to receive their results. A timeout would occur if the timeout interval had elapsed without the peer receiving all of the query results. It was noted in Chapter 3, Section 3.4.1 that this interval was set at 700 seconds for these experiments. The average percentage of downloaders timing out over the five runs is shown in Figure 4.9.

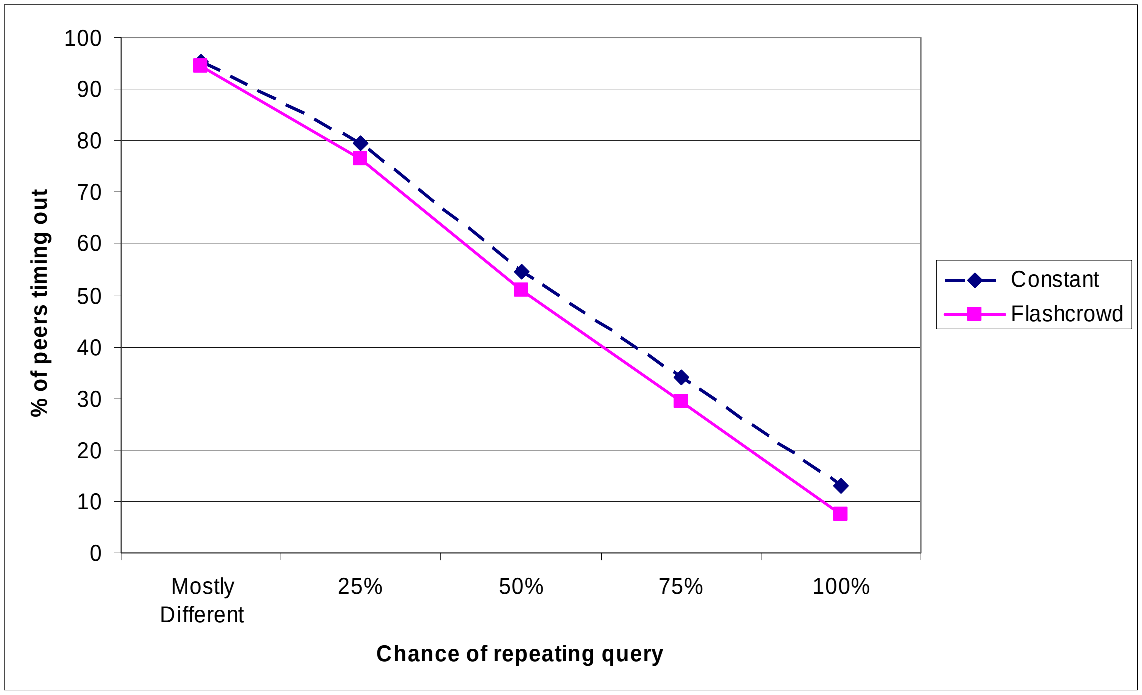


Figure 4.9 – The percentage of peers that timed out and had to re-contact the Tracker

Here, it can be observed that, if the queries are mostly different, over 90% of peers will time out because the seed is very busy. However, if all the queries are the same, the number of peers timing out is approximately 13% in the constant workload and 7% in the flashcrowd.

Also, regardless of the chance of a repeating query, the flashcrowd has fewer peers timing out than the constant workload. Again, this is because of the initial part of the download period; fewer peers competing for a free slot at the seed during this time allows more peers to complete and therefore fewer will time out.

To examine the effect of a peer's start time on the overall response time, a new set of statistics was collected. This used the constant workload at the same rate as in the previous experiments, with a 75% chance of a repeating query. This allowed the opportunity to examine downloaders whose queries could be answered by many uploaders and also those that have queries answerable only by the seed. It also allowed the opportunity to see what impact, if any, the different types of queries had on each other. The average response time was recorded with the downloaders' start times. Due to the high number of peers, the start times have been grouped in five second intervals. Thus, a figure of 60 on the x-axis includes the average of all response times for all peers which join the download after and including 60 seconds following the release of the data, but before 65 seconds following the release of the data. Again, five experiments were performed and the averages given are the average response times over these five runs. Two sets of statistics were collected from the same simulation run – one for the downloaders that had the repeating query and one for those which had random queries. The latter set is shown in Figure 4.10.

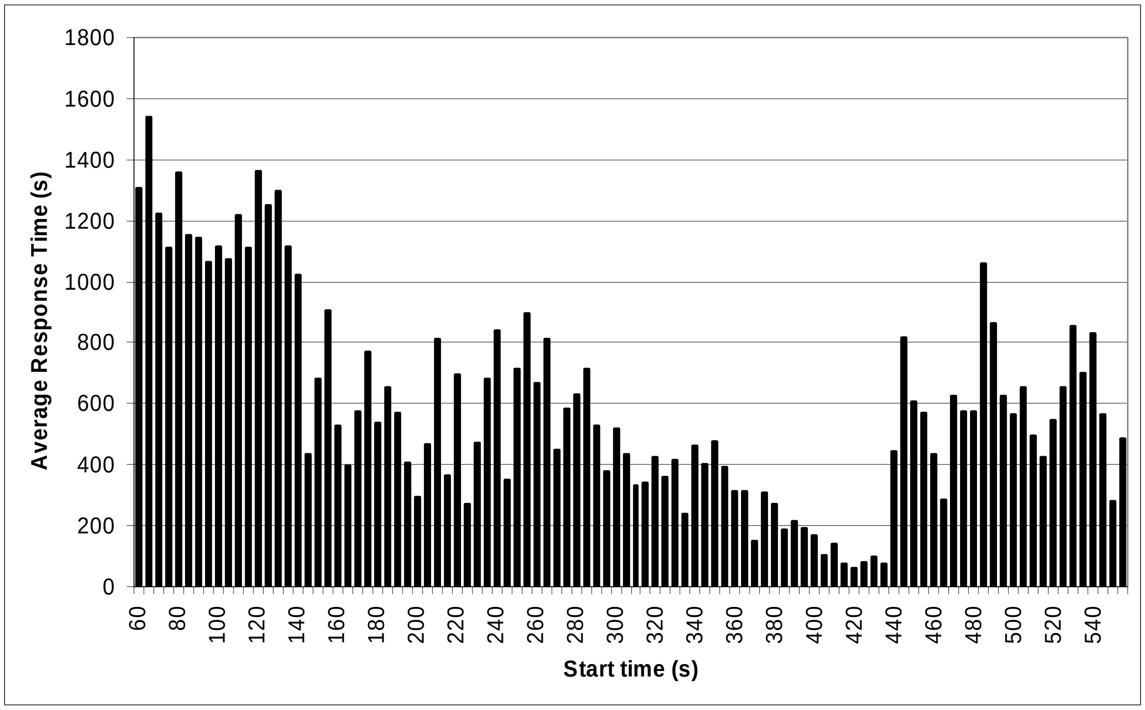


Figure 4.10 – Average response time compared to start time (random queries)

It can be seen that the random nature of these queries means that the response times can fluctuate considerably. With such random queries, there is no way of predicting if there will be another possible uploader apart from the seed. The slowest response times were obtained by peers joining the download just after the start, as there would be two queries per second arriving at the seed with no other alternative uploaders. It is also interesting to note that peers joining after 440 seconds experience a higher response time on average to those joining slightly earlier. This is the point at which the seed must be approaching its busiest period and further investigations to explain this follow later in this section.

When observing the same statistics for those downloaders from the same simulation run, with the repeating query, shown in Figure 4.11, the effect of the busy seed at the start of the download period becomes evident.

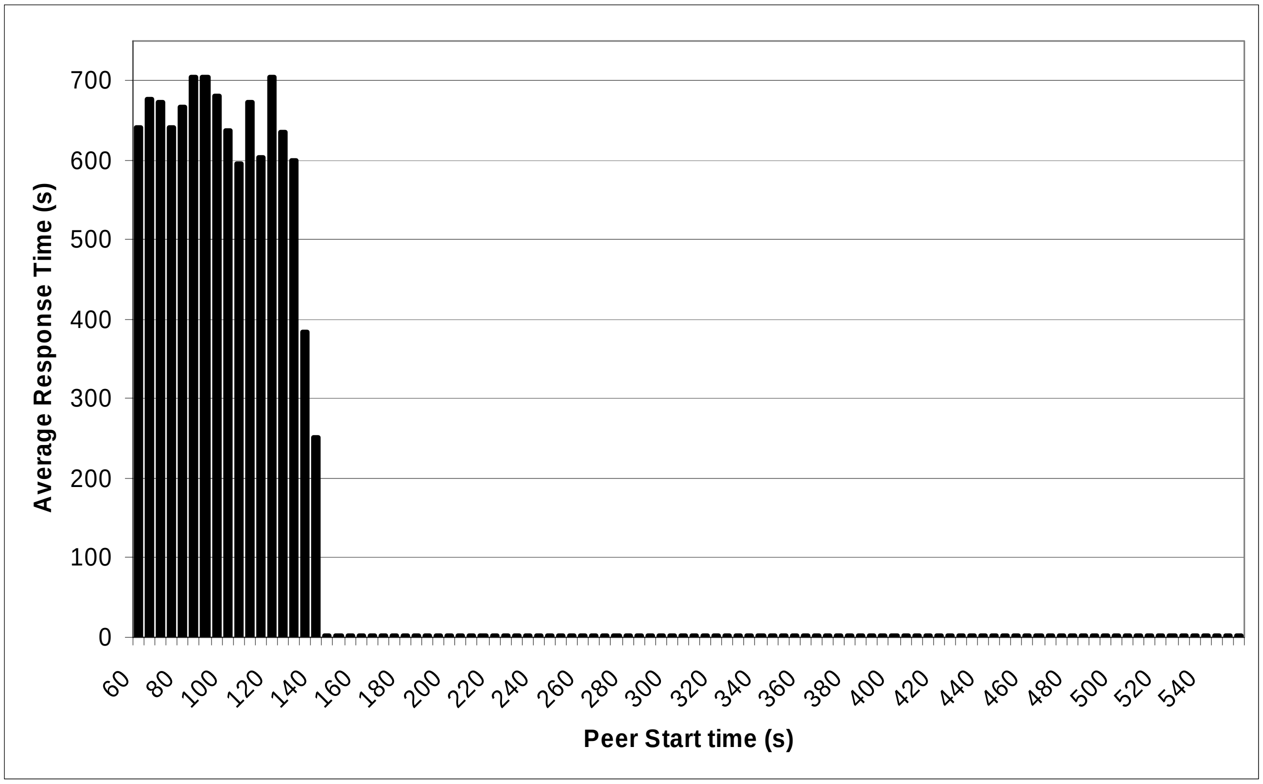


Figure 4.11 – Average response time compared to start time (repeating query)

Peers joining during the start of the download period experience a high response time of several minutes. Again, the randomness of the system does result in some fluctuations. However, once one peer begins to advertise the query, response times plummet to the figure of approximately 2.6 seconds described earlier in this chapter. Figure 4.12 zooms in on the download period following the reduction in response times.

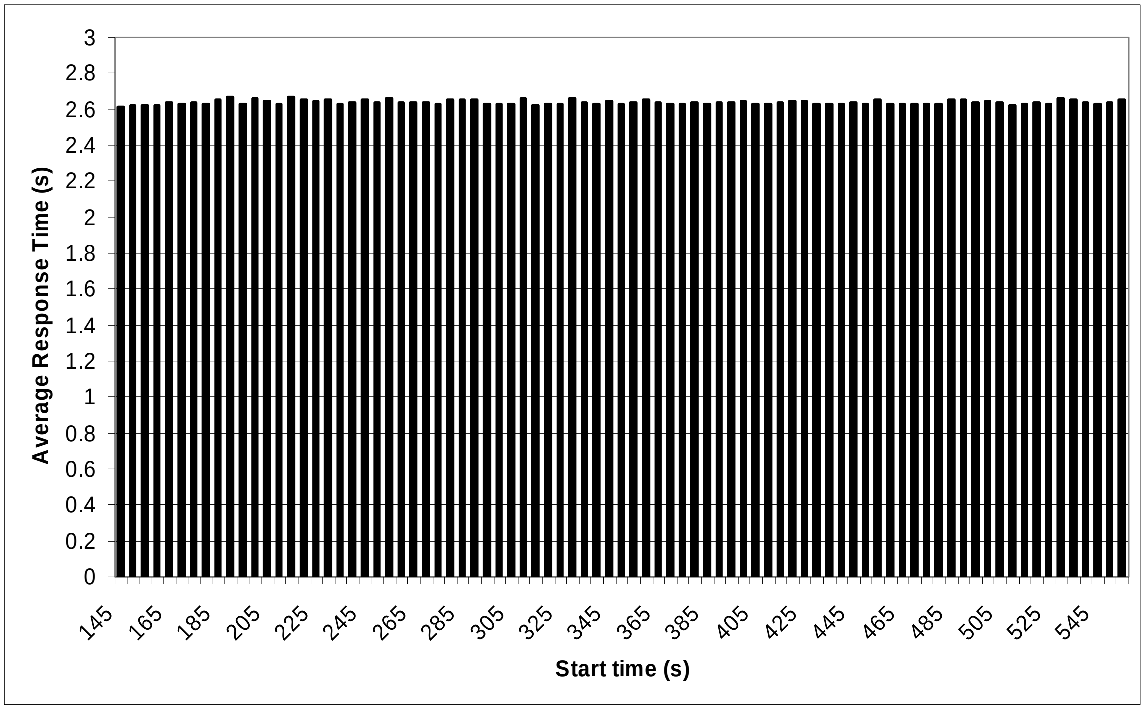


Figure 4.12 – Average response time compared to start time (repeating query) later in the download period

There are a few very minor fluctuations (by about a millisecond) which are likely to be caused by downloaders who receive the piece from an uploader with a slightly slower upload rate and also by downloaders that are choked by the first uploader they try and then obtain the results from another. Apart from that, the times remain consistent.

All of these experiments have indicated that the number of queries per second received by the seed is of importance to the average response time. To investigate the exact behaviour of the seed during a simulation run, a new set of simulation runs were carried out and different statistics were obtained. This experiment was performed using the constant workload – the worst case scenario for the seed, where it suddenly starts receiving a continually high volume of queries – and the probability of a repeating query again was 75%. Every second, the seed noted how many queries it received and how it responded to these requests – with a choke message (Figure 3.23) or with a piece response message (Figure 3.21). Note that there was no distinction made between piece responses containing data and piece responses containing an empty results set. Again, the experiment was repeated five times and an average taken. The resulting statistics are shown in Figure 4.13:

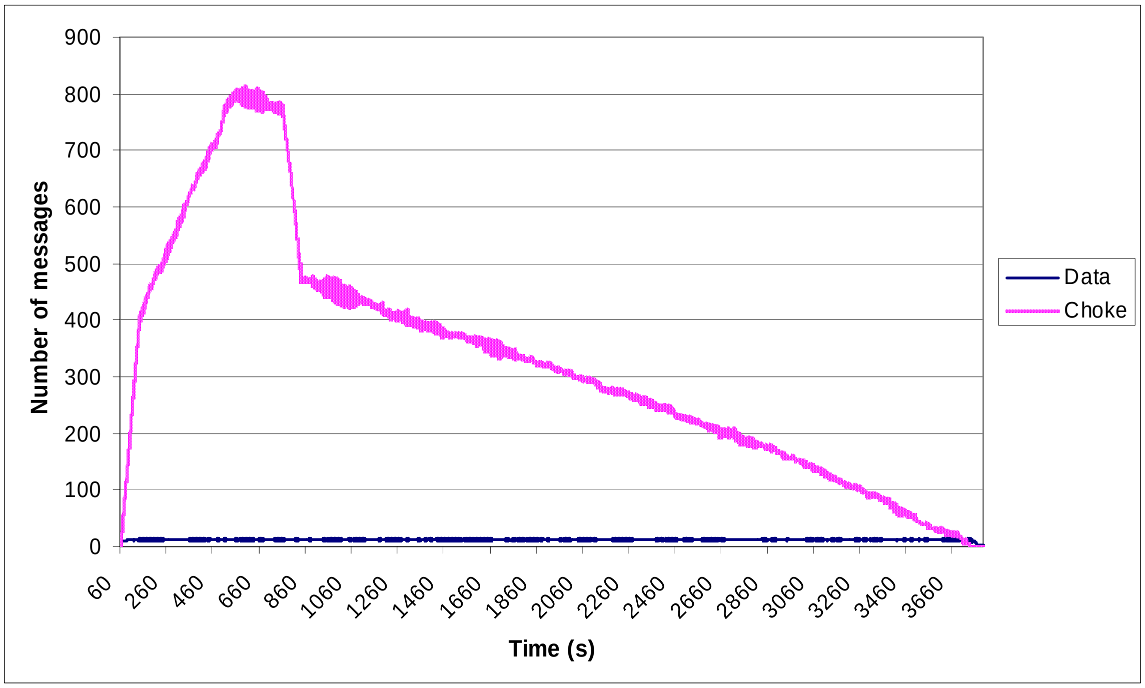


Figure 4.13 – The number of messages received by the seed during the download period

It can be observed that the seed is continually despatching data to downloading peers. However, it is also turning away many more requests, sometimes over 800 requests in a single second. Initially, these requests will come from all downloaders in the system because they have no alternative source of data. Those choked by the seed will continue to submit requests as they have no other choice, hence the large volume of requests received by the seed. During the download period, the number of requests initially grows rapidly because all peers currently downloading would be querying the seed. The growth eases off slightly once peers start uploading the first 10 tuples because all new downloaders requesting that query will not be querying the seed from this point. Once the timeout period has elapsed, all downloaders that have not received their results will re-contact the Tracker. Those downloaders requesting the first 10 tuples will no longer have to submit any more queries to the seed because there will now be other uploaders advertising that query. This is why there is a sudden drop in the number of requests received by the seed as downloaders begin to time out. Once all downloaders requesting the first 10 tuples have timed out and found new uploaders, the only downloaders left querying the seed are those that have submitted random queries. The number of requests gradually falls as the seed keeps methodically despatching data until all downloaders receive their results. Eventually, the seed stops receiving more requests than it can handle, before it eventually stops receiving requests at all. This is the point when all of the downloaders have received their results.

These experiments highlight the potential issue of a busy seed receiving many more requests than it can service. Whilst this might not cause technical problems at the seed (it can simply send choke messages to any downloader not on its list) it does impact on the query response times at the downloading peers. Repeating queries help lighten the load at the seed, as can be seen from the drop in the number of choke messages in Figure 4.13 when downloaders begin to time out. Note that queries would not necessarily have to repeat exactly to benefit, there is also the proper subset case discussed in Chapter 3, Section 3.3.3. Such examples will also help lighten the load at the seed. One possible option would be for the seed to perform more concurrent uploads, which shall be discussed later in this chapter, in Section 4.8.

4.6 MULTI-TABLE, SMALL QUERY EXPERIMENTS

Chapter 3, Section 3.5, introduced two possible join algorithms which could be used in the Wigan system. This section will investigate both of these, in a similar manner to the single-table experiments discussed in the previous section.

This set of experiments was a repeat of the previous ones, except that the queries now involved a join. Each peer again selected 10 tuples from the Supplier table, some choosing the same 10 tuples. These 10 tuples were again the first 10 tuples in the table. The probabilities that each peer would select the first 10 tuples were the same as in Section 4.5. However, in these experiments, the query also included the name of the supplier's country of origin. This was a single tuple which was obtained from the Nation table, a small table containing 25 tuples in total. Again, there was a probability that each peer would select the first country in the table. The probability of a repeating query was kept the same for both tables, thus at 100% all peers selected the first 10 suppliers and the first country. If the probability was 0, then the peers would select 10 random suppliers and one of the remaining countries. Note that with the Nation table, the odds of a repeating query were only 1 in 24, so this also afforded the opportunity to examine how Wigan behaved with very small tables. In the uncached join algorithm, each downloader submitted two queries, one on each of the two tables and joined them locally. Thus, the repeating queries would be:

1)
```
SELECT <<Supplier contact information>>
FROM supplier
WHERE s_suppkey >= 1
AND s_suppkey < 11
AND s_nationkey = 1
```

2)
```
SELECT n_nationkey, n_name
FROM nation
WHERE n_nationkey = 1
```

In the cached join, each peer submitted a single query. The repeating query would therefore be:

```
SELECT <<Supplier contact information>>, n_name
FROM supplier, nation
WHERE s_suppkey >= 1
AND s_suppkey < 11
AND s_nationkey = 1
AND s_nationkey = n_nationkey
```

In the following figures, the statistics for the uncached join have been split to show the separate results for the Supplier and Nation tables. The average response times are shown in Figure 4.14.

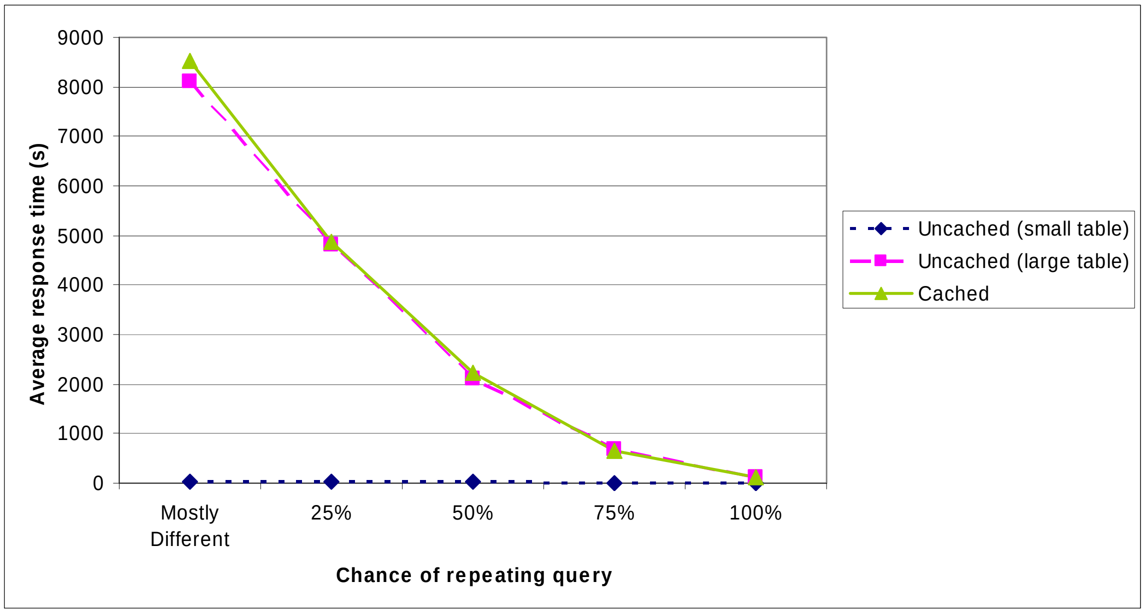


Figure 4.14 – Average response times for the algorithms

The first observation is that information from the small table (Nation) arrives very quickly, because it is a fraction of the size of the large table. The entire Nation table can fit on one piece, so even if a downloader has to contact the seed, there is only one piece to receive. The only delays will be if the seed is busy and the downloader has to wait.

The second important point to note is that there is very little difference in performance between the response times for the cached join algorithm and for the large table in the uncached join algorithm. This is because the query on the large table in the uncached join algorithm contains the same number of pieces as the single query in the cached join. The only difference is that in the cached join, data from the Nation table arrives already joined to the data from the Supplier table. In the uncached join algorithm, data from the Nation table is likely to arrive at a similar time or slightly earlier than the data from the Supplier table. If the downloader was obtaining the two query results from separate uploaders, the results are likely to arrive at the same time, as each consists of a single piece. If the downloader is obtaining both sets of results from the seed, the small table results are likely to arrive earlier because the seed has fewer pieces to scan.

4.7 CHOOSING A JOIN ALGORITHM

Section 4.6 indicated that the choice of join algorithm is largely irrelevant for small multi-table queries. If a downloader can find an uploader advertising the same query, they will be waiting for two pieces to arrive concurrently in the uncached join algorithm or for one single piece in the cached join algorithm. Similarly, if there is a bottleneck at the seed the downloader is still waiting for its results, the only difference is that in the uncached join algorithm, the results from the Nation table will probably have arrived before those from the Supplier table. The cached join algorithm might be slightly faster at times because there are fewer requests being sent to the seed at the start of the download period and also because if there is another peer advertising the downloader's query, the downloader will have to wait for one set of results to arrive, not one set from each table. However, the difference in performance is very marginal.

There are some occasions, however, where the difference in performance between the algorithms is much greater. Two cases will be presented here to highlight where each of the two algorithms performs well and badly.

If the majority of downloading peers submit single table queries, the uncached join is the better option for any peer which wishes to submit a multi-table query. This is because there will be a large number of uploaders available that can provide the single-table queries, but fewer – or even no uploaders – except for the seed that can provide the multi-table query. In this experiment, 998 of the 999 downloaders randomly picked one of two queries with equal probability. One of these queries was on the Supplier table and the other on the Nation table. The final downloader submitted a multi-table query which could be answered by combining the results of both of the single-table queries. In one experiment, the final downloader used the uncached join and in the other, it used the cached join. Each experiment was repeated five times and the response time for the final downloader, averaged across the five runs, is shown in Figure 4.15. The workload model used was the constant workload with a rate of two queries per second as used earlier in this chapter.

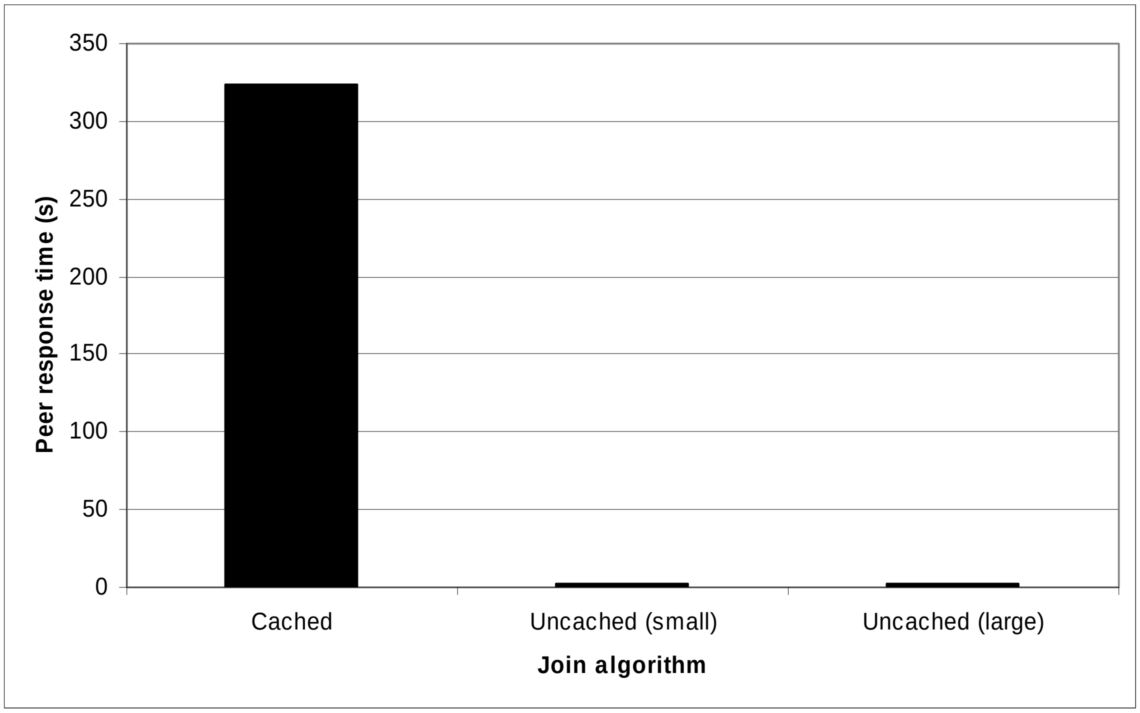


Figure 4.15 – Most peers submit a single-table query

When a downloader uses the uncached join algorithm, it is informed by the Tracker that there are several peers advertising each of the two sets of query results that it requires. The results from each table can fit onto a single piece and hence arrive very quickly. This is why there is little performance difference between the two tables. However, when the downloader uses the cached join algorithm, it is forced to obtain all data from the seed because no other peer is advertising a multi-table query. The downloader will have to wait for a free slot at the seed; which will be busy responding to queries from other peers as well. Once the downloader begins to receive data from the seed, as noted in Chapter 3, the seed will have to send one response for each piece of the table.

However, if there is a considerable difference in size between the tables involved in a join and the query contains only conditions on the smaller table, the cached join becomes a more attractive proposition than the uncached join. This is because the larger of the two tables is returned in its entirety to the downloader. In this set of experiments, all peers submitted the same query, to find suppliers who came from a particular country. The name of a country is in the Nation table, not the Supplier table. In the cached join algorithm, all peers would submit one query. However, in the uncached join algorithm, the two queries would take the form:

```
1) SELECT <<relevant country information>>

FROM nation WHERE name = <<country to search for>>

2) SELECT <<relevant supplier information>>

FROM supplier
```

The information from the Nation table would still be a single tuple. However, as noted above, the only condition is on the Nation table which means that each downloader must obtain the relevant columns for every tuple in the Supplier table. The workload model remained the same, with a constant workload of two queries per second. Each experiment was repeated five times and an average was taken. The average response times for this experiment are shown in Figure 4.16.

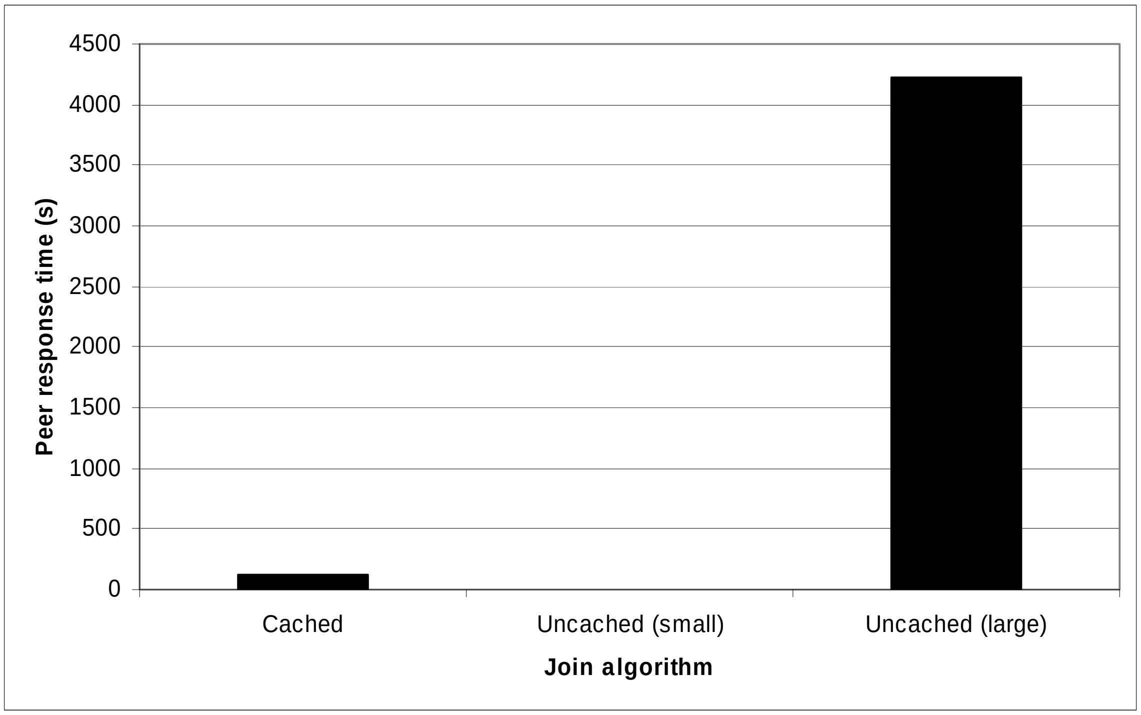


Figure 4.16 – The query condition on the small table

The response time for the small table is very fast, because there is only one tuple. The first downloader will receive this very quickly and all subsequent requests will therefore avoid the seed. However, when using the uncached join algorithm, downloading peers will be kept waiting for a very long time to receive the Supplier table. The continual stream of queries means that the seed is initially very busy. Each downloader will occupy one of the seed's slots for uploaders for slightly longer than normal because each piece the seed dispatches will contain data and will therefore take slightly longer to arrive than empty responses. Even though all peers submit the same query, this is of little advantage here because all downloaders must request all 200 pieces of the table from the uploaders. In the cached join algorithm, there is still the initial query stream at the seed, though as some responses from the seed will contain no tuples, there should be a slight speed-up compared to the uncached join. Once a peer receives the results and starts advertising, no new downloaders will be routed to the seed. A downloader connecting to an uploader advertising the query does not have to receive 200 pieces. The results of the query will fit onto eight pieces and this contributes to a significant speed-up.

Chapter 3 explained that one disadvantage of the uncached join algorithm is that there may be some tuples which are requested but are not actually needed. This is what is happening in this scenario. The downloading peers receive all 10,000 tuples from the Supplier table, but over 9,600 of these are discarded when the peers perform a local join. This also means that there are more I/O's in the uncached join algorithm. The average number of I/O's for all peers excluding the seed in this set of experiments is shown in Figure 4.17. These statistics are an average from the same five simulation runs used to generate the results in Figure 4.16. The total for the uncached join includes both queries and the results for each algorithm include read and write I/O's. Downloaders calculated write I/O's every time they received tuples from an uploader. Uploaders calculated read I/O's every time they scanned data, regardless of whether that data was returned to a downloader or not. Due to the conditions on the query, some data may be scanned by an uploader but not selected and returned to the downloader. For simplicity, no indexing or caching at the uploaders was assumed.

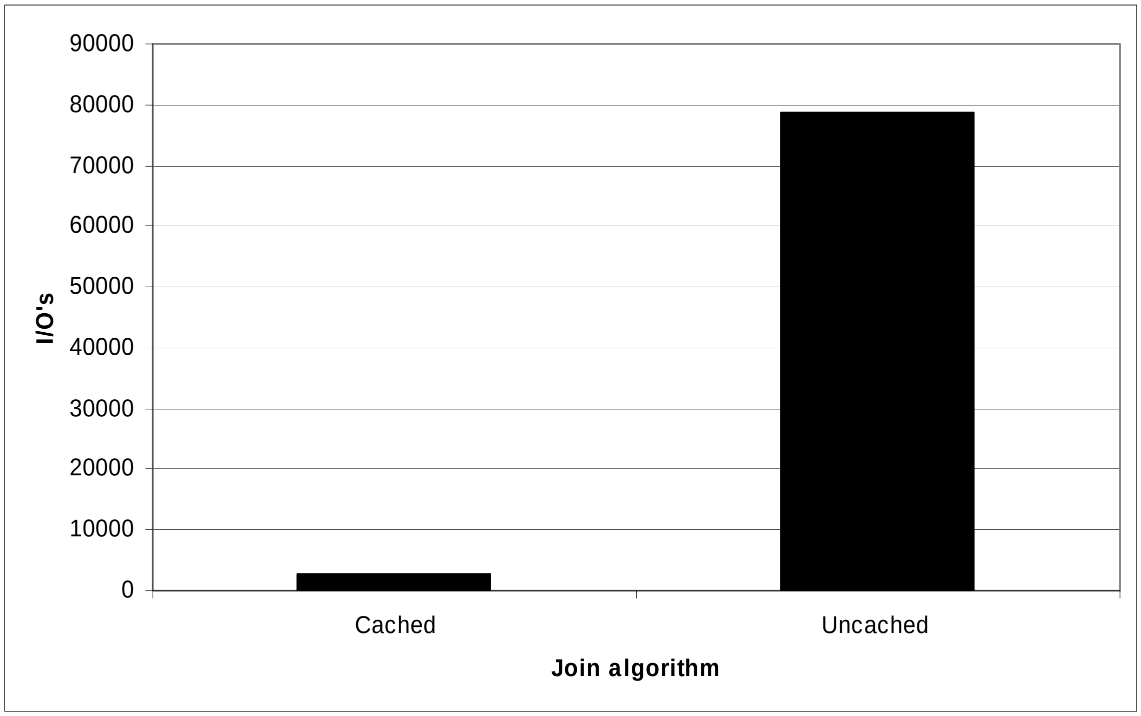


Figure 4.17 – The average number of I/O's

The effect of the additional tuples in the uncached join algorithm is clear. Uploaders must perform more read I/O's whilst downloaders must perform more write I/O's. Given that peers upload after they complete their download, the number of I/O's in the uncached join algorithm is much higher than in the cached join algorithm. Many of these I/O's in the uncached join algorithm are actually unnecessary because the data will be discarded once the downloaders perform a local join.

These experiments highlight an important issue in the Wigan system. Ideally, a downloader would dynamically select the best join algorithm for its query. In the experiments shown in Figure 4.17, the downloader should have some idea that the uncached join may not be good idea, because they will know they need to request every tuple in a table. However, this might not be a problem for a very small table, especially one which could fit entirely on one piece. If the Tracker published statistics showing the size of each table, this could help the downloader choose the best algorithm.

In the experiments shown in Figure 4.15, there is little the downloader can do because they may well be unaware of what advertisements are available before submitting their query. A published query log, similar to the SkyServer [12] logs could help. If a downloader examined the Tracker logs before submitting a query, they would be able to see if others have been submitting multi-table queries or not. Also, if a downloader uses the cached join and only the seed can answer the query but other peers are advertising the tables in question, the Tracker could suggest that the downloader resubmit separate single-table queries. Chapter 5 will discuss extensions to the Wigan system to prevent a join algorithm being used in its worst case.

4.8 GREATER PARALLELISATION OF SEED DOWNLOAD

Chapters 2 and 3 explained that the BitTorrent seed normally uploaded to a maximum of five peers at any one time. Figure 4.13 illustrated a scenario where the seed was receiving considerably more queries than it could respond to. This is a potential system bottleneck as described earlier in this chapter. This section introduces more parallelism to the seed and examines a seed which can accommodate a greater number of concurrent uploaders. This could be a method of overcoming the bottleneck problem. In this set of experiments, the seed was adjusted so that it could handle 10 times as many peers, i.e. upload to a maximum of 50 peers concurrently.

When examining the concept of a seed able to upload to more concurrent uploaders in conjunction with the queries used in earlier in this chapter, the performance advantage offered by such a seed would be most evident in the scenario where the queries are mostly different. This is because in this scenario, almost all queries are routed to the seed which quickly reaches its maximum number of permitted downloaders. If the seed could upload to 50 peers at once, then it would not become reach its maximum number of downloaders so quickly. More peers could begin downloading before the seed reaches the maximum number of downloaders. If more peers are downloading from the seed, then there are fewer peers waiting for the seed to have a free slot.

Although the greatest improvement in performance is expected to occur when the queries are mostly different, there is still likely to be some improvement compared with a standard seed if the queries were all the same. Peers submitting their query later in the download period would be able to obtain their data from an uploader advertising the query and would not need to interact with the seed at all. However, those joining near the start of the download period would be able to benefit from the greater capacity at the seed and could still receive their results faster.

To investigate the impact of a seed being able to upload to more peers concurrently, the experiments from Section 4.5, where the queries were all the same and the experiments where the queries were mostly different were repeated. All other parameters were kept unaltered; the only difference was the increased number of downloaders at the seed. Therefore, there were still 999 peers and one seed, each query was still either 10 randomly chosen rows from the Supplier table or the first 10 tuples from that table and the workload profile used was the constant workload with a rate of two queries per second. The average response time was compared with that for the standard seed, hosting up to five concurrent uploaders. Again, each experiment was repeated five times and an average was taken.

When all the queries were the same, the average response time was as shown in Figure 4.18:

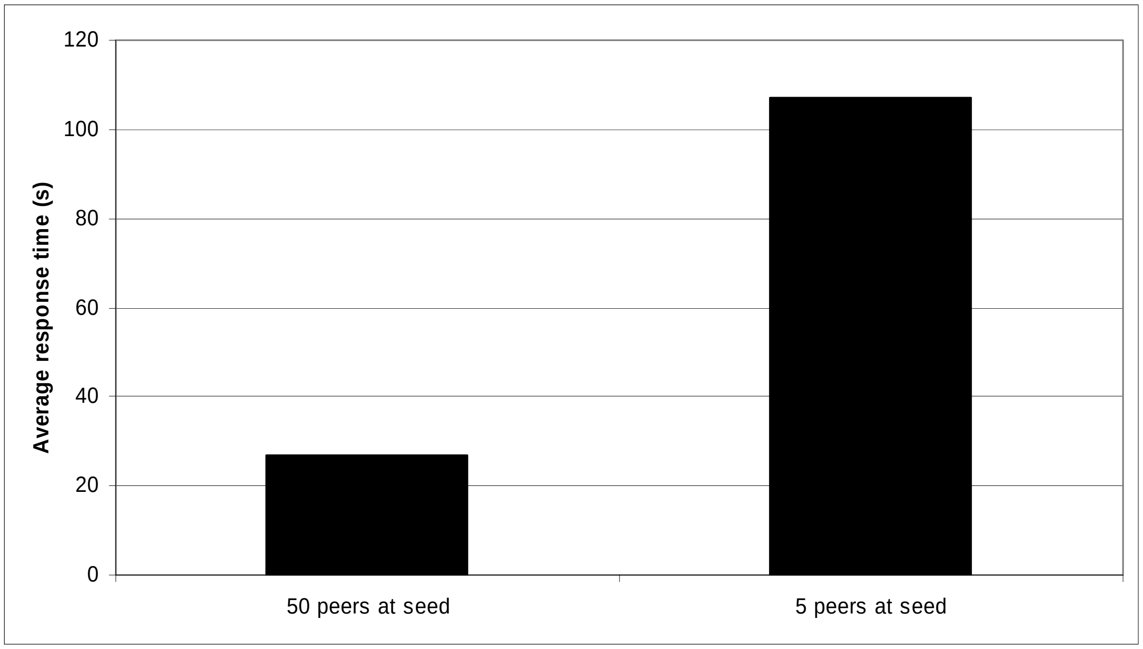


Figure 4.18 – Average response times for identical queries with a more powerful seed

When the seed can upload to 50 peers, the average response time is about four times faster than when the seed can only upload to five peers. In addition, when analysing the additional statistics collected during the simulation runs, it was observed that no peers timed out. All received their results before the timeout interval expired. When the seed could only upload to five peers at any one time, approximately 13% of peers timed out.

Figure 4.18 has indicated that a 10-fold increase in capacity has led to a fourfold increase in performance. However, as noted above, not all peers benefit from the increased number of downloaders at the seed because many do not need to interact with the seed whilst downloading and hence the case where the queries are mostly different should show a larger speed-up factor. The average response times for this scenario are shown in Figure 4.19:

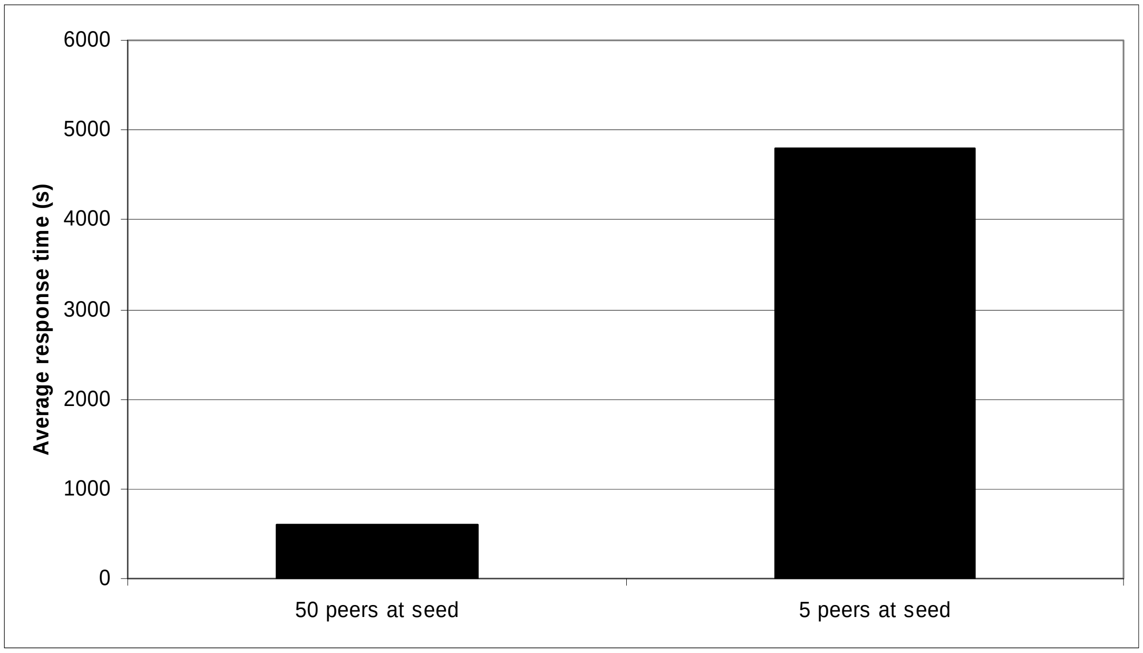


Figure 4.19 – Average response times for mostly different queries with a more powerful seed

Again, it can be observed that having a seed capable of uploading to 50 peers reduces the average response time. This time however, the increased parallelisation at the seed results in response times which are nearly eight times faster, not four times faster; they have fallen from about 1 hour 20 minutes to less than 10 minutes. In addition, the number of peers timing out has fallen from 951, i.e. nearly all, to 405, less than half.

The experiments from Figure 4.11 were then repeated to provide a detailed analysis of how the revised seed affected the overall response times. This was a constant workload with a rate of two queries per second and a 75% chance of a repeating query. The average response times for repeating queries are shown in Figure 4.20.

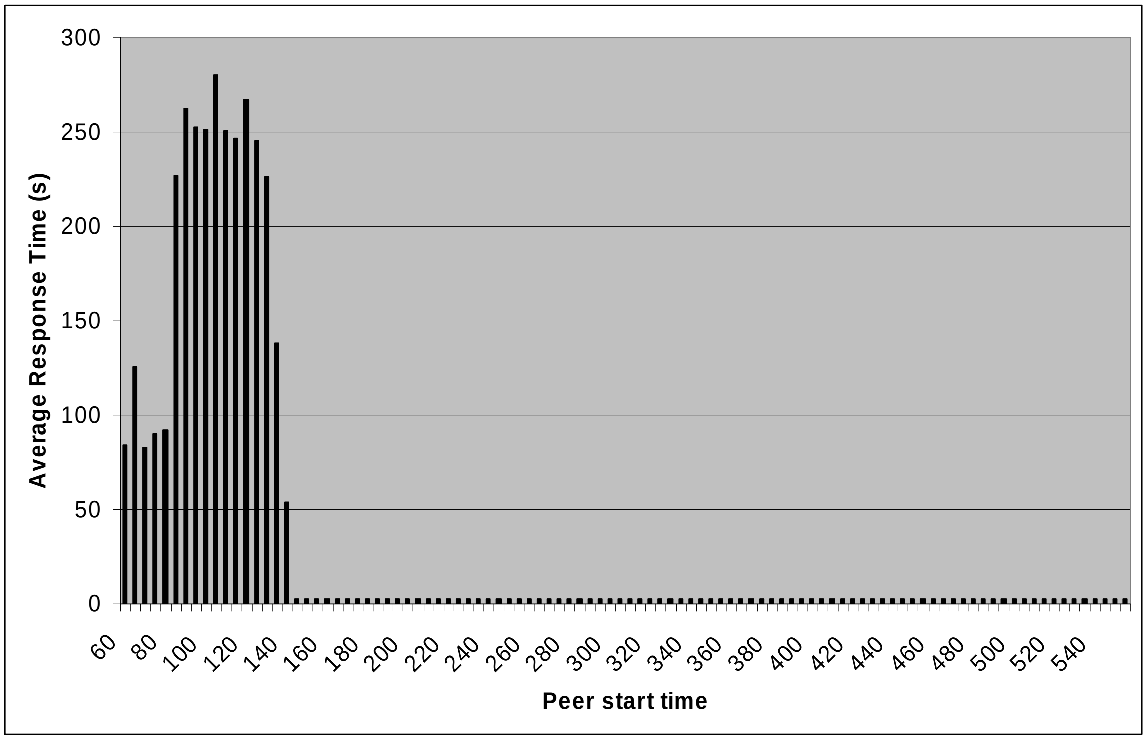


Figure 4.20 – average response times compared to start times

Examining Figure 4.20 in comparison with Figure 4.11 shows that, once again, the peers joining near the start of the download period experience a higher response time compared to those joining later. The amount of time taken before the response times fall and stabilise is still the same. However, there are two important differences between the graphs. Firstly, the average response times for peers joining near the start of the download period, with the increased parallelisation at the seed, are not as high. This reflects the reduction in response times, illustrated earlier in this section, resulting from the greater parallelisation at the seed. In Figure 4.11, some response times were higher than 700 seconds, the timeout interval in these experiments, whereas here this does not occur.

It is also possible to observe that in Figure 4.20 the average response times take longer to peak, whereas in Figure 4.11, the very high response times began at the very start of the download period. This is because it takes longer for the seed to reach the maximum number of concurrent downloaders and, during this time, more peers will complete.

These experiments indicate that a seed capable of uploading to more peers can help overcome the bottleneck of a busy seed, particularly if there are many queries that can only be answered by the seed. If the system had fewer peers joining per second, a seed like this would lead to even better performance because it might not reach the maximum number of concurrent downloaders before one of its downloaders completes. Note that this concept of a powerful seed capable of uploading to more peers is closer to a Client-Server system than to BitTorrent. However, it differs in that there are other peers available to help reduce the load on the seed. One possibility would be for the owner of the seed to determine and vary the number of concurrent downloaders permitted and this will be discussed further in Chapter 5.

4.9 COMPLETELY RANDOM QUERIES

Many of the experiments introduced so far in this chapter have included peers submitting a query request for a fixed number of random tuples. This section will examine what happens when peers submit totally random queries on a particular table. In these experiments, there was a seed and 999 other peers, each submitting a query on the Supplier table, which contains 10,000 tuples. Each peer selected a random starting tuple and a random number of tuples to request. If the starting tuple plus the number of tuples was larger than the size of the table, the end tuple was set to be the last tuple in the table. The workload profile was the constant workload at a rate of two queries per second.

In this set of experiments, an exact match between a downloader's query and an existing advert was most unlikely. The more likely scenario was that of the proper subset discussed in Chapter 3, i.e. for a downloader's query, there would be an advert for another query with a larger result set which could resolve the downloader's query. An example of this would be if a peer requested a large part of the table, i.e. the starting tuple in the query was one of the first tuples in the table and the last tuple in the query was one of the last in the table. A peer advertising a query like this would be able to answer large numbers of queries across the table. It is also possible that the seed would be the only option for some downloaders if their query was not covered by any of the other adverts. Chapter 3 noted how the Tracker would return a peer list with the uploaders split into groups, ordered by query, with the closest to an exact match first. In this set of experiments, it would be likely that each of these uploader groups would consist of only one peer, given the low odds of two peers requesting the same query.

This high chance that each group of uploaders contains only one peer means that there is a chance that a peer will have to change query groups. This occurs when a downloader is choked by an uploader, that uploader is the only one in the current group and there are more groups to try. An example of multiple query groups was illustrated in Chapter 3, Section 3.3.3. One important aspect of these experiments concerns what action a peer should take if it is snubbed, i.e. choked by all uploaders in all of the groups. The final uploader that each peer tries will be the seed because it always has lowest priority as it is the furthest from an exact match to any query, with the exception of a query requesting the entire table. One option would be for a peer to keep requesting data from the seed until either a timeout occurs or the seed begins providing data. A second option would be to re-contact the Tracker. However, it is possible that if a downloader only knows of the seed and one other uploader and is choked by both, there may be no other peers that could provide the results. Therefore, the downloader would find themselves in the same position again, being snubbed by the same uploaders. To prevent this, a short delay before contacting the Tracker again could help. The delay between the choke and the re-contact was set to one minute as this would allow some time for more peers to complete the download. Both options were tested in this set of experiments – repeatedly querying the seed and contacting the Tracker again.

The number of peers is also an important factor. More peers can mean more queues forming around the system as uploaders attempt to serve an increasing number of downloaders, but equally could mean more choice for later downloaders. In this experiment, a small network of 100 peers was trialled initially, followed by larger networks of 500 and 600 peers. The average response times are shown in Figure 4.21:

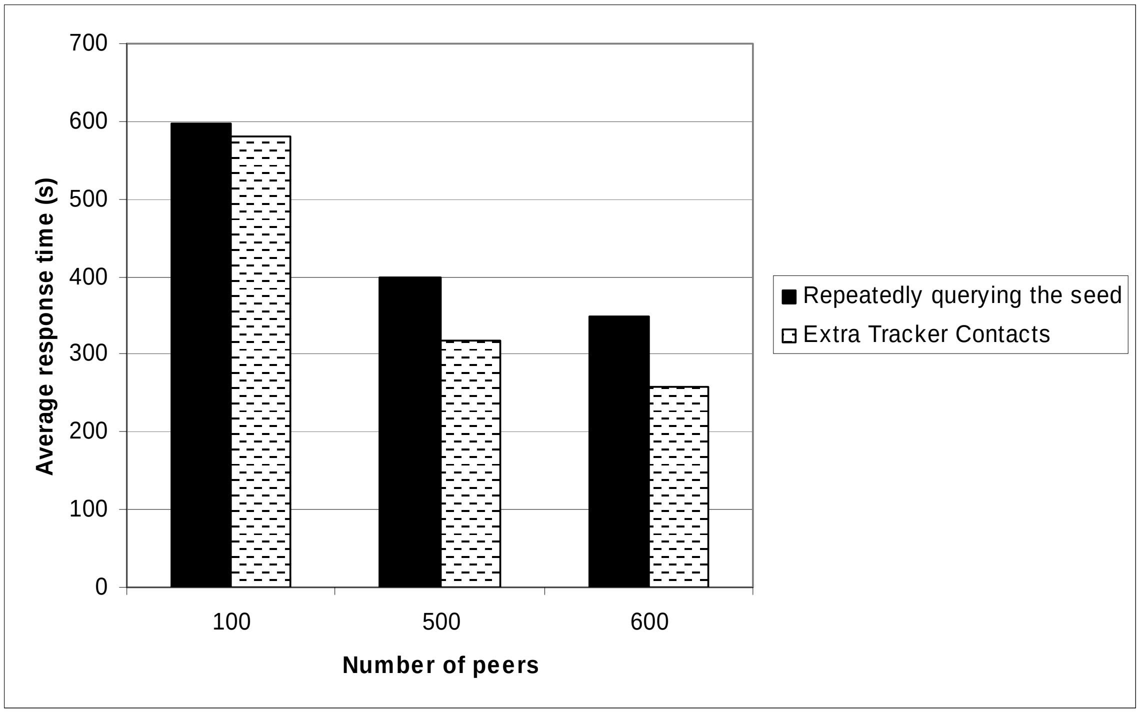


Figure 4.21 – Average response times for totally random queries

The first observation is that if snubbed, it is faster for a peer to re-contact the Tracker instead of continually querying the seed. This is because the seed will also be dealing with peers who cannot obtain data from anywhere else and hence will be busy, as was the case in many previous experiments illustrated in this chapter. Getting a new list from the Tracker after a minute's wait allows time for other downloaders to complete and advertise through the Tracker. Hence more peers will be advertising at the Tracker and thus there will be a greater chance of finding an uploader other than the seed that can answer the query. Re-contacting the Tracker immediately would not have given sufficient time for other peers to advertise a query that could be used to generate the result.

The second point to note is that response times fall as the number of peers increases. This is because, as stated earlier, more peers mean more choice for future downloaders. When the number of peers increases, so does the number of different queries being advertised in the system and thus there is a greater chance that a downloader will be able to find an uploader other than the seed that can answer their query. During the simulation runs, statistics were also collected showing the number of requests received by the Tracker for which only one advert (i.e. just the seed) could resolve the query and the number of requests received by the Tracker which could be resolved by multiple peers. Note that these statistics include repeat queries from those peers which had timed out and contacted the Tracker again. The statistics are shown in Figure 4.22:

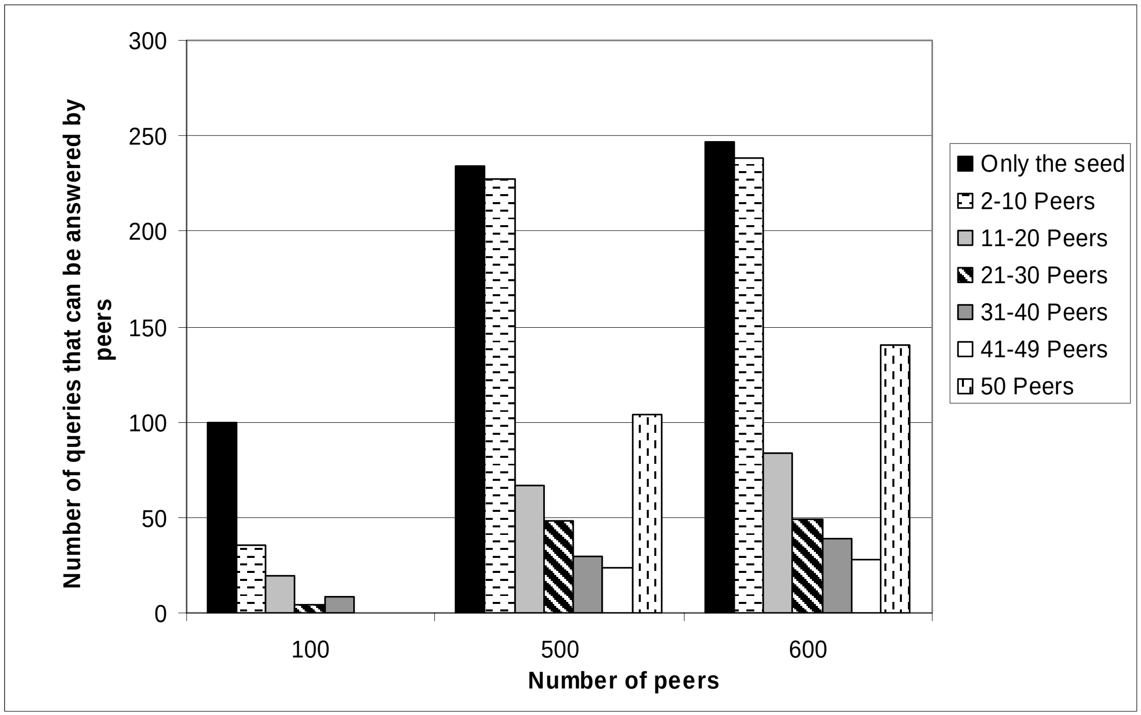


Figure 4.22 – Percentage of queries answerable by the seed and multiple peers

When the number of peers increases, there are fewer queries which can be answered by only the seed and more which can be answered by other peers. This shows that if the number of peers is large enough, there is a chance that there will be another peer or peers available to answer a query. Indeed for both 500 and 600 peers, the majority of queries are answerable by other peers apart from the seed. Looking at the breakdown in more detail, it can also be observed that the percentage of queries which can be answered by 50 peers increases as the number of peers increases, indeed with 100 peers, no queries can be answered by 50 peers.

Finally, it can be observed from Figure 4.21 that the average response times are quite high. The initial logjam at the seed accounts for some of this problem, however, there are also peers which must continue to queue at the seed because nobody else has the query they want. Note that the average response times are better than the "Mostly different" scenario from Figure 4.8 because although there are fewer peers in the experiments described in this section, those from Figure 4.8 are for queries which all return 10 tuples. There is no chance of a downloader's query being a subset of an existing advertisement.

These experiments highlight the performance of the system in what is potentially one of the worst cases – completely different queries which stand little chance of being repeated. Although there is a chance that there may be multiple peers capable of answering many queries, the average response times are not good because those downloaders which cannot receive their results from any peer apart from the seed do have an impact. The high number of requests per second in the system also causes queues to form, so this indicates that Wigan is less suited to a busy system where all the queries are entirely random.

However, the way in which the number of peers influences response time shows how peers in a P2P system such as Wigan contribute to scalability by providing servicing the requests of other peers.

4.10 CONCURRENT PIECE REQUESTS

Chapter 3 explained how pieces of data were requested sequentially in Wigan. This section will examine the effect of making concurrent piece requests. This could be of benefit if there are several pieces to receive and there are multiple uploaders which could provide the query.

A modified version of the simulator was created which would enable downloading peers to make multiple piece requests. The number of concurrent requests sent was a parameter, and for these experiments, it was set at five, which is the number of sub-pieces requested by BitTorrent. Note that five requests was the upper limit, if the list of uploaders given to a downloader contained one to four uploaders, the downloader could only send a request to each of them. The downloader would not send multiple concurrent requests to the same uploader although it would send sequential requests. For example, consider a downloader that has sent five concurrent requests to uploading peers. When a piece arrives from an uploader, the downloader will then send that uploader a request for the sixth piece. In these experiments, all peers submitted the same query – although this query varied from experiment to experiment – and the same workload was used as in most of the previous experiments outlined in this chapter – a continual stream of two queries per second. Each experiment was repeated five times and an average was taken.

The first set of experiments featured a query whose results contained 1,000 tuples which, at 50 tuples per piece, fitted onto 20 pieces. The average response time is shown in Figure 4.23.

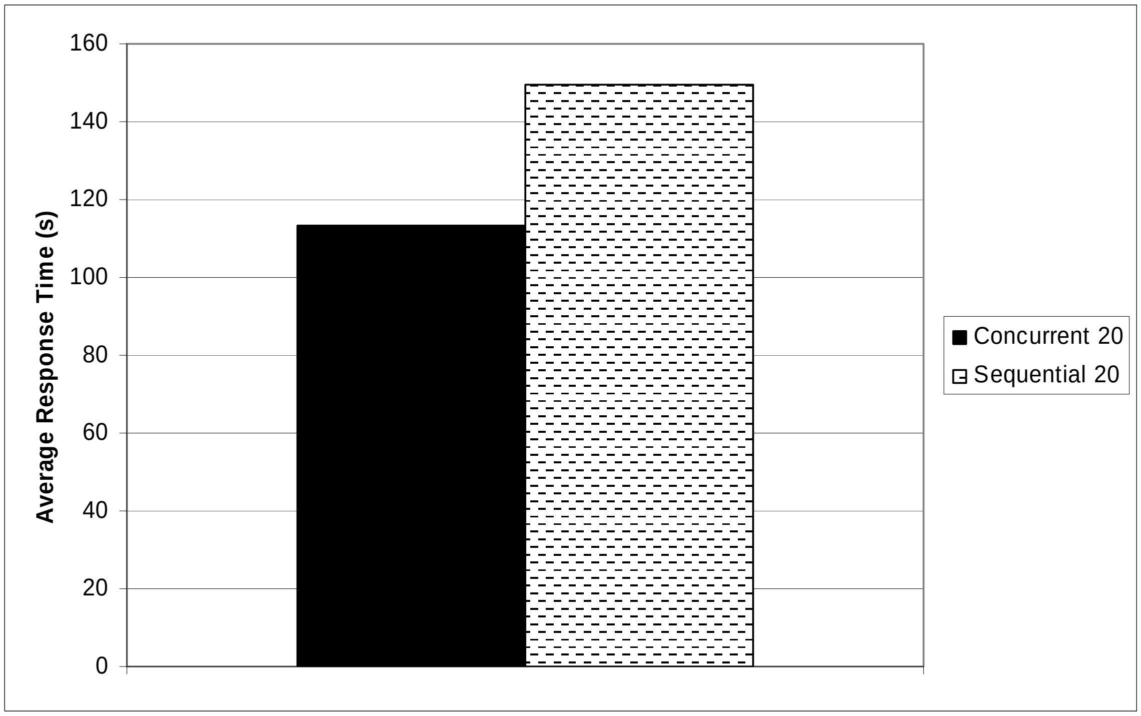


Figure 4.23 – The average response time for a 20 piece query

Making concurrent requests clearly has a performance advantage. This advantage cannot come from the opening part of the download period where only the seed is available to provide data; this remains as busy as before. The speed-up begins once the second peer to download the query begins advertising. At this point, the next downloaders to submit a query have two uploaders they can send requests to concurrently. This is why the speed-up does not begin when the first peer completes the download – in its query group, there will just be the one uploader that can answer the query. However, once the second peer to download the query starts advertising, all new downloaders can make concurrent requests.

In order to explore how result size affects the performance gains of concurrency, a query whose result set was twice the size of the previous query was tried next. All other parameters remained unchanged.

The results are shown in Figure 4.24 and have been combined with the results for the 20 piece query in Figure 4.25:

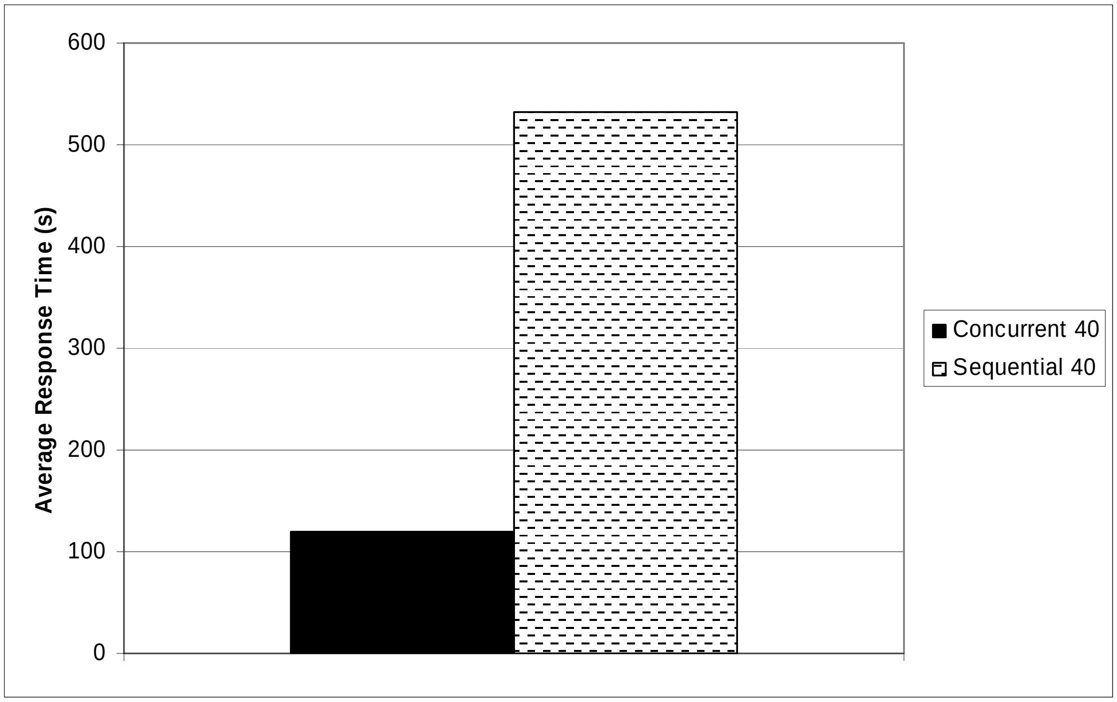


Figure 4.24 – The average response time for a 40 piece query

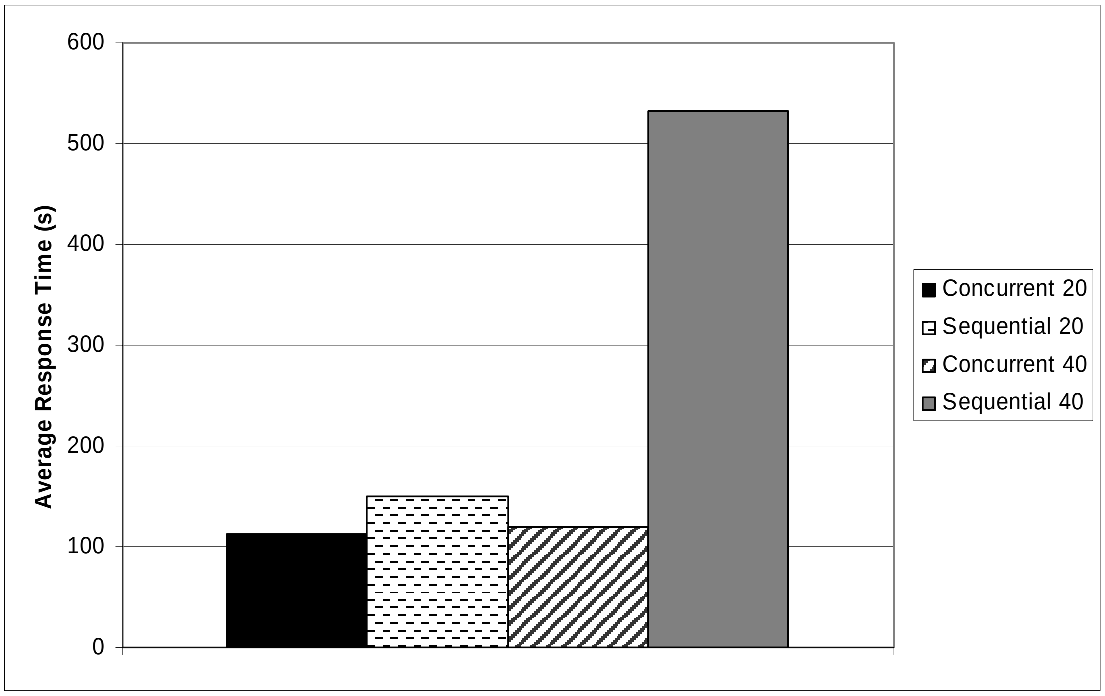


Figure 4.25 – a comparison of 20 and 40 piece query response times

The concurrent algorithm is considerably faster than the sequential. Making concurrent piece requests is over four times faster than requesting pieces sequentially.

One feature of the concurrent algorithm is that it scales very well compared to the sequential algorithm. Figure 4.25 has been re-arranged to show the two concurrent results and the two sequential results next to each other - Figure 4.26 – to illustrate this.

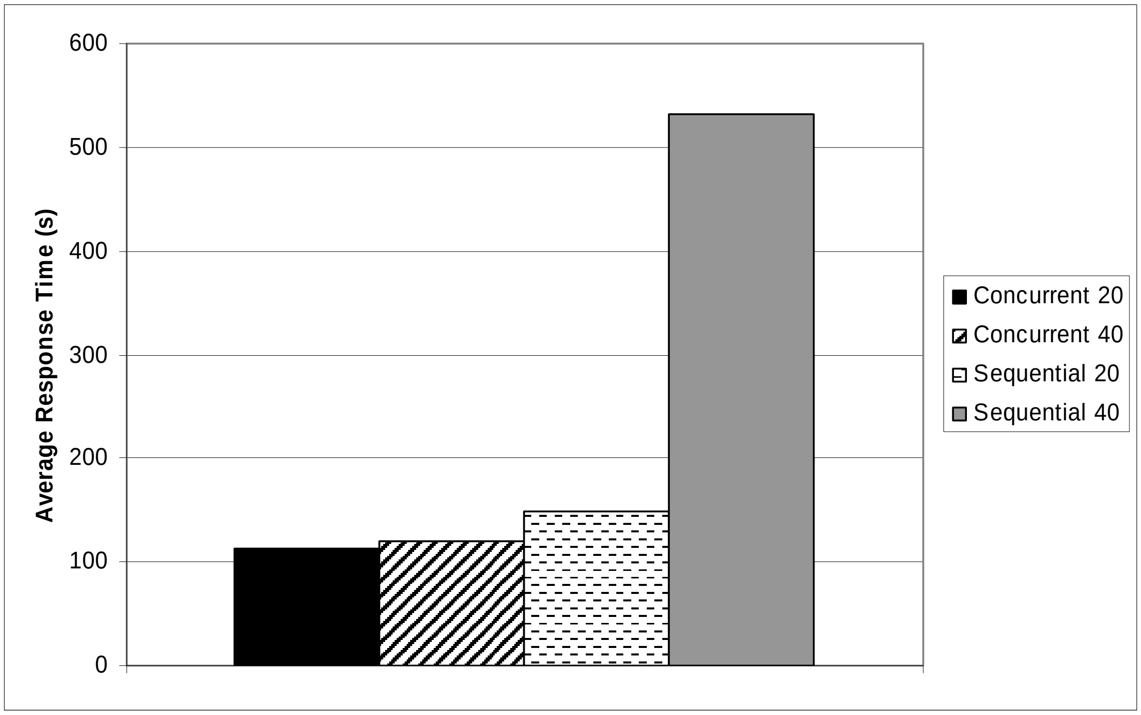


Figure 4.26 – The relative increase between the algorithms

The reason for the large delay in the sequential algorithm is because of the increased size of the query. The initial backlog at the seed will not be the problem; this will be the same as in previous experiments. Instead, the delay will begin when there are a small number of peers available to upload the query. During this period, all new downloaders contacting the Tracker will be informed about the small group of uploaders and will attempt to obtain data from them. It will take longer for a downloader to request 40 pieces than it would to request 20 pieces, so a downloader will occupy a slot at an uploader for longer. These uploaders all have a limited number of slots for downloaders, and if these slots are being occupied for longer, other downloaders will have to wait. It is this wait which leads to the increase in the response time.

To study the overall pattern of the algorithms' behaviour, the experiments were repeated for 10 and 80 piece queries. The overall results are shown in Figure 4.27:

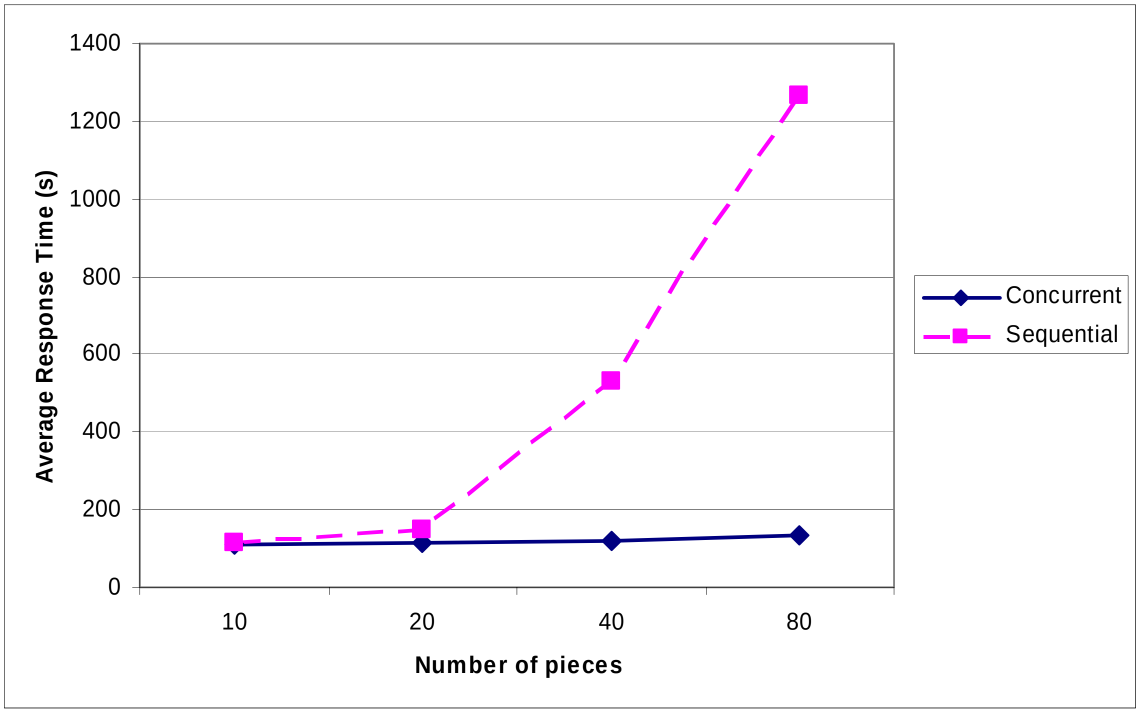


Figure 4.27 – Concurrent and sequential piece requests for different sized queries

It can be seen that for the 10 piece query, the two algorithms offer similar results. This is because with a smaller number of pieces, the sequential algorithm has fewer requests to make when compared to queries which have a larger result set, and hence a downloader will complete the download faster. This in turn results in adverts being placed at the Tracker more quickly and fewer peers joining later in the download period will have to contact the seed. The concurrent algorithm still has a slight performance advantage because it will have fewer requests to make than the sequential. For example, if the downloader can send the maximum number of concurrent requests (five in these experiments), it can be simultaneously receiving half the result set from different uploaders.

The scalability of the concurrent algorithm shown in Figure 4.26 is still here across the larger range of pieces. There is an increase in response times caused by the need to send more requests, but this increase is very gradual. It can also be observed that with an 80-piece query, the concurrent algorithm is more than five times faster than the sequential, despite the fact that a peer using the concurrent algorithm will send at most five times fewer sequential requests than a peer using the sequential. The extra speed-up stems from the fact that with larger queries in the sequential algorithm, downloaders will take longer to receive their data hence when there are only a few uploaders, backlogs will develop.

4.11 THE EFFECT OF CHANGING THE PIECE SIZE

The experiments in Section 4.10 analysed the behaviour of Wigan when concurrent piece requests were submitted for queries of varying numbers of pieces. This section will examine the effect of changing the size of these pieces.

There is a spectrum of possibilities from one piece per tuple to one piece per table. Larger pieces do have some advantages. At the start of the download period, peers have to request fewer pieces from the seed, which scans more data for a single request. As the piece size decreases, the number of messages, in particular piece requests, increases and more importantly so does the overhead of communications.

However, there are some disadvantages of bigger pieces. Larger pieces contain more data and therefore take longer to travel from the uploader to the downloader than smaller pieces. Larger pieces also have a greater impact on the performance of uploading peers, because the uploader has to perform more I/O's to process requests. Whilst the amount of data requested by a downloader clearly depends on the query size and not the piece size, an uploader receiving continual requests for large amounts of data could be doing more work than one receiving requests for smaller amounts of data, if it gets the same number of requests. If this impacts on the performance of the uploader, the owner of the uploader may decide to disconnect from the Wigan system, particularly if they want to use their machine to perform some other operations unconnected with Wigan. Larger pieces also mean that choking has a greater impact. As the piece size increases, the level of possible parallelism decreases. Consider a downloader which has sent some concurrent piece requests. If it is choked by one uploader, it will have to request that piece again from elsewhere, but it will still receive the other pieces it has requested from the remaining uploaders. If the pieces were bigger and the downloader was choked by one uploader, it would have to send a new request to a different uploader for a potentially larger part of the result set. For example, if a query was divided into 10 pieces and a downloader was choked by an uploader, the downloader would have to find a new uploader to request a tenth of the query results from. However, if the same query was divided into only two pieces and the downloader was choked by an uploader, would need to find a new uploader to request half the query results from. The extreme case occurs when the pieces contain an entire table and thus all queries can fit on one piece. This forces the use of sequential requests and if a downloader is choked by an uploader, it will have to send out a new request to a different uploader for the entire query.

To explore the impact of piece sizes, an experiment from Section 4.10 was repeated. In this experiment, all 999 ordinary peers submitted requests for the same query, whose result contained 2,000 tuples. Different piece sizes were tested, ranging from the smallest possible – one tuple per piece – to the largest possible – all 10,000 tuples in one piece, i.e. one piece holding the entire Supplier table. The piece sizes are shown in Table 4.4.

Table 4.4 – Differing piece sizes

Tuples per Piece	**Number of pieces in table**	**Number of pieces in query**
1	10,000	2,000
10	1,000	200
30	334	67
50	200	40
100	100	20
400	25	5
500	20	4
750	14	3
1,000	10	2
10,000	1	1

Figure 4.28 shows the average response times for these piece sizes, while Figure 4.29 gives more detail on the results for the larger piece sizes.

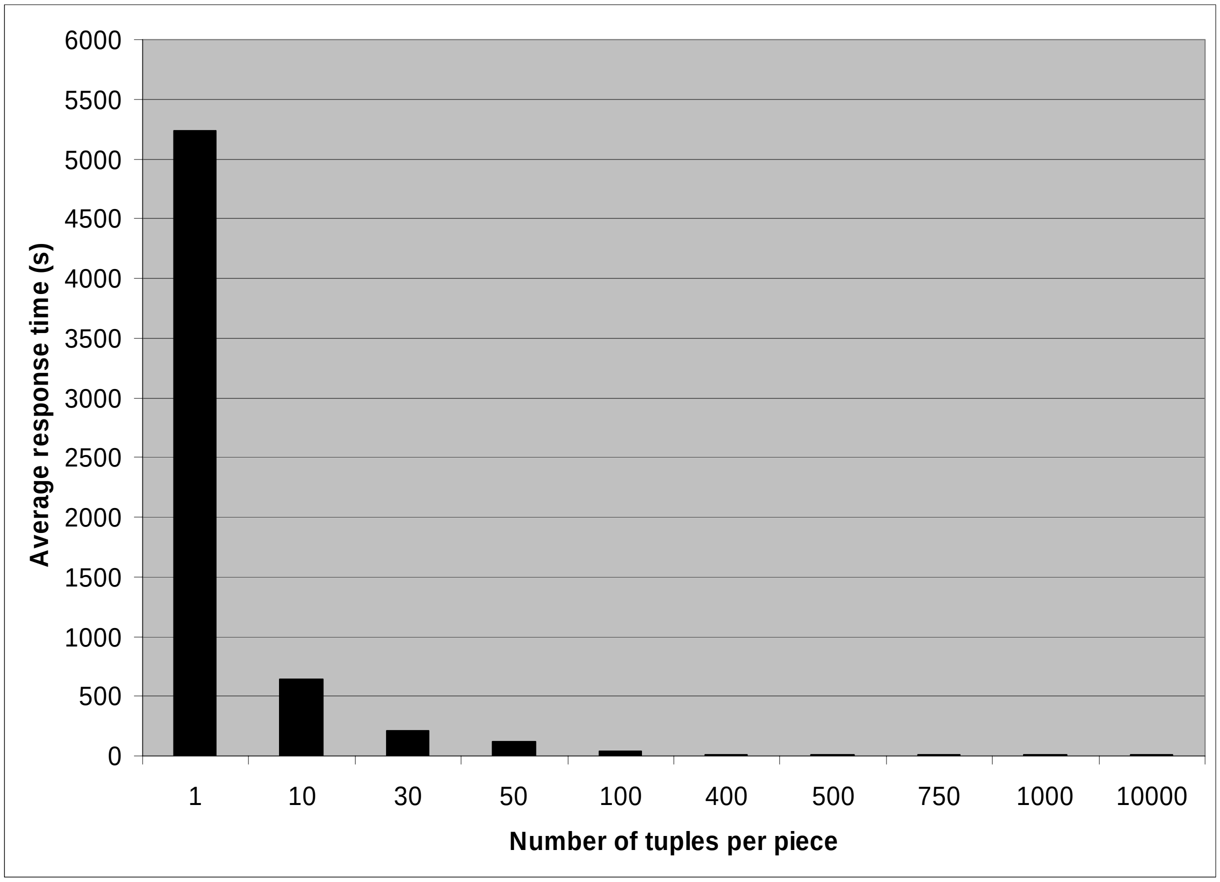


Figure 4.28 – the effect of changing the piece size

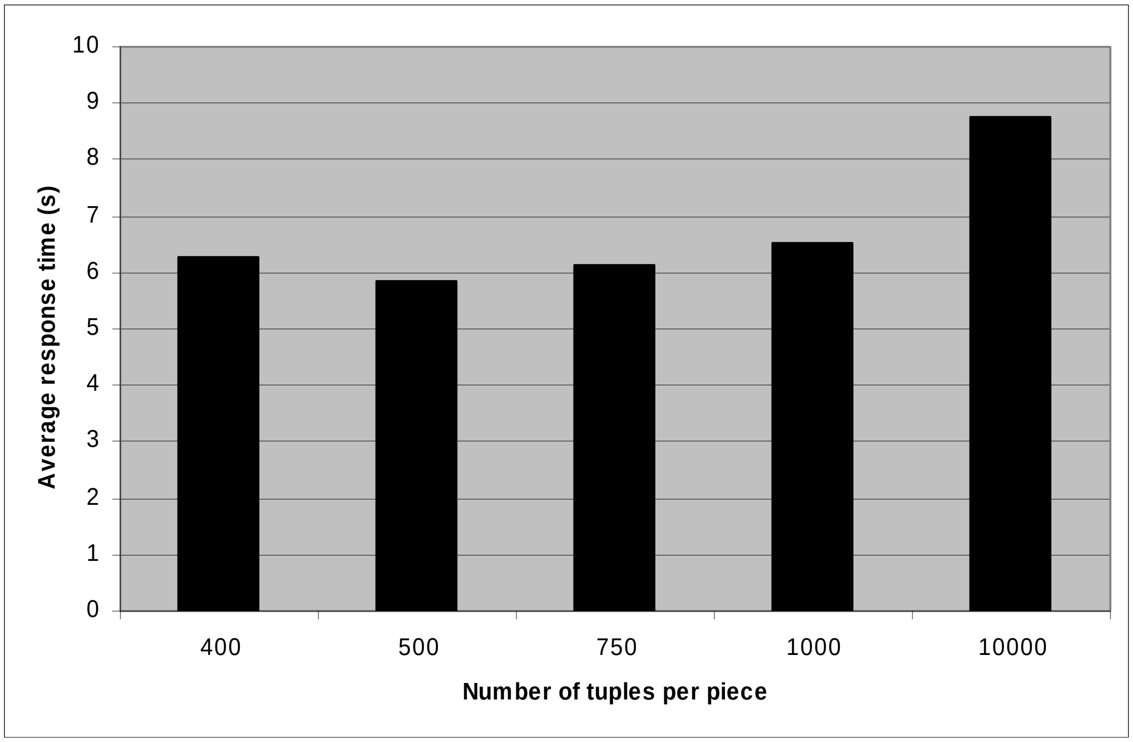


Figure 4.29 – details of the response times for larger pieces

It can be seen that a large number of very small pieces creates a very high response time. This is due to peers having to submit a larger number of requests because of the larger number of pieces. There is more parallelism, but the overhead of this additional parallelism is so high that performance suffers. The worst case of this occurs when downloading from the seed; because the query result is 8,000 tuples smaller than the size of the table, there will be a large number of requests which are met with an empty results set. Even when downloading from a peer with exactly the same query, a large number of requests must still be sent.

A piece size of one tuple has a particularly poor effect on response time. Initially, when all peers have to query the seed, they must each send 10,000 requests, one for each tuple of the Supplier table. The majority of these requests (8,000) receive an empty result set as a response because, as noted above, there are only 2,000 rows matching the query. Submitting 10,000 requests takes a long time with the result that no peer has completed by the time the final downloader, Peer 999, submits its query request. The advantage of having additional uploaders available to reduce the load on the seed cannot be exploited at this point. The seed has to try and serve all 999 downloaders, which it clearly cannot do quickly. Eventually, after more than an hour, some peers complete their download and advertise at the Tracker. When other downloaders time out and contact the Tracker again, they will be informed by the Tracker that there are other uploaders available. The downloaders will then contact these uploaders and attempt to obtain the data. This will involve sending 2,000 requests – one for each tuple in the results – which again will take some time, although it will be faster than having to despatch 10,000 requests. Each downloader can send out five concurrent piece requests to uploaders. Only a few uploaders are available at this time, so these will also receive a large number of requests. The large number of downloaders, compared to uploaders, means that the system as a whole must be receiving more requests than it is able to respond to, hence download times are increased considerably.

Conversely, a smaller number of larger pieces produces a lower response time. This is of particular significance when the number of pieces is less than five (a piece size of 400 tuples) because a peer will be able to send out one set of concurrent requests which will cover the entire query. However, one or two very large pieces are not optimal due to the disadvantages caused by the loss of parallelisation described above. A piece size of 500 tuples produces the best result for this query, indicating that this is the best balance point between the number of pieces, the overhead of communication and the level of parallelism. To examine the impact of changing the piece size in more detail, the experiments used to generate Figure 4.11 were repeated with a piece size of 500 tuples per piece. This was a constant workload with a rate of two queries per second and a 75% chance of a repeating query. The average response times for the repeating queries are shown in Figure 4.30.

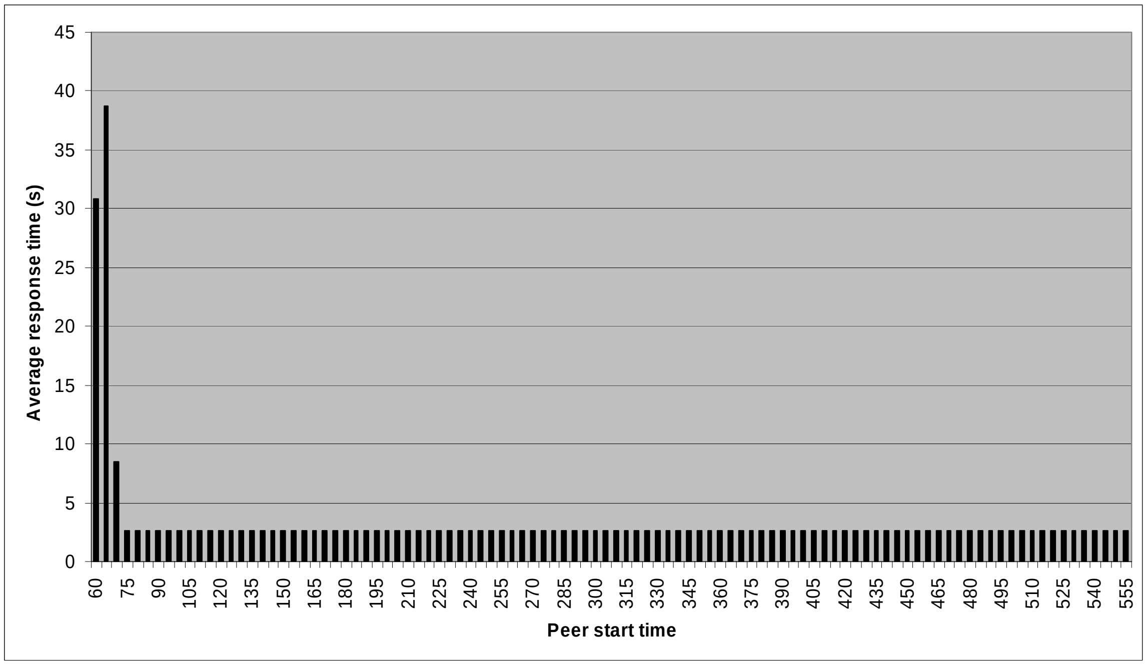


Figure 4.30 – average response times for repeating queries with a larger piece size of 500 tuples

Firstly, it can be observed that the overall pattern of the graph is the same in that peers joining at the start of the download period experience a higher response time than those joining later. Once repeating queries can be obtained from other peers, the response times stabilise.

However, there are two crucial differences between Figure 4.11 and Figure 4.30. Firstly, the delays experienced by peers joining near the start of the download period are not as severe as in Figure 4.11; they are in the order of seconds, not minutes. Secondly, the response times fall and stabilise much earlier. This indicates that bigger pieces can provide an effective means of reducing the initial overhead at the seed because they reduce the number of requests that peers submit to the seed. This results in peers advertising earlier and thus providing more choice for future downloaders.

4.12 REPLICATING THE SEED

Section 4.8 indicated one means of improving the performance was to introduce a seed that could handle more downloaders simultaneously. This meant that it took longer for the seed to reach its maximum number of downloaders and led to a reduction in the average query response time, when compared to the original experiments performed in Section 4.5. An alternative approach would be to introduce replicas of the standard seed (i.e. a seed which could upload to a maximum of five peers) and introducing a means of load balancing. Like a single seed that could upload to more downloaders simultaneously, multiple seeds would provide more capacity to handle the initial rush of queries, however multiple seeds would also permit peers to make concurrent piece requests from the start of the download period. This could enable downloaders that submit the repeating query near the start of the download period to complete their download faster and hence begin advertising sooner. Once peers start advertising the repeating query, other downloaders will be directed to them and not the seed. Concurrent piece requests could also help peers submitting random queries because these peers are likely to have to download all data from the seed and could therefore request multiple pieces simultaneously from different seeds.

In order to investigate this further, the experiment shown in Figure 4.11 was repeated, with differing numbers of seeds available. The maximum number of seeds used was 50, the maximum number of peers which could be returned to a downloader by the Tracker. Having 50 seeds means that all downloaders would receive a full piece list, regardless of their start time or the query they submit. All other parameters from Figure 4.11 were kept the same to ensure a fair comparison. It is important to note that in these experiments, all seeds remained online for the entire duration of the download period and, unlike the peers, did not disconnect after a random amount of time. This assumption was made because, as discussed in Section 3.3.1, the seed is assumed to belong to the database owner or administrator and is therefore able to answer queries at any time. The average response times for the differing numbers of seeds are shown in Figure 4.31.

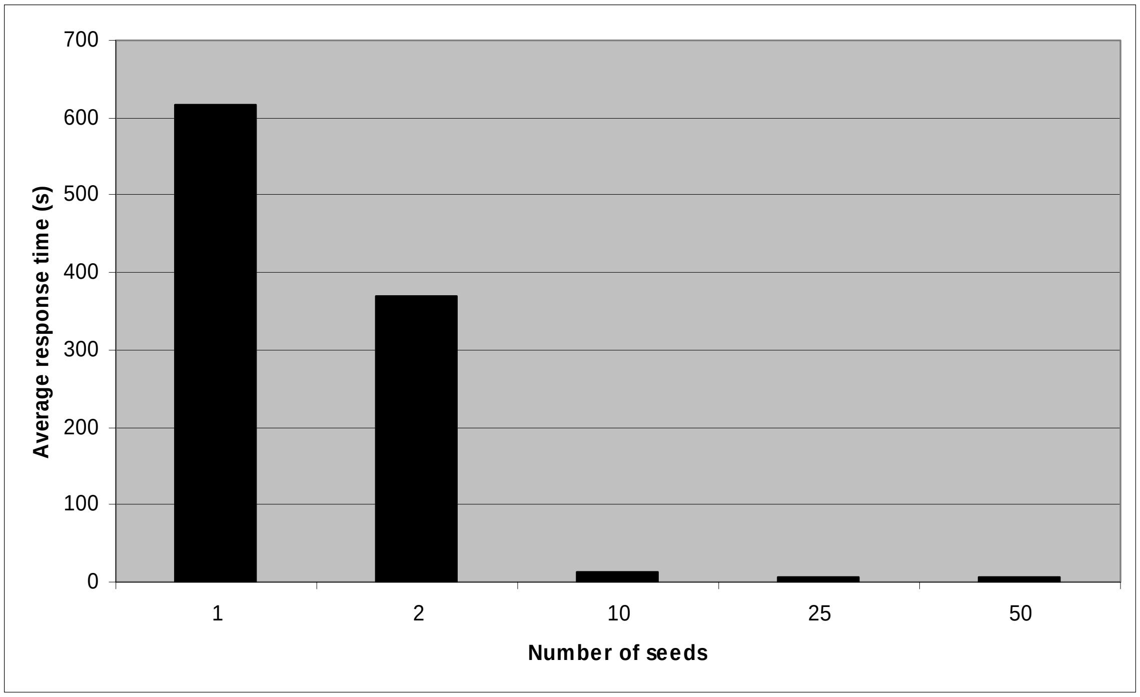


Figure 4.31 – average response times for differing numbers of seeds

Firstly, it can be observed that, as the number of seeds increases, the average response time falls. The biggest decrease in the average response times comes between two and 10 seeds. Increasing from 10 to 25 seeds reduces the average response time further from just over 13 seconds to approximately 7.3 seconds. However, the increase from 25 to 50 seeds reduces response time by approximately 0.5 seconds. Given that the increase from 25 to 50 seeds represents a doubling in seed capacity, this is not much of a reduction and hence it is likely that the database owner would not want to purchase these extra 25 seeds for such a small reduction in the response time, instead it is more likely that they would have 10 or 25 seeds. The average response times for both 10 and 25 seeds are more than 10 and 25 times faster than the average response times for just a single seed, thus it could be argued that the extra seeds are worth the investment.

To investigate the extra capacity offered by the additional seeds, a graph of 1/average response time was plotted. This is Figure 4.32.

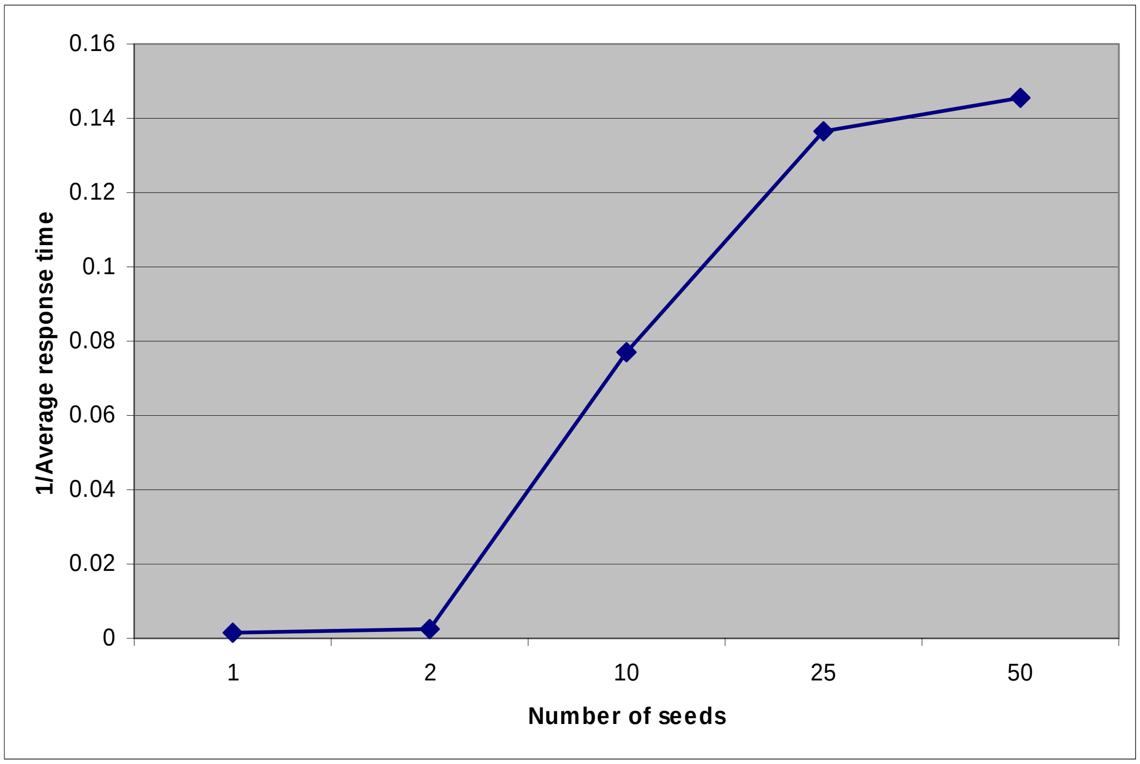


Figure 4.32 – 1/Average response time compared to the number of seeds

As the number of seeds increases, the capacity increases in a linear manner from 2 – 25 seeds. However, before and after this point, the increase in capacity reduces. From 1-2 seeds, this is because the increase in capacity is not sufficient to permit the system to cope with the large stream of requests at the start of the download period. From 25-50 seeds, the reduction in capacity occurs for the opposite reason – 25 seeds are already capable of absorbing almost the entire initial query stream and hence the large capability offered by the 50 seeds is not fully exploited.

Direct comparisons with Figure 4.11 will now be made. The statistics for 10 seeds are shown in Figure 4.33 and those for 25 seeds in Figure 4.34.

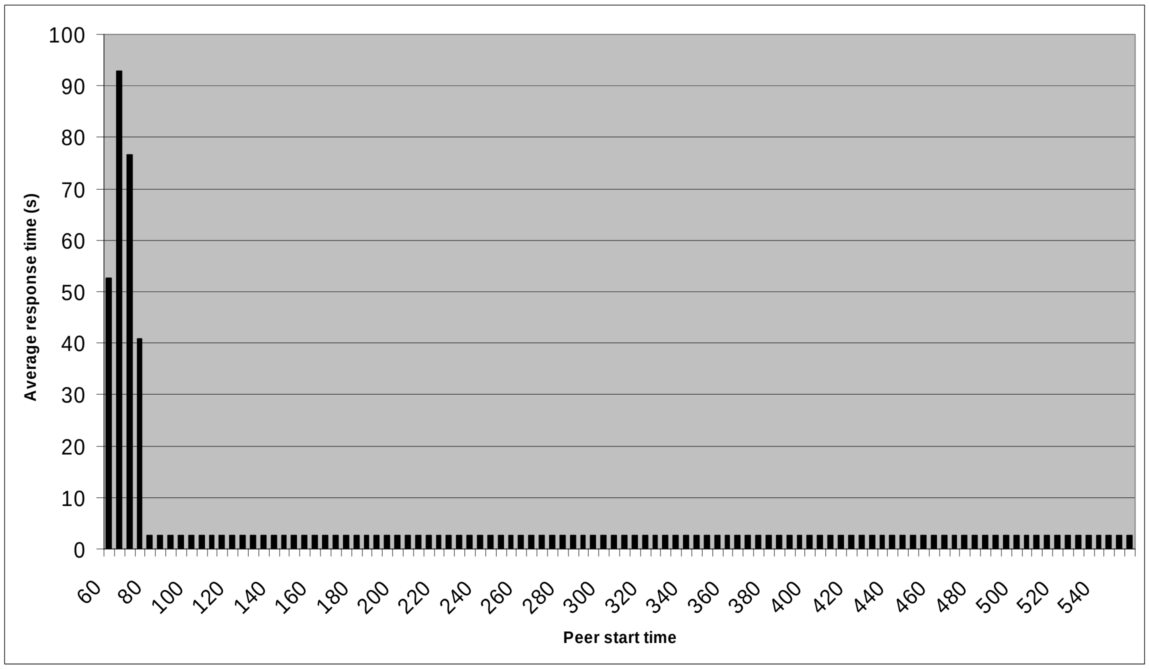


Figure 4.33 – average response times for 10 seeds

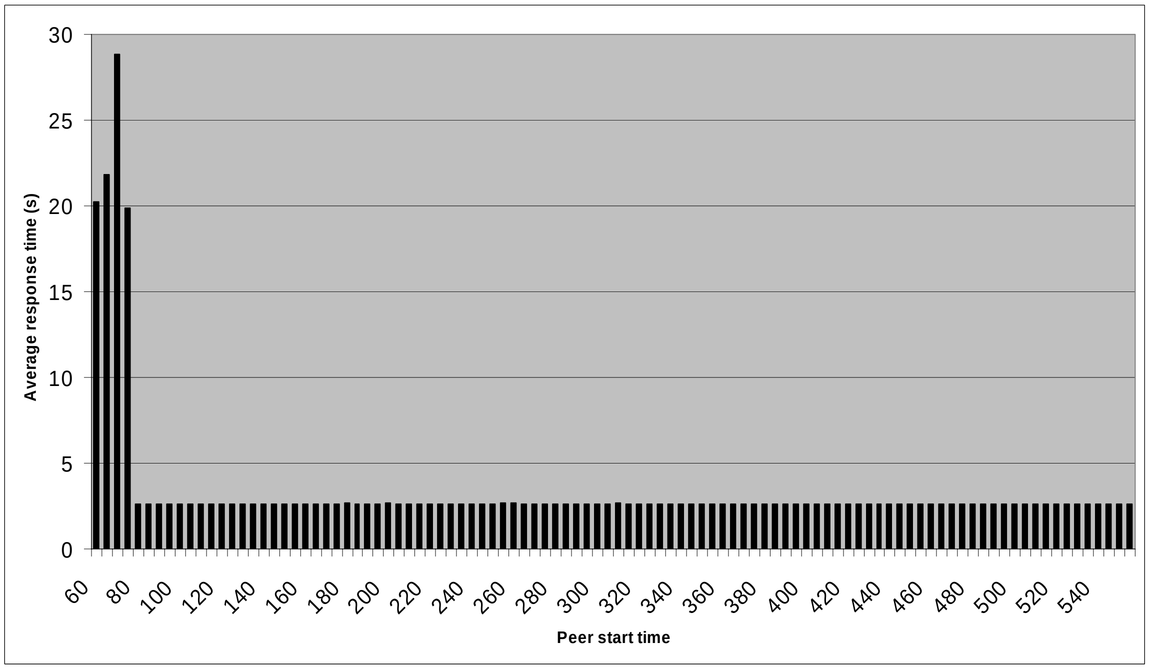


Figure 4.34 – average response times for 25 seeds

Once again, the overall shape of the graphs has not changed, peers joining at the beginning experiencing a higher response time before the times fall and stabilise. However, when compared to Figure 4.11, the response times are much lower – less than a minute in the case of 25 seeds – and also fall and stabilise much faster. When comparing Figure 4.33 with Figure 4.34, it can be seen that, although the response times do not stabilise any faster with 25 seeds, the peak is much lower.

4.13 EARLY RELEASE OF DATA

Chapter 3, Section 3.4.2, explained how, in Wigan, peers do not start advertising at the Tracker until they have received the complete results set because a peer cannot predict exactly how many pieces it will receive as it may be downloading from a peer advertising a different query. It was noted in Section 3.4.2 that this is different to BitTorrent, where peers can begin advertising as soon as they have received a single piece of a file. This section will further discuss the problems associated with early release of data by peers and also examine a scenario where it may be possible to release data early and will then assess the implications of this. Note that additional terminology will be required in this section, because there will now be some peers that are both uploaders and downloaders. Therefore, the BitTorrent term "Leecher" will be used to describe a peer that is both downloading and uploading simultaneously. The terms "Uploader" and "Downloader" will continue to refer to peers that are only uploading and only downloading respectively.

A potential problem with storing data, as noted in Section 3.4.2, was the fact that there is no control of how different peers store data in their local database systems. In Wigan, as explained in Section 3.4.2, this problem is overcome by peers creating mappings between tuple ID and piece number after they receive their query results. In order to permit early release of data, peers must create the row to piece mappings after they receive a single piece.

Section 3.4.2 explained that if a peer was receiving data from an uploader advertising a different query to the one it is requesting, for example the seed, there may be some tuples held by the uploader that do not match the downloader's query, thus the pieces received by a downloader may not be completely full of data. For efficiency reasons, the piece structure is changed once the download is complete as described in Section 3.4.2. However, in order to permit early advertising, the piece structure must, in effect, be changed dynamically by leechers as they receive data. Leechers must update their row to piece mappings when they have received the equivalent of one full piece. If the first two pieces received by a leecher were half full of data, the leecher would therefore have one full piece of data in the new piece structure. Assuming 50 tuples per piece, this is illustrated in Figure 4.35 ,where a number in a square represents the number of tuples in that piece that match the conditions of the query.

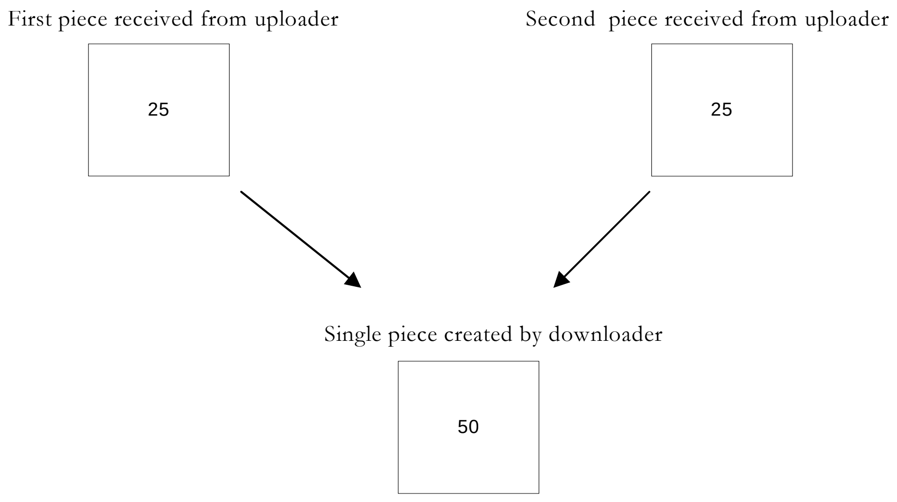

First piece received from uploader
Second piece received from uploader
25
25
Single piece created by downloader
50

Figure 4.35 – a downloader creates a single piece from two half-empty pieces

A leecher would contact the Tracker after receiving some portion of a query (discussed later in this section) to begin advertising. If a downloader contacted a leecher to ask how many pieces it had received, the leecher could then return how many full pieces of data it had received. However, it would be impossible for the leecher to state how many pieces it was going to receive in total. This is not a problem in BitTorrent, because the piece structure of a file does not change. If the downloader does not know how many full pieces' worth of data it is going to receive, one of the advantages of the Wigan system is lost – the ability for peers to be aware of when they have received the complete results set. The downloader would either have to poll the leecher – i.e. ask it at regular intervals if it has received another complete piece – or leave its details with the leecher so that the leecher can forward on a new piece as soon as it receives one.

Concurrent piece requests would be even more problematic because multiple peers requesting the same query could receive the results in a different order. Consider two peers Peer X and Peer Y, both of which are requesting the same query, Q. For the sake of simplicity, assume there are only three uploaders, advertising a query, Q1 of which Q is a subset. One uploader, UF has a particularly fast upload rate, the second, US has a particularly slow upload rate and the third, UM has an upload rate in between that of UF and US. Peers X and Y send three concurrent piece requests, one to each uploader. Peer X requests piece 1 from UF, piece 2 from US and piece 3 from UM, whereas Peer Y requests piece 1 from US, piece 2 from UM and piece 3 from UF. This is shown in Figure 4.36.

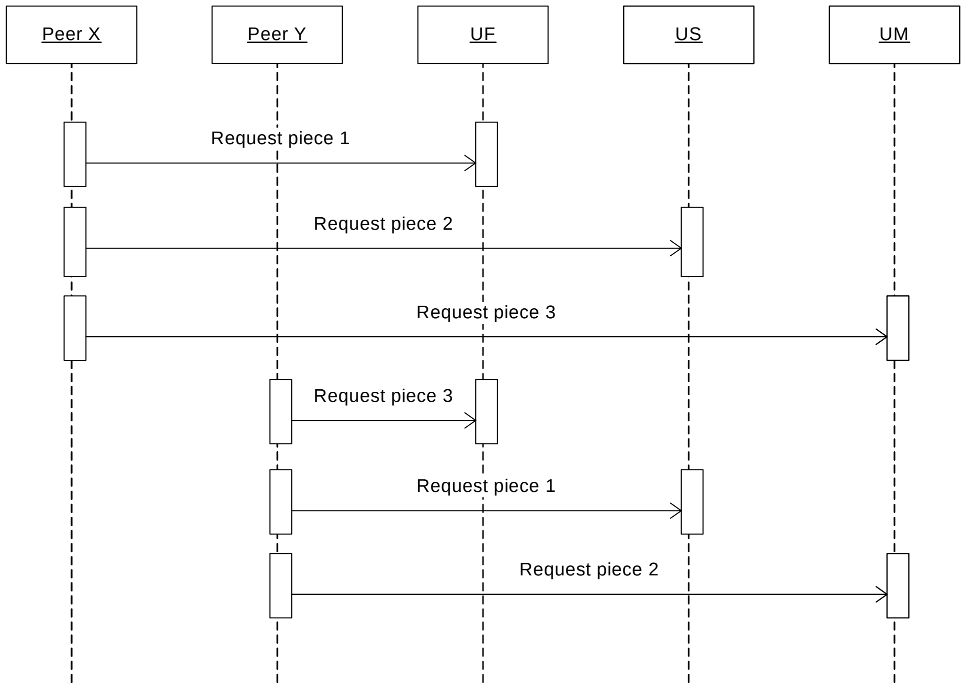


Figure 4.36 – two downloaders send concurrent requests for different pieces to different uploaders

If the peers were permitted to advertise after receiving a single piece, both X and Y would contact the Tracker as soon as they receive a piece of data. In this example, X and Y would receive the responses to the requests sent to UF first. Thus, when advertising a single piece, Peer X has piece 1 and Peer Y has piece 3. When these peers receive the second piece of data (from UM), Peer X has pieces 1 and 3, whereas Peer Y has pieces 2 and 3. Note that Peer Y has received its pieces in reverse order. Thus, when it receives piece 2 from UM, it will have to re-calculate the tuple-to-piece mappings based on the tuple ID. Peer Y is unaware of the tuple-to-piece mappings of the data it will receive, because the uploaders' query, Q1, is not identical to the query, Q, which it is requesting. Hence, it could not predict that it would have to re-calculate the tuple-to-piece mappings when the second piece arrived and also is unaware that it will have to do the same again when it receives piece 1.

Now consider a new downloader, Peer Z which decides to try and obtain data from Peers X and Y. Both X and Y inform Peer Z that they possess two pieces of data, but Peer Z is not aware that the data held by each peer is not the same. Similarly, Peers X and Y are not aware that they will have to restructure the data when they receive their final piece. Peer Z asks Peer X to send its first piece and Peer Y to send its second. Peer Z therefore receives pieces 1 (from X) and 3 (from Y). This is shown in Figure 4.37.

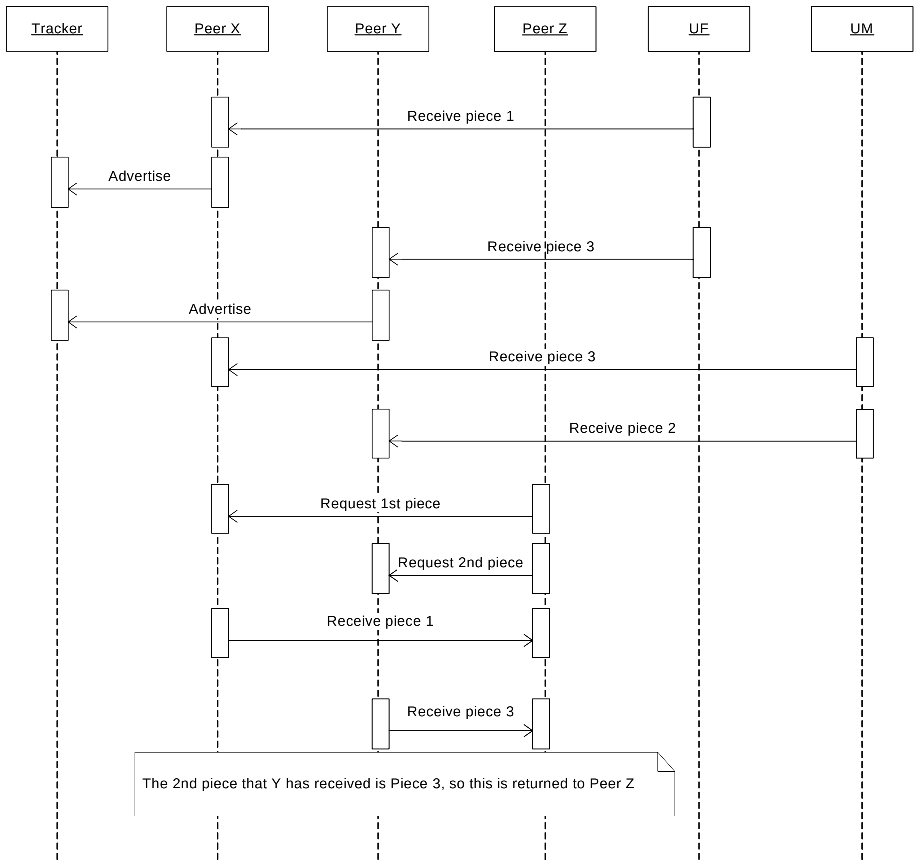


Figure 4.37 – a downloader receives pieces out of numerical order

Eventually, X and Y receive the results from US. They now have all three pieces, which they order based on the tuple ID. Once Peer Z discovers that the peers have received another piece, it sends a request to one of them. Both peers, having re-calculated the tuple-to-piece mappings on receipt of the results from US, would send piece 3 if asked to send their third piece. This means that Z has received one copy of piece 1, two copies of piece 3 and has not received piece 2 at all. This problem is caused by the fact that X and Y cannot predict what results they will receive and then do not receive their data in order; thus, later in the download period could receive data from near the beginning of the query results set. This means they must re-order their data, potentially after it has already been uploaded to other peers.

Another potential problem when downloading from leechers is that the download is in turn dependant on the progress of the leechers' download. If the leecher is choked or snubbed by its uploaders, the downloaders receiving data from that leecher will in turn be unable to receive any more data.

Permitting peers to advertise early also results in more work for the Tracker. The Tracker must in some way distinguish between peers with complete and incomplete results, thus ensuring the downloader does not view the current amount of data received by a leecher as the size of the overall query results. If there are large numbers of uploaders and leechers available to answer a particular query, the Tracker must also have some policy for selecting a collection of adverts to return to the downloader. Does it choose randomly, give priority to leechers or give priority to uploaders? Note that only providing information about leechers causes the risk that the downloader cannot obtain the entire results set.

Another issue concerns how early a peer can start advertising. Given that a downloader may be receiving data from a peer that does not possess the same query it is requesting, a downloader may be unable to predict when it has received a particular proportion – for example half – of the query results. If the downloader is receiving data from the seed and all the tuples matching its query are at the end of a table, the downloader may receive many empty pieces followed by some full pieces. Advertising after receiving half of the pieces (in the seed's piece structure) in this case would not be possible.

However, many of these problems are reduced or even eliminated if the leechers are advertising exactly the same query requested by the downloader. In this case, peers will not need to change the piece structure and can easily identify when they have received a particular proportion of the results. The piece numbered 1 received by a peer will be the same as the piece numbered 1 that the peer will upload. Thus, it is easy for a leecher to inform a downloader of which pieces it possesses, even if these pieces are being requested concurrently, and thus may not arrive in sequential order. This is unlike the scenario illustrated in Figure 4.37, where the leechers were receiving data from uploaders that were advertising a different query and therefore could not predict what the final piece numbers would be until they had received the entire result set.

The potential benefits of early release would be exploited at the time in the download period when a small number of peers have downloaded the results. Normally, these peers would then be bombarded with requests from downloaders until other peers receive the complete query results, but in this scenario, the load would be spread amongst other peers which, whilst not possessing the complete results, can upload the pieces they have already received.

To investigate this further, the simulator was altered. Peers receiving data from others with exactly the same query as theirs began advertising when they had received 25%, 50% and 75% of the query results and also after receiving only a single piece of data. All peers submitted the same query, which had a result set that fitted onto 40 pieces. A 40 piece query was used because this was one of the sizes that had been used when testing concurrent piece requests (and thus the results were available for both concurrent and sequential requests for the case where peers only advertise after receiving the entire query results) and also because it was the size used to examine the impact of changing the piece size, thus offering a comparison with that approach. Furthermore, it was very easy to calculate when one piece, 25%, 50% and 75% of the results had been received. A single-piece query would not suffice here because it would not be possible to advertise early. Concurrent piece requests were used in the experiments as these had already been shown to offer a considerable speed-up over sequential requests.

The Tracker was configured to look for adverts which were identical to the downloader's query and that were placed by peers with a currently incomplete results set. This required an additional field in the Tracker's database to distinguish whether a peer had a complete results set or not. Given that a set of leechers may not be able to return the entire query, the Tracker gave priority to uploaders over leechers. However, if there were less than 50 uploaders available, leechers were added to a new query group and included in the results returned by the Tracker to the downloader. The use of a separate query group enabled the downloaders to distinguish which peers had received the complete results and which had not. They would therefore be aware of the complete size of the query results. However, when requesting pieces, downloaders would not distinguish between uploaders and leechers, because the data received from both would be the same. The average response times are shown in Figure 4.38.

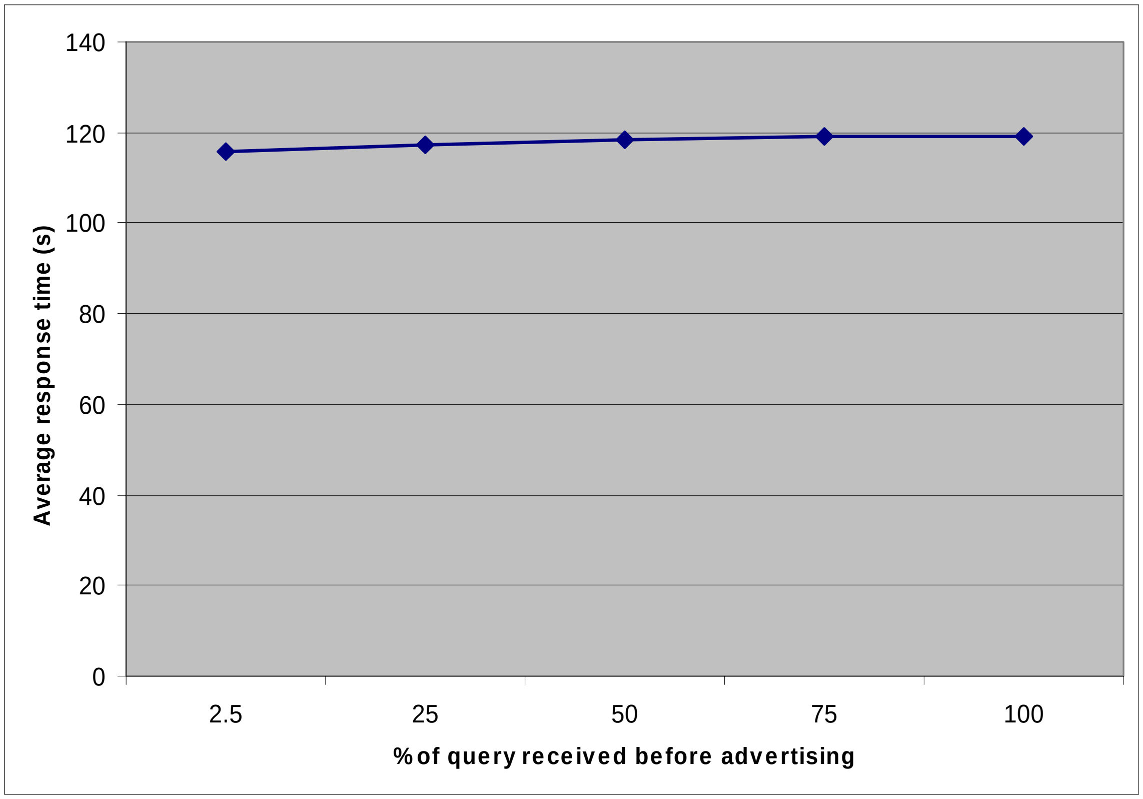


Figure 4.38 – average response times for early advertising of data

Firstly, it can be observed that the response times increase slightly as the peers wait longer before they begin to advertise. This is because, when peers advertise after receiving the entire query, there is a period, once a few uploaders have come online, that these uploaders are being swamped with requests from other downloaders. However, if peers can advertise early, there is more choice available to downloaders, which do not have to rely on a small number of uploaders.

Secondly, it can also be observed that the speed-up factor is not very high. Advertising after 2.5% of the query (i.e. one piece) has been received produces a decrease in average response time of approximately four seconds compared to not advertising until all of the results have arrived, which is a speed-up of less than 4%. Increasing the piece size had a far greater impact on the response time. This is because the impact of increasing the piece size could be exploited from the very beginning of the download period. Downloaders joining at this point could use the larger piece size to obtain more data from the seed at once, which meant that downloaders did not connect to the seed for as long and thus were able to advertise earlier. In turn, this provided more uploaders for future downloaders to use. However, the early release of data can only be exploited when a downloader receives data from a leecher in the process of receiving the same query requested by the downloader. The seed has a different query to that requested by the early downloaders and hence those downloaders joining at the start of the download period cannot advertise early. Early release of data does not overcome the problem of the seed receiving more requests than it is able to process.

4.14 CACHE-WARMING

In the previous experiments, slow response times in the Wigan system have been attributed to the opening part of the download period when only the seed is available to answer queries. During this time, the seed is receiving more queries than it is able to process as shown earlier in Figure 4.13. This could be described as a cold cache scenario, i.e. there are no adverts at the Tracker other than the seed's which can resolve a query. This set of experiments will introduce a warm cache scenario, where certain peers will advertise certain queries before the continual query stream begins. This offers the opportunity to explore how Wigan behaves once the initial backlog of queries has been cleared. The continual query stream of two queries per second caused the seed some problems, for example the high response times experienced by peers joining near the start of the download period, shown in Figure 4.11; however, with a warm-cache it could be feasible to have a continual query stream with more queries per second because the system should have some additional capacity. Each peer can upload to four downloaders until the optimistic unchoke occurs after 30 seconds after which it can upload to five as described in Chapter 2, Section 2.4.

One area of interest with Wigan is how it compares to a Client-Server system over a range of workloads. To make a comparison, a second simulator – again using SimJava – was created to model a Client-Server database system. The simulator in this set of experiments featured a Client-Server database which could process 100 I/O's per second. This would allow a comparison to discover any areas where Wigan could offer a performance advantage and also any areas where it would not.

The first set of experiments were to verify that that cache-warming could overcome the queuing at the seed. There were 2,600 peers, one was the seed and the rest all requested exactly the same small query. Five cache-warmers were used. This is not too small a number of cache-warmers. If the number of warmers was too small, there will still be insufficient capacity to cope with the continual stream of queries. Equally, if there were too many warmers, there is a risk that the warmers themselves take too long to obtain their results and some are still queuing at the seed when the continual query stream begins. Therefore, the first five peers started 90 seconds earlier than the rest to allow sufficient time for them to download before the continual query stream of 310 queries per minute began. This figure was chosen as it was a considerable increase over the previous experiments. In the Client-Server system there was one server and 2,599 clients. To ensure there was a fair comparison, the first five clients in the Client-Server system submitted their queries 90 seconds before the others. The average response times are shown in Figure 4.39:

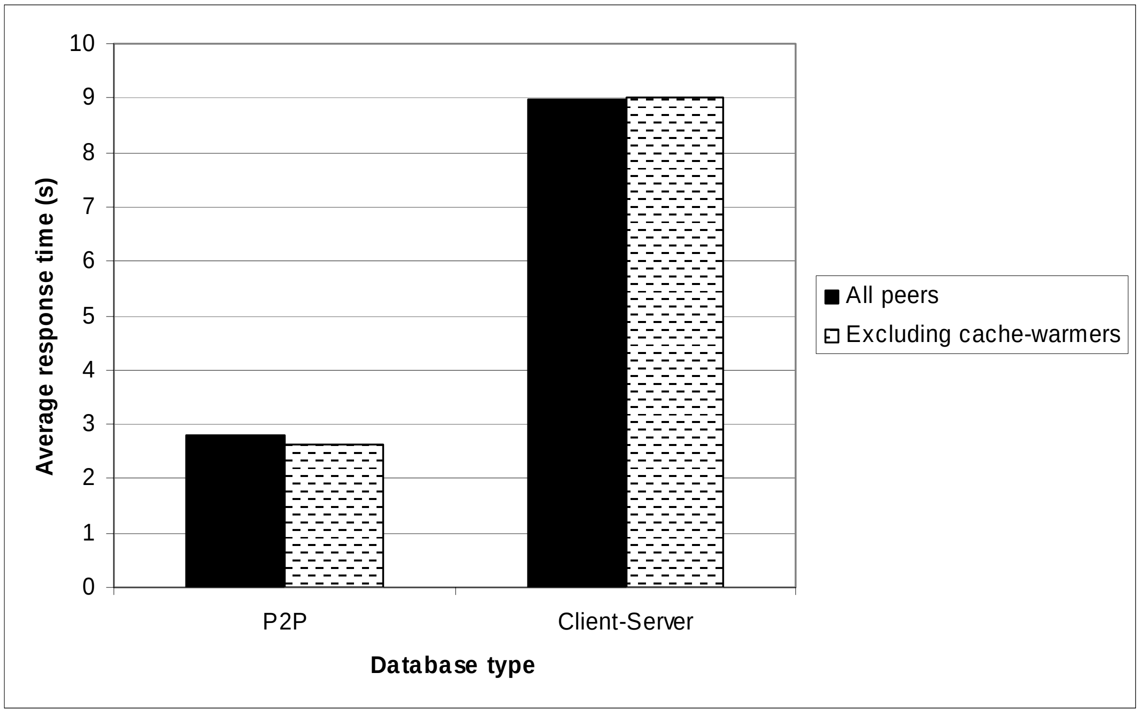


Figure 4.39 – Average response times for the initial experiments with cache-warming peers

It can be seen that the P2P system is the faster. This is because the Client-Server system receives more queries than it is able to process in a second. Some clients – particularly those that submit their queries early when the system is quiet – will receive their results quickly, but others' requests will be queued at the server. Note that because there are a finite number of clients submitting queries, and thus the stream of queries does end, all clients will receive a response, though as noted above, this response could be delayed. It is interesting to note that the clients which experience the fastest response times in the Client-Server system will, as noted above, be those who submit their queries early. However, in Wigan, those that experience a faster response time will be those that have submitted queries later in the download period when more advertisements are available. When examining the average response time, excluding the cache-warmers, this was lower in Wigan than the overall average response time. This is because the cache-warmers take longer as they have to obtain their data from the seed which must scan the entire table, whereas other peers can obtain their results from a peer advertising exactly the same query. Conversely, in the Client-Server system, there is little difference between the performance of the first five clients and the remainder. This is because all clients obtain data from the single server regardless of their start time.

Having seen there was the potential for some performance advantage over Client-Server using cache-warming peers, further experiments were undertaken. The next stage was to examine cache-warming for multiple queries. In this experiment, there were five different single-piece queries. Three ranged over the Supplier table, and the remaining two involved a join between the Supplier and Nation tables. These experiments used a slightly busier system, with 20 more queries per minute, which offered the opportunity to see how the system scaled over the initial experiments. In Wigan, these latter queries were submitted using the cached join algorithm described in Chapter 3, Section 3.5. The same set of queries was used for both the P2P and the Client-Server experiments. In Wigan, 2,600 peers were used. One of these was the seed and the first five warmed the cache. These five peers each submitted one of the five queries. This time, there was a two minute gap to allow these five peers more than enough time to download and begin advertising. This allowed for the fact that the fifth peer would be unable to begin downloading until after the seed performed an optimistic unchoke. If a cache-warmer had not completed its download before the continual query stream began, there would be one of the five queries unavailable from any other peer except the seed. The result would be that the seed would start to receive many requests, potentially more than it could handle, until a peer completed the download. The remaining peers picked one of the five queries at random and submitted queries at a rate of six queries per second.

In the Client-Server system, there was one server and 2,599 clients. To make a fair comparison with Wigan, the first five peers each submitted one of the five queries and there was a two minute gap before the remaining peers began submitting one of the five queries chosen at random. The rate was kept at six queries per second.

The response times are shown in Figure 4.40. Again, two sets of response times are shown for each system in Figure 4.40 – including and excluding the cache-warmers.

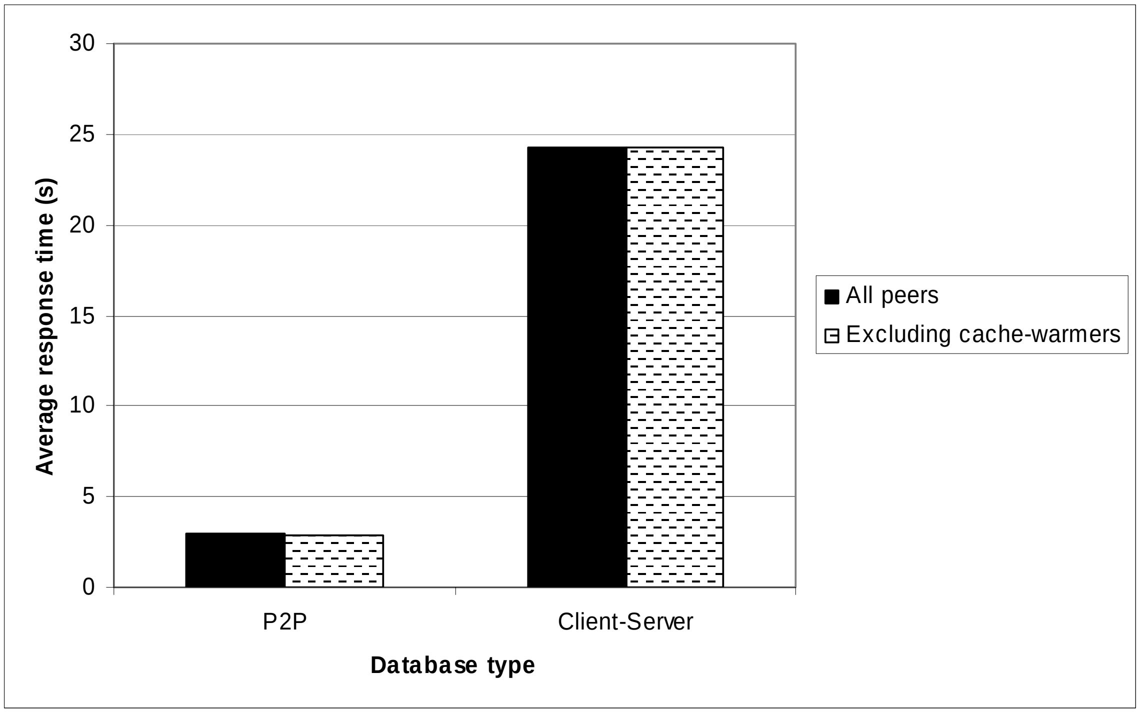


Figure 4.40 – Average response times for the warm-cache experiments with five queries

The percentage of peers receiving results in less than five seconds is shown in Table 4.5.

Table 4.5 – the percentage of peers receiving results in < 5 seconds

P2P	**Client-Server**
99.47	9.27

The increase in the number of queries per minute has resulted in an increase in response times for both systems as can be seen by comparing Figure 4.39 with Figure 4.40. However, in the Client-Server system, this increase is more than double, whereas in Wigan, the increase is very slight. This is because all queries in the Client-Server system are routed to the server. The system was already busy in the previous set of experiments, so these extra peers will add to the delays and the queue will build up. In contrast, as the Wigan network grows so does its capacity to handle new downloaders. The cache-warming peers are able to handle the first rush of downloaders when the continual query stream begins and once these start advertising, they can also provide queries to future downloaders.

It is likely that in a real-life workload, not all peers will be submitting one of five queries. The next set of experiments was a repeat of the previous, but with one difference – one in every 100 peers submitted a random query on the Supplier table. The random queries could be for any number of tuples and there was no guarantee of how these random queries would overlap with each other or with any of the five repeating queries if at all. In this set of experiments, sequential piece requests were used instead of concurrent, because the majority of queries were single-piece and also because the random queries would usually have to be downloaded from only the seed unless it was a proper subset of one of the existing queries. The results are shown in Figure 4.41:

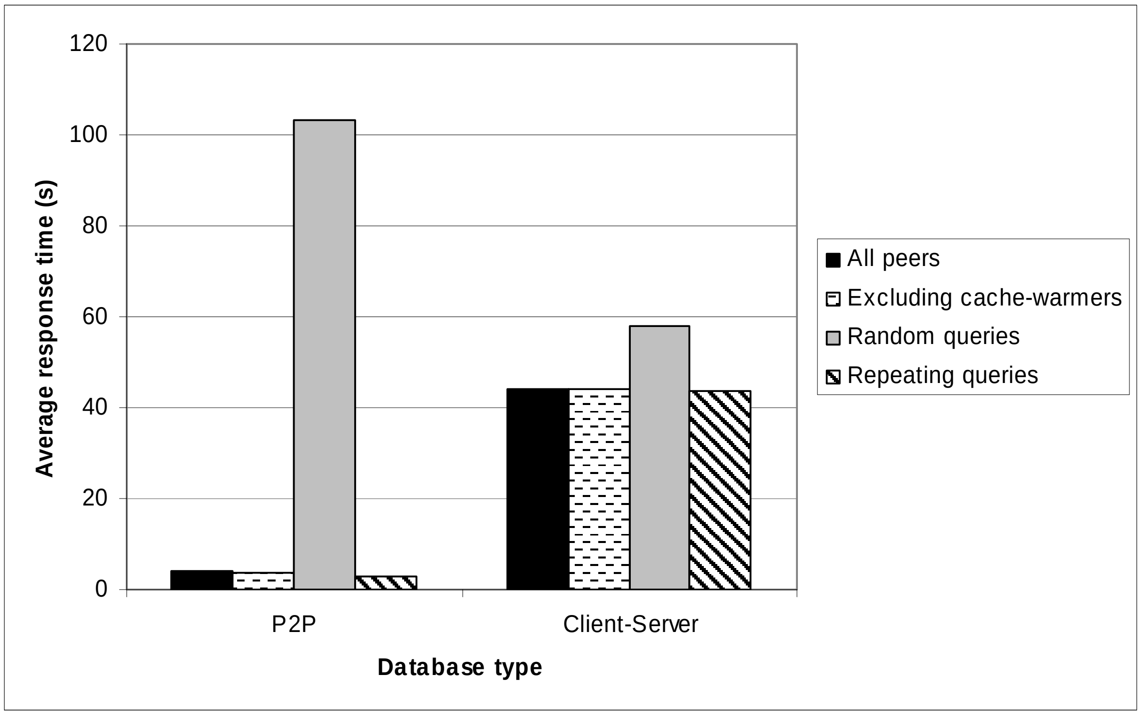


Figure 4.41 – Average response times for the warm-cache experiments with five queries and random queries

The overall average response times show that P2P continues to be faster. It can also be observed again that the average response time in Wigan excluding the cache-warming peers is lower than the average response time with the cache-warmers included. This is because, as noted earlier, Wigan peers joining later in the download period experience a faster response time due to an increased number of adverts available.

Peers submitting random queries experience slower response times than those submitting the repeating queries in both systems. In the Client-Server system this is because the random queries could be larger than the others and therefore take longer to arrive from the server. However, Wigan is considerably slower than the Client-Server system for peers with random queries because there will be more pieces to request and also because these queries will be going to the seed, a queue will develop there.

4.15 THE TRACKER

All of experiments introduced in this chapter have examined the performance experienced by the peers and also, in places, examined the load experienced by the seed. This section will focus on the Tracker and its performance.

In particular, it is important that the Tracker does not become a bottleneck in a similar manner to the seed, or the server in a Client-Server database system. Previously, in Section 4.4, the time taken for a peer to contact the Tracker and receive a response was noted. This time included the processing time at the Tracker and the time taken for the messages to pass between Tracker and downloader. This section will investigate the processing time in more detail.

In order to examine the Tracker, some analysis was undertaken to determine exactly what the Tracker was doing during the course of the download period, in particular what messages it was receiving, i.e. requests for peers from new or timed-out downloaders, completion messages and disconnection messages. This was achieved by adding a new table to the Tracker's database into which the Tracker reported what messages it received at what time and permitted a collection of queries to be generated which modelled those received by the Tracker during the course of the download period.

The Tracker code was copied to run outside the simulator. This abstracted the issue of the Tracker away from the overall Wigan system and thus permitted thorough testing of this code without having to wait for large numbers of peers to complete downloading. The queries sent to the Tracker were those collected during the analysis described in the previous paragraph. Timing code was added and initial tests showed that the Tracker was processing queries in a matter of milliseconds.

In the previous test, the queries were despatched at the same rate as those used in the simulation runs. However, the test program was then modified so that all the requests were fired at the Tracker consecutively without a break, which was equivalent to all downloaders starting the download at exactly the same time. This was a much higher rate of requests than the Tracker experienced during the previous experiments presented in this chapter, where the maximum number of queries per second received was six, in the cache-warming experiments, although two queries per second has been more common. The Tracker's statistical database was also modified so that a timestamp was inserted as each request was executed. This was to permit the observation of the number of queries per second that the Tracker database was able to process.

Once all of the queries had been sent to the Tracker database, it was discovered that the database was processing between 300 and 400 queries per second. This is far higher (by a factor of 50 or more) than the query rate received by the Tracker during the other experiments presented in this Chapter. The experiment was repeated with more requests (the equivalent of 2,000 peers, a higher number than was used in any of the experiments in this Chapter) and the database continued to process between 300 and 400 queries per second.

This analysis indicated that the figure quoted in Section 4.4, for contacting the Tracker and receiving a response was composed almost entirely of the network cost for sending a message. At the level of usage in the experiments in this chapter, the Tracker is not a bottleneck and is capable of supporting all downloaders and their requests. If more than 300 queries per second is needed for large systems, replicated Trackers could be deployed.

4.16 SUMMARY

This chapter has introduced experiments and analysis on the Wigan architecture and has explored some variations to the basic architecture, for example introducing concurrent piece requests. Wigan offered a performance advantage over the Client-Server database once there were uploaders available to answer queries and the Client-Server database was receiving more queries per second than it was able to process. Whilst the Tracker was not a bottleneck in these experiments, a major factor affecting performance in Wigan was the opening part of the download period when all queries are routed to the seed. In this phase, it is likely that some peers will be unable to access the seed at the first attempt and will have to wait until the seed has a free slot, or until they are unchoked by the seed or until a timeout occurs. This has a major effect on the initial performance, but the system does recover and perform well afterwards, as illustrated by the cache-warming experiments in Section 4.14. This is less of a problem in BitTorrent because a BitTorrent peer is able to start uploading as soon as it has received a single piece of the file, however, this is not always possible in Wigan, in particular, it is not possible when downloading from the seed. The performance of Wigan can be improved by allowing the seed to upload to more peers concurrently and if the query contains more than one piece, requesting these pieces concurrently from different uploading peers. Further improvements can be made by increasing the number of concurrent downloaders at the seed, by replicating the seed and also by changing the piece size.

The results also showed that multi-table queries are possible in Wigan; in the general case, either of the two algorithms presented in this chapter will work equally well, although each has its own best and worst cases.

A summary of the overall results will be presented in Chapter 5 along with some recommendations for future work.

5 CONCLUSIONS AND FUTURE WORK

5.1 INTRODUCTION

This chapter concludes this thesis with a summary of the Wigan system architecture and the evaluation of the Wigan system. Some suggestions for future work are discussed and the main contributions from this research are highlighted.

5.2 CONCLUSIONS

This thesis began by examining the problem of a database user submitting a query to a busy database server and experiencing an unacceptable response time because the workload forced the server to queue query requests. The proposed solution to this problem was the Wigan Peer-to-Peer Database, derived from the BitTorrent file-sharing protocol, and the aim of this thesis was to design, implement and evaluate the P2P RDBMS.

Unlike almost all of the existing P2P database systems discussed in Chapter 2, Wigan was not devised as a means of federating separate databases with different schemas, managed by different organisations but instead is designed to take a single relational database and make it more scalable by distributing it in a P2P manner, thus combining techniques used in the fields of database systems and P2P computing.

BitTorrent algorithms were used as the starting point for designing the Wigan algorithms, but the differences highlighted in Chapter 1 between a file-sharing system and a database meant that most BitTorrent algorithms could not be copied directly to Wigan. Examples of such differences included the ability for a single query to be answered by many different adverts because queries may be total subsets of each other, the fact that the piece structure may change in between the downloading and the uploading phases and the occurrence of concepts such as joins and aggregations which do not occur in file-sharing systems. However, it proved possible to successfully design new algorithms for the database environment as discussed in Chapter 3.

The algorithms presented were implemented using a simulator to enable evaluation to take place. It was discovered that Wigan could indeed produce faster response times than a Client-Server database in some circumstances. Once the cache was warm and there were uploaders available to answer downloaders' queries, Wigan peers began to experience a faster response time than Client-Server clients. In Wigan, this results in a high level of capacity because every single uploading peer can serve multiple concurrent downloaders, thus there is the potential to cope with a high of demand for popular queries. In contrast, in Client-Server systems, only the server can answer queries

In addition, it was shown that concurrent piece requests (Section 4.10) can reduce the response time for large queries, again if there are multiple uploading peers available, because they permit downloaders to obtain multiple pieces concurrently from a set of uploaders.

However, as noted above, this improvement in performance occurred when the cache was warm, i.e. there were uploaders available to answer queries and downloaders were not forced to rely on the seed. One factor limiting the performance of Wigan was the time shortly after the release of the data when all peers had to query the seed. During this time, backlogs at the seed had an impact on the average response time of a query. Therefore, further experiments were undertaken to evaluate possible solutions to this problem.

Greater parallelisation of the seed download (Section 4.8) improves performance at the seed by allowing the seed to upload to more peers concurrently, thus delaying the point at which the seed becomes too busy to accommodate new downloaders. During this delay, more peers are able to receive their data and start uploading, thus providing a potential data source to future downloaders.

Changing the piece size (Section 4.11) is an alternative means of improving performance. Large and small piece sizes do have their respective advantages and disadvantages. If the piece size is too small, more requests are needed and the overhead of these request messages is high. Larger pieces mean that downloaders send fewer requests to the seed at the start of the download period and thus are not connected for as long. However, the piece size must not be too large because large messages take longer to travel across the network. Also, if the piece size is kept below the maximum (one piece per table), then those peers which do not have to contact the seed can still exploit the advantages of concurrent piece requests.

Replicating the seed (Section 4.12) allows downloaders to exploit the advantages of concurrent piece requests from the very beginning of the download period. This again permits peers joining at the beginning of the download period to receive their results faster, which in turn results in more uploaders becoming available sooner.

Unlike a Client-Server system, an increase in the number of peers will lead to a reduction in the query response times, providing that all queries do not have to be routed to the seed. This is because more peers offer more choice; the exact opposite of a Client-Server database where more clients can lead to bigger queues at the server. Although more peers does mean more load, the scalability of the Wigan system means that the system is able to cope with this. An example can be seen by comparing Figure 4.39 and Figure 4.40, where the load in the system increased and Wigan's response time scaled very well, especially when compared to the Client-Server database.

Results obtained in Chapter 4 suggest that a P2P database server based on BitTorrent can outperform Client-Server in certain circumstances. However, whilst these investigations have yielded some interesting and promising results, it has not been possible to explore all potential areas of interest within the timescale of this research. There are still many interesting and open problems in this area, which will be explored in Section 5.3, whilst Section 5.4 will conclude this thesis.

5.3 FUTURE WORK

This section will investigate possibilities for future research into the Wigan P2P database system. Certain ideas presented in this section will build upon suggestions made in previous chapters.

5.3.1 UPDATES

Section 1.3 introduced the differences in requirements between a file-sharing system and a P2P database and noted that one major difference concerned the issue of data updates.

Database data may be updated by a user or an administrator, though there are some database systems where only an administrator would update data, for example an e-Science database like the SkyServer [12]. Database updates may involve inserting, deleting or amending tuples in the database tables. Referential integrity rules in a relational database mean that altering data in one table may involve cascading that change to other tables. Chapters 3 and 4 in this thesis have examined the issues concerning query processing in the Wigan system and have assumed that the data is static. This section will discuss the issues associated with data updates in the Wigan system and suggest approaches to handling them.

An update is normally enclosed in a transaction which, in database systems, is a unit of work that "must be executed atomically and in isolation" [1]. A transaction may consist of queries, updates or both queries and updates. An example of the latter would be a query on a bank's database checking that the amount of money in an account is greater than the amount that the customer wishes to withdraw, and if it is, an update occurs – reducing the account total accordingly. The characteristics of the Wigan system mean that existing database server transaction solutions alone are insufficient; updates present more of a challenge in a P2P system as will now be discussed.

Firstly, there is the question of which peers are permitted to perform updates. If clients are permitted to perform updates, the problem exists that these clients are not necessarily aware of which other clients have already downloaded the same data; which means that there is a risk that multiple clients may perform conflicting updates at the same time. A possible solution to this problem would be for the clients to send update requests to the seed, which can then perform the update operation. If the seed receives update messages in this manner, its owner will be aware of exactly what updates the clients wish to make to the data and, because only the seed is actually performing the updates, conflicts would not occur because the updates could be enclosed in a transaction. However, the time taken for an update to complete may increase because each peer would have to contact the seed with its update message and then wait for the seed to actually perform the update. In addition, by adding further tasks to the seed's workload, there is the potential for the seed to become a bottleneck, particularly if, as discussed in Chapter 4, it is also busy uploading data to downloading peers.

Note that in the scenario described earlier in this section where only an administrator is permitted to perform updates, it can again be assumed that all updates are performed by the seed, because it is the original source of the data. Given that there is only one peer – the seed – that is allowed to perform any updates, the problems of conflicting updates occurring concurrently would not occur, assuming they were done in transactions.

In addition to the actual update process, there are also many potential consistency problems which could arise following a data update. Firstly, there may be some adverts placed by uploading peers at the Tracker which become out of date because those uploaders received the data before the update occurred. Furthermore, there may be downloaders receiving data from other uploaders whilst the seed is performing an update. These downloaders may receive data that is out of date and then advertise this data to other peers on completion of their download. If there are many peers advertising the same query, there is a risk that some of these peers possess the original results (i.e. results obtained before the update occurred) and some possess the updated results. This could lead to many consistency problems, particularly if a downloading peer was obtaining data from a collection of peers, some of which had the updated data and some of which did not. If the update involved a deletion, there would be the risk that the downloading peer received some or all of the deleted rows, while if the update involved an insertion, there is the risk that the downloader does not receive some or all of the new rows and finally, if the update involved amending some tuples, there is the risk that the downloader receives the old values, or a mix of old and new values. This is a problem because not only does the downloading peer receive incorrect information, this incorrect information could then spread around the Wigan system when the peer starts advertising at the Tracker. There needs to be a mechanism in place to prevent downloaders obtaining inconsistent data from before and after an update has occurred.

Another potential problem, which could arise if data was being inserted or deleted, is a change in the piece structure. For example, inserting rows may increase the number of pieces whilst deleting rows may decrease the number of pieces. This would only be a problem for peers downloading data from the seed whilst it was performing an update. Even if the seed enclosed the update in a transaction to prevent peers downloading data whilst the update itself was occurring, inconsistencies could still arise afterwards. For example, if the update involved deleting a tuple in the first piece of a table and the downloader had already received the first piece before the update, the downloader would be unaware that the results it had received were now out of date.

It is the issue of awareness that is a big challenge regarding updates because there may be a large number of peers, both downloaders and uploaders, around the Wigan system that are unaware of what updates the seed is performing and when. One possible solution would be for the Tracker to forward the SQL "UPDATE" statement executed by the seed to all relevant uploaders. However, as with all P2P systems, there is no guarantee that the notification reaches all peers, e.g. because of peer or network failure. Therefore, the update would need to be performed as one big distributed transaction. This could, however, be problematic because, in order for the transaction to commit, i.e. complete successfully, every peer would have to perform the update. Given that, as stated above, there may be peers that do not receive the notification message, there is a strong chance that at least one peer would not perform the update and hence the transaction could not be committed.

The Tracker is aware of exactly which peers are downloading and uploading which queries, whereas the seed is only aware of which peers it is currently uploading to. Therefore, all possible approaches regarding updates would have to involve the Tracker. Algorithms which could be investigated and evaluated include:

A simple invalidation approach, where the Tracker invalidates, i.e. deletes, all adverts for out of date data so that only the most recent data is available to downloaders. This could be implemented at the granularity of database, table or sub-table. These options are progressively more complex to implement, but would result in fewer adverts being invalidated. For example, invalidating all adverts for a particular database after an update occurs would be straightforward, but could result in a large number of adverts being invalidated, some of which may be unaffected by the update. Regardless of the invalidation strategy, there is the problem that, when a peer contacts the Tracker with a new advert, the data it is advertising would be out of date if the update occurred whilst that peer was downloading. Such adverts would have to be detected and withheld by the Tracker.

Alternatively, there are more complex algorithms, for example involving version numbering, which could allow different versions of the database, dating from different time periods, to co-exist. A generic overview of such an algorithm is as follows. The Tracker would need to store the database version number for all queries that are being advertised. When the seed initially advertises the data at the Tracker, this is Version 1. When an update occurs, the version number is incremented and the Tracker would update its records to reflect the fact that the seed now has a new version of the database. Note that the seed's owner may decide to keep previous versions of the database, in case a downloader wishes to acquire data from an earlier point in time or they may alternatively decide to keep only the most recent version. Whichever option is chosen, the seed would send a message to the Tracker informing it that an update had occurred and requesting the Tracker update its database accordingly.

If a peer downloads its data from a collection of uploaders which are all advertising the same version of the database, the query results will be consistent, which means that the advert placed by that peer will also be consistent. When an update occurs, this advert will still be consistent; it will just reflect a previous version of the database. Inconsistencies could only arise if a downloader attempted to obtain pieces concurrently from different uploaders advertising different versions, so this must be prevented. One means of achieving this would be for the Tracker to ensure that each query group returned to downloading peers (Chapter 3, Section 3.3.3) contains only uploaders that are advertising identical versions of the database.

As noted above, a version numbering approach will allow different versions of the same database to co-exist. Users may be permitted to include version numbers in their queries, for example a downloader may indicate that it only wants the Tracker to inform it of uploaders advertising the most recent version of the database. A similar approach is used in the dynamic caching system MTCache [43] where, as discussed in Chapter 2, Section 2.6, it is possible for users to specify constraints as to how out of date their data can be.

5.3.2 PARTIAL SUBSETS

Chapter 3 introduced the concept of partial subsets when describing the matching process between queries and adverts which occurs at the Tracker. A partial subset occurs when the advert can resolve some of, but not the entire query (Figure 5.1).

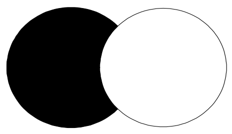


Figure 5.1 – A query (white circle) which is a partial subset of an advert (black circle)

In SQL, an example would be:

Query: "`SELECT item FROM parts WHERE cost <= 10`"

Advert: "`SELECT item FROM parts WHERE cost <= 5`"

One extension to the existing Wigan system would be for the Tracker to return partial subsets to the downloading peer. When examining which uploaders can provide each table, instead of rejecting peers that do not have all the correct tables, columns or rows, those which could resolve part of the query would be included in the result set. In the same way in which the Tracker currently arranges the peer list in groups, something similar must be done to ensure the downloader possesses enough information about where it can obtain the data. A downloader must be told by the Tracker exactly which uploaders possess which parts of the query. The Tracker must not provide one partial subset if the remainder of the query cannot be obtained from the anywhere else.

Partial subsets resemble the uncached join algorithm introduced in Chapter 3, which uses multiple single-table queries to obtain the results of a multi-table query. One drawback of the uncached join algorithm is that unrequired data could be returned to the downloader to be joined. However, this problem will not occur with partial subsets unless of course the downloader is using a collection of partial subsets for each of the single-table queries in the uncached join algorithm. Consider this query and advert:

Query: "`SELECT item FROM parts WHERE cost >= 2 AND cost <= 10`"

Advert: "`SELECT item FROM parts WHERE cost <= 5`"

Another advert would clearly be required which contains details of items costing more than £5. The advert shown here also contains some data which is not required by the downloader – those items costing less than £2. However, when the uploader that placed the advert received and executed the query, the conditions in the "WHERE" clause would prevent information about all items costing less than £2 being returned to the downloader.

It may be viewed as unnecessarily complex to obtain a query as a collection of partial subsets if there is at least one uploader available that can provide the entire query. However, one scenario where partial subsets might be advantageous would be in the case of a query answerable only by the seed. Chapter 4 highlighted how busy the seed could become during the download period. If a downloader discovered that their query was answerable in its entirety by the seed or as a collection of partial subsets from other peers, the downloader may decide to choose the latter option. Not only would this reduce the load on the seed, but the downloader may experience a faster response time if the other peers were not as busy as the seed. Another scenario in which partial subsets could be of use is if the query has a potentially large result set and there are only a small number of possible uploaders which means that the advantages of concurrent piece requests cannot be exploited fully. In this case, concurrently obtaining parts of the data from partial subsets could be faster. The use of partial subsets also allows for the scenario envisaged by the creator of BitTorrent where the original seed could be removed but downloads continue and could be useful if the seed suffered a fault. However, this would assume that the entire database could be constructed by combining the adverts, in a similar manner to the universal relation used by the query difference operator [55] described in Chapter 2, Section 2.7. This may not be possible as there may be some parts of the database which are not available on other peers. In addition, even if the entire database was available by combining adverts held by other peers, there may be parts of the database held by only a single peer and therefore the availability of that data would depend on the availability of that peer. Whilst the same can be said of data stored at the seed, the seed is a database server with an appropriate backup system, whereas a peer is a user's PC that is not likely to have this.

Another potential use of partial subsets would occur if a more advanced security system existed in the Wigan network (see Section 5.3.8). If this was the case, peers might use security policies to decide where to download data and thus may decide to use a collection of partial subsets of trusted peers instead of receiving the complete results from one other peer.

When examining the issue of partial subsets, one point of importance is how involved the Tracker should become in the download process. There will always be at least one peer that can resolve any query in its entirety. However, if there is only one suitable uploader (the seed), should the Tracker split up the query automatically into various parts and attempt to find additional peers that could resolve those parts? Alternatively, should it just return the one uploader and allow the downloader to resubmit the query if they wish? Note that this would involve the downloader's Wigan software making a dynamic decision to resubmit the query instead of accepting the peer list supplied by the Tracker and commencing the download. Clearly, this latter approach could inconvenience the downloader because it involves the downloader's software having to perform more tasks and re-contact the Tracker. A better option would be for the Tracker to attempt to find peers that could answer the query in its entirety, and if there was less than the maximum number of permitted uploaders on the peer list (50 in the implementation used in this thesis), fill the list with any collections of partial subsets. Clearly, this would require a more advanced Tracker, which would need to iterate through all available adverts with the correct columns to find out if any of these adverts could answer part of the query.

There is also the choice about how the Tracker groups the adverts. Consider the query "`SELECT item FROM parts WHERE cost >= 50 AND cost <= 100`". Assume that a downloader submits this query and that there are currently five advertisements at the Tracker which could answer all or part of the query:

1) `SELECT item FROM parts WHERE cost >= 75 AND cost <= 100`

2) `SELECT item FROM parts WHERE cost < 75`

3) `SELECT item FROM parts WHERE cost > 50 AND cost <= 100`

4) `SELECT item FROM parts WHERE cost <= 50`

5) `SELECT * FROM parts` (The seed).

The results could be obtained from combining the results of Adverts 1 and 2, by combining the results of Advert 3 and Advert 4 (to find the items which cost £50 exactly) or by using Advert 5. There are three possible ways of grouping the adverts in the peer list, each demonstrating a deeper level of involvement by the Tracker. Firstly, it could simply return a list of peers grouped by advert, stating what each group of peers was advertising. In this case, the downloader's Wigan software would calculate which combinations of adverts would provide the entire query results and pick an option to execute. Alternatively, the Tracker could return adverts in a different grouping, so that each grouping would return the entire query. In the example here, the first group would contain all peers advertising Advert 5, the second would contain all those advertising Adverts 1 and 2 and the third would contain all those peers advertising Adverts 3 and 4. This also gives the downloader a choice of how to obtain the query, but saves the downloader some time because it does not have to work out how to combine the groups in order to obtain the results; instead it just selects its preferred option and begins sending handshakes. Finally, the Tracker could select which of the three groupings was the best option and only return that information to the downloader. For example, if the Tracker believed the seed was not very busy, it could only inform the downloader about Advert 5. This saves even more time at the downloader, but it does result in the Tracker making an important decision on the downloader's behalf. If the Tracker made an unsuitable decision, the downloader would be unaware of potentially better options. It could be argued that all decisions about how to download the query should be made by the downloader.

The process of choosing a collection of partial subsets is, in effect, introducing costed query plans into the Wigan system, both at the Tracker and at the downloader. Query plans, cost models and optimisation will be discussed in more detail in Section 5.3.4.

Care must also be taken to avoid the Tracker becoming a bottleneck. If the Tracker has to make more decisions, it will take longer to process each request. If the system was very busy, queues may form at the Tracker if it takes too long to process a request, thus increasing overall response time and decreasing throughput.

In conclusion, there may be scenarios where obtaining query results from a collection of partial subsets could be a benefit. However, care must be taken when implementing such an algorithm to prevent the Tracker restricting downloader choice or becoming a system bottleneck.

5.3.3 DYNAMICALLY CHOOSING A JOIN ALGORITHM

Chapter 3, Section 3.5 introduced two different join algorithms and Chapter 4, Section 4.7, highlighted a case where each algorithm performs badly. One useful modification would be to have the system choose automatically which join algorithm to use, thus preventing a user inadvertently using one of the algorithms in its worst case and experiencing a very slow response time. This process would have to involve the Tracker because it is aware of exactly who is advertising what. This could help in the worst case for the cached join. The Tracker's database would indicate if a downloader submitted a multi-table query and there were only single-table queries being advertised and thus the Tracker could split up the downloader's query and find a peer list for each table. This is in effect converting dynamically from the cached to the uncached join.

The worst case for the uncached join occurs when a peer needs to download a large table in its entirety. It would be more complex to dynamically change algorithms in this instance, because the downloader submits several single-table queries and the Tracker could not determine if the downloader was planning to perform a join or not. One option would be for the Tracker to assume, if it receives multiple requests from a single downloader in very quick succession, that the downloader wants to join the results of the queries. The Tracker could then also list any possible options for a combination of the single-table queries.

An alternative would be for the downloader to make a dynamic choice. This would be handled by the downloader's Wigan software, without the human owner of the peer having to make a decision manually. This would prevent the Tracker having to try and predict what the downloader wants to do – after all, it is possible that a downloader will submit several single-table queries at one particular moment in time without planning to join the results. If the list of uploaders provided by the Tracker indicated that a large result set – such as an entire table – would need to be downloaded, the downloader could resubmit a collection of single-table queries. This would, however, require the downloader to submit another request to the Tracker, although this is still likely to be faster than trying to obtain a large table in its entirety.

5.3.4 COST MODELS AND OPTIMISATION

Section 5.3.2 discussed possible ways of combining query results from partial subsets and noted that, if partial subsets were used, there may be occasions where the same data could be obtained from many different combinations of peers and advertisements. Similarly, Section 5.3.3 explained how there may be circumstances where a user would experience an improvement in the response time of a multi-table query by choosing one particular join algorithm. Both of these sections described issues relating to a common problem in database systems – that of query optimisation, i.e. how to ensure that a query executes in an efficient manner and large numbers of unrequired tuples are not either scanned by an uploader or, in the case of the uncached join algorithm, actually returned to a downloader.

Currently in the Wigan system, there are some optimisation decisions that have already been taken during the design phase. One example of such a decision is the use of query groups, giving priority to an exact match, as discussed in Section 3.3.3. This prevents downloaders trying to obtain data from the seed when other possible uploaders are available that have less data to scan and also prevents the seed from remaining a bottleneck whilst other uploaders are not being utilised by downloaders. Even if the seed is not included in a peer list, the query groups will still give priority to those uploaders that have the least amount of data to scan. A similar means of optimisation is used if there are more uploaders available than will fit on a peer list – 50 in the experiments presented in this thesis. In order to reduce the peer list to the maximum permitted size, the Tracker removes uploaders starting from the rear of the list, i.e. the uploaders whose queries are furthest away from an exact match, as illustrated in Chapter 3, Figure 3.14. Stepping back through the query groups and moving to the next-closest match if a downloader gets choked is also an optimisation because it prevents downloaders attempting to obtain data from the seed immediately if there is another alternative.

Indeed, the choking algorithm is another example of optimisation. This makes uploaders give priority to faster downloaders. It prevents slots at the seed being occupied for long periods of time by very slow peers and will also benefit future downloaders. If the first peers to complete are fast, they will in turn be able to upload content quickly to downloaders joining later in the download period, thus increasing the throughput.

However, there are further opportunities to apply query optimisation techniques within the Wigan system. Wigan peers currently pick uploaders at random and do not generate cost models at all. An increase in the number of possible uploaders, and also in the complexity of queries, results in downloaders having more options as to how their query is executed. The examples from Sections 5.3.2 and 5.3.3 illustrated scenarios where optimisation could be used to improve the efficiency of the whole query execution process – e.g. which join algorithm to use – but even in the simple case of selecting uploaders from a query group, a cost model may be used to optimise the peer selection process, i.e. downloaders use some cost model to determine which uploaders to select from the list. An example would be, if a downloader had five uploaders to choose from and five pieces to request, which piece to request from which uploader? If the final piece was smaller than the maximum piece size, and one of the uploaders was known to have a slow upload rate, requesting the last piece from the slow uploader could be the most efficient option because this ensures the slowest uploader does the least work.

Cost models are used in distributed Client-Server databases to help construct an efficient query plan. The classic Client-Server cost model described by Kossmann [80] uses I/O, CPU and network cost to determine a possible query plan. Other cost models are based on response time [81] and quality of service (QoS) [82] metrics. Any of these could be used in a more advanced Wigan system to enable downloaders to decide from where to obtain query results. Over time, as suggested earlier in this chapter, a downloader may become familiar with certain uploaders and re-use those that have provided it with a fast upload rate or a fast query execution time, whilst rejecting those which are very slow or keep failing. New peers joining the Wigan system will naturally be unaware of existing uploaders' record at providing data. However, they could still make an estimate if they are aware of the uploaders' geographical location – if for example, in general a local peer will be able to provide a better response time than one further away. A further extension could see peer statistics, such as average upload speed, summarised for the peers to access.

In conclusion, query plans, cost models and optimisation are wide-ranging areas and, although there are examples of optimisation in the current Wigan system, there is scope to apply these areas to Wigan in the future in order to investigate ways of executing a query in the most efficient manner.

5.3.5 INDEXING

Indexing is a mechanism used in database systems to speed up the process of data retrieval. The aim is to prevent the server iterating through a large table just to find a small number of rows which match a particular query, in the same way that an index to a book prevents a reader having to examine every page to find the information they are looking for. The simulator described in Chapter 4 had no indexing. However, indexing could be used in Wigan to speed up data downloads, particularly at the seed. The seed is the only peer which definitely has to scan an entire table. This could be a time-consuming process, especially if evaluating a query on a very large table, such as the TPC-H Lineitem table. Other peers are not likely to possess as much data at the seed; however it is still possible that there are peers advertising a complete table or a query with a very large result set. Indexing could still be used by these peers to assist them in the uploading process.

An interesting concept involves indexing at the Tracker. Chapter 4 explained how the Tracker maintained a database of adverts, noting the tables, rows, columns and any conditions on those columns. It is important to prevent the Tracker becoming a bottleneck and indexing could also be used in this database to improve performance at the Tracker.

5.3.6 FURTHER INVESTIGATIONS INTO PIECE SIZE

Chapter 4, Section 4.11 introduced experiments which examined how the size of the pieces impacted on the response times experienced by peers in the Wigan system. However, this assumed a fixed piece size which remains unchanged regardless of the query and also regardless of whether a downloader is requesting data from the seed or another uploader. Further investigations could examine whether dynamically changing the piece size based on these factors would be of benefit, e.g. have the seed use one piece size and uploaders another. The seed might have a very large piece size to enable a high throughput of downloaders, only sending a few requests, whereas the uploaders might use smaller piece sizes to allow peers to send concurrent piece requests. The only requirement would be that downloaders receive data concurrently from peers with the same piece structure.

An alternative option would be to keep the piece size constant, but to permit the seed to return multiple pieces. This could be achieved by downloaders sending multiple piece requests simultaneously (i.e. not waiting for a piece to arrive from the seed before sending the next request) or alternatively by amending the piece requests to include multiple piece numbers. Either approach could result in downloaders spending less time occupying a slot at the seed and thus enable them to start advertising earlier.

5.3.7 ADAPTING THE NUMBER OF CONCURRENT DOWNLOADERS

Chapter 4, Section 4.8 introduced the possibility that the seed could upload to more than five downloaders at once and then presented experiments showing the effect of a seed that could upload to 50 downloaders concurrently. One potentially useful extension would be a way of adapting the number of downloaders to which the seed could upload during the download process. This might be a dynamic decision made by the software, or a manual decision taken by the owner of the seed to respond to some change in demand from downloaders. During the start of the download period when the seed is busy, it would be useful if the seed was able to upload data to more downloaders. Once additional peers begin advertising, the load on the seed would reduce if there were large numbers of repeating queries. At this point, the seed may restrict the number of permitted downloaders. This is one decision that could easily be made dynamically by the software – begin turning away more requests after a certain amount of time had passed.

A possible scenario where the number of concurrent downloaders could be restricted would occur if the seed requires some form of maintenance, or the owner of the seed wishes to perform some other operations (not connected with the Wigan database) and does not want large numbers of incoming piece requests slowing the machine down. This is a decision which would be taken manually by the owner.

Another manual decision could be taken by the owner of the seed if there is a sudden rush of requests midway through the download period and having a seed capable of uploading to only five peers is insufficient to meet demand.

All of the scenarios described in this section are examples of why the ability to amend the maximum number of concurrent downloaders would be useful. In many ways, this is a means of amending the level of centralisation – more concurrent downloaders at the seed is a more centralised approach. The aim would be to vary the level of centralisation in order to obtain the best possible query response time for downloading peers.

5.3.8 FAULT TOLERANCE

Chapter 3, Section 3.6 discussed the effect of peer failure. Whilst in most cases, the failure of a single peer will not lead to the failure of the entire system; a peer failing could still have an impact. Future developments of the Wigan system could include some more advanced mechanisms to handle fault tolerance.

One logical approach would be to have backup versions of both the Tracker and the seed on standby to take over should the other fail. Distributed seeds or Trackers could also be used. This means that although there would be a single logical seed or Tracker, it would be physically distributed across multiple machines as is the case with a distributed database.

If a peer should fail, those peers it has connected to will become aware of the fact when keep-alive messages – or handshakes if the peers have not yet established a connection – are not met with a response. The Tracker will only become aware after the update interval between Tracker contacts has passed without the failed peer sending an update message. Until that occurs, the Tracker will not delete the failed peer from its list of advertisements. If an uploading peer fails shortly after sending an update message to the Tracker, the Tracker will have a failed peer in its list of advertisements for almost the entire update interval. Details of the failed uploader could be given to downloaders during this time. One useful extension to the Wigan architecture could be for a downloading peer to notify the Tracker if one of its uploaders either fails to respond to a keep-alive message or a handshake. The Tracker could then remove the peer from its list of advertisements. It is possible that an uploader is only temporarily unavailable, e.g. if the owner was restarting the peer following some software update, hence it might be safer for a peer to attempt two or three keep-alive messages or handshakes before informing the Tracker about the lack of response from the uploader.

5.3.9 SECURITY

Security is an important part of any system and, whilst outside the scope of the simulator implementation described in this Thesis, there are many security aspects concerning Wigan which would warrant further investigation.

Firstly, there is the security of an individual peer. There is the risk of a malicious user accessing a peer and viewing or even amending the data it has downloaded. If a malicious user was able to access and amend data held by an uploading peer, inconsistencies could spread around the Wigan network.

There is also the security of the system as a whole to consider. Only bona fide peers should be able to contact the Tracker to obtain a peer list and then start downloading. In addition to the threat from hackers outside the owning organisation attempting to set themselves up as a peer and obtain data, there may also be the need to prevent valid Wigan users from accessing certain parts of the database. In many database systems, there is sensitive information – such as financial data and personnel records – which can only be accessed by certain users. The same could be true of a Wigan database. Some authentication and authorisation would be needed before a user submits a query. The Tracker would have to participate in this process because this is the place that all downloaders send their queries. This validation could take the form of downloaders having to authenticate themselves at the Tracker before they are allowed to submit a query. The Tracker would use this authentication to verify that the downloader is indeed permitted to submit queries to the Wigan system before returning the peer list. Further validation could also take place at the uploader. When a downloader sends the first piece request to an uploader, it could have to authenticate itself there as well before the uploader sends the first piece. This would prevent malicious users attempting to bypass the Tracker and contacting an uploader directly to request data.

BitTorrent has the facility for a human moderator to examine files placed by seeds to prevent users placing incorrect or corrupt files. This facility was praised by Pouwelse in his analyses of BitTorrent [29, 30] for preventing the spread of such corrupt files around the network. However, the data advertised by the Wigan seed is more complex than the file advertised by the BitTorrent seed, hence moderating would also be more complex. One related issue with the Wigan system is that hashing techniques used in a file-sharing network, whereby a hash function is applied to a file, would not work in every scenario in Wigan. In BitTorrent, as discussed in this thesis, a file does not change throughout the download period. Thus, it is easy to examine the hash-codes of both the original file and the file obtained by a downloader to verify that they are indeed the same file. However, in Wigan this is not the case because it is possible that a downloader obtains a different query to that held by the uploader or uploaders it has received data from. The results would not match a hash of the uploader's query but this does not mean the query results are incorrect. If the database is small, a human moderator could check that the seed has placed the correct data when Wigan system is started; however, for large databases this would be difficult. Regardless of the database size, the moderator could perform random checks on data held by peers other than the seed to check there are no inconsistencies. One possible means of achieving this would be for the moderator to execute test queries to ensure the peers are providing correct data. The moderator would need to be the owner of the seed because they would know what results the test queries should return.

Given that data in Wigan flows across a network from uploader to downloader, there is a need for network security. However, this should not be over and above the current security requirements for distributed client-server databases and other applications – such as online shopping – which require personal or commercially sensitive information to be transmitted across a network. It would be possible to insert checksums (Cyclic Redundancy Checks) into Wigan messages, to verify that the data has not been accidentally corrupted in transit, e.g. because of a network problem. However, these checksums do not detect deliberate attempts to corrupt data. Other mechanisms would be required to verify messages had not been tampered with whilst travelling across the network. Note that this would also include messages to and from the Tracker to prevent a malicious user amending query groups or the IDs of new uploaders in order to divert downloaders to peers of their choosing. Some form of authentication, for example X.509 certificates, would be required to verify messages. The Tracker could authenticate all requests it receives to verify that the message is genuine. There are many encryption algorithms that could be used to encrypt Wigan messages and therefore prevent malicious users viewing the contents. In addition, if the Tracker receives multiple requests for a particularly popular query, it could return peer lists to downloaders which contain non-overlapping sets of uploaders if possible. This would allow the moderator to validate whether independent sets of uploaders are returning identical query results.

Note that this is a scenario where care would have to be taken to avoid security checks slowing down the system so that the performance suffers too much.

A more advanced security framework would see each peer possessing its own security policy. This policy could be based on a variety of factors. A peer might, for example, only want to upload to and download from peers in its organisation, department or office. Alternatively, a peer that has used the Wigan system frequently on previous occasions may only want to upload from and download to peers that it has dealt with before. Such a security policy framework could be implemented in various ways. One possibility would be for a downloader to examine the list of uploaders returned by the Tracker and select uploaders based on its security policy. Alternatively, a peer could specify security constraints in its initial message to the Tracker to ensure the Tracker filters out possible uploaders that violate the downloader's security policy.

5.4 CONTRIBUTIONS OF THIS RESEARCH

This section concludes this thesis by summarising the overall results and describing the contributions of this research. The result of this research is the construction of a (read-only) database system based on the BitTorrent protocol that is correct and complete. It is complete in that all queries that can be asked always (eventually) get an answer unless there is a major failure. It is correct in that the answer that is given is always the correct answer to the query.

The investigation into the Wigan system has produced a number of interesting new results, in particular:

- It is possible to extend and adapt BitTorrent algorithms to create a P2P database server system. The research has pinpointed those areas where the differences between file-sharing and a DBMS mean that BitTorrent is inadequate and has then proposed and evaluated solutions to these problems. The key issues are:
    - The system's performance depends, as might be expected, on the degree of overlap that exists between queries. If there is sufficient overlap, the system can perform well, scaling as the number of clients increases and dramatically reducing the load on the server.
    - The start of the download period, when only the seed can answer queries, causes performance problems. This can be mitigated by a variety of techniques introduced in Chapter 4 including increasing the number of concurrent downloaders allowed at the seed, changing the piece size and replicating the seed.
    - Once the initial bottleneck has been overcome and the caches in the peers are warm, Wigan can offer a faster response time than a heavily-loaded Client-Server system and is capable of supporting larger numbers of downloading peers.

The research presented in this thesis has therefore examined the potential of P2P techniques to overcome the problem of the scalability of a database system being entirely dependant on the performance of the database server. The research aim, introduced in Chapter 1 Section 1.2, has been met and the P2P approach has been shown to have promise. Further work is needed to realise the potential, and an extensive list of possible future work has been identified. A final conclusion is that this research has shown that peer-to-peer databases are a promising approach, worthy of further study as they allow client resources to contribute to increasing the overall performance of a database system.

REFERENCES

1 H. Garcia-Molina, J. D. Ullman and J. Widom. *Database System Implementation.* 2000, Prentice Hall. ISBN 0-13-040264-8

2 *BitTorrent.* Available at: http://www.bittorrent.com/index.html

3 *KaZaa.* Available at: http://www.kazaa.com

4 *The Gnutella Community.* Available at: http://www.gnutella.com/

5 *Napster dead, here comes Torrent.* Available at: http://timesofindia.indiatimes.com/articleshow/1179610.cms

6 *The BitTorrent Protocol.* Available at: http://www.bittorrent.org/beps/bep_0003.html

7 *The BitTorrent Specification.* Available at: http://wiki.theory.org/BitTorrentSpecification

8 D. Zeinalipour-Yazti, V. Kalogeraki and D. Gunopulos. *Exploiting locality for scalable information retrieval in peer-to-peer networks.* In: *Information Systems.* 2005, **30** (4): pp.277-298.

9 P. Lee and T. Anderson. *Fault Tolerance: Principles and Practice, 2nd Edition.* 1990, Springer-Verlag.

10 A. Avizienis, J.-C. Laprie, B. Randell, et al. *Basic Concepts and Taxonomy of Dependable and Secure Computing.* In: *IEEE Transactions on Dependable and Secure Computing.* 2004, **1** (1): pp.11-33.

11 *Sloan Digital Sky Survey.* Available at: http://www.sdss.org/

12 *SkyServer.* Available at: http://cas.sdss.org/dr6/en/

13 C. Mohan. *Caching Technologies for Web Applications.* In: *VLDB 2001, 27th International Conference on Very Large Databases, 2001, Rome.* Morgan Kaufmann.

14 C. Henderson. *Building Scalable Web Sites.* 2006, O'Reilly. ISBN 978-0-596-10235-7

15 S. Androutsellis-Theotokis and D. Spinellis. *A Survey of Peer-to-Peer Content Distribution Technologies.* In: *ACM Computing Surveys.* 2004, **36** (4): pp.335-371.

16 R. Schollmeier. *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications* In: *First International Conference on Peer-to-Peer Computing (P2P '01), 2001, p*p.101-102. IEEE Computer Society Press.

17 B. Yang and H. Garcia-Molina. *Comparing Hybrid Peer-to-Peer Systems.* In: *VLDB 2001, 27th International Conference on Very Large Databases, 2001, Rome. p*p.561-570. Morgan Kaufmann.

18 L. Bischofs, S. Giesecke, M. Gottschalk, et al. *Comparative evaluation of dependability characteristics for peer-to-peer architectural styles by simulation.* In: *The Journal of Systems and Software.* 2006, **79** (10): pp.1419-1432.

19 S. Bowen and F. Maurer. *Designing a Distributed Software Development Support System Using a Peer-to-Peer Architecture.* In: *The 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment, 2002, p*p.1087-1092. IEEE Computer Society.

20 P. Maniatis, M. Roussopoulos, T. J. Giuli, et al. *The LOCKSS Peer-to-Peer Digital Preservation System.* In: *ACM Transactions on Computer Systems.* 2005, **23** (1): pp.2-50.

21 G. Koloniari and E. Pitoura. *Peer-to-Peer Management of XML Data: Issues and Research Challenges.* In: *ACM Sigmod Record.* 2005, **34** (2): pp.6-17.

22 W. Liu and R. Boutabaa. *pMeasure: A peer-to-peer measurement infrastructure for the Internet* In: *Computer Communications.* 2006, **29** (10): pp.1665-1674.

23 M. Roussopoulos, M. Baker, D. S. H. Rosenthal, et al. *2 P2P or Not 2 P2P?,* 2003. Hewlett-Packard, Available at: www.eecs.harvard.edu/~mema/courses/cs264/papers/iptps2004.pdf

24 I. Clarke, O. Sandberg, B. Wiley, et al. *Freenet: A Distributed Anonymous Information Storage and Retrieval System.* In: *ICSI Workshop on Design Issues in Anonymity and Unobservability, 2000, Berkeley.*

25 E. Adar and B. A. Huberman. *Free Riding on Gnutella,* 2000. Available at: http://www.hpl.hp.com/research/idl/papers/gnutella/index.html

26 *Azureus - Java BitTorrent Client.* Available at: http://azureus.sourceforge.net/download.php

27 B. Cohen. *Incentives Build Robustness in BitTorrent.* In: *Workshop on Economics of Peer-to-Peer Systems, 2003, Berkeley.*

28 S. Jun and M. Ahamad. *Incentives in BitTorrent induce free riding.* In: *2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems, 2005, Philadelphia. p*p.116-121. ACM Press.

29 J. Pouwelse, P. Garbacki, D. Epema, et al. *The BitTorrent P2P File-Sharing System: Measurements and Analysis,* 2004. Delft University of Technology, Available at: http://pds.twi.tudelft.nl/~pouwelse/Bittorrent_Measurements_6pages.pdf

30 J. Pouwelse. *The BitTorrent P2P File-Sharing System.* In: *The Register,* 18th December 2004. Available at: http://www.theregister.co.uk/2004/12/18/bittorrent_measurements_analysis/

31 D. Cullen. *SuprNova.org ends, not with a bang but a whimper.* In: *The Register,* 19th December 2004. Available at: http://www.theregister.co.uk/2004/12/19/suprnova_stops_torrents/

32 D. Qiu and R. Srikant. *Modelling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks.* In: *ACM SIGCOMM '04, 2004, Portland. p*p.367-377. ACM Press.

33 M. Izal, G. Urvoy-Keller, E. W. Biersack, et al. *Dissecting BitTorrent: Five Months in a Torrent's Lifetime.* In: *PAM 2004: Passive and Active Network Measurement, 5th International Workshop, 2004, Antibes Juan-les-Pins. p*p.1-11. Springer-Verlag.

34 A. Legout, G. Urvoy-Keller and P. Michiardi. *Understanding BitTorrent: An Experimental Perspective,* 2005. INRIA, INRIA-00000156, VERSION 3. Available at: http://hal.inria.fr/inria-00000156/en

35 A. R. Bharambe, C. Herley and V. N. Padmanabhan. *Analyzing and Improving BitTorrent Performance,* 2005. Microsoft Research, MSR-TR-2005-03. Available at: http://research.microsoft.com/~padmanab/papers/msr-tr-2005-03.pdf

36 C. Gkantsidis and P. R. Rodriguez. *Network Coding for Large Scale Content Distribution.* In: *IEEE/INFOCOM '05, 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 2005, Miami. p*p.2235-2245, Volume 4

37 Microsoft Research. *Avalanche.* Available at: http://research.microsoft.com/camsys/avalanche/

38 R. Knies. *Microsoft Research Cambridge Puts Visual Studio Beta on Fast Track.* In: *Microsoft Research News & Highlights,* 2007. Available at: http://research.microsoft.com/displayArticle.aspx?id=1772

39 D. Arthur and R. Panigrahy. *Analyzing BitTorrent and related peer-to-peer networks.* In: *Seventeenth annual ACM-SIAM symposium on Discrete algorithms 2006, Miami. p*p.961-969. ACM Press.

40 M. Altinel, C. Bornhövd, S. Krishnamurthy, et al. *Cache Tables: Paving the Way for an Adaptive Database Cache.* In: *VLDB 2003, 29th International Conference on Very Large Databases, 2003, Berlin. p*p.718-729. Morgan Kaufmann.

41 Q. Luo, S. Krishnamurthy, C. Mohan, et al. *Middle-tier database caching for e-business.* In: *ACM SIGMOD International Conference on Management of Data, 2002, Madison, Wisconsin. p*p.600-611. ACM Press.

42 P.-A. Larson, J. Goldstein and J. Zhou. *MTCache: Transparent Mid-Tier Database Caching in SQL Server.* In: *ICDE 2004 - 20th International Conference on Data Engineering, 2004, Boston. p*p.177-189. IEEE Computer Society.

43 H. Guo, P.-A. Larson, R. Ramakrishnan, et al. *Support for Relaxed Concurrency and Consistency Constraints in MTCache.* In: *SIGMOD 2004, 2004, Paris. p*p.937-938. ACM.

44 T. Malik, R. Burns and A. Chaudhary. *Bypass Caching: Making Scientific Databases Good Network Citizens.* In: *ICDE 2005, the 21st International Conference on Data Engineering, 2005, Tokyo. p*p.94-105. IEEE.

45 G. Skobeltsyn and K. Aberer. *Distributed cache table: efficient query-driven processing of multi-term queries in P2P networks.* In: *International workshop on Information Retrieval in Peer-to-Peer Networks 2006, Arlington. p*p.33-40. ACM Press.

46 P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, et al. *Data Management for Peer-to-Peer Computing: A Vision.* In: *Workshop on the Web and Databases (WebDB'02), 2002, Madison.*

47 L. Serafini, F. Giunchiglia, J. Mylopoulos, et al. *Local Relational Model: A Logical Formalisation of Database Co-ordination,* 2003. University of Trento, DIT-03-002. Available at: http://eprints.biblio.unitn.it/archive/00000347/01/002.pdf

48 E. Franconi, G. Kuper, A. Lopatenko, et al. *A Robust Logical and Computational Characterisation of Peer-to-Peer Database Systems.* In: *VLDB International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'03), 2003, Berlin.* Springer-Verlag.

49 E. Franconi, G. Kuper, A. Lopatenko, et al. *A distributed algorithm for robust data sharing and updates in P2P database networks.* In: *EDBT International Workshop on Peer-to-peer Computing and Databases (P2P&DB'04), 2004, Heraklion, Crete. p*p.446-455. Springer-Verlag.

50 D. D. Vecchio and S. H. Son. *Flexible Update Management in Peer-to-Peer Database Systems.* In: *IDEAS'05, 9th International Database Engineering and Applications Symposium, 2005, Montreal. p*p.435-444. IEEE Computer Society Press.

51 Z. Majkic. *Weakly-coupled ontology integration of P2P database systems.* In: *The First International Workshop on Peer-to-Peer Knowledge Management, 2004, Boston.* CEUR-WS.org.

52 F. Giunchiglia and I. Zaihrayeu. *Making Peer Databases Interact - A Vision for an Architecture Supporting Data Coordination.* In: *CIA 2002, the 6th International Workshop on Cooperative Information Agents VI, 2002, Madrid. p*p.18-35. Springer-Verlag.

53 K. Aberer, A. Datta, M. Hauswirth, et al. *Indexing Data-oriented Overlay Networks.* In: *VLDB 2005, the 31st International Conference on Very Large Data Bases, 2005, Trondheim. p*p.685-696. ACM.

54 *The P-Grid Project.* Available at: http://www.p-grid.org/index.html

55 M. Minock, M. Rusinkiewicz and B. Perry. *The Identification of Missing Information Resources through the Query Difference Operator.* In: *4th International Conference on Co-operating Information Systems (COOPIS 99), 1999, Edinburgh. p*p.304-314. IEEE Computer Society Press.

56 A. Y. Levy. *Obtaining Complete Answers from Incomplete Databases.* In: *VLDB 96, 22nd International Conference on Very Large Databases, 1996, Mumbai. p*p.402-412. Morgan Kaufmann.

57 S. Gribble, A. Halevy, Z. Ives, et al. *What Can Databases do for Peer-to-Peer?* In: *WEBDB'01, the 4th International Workshop on the Web and Databases, 2001, Santa Barbara.*

58 I. Tatarinov, Z. Ives, J. Madhavan, et al. *The Piazza Peer Data Management Project.* In: *ACM Sigmod Record.* 2003, **32** (3): pp.47-52.

59 D. Alvarez, A. Smukler and A. A. Vaisman. *Peer-To-Peer Databases for e-Science: a Biodiversity Case Study.* In: *20th Brazilian Symposium on Databases, 2005, Federal University of Uberlândia.* UFU.

60 *JXTA.* Available at: http://www.sun.com/software/jxta/

61 W. Hoschek. *A Unified Peer-to-Peer Database Framework for Scalable Service and Resource Discovery.* In: *Grid 2002, the 3rd International IEEE/ACM Workshop on Grid Computing 2002, Baltimore. p*p.126-144. Springer-Verlag.

62 R. Huebsch, J. M. Hellerstein, N. Lanham, et al. *Querying the Internet with PIER.* In: *VLDB '03, the 29th International Conference on Very Large Databases, 2003, Berlin. p*p.321-332. Morgan Kaufmann.

63 R. Huebsch, B. Chun, J. M. Hellerstein, et al. *The Architecture of PIER: an Internet-Scale Query Processor.* In: *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, 2005, Asilomar, California.*

64 B. Chun, J. M. Hellerstein, R. Huebsch, et al. *Design Considerations for Information Planes.* In: *WORLDS '04, the 1st Workshop on Real Large Distributed Systems, 2004, San Francisco.*

65 G. Kokkinidis and V. Christophides. *Semantic Query Routing and Processing in P2P Database Systems: The ICSFORTH SQPeer Middleware.* In: *EDBT International Workshop on Peer-to-peer Computing and Databases (P2P&DB'04), 2004, Heraklion, Crete.* Springer-Verlag.

66 W. S. Ng, B. C. Ooi, K.-L. Tan, et al. *PeerDB: A P2P-based System for Distributed Data Sharing.* In: *19th International Conference on Data Engineering, 2003, Bangalore. p*p.633-644. IEEE Computer Society.

67 B. C. Ooi, Y. Shu and K.-L. Tan. R*elational data sharing in peer-based data management systems.* In: *ACM SIGMOD.* 2003, **32** (3): pp.59-64.

68 *BestPeer.* Available at: http://www.bestpeer.com

69 V. Papadimos, D. Maier and K. Tufte. *Distributed Query Processing and Catalogs for P2P Systems.* In: *1st Conference on Innovative Data Systems Research, 2003, Asilomar, California.*

70 F. Pentaris and Y. Ioannidis. *Query optimization in distributed networks of autonomous database systems.* In: *ACM Transactions on Database Systems.* 2006, **31** (2): pp.537-583.

71 C. Rouse and S. Berman. *A Scalable P2P Database System with Semi-Automated Schema Matching.* In: *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06) 2006, Lisbon. p*p.78. IEEE Computer Society.

72 C. P. d. Laborda and C. Popfinger. *A Flexible Architecture for a Push-based P2P Database.* In: *16th GI-Workshop on the Foundations of Databases, 2004, Monheim. p*p.93-97. Universitat Dusseldorf.

73 C. P. d. Laborda, C. Popfinger and S. Conrad. *Digame: A Vision of an Active Multidatabase with Push–based Schema and Data Propagation.* In: *GI-/GMDS-Workshop on Enterprise Application Integration (EAI'04), 2004, Oldenburg.*

74 Transaction Processing Council. *Transaction Processing Council TPC-H Benchmark.* Available at: http://www.tpc.org/tpch/

75 M. T. Schlosser, T. E. Condie and S. D. Kamvar. *Simulating a File-Sharing P2P Network*. In: *First Workshop on Semantics in P2P and Grid Computing, 2002,*

76 University of Edinburgh, Institute for Computing Systems Architecture, Division of Informatics. *The SimJava tool*. Available at: http://www.dcs.ed.ac.uk/home/hase/simjava/

77 *MySQL*. Available at: http://www.mysql.com

78 *JDBC*. Available at: http://java.sun.com/j2se/1.5.0/docs/guide/jdbc/

79 *SQL Server*. Available at: http://www.microsoft.com/sql/default.mspx

80 D. Kossmann. *The State of the Art in Distributed Query Processing*. In: *ACM Computing Surveys*. 2000, **32** (4): pp.422-469.

81 S. Ganguly, W. Hasan and R. Krishnamurthy. *Query Optimisation for Parallel Execution*. In: *1992 ACM SIGMOD International Conference on Management of Data, 1992, San Diego*. *p*p.9-18. ACM Press.

82 H. Ye, B. Kerherve and G. v. Bochmann. *QoS-Aware Distributed Query Processing*. In: *10th International Workshop on Database and Expert Systems Applications (DEXA 99), 1999, Florence*. IEEE Computer Society.