WAYFINDER: A FEDERATED INFORMATION SHARING AND MANAGEMENT SYSTEM

BY CHRISTOPHER PEERY

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Thu D. Nguyen
and approved by

New Brunswick, New Jersey
October, 2009
ABSTRACT OF THE DISSERTATION

Wayfinder: A Federated Information Sharing and Management System

by Christopher Peery
Dissertation Director: Thu D. Nguyen

The decreasing costs of computing devices, increases in connectivity, and improved performance are altering the computing environments of users in fundamental ways. Users are no longer restricted to operating single devices in isolation but rather distribute and access information across many devices and develop complex sharing patterns among groups of users. Unfortunately, while these trends are significantly enriching the user’s computing experience, they are also increasing the data management overhead as users must explicitly reason about data placement and replication across multiple devices and logical sharing groups.

In this thesis, we present the Wayfinder file system, which was designed to simplify the management of information in federate systems. Wayfinder focuses on three critical management deficiencies present in most current federated environments: 1) the lack of a consistent view for stored information across devices, 2) the required manual management of replicated information, and 3) the limited search/ranking capability for finding relevant information.

The Wayfinder file system addresses these deficiencies by providing three synergistic abstractions: 1) a global namespace that is uniformly accessible across connected and disconnected operation, 2) a user-centric automatic availability management to ensure
continuous access to information based on data-centric availability policies, and 3) a multi-dimensional fuzzy search framework that significantly improves relevance ranking. We will show that these abstractions simplify the management burden by requiring users to reason only about the data and its properties while ignoring the underlying physical complexities of the system.

Underlying all three abstractions is a common implementation layer that adheres to three principles. First, any subset of nodes in a Wayfinder community can interact normally when they are interconnected, regardless of the membership of the subset. Second, all protocols and interactions are tolerant of a weakly consistent model allowing them to suffer unexpected devices departures. Finally, devices are assumed to be owned by specific users and so should prioritize the needs of their owners in the presence of resource constraints, while using excess resources to benefit the community as a whole.
Acknowledgements

During my time at Rutgers I have had the pleasure of interacting with an amazing group of individuals. Two people in particular stand out; my wife, Sharon, and my advisor, Prof. Thu Nguyen. Sharon has been an unwavering source of encouragement and support in my life. This thesis might not have been completed if not for her love and endless confidence in me. Thu has stuck with me through the years and continuously challenged me to do my best. This thesis is a testimony to his patience and to the many things I have learned from him. I thank them both for all they have done for me.

I also thank all the members of Panic-Lab and Dark-Lab, past and present; Matias Cuecna Acuna, Kiran Nagaraja, Xiaoyan Li, Kien Le, Wei Wang, Tuan Phan, Taliver Heath, Eduardo Pinheiro, and Fabio Oliveira. You made being at the lab a memorable and culturally enriching experience. Our friendships, lab discussions, and many experiences allow me to look back at my time at Rutgers with fondness and laughter.

I thank the other professors I have had the privilege of working with; Prof. Rich Martin, and Prof Amélie Marian, for their instruction and advice.

I thank the staff of the Computer Science Department and LCSR for helping me solve the countless problems that graduate students have to deal with.

I thank the “Argentinian Gang”; Matias, Cecilia, Miguel, and Silvia. Your friendship and acceptance will never be forgotten.

I thank my friends outside of Rutgers; Foster, Courtney, Rita, Dave, and Frank. You provided me with an escape, with words of encouragement, and with friendship when I needed them most.

Finally, I thank my parents to whom this thesis is dedicated. They have always been there for me and I owe much of who I am and where I am today to them.
Dedication

To my parents, who were the first to teach me the value of an inquisitive mind.
Table of Contents

Abstract ................................................. ii
Acknowledgements ....................................... iv
Dedication ................................................ v
List of Tables ........................................... xi
List of Figures .......................................... xiv

1. Introduction ......................................... 1
   1.1. Motivation and Goals ............................. 1
   1.2. Thesis Structure ................................. 3
   1.3. Contributions .................................. 11

2. Background and Related Work ....................... 12
   2.1. Federated Systems ............................... 12
   2.2. PlanetP ......................................... 14
   2.3. Distributed File Systems ....................... 19
   2.4. Global Directory Structure .................... 21
   2.5. Consistency Model .............................. 21
   2.6. Disconnected Operation ......................... 22
   2.7. Content-based addressing ....................... 24
   2.8. Personal Information Search ................... 24
   2.9. Distributed Information Search ............... 25
   2.10. Availability .................................. 26
   2.11. Cooperative Caching ........................... 28
3. Federated File System .............................................. 29
   3.1. Overview .................................................. 29
   3.2. Example Usage ............................................ 31
      3.2.1. Usage Example: Wiki ............................... 31
      3.2.2. Usage Example: Shared Search Structure ......... 32
   3.3. File System Design ........................................ 34
      3.3.1. Files .................................................. 36
      3.3.2. Directories .......................................... 39
   3.4. Caching .................................................... 40
   3.5. Consistency Model ......................................... 41
      3.5.1. Files .................................................. 42
      3.5.2. Directories .......................................... 44
   3.6. Security Model ............................................ 46
   3.7. Storage Tables ............................................ 48
      3.7.1. Metadata Table ...................................... 48
      3.7.2. Structural Table .................................... 49
      3.7.3. Content Table ...................................... 49
   3.8. Performance ................................................ 50
      3.8.1. Experimental Setup .................................. 50
      3.8.2. Micro Benchmark .................................... 50
      3.8.3. Macro Benchmark ................................... 52
      3.8.4. Scalability and Robustness ......................... 54
      3.8.5. Storage Overhead ................................... 55
   3.9. Summary .................................................... 56

4. Automatic Availability Management ............................. 57
   4.1. Replication Algorithm ...................................... 58
      4.1.1. Terminology and Assumptions ....................... 61
      4.1.2. Replication .......................................... 62
4.1.3. Eviction .................................................. 64
4.1.4. Updates .................................................. 65
4.1.5. Estimating Online Availability ......................... 67
4.1.6. Algorithm Implications ................................. 67
4.2. Implementation ............................................. 69
  4.2.1. Locating Replicas ..................................... 69
  4.2.2. Maintaining Node Groups ............................. 69
  4.2.3. Tracking Device Online Availability .................. 69
  4.2.4. File Tagging ........................................... 70
  4.2.5. Maintaining Offline and Ownership Availability .... 70
  4.2.6. Maintaining Online Availability ..................... 71
  4.2.7. Eviction ............................................... 72
  4.2.8. Updates ............................................... 72
4.3. Evaluation .................................................. 72
  4.3.1. Methodology ........................................... 73
4.4. Online Availability for Creation and Write Bursts ........ 74
4.5. Online Availability for a Collaborative Workload .......... 76
4.6. Ownership Availability for a Collaborative Workload ...... 82
4.7. CPU and Network Utilization ............................... 83
4.8. Summary ................................................... 87

5. Single Node Multi-Dimensional Search ....................... 88
  5.1. Introduction ............................................... 88
  5.2. A Unified Multi-dimensional Scoring Framework .......... 90
    5.2.1. Query Model ......................................... 91
    5.2.2. Scoring Content ...................................... 92
    5.2.3. Scoring Metadata ..................................... 92
      5.2.3.1. DAG Representation of Metadata Relaxations .... 93
      5.2.3.2. Scoring Metadata Relaxations .................... 96
6.1.2.2. Query Evaluation ........................................ 126
6.1.2.3. Result Merging ........................................ 126
6.1.3. Partitioned Federated Search .............................. 127
6.2. Implementation .............................................. 128
  6.2.1. Maintaining Global Indices ............................ 128
  6.2.2. Summary Sizes .......................................... 129
    6.2.2.1. Data Sets ........................................ 129
    6.2.2.2. Lossless Summary Compression ...................... 130
    6.2.2.3. Lossy Summary Compression ......................... 134
6.3. Experimental Evaluation ................................... 137
  6.3.1. Experimental Setting .................................. 138
    6.3.1.1. Platform .......................................... 138
  6.3.2. Federated Query Evaluation ............................ 138
    6.3.2.1. Federated System ................................ 139
    6.3.2.2. Data Replication ................................ 139
    6.3.2.3. Queries ........................................... 140
    6.3.2.4. Federated Query Performance ...................... 141
    6.3.2.5. Stability ......................................... 146
    6.3.2.6. Global Index Maintenance ......................... 149
6.4. Summary .................................................. 151
7. Conclusions .................................................. 153
References ..................................................... 159
Vita ............................................................. 165
List of Tables

3.1. Micro-benchmarks for NFS operations in both single and multi-node communities while using both persistent and in-memory storage for the tables outlined in Section 3.7. The benchmark comprises a set a user coming into a community, creating numerous files and directories at different locations, writing to them, and then deleting them.................................................. 51

3.2. Results of the Modified Andrew Benchmark using the Linux NFS, original JNFSD and the JNFSD linked with Wayfinder running in isolation and in a community of 12 nodes.............................................. 53

4.1. The median time required for Wayfinder to achieve the target online availability \( TOAC \) for a group of newly created files. .................. 75

4.2. Percentage of CPU utilization while running Wiki Trace in both the HW and CO environments with different restrictions on the size of the local hoard. ................................................................. 84

4.3. Count of operations performed by replication algorithm during the Wiki Trace. Replication Request is the actual sending of a replication request from a champion. Download Attempt is the number of actual file downloads that occurred as a result of either an access or replication request. File Eviction is the number of performed replica evictions. ............ 84
4.4. Breakdown of the amount of data written during the wiki experiment for different environments. Included is the total amount of information written (in MBs), the average write bandwidth (in KB/s) for devices and a breakdown (in MBs) of the total bandwidth into replication related activities. Content is the retrieval of remote file content. Search is the amount of information exchanged when during distributed query processing in PlanetP. Replication is the amount of information sent to issue replication requests.

5.1. The rank of a target file—the novel Sea Wolf by Jack London—returned by a set of related queries. In the presence of ties in the relevance scores, the highest rank the target file could have is given. The queried dimensions include Content, Type (Metadata), Date (Metadata), and Structural. The initial content query Q1 provides the set C containing the 4 query terms {jack, london, sea, wolf}. Structural values are abbreviated. The complete path of our target file is /Personal/Ebooks/Novels/JackLondon/.

Queries which contain a “*” in the first column represent those in which the target file would not be considered as a relevant answer given today’s typical filtering approach.

5.2. Query performance for various single- and multi-dimensional queries for both in-memory indexes/tables and persistent indexes/tables. k is set to 20.

6.1. Summary of the data sets used in our evaluation. Presented are summary statistics for metadata, content, and structural information.

6.2. Table showing the number of hashed values that experienced collision when encoding structure and content values using an MD5 hash and truncating it to various sizes.
6.3. The space requirements for encoding the summaries of the data sets outlined in Table 6.1 under various schemes. Naive refers to using 4 bytes per integer value and a tuple \((size, value)\) for any strings. Hash encodes textural information as a hash value of the specified size. Byte refers to byte-aligned encoding and Gamma to a gamma encoding. For rows identified by two encodings (i.e., Byte, Gamma), the first is the encoding of the key value and the second the corresponding file count for each tuple in the summary. All versions use a run-length encoding on the key values.

6.4. The number of unique keys for (a) modification times and (b) file sizes that results from aggregating the unique values of the respective tables into increasing larger ranges.

6.5. The number of bytes needed to encode the date and file size information for several data sets after tables values have been aggregated. The percentage after each number is the percentage of reduction achieved compared to the encodings given in Table 6.3.

6.6. The number of valid queries generated for each category of files. Shown are the number for exact queries. If the number of relaxed queries differ, the difference is given in parenthesis.

6.7. Summary of the changes in rank of the target files used in the construction of our set of queries when comparing a single node (SE) query evaluation to that of a federated system (FE) broken down by query categories. Shown is the behavior for the set of exact queries (a), relaxed queries (b), and exact queries using an aggregated index(c). Improved represents that the target file attained a higher rank, for Worsened it attained a lower rank, for Unchanged the rank was the same, and Not Found represents that the target file was not found in either setting. For these queries, \(K\) was set to 20.
List of Figures

2.1. Layout of storage system in PlanetP. Shown is the distributed table and the gossip-based communication component for a single running instance. The latter component ensures that any updated information is received and shared with the rest of the community. ........................................... 16

2.2. Layout of the storage system in PlanetP which has been altered to allow multiple user-defined global tables and to permit the storage of XML documents. Each table is designed to store a particular type of XML document. Remote summaries are passed, and stored, only between tables of the same type. ......................................................... 17

2.3. Example XPath query to retrieve information from PlanetP. ......................... 18

3.1. Wayfinder dynamically constructs a shared namespace across any set of connected devices by merging their local hoards. This figure shows 5 nodes originally being connected so that the shared namespace is the merged view of hoards H1 through H5. When the community is partitioned into 3 connected subsets, Wayfinder maintains a merged view for each subset. When the subsets reconnect, Wayfinder dynamically re-merges the shared namespace. Shaded boxes indicated connected partitions. ......................... 30

3.2. An overview of Wayfinder’s architecture across two devices each with their own local hoards (denoted by cylinders). Portions of the global tables for storing metadata and structural information are shown with their relationship to information in the local hoard. .............................. 36

3.3. XML encoding of the metadata portion of a waynode representing a file. ...... 37

3.4. XML encoding of the content summary of a waynode representing a file. ...... 37

3.5. XML encoding of the structural portion of a waynode representing a file. ...... 38
3.6. Example of a file’s history vector containing information for several applied and conflicting updates.  ........................................... 43

3.7. (a) Time required to scan a namespace plotted against community size. (b) Scan time in the presence of DHT failures. X-axis indicates a sequence of successive scans with solid vertical lines denoting a scheduled failure point. The label associated with each solid line indicates the number of nodes involved. Dashed lines represent the minimum and maximum times across twelve samples.  ........................................... 55

4.1. (a) Basic and (b) modified replication algorithms for a user \( u \) and his device set. Files in \( \text{Own}_u \) must be stored in the PSpace of at least one device in \( \text{DS}_u \) (or just the champion (b)). Files in \( \text{WS}_u \) are stored in the PSpace of all devices in \( \text{DS}_u \). Each device in \( \text{DS}_u \) (or just the champion (b)) then pushes files in \( \text{Own}_u \) to the CSpace of other devices in the community.  ........................................... 60

4.2. The availability of a file (both as a percentage and the number of nines) plotted as a function of the number of replicas placed on devices with equal availability. Each line represents a specific device availability. Figure (b) is an enlarged view of the upper-left region of Figure (a).  ........................................... 67

4.3. Average file online availability, average version availability (for the latest version of each file at each instant in time), and minimum availability plotted as functions of experiment time for (a) CO with infinite space, (b) HW with infinite space, (c) CO with constrained space, and (d) HW with constrained space.  ........................................... 78

4.4. Division of files between PSpace and CSpace for (a) CO with infinite space, (b) HW with infinite space, (c) CO with constrained space, and (d) HW with constrained space.  ........................................... 79

4.5. Online version availability for a representative file from HW with infinite space.  ........................................... 81
4.6. Ownership availability of a representative file vs. experiment time for the HW environment with (a) infinite space and (b) constrained space. Figures plot the number of replicas contained within a user’s device set and how many of these replicas are up-to-date given changes performed by the user (Group Access) and remote changes (Remote Access).

5.1. Fragments of the indexing DAGs for (a) file type (extension) metadata, and (b) file date metadata. Both represent portions of the complete DAGs with several levels removed for simplicity of presentation. For file dates, we present the length of the relaxation intervals at each level of the DAG on the box at the far right. Highlighted nodes indicate the sequence of relaxations for a file type query of “.cpp” and a date query of “3:00 PM on Feb. 5, 2006”.

5.2. Fragment of the indexing DAG for file sizes. The DAG spans the range of file size up to 8 MB. Portions of the DAG have been removed to simplify the presentation. Shown are the branches associated with files size of 16 bytes and 512 KB. Dashed lines represent locations where several nodes have been removed for space considerations. Highlighted nodes indicate the sequence of relaxations for a file size query “512 KB”.

5.3. The structure DAG for the structural query condition Personal/Ebooks/JackLondon. Solid lines represent parent-child relationships. Dotted lines represent ancestor-descendant relationships, with intermediate nodes removed for simplicity of presentation.

5.4. Relevance score plotted as a function of ranking for (a) four content queries, (b) three structure queries, and (c) three size metadata queries.

5.5. Score (a-b) and rank (c-d) of a target file returned as a result of a constant 2-dimensional query as the file is relaxed away from the query conditions across the two query dimensions. Score and rank of the same file target file plotted for a content-only query (e) and a structure-only query (f).

5.6. $\rho_{\text{min}}$ value for various multi-dimensional queries as a function of $k$. ...
5.7. (a) The mean time for queries targeting email and documents plotted as a function of the data set size. (b) CDF of the total query time for all queries targeting email and documents. $k$ is set to 20.

5.8. CDF of the total time for evaluating queries for different values of $k$.

6.1. (a) Equation for computing Inverse Document Frequency (IDF) and (b) a plot of this function as the value of $T$ increases with a value of 1000 for $N$.

6.2. The three stages of compressing an index/table summary (a) using a run-length encoding (b) on index/table values followed by either a (c.1) byte-aligned encoding scheme or (c.2) gamma encoding scheme. For the byte-aligned encoding, the underlined bits indicate (in binary) how many bytes are used.

6.3. Distribution of scores returned when querying for files having size of 1 KB under different levels of aggregation. Base represents no aggregation, Small Aggr. represents using a 32 bytes factor and Large Aggr. using a 1024 byte factor.

6.4. A pictorial representation of how scores are assigned when Global IDF Summary indexes are aggregated. Shown is a sequence of file sizes along a range (middle), the span of several relaxation steps over these files sizes (top), and the aggregation intervals for a factor of 32 bytes. The labels $S1, S2, S3$ explicitly identify three specific sizes.

6.5. (a-b) The Result Overlap and (c-f) normalized cumulative ranks of any missed files when comparing the set of exact queries evaluated on a single node (SE) to an evaluation in a federated setting (FE). For the cumulative miss rate, every fifth rank is denoted by a darkened bar.

6.6. (a) The Result Overlap and (b) normalized cumulative ranks of any missed files when comparing the set of relaxed queries evaluated on a single node (SE) to an evaluation in a federated setting (FE). For the cumulative miss rate, every fifth rank is denoted by a darkened bar.
6.7. Result Overlap when comparing the results of query evaluation on a single node (SE) to the results of evaluation in a federated setting (FE) in which the global indexes have been aggregated. Aggregation is done based on a small configuration and queries used are exact. . . . . . . . . . . . 146

6.8. Result Overlap for (a) content-only and (b) multi-dimensional when comparing the results of query evaluation on a single node (SE) to the results of evaluation in a federated setting (FE) for various values of \( k \). . . . 147

6.9. Number of global index values that changed daily for content (a) and metadata (b) a period of 19 days. Changes are computed on a daily basis. Date, Size, Directory indicate changes to specific dimensions. File represent the number of files that were either added or removed. At the start of the trace there are approximately 4,696,362 terms in the content table. . . . . . . . . . . . . . 149

6.10. Number of global index values that changed daily for metadata during a period of 126 days (a) and a subset of this trace spanning 30 days (b). Changes are computed on a daily basis. Date, Size, Directory indicate changes to a specific dimension. File represent the number of files that were either added or removed. . . . . . . . . . . . . . . . . 150
Chapter 1

Introduction

1.1 Motivation and Goals

Information management concerns the collection, organization, re-distribution, and maintenance of information [39]. Since most information today is stored digitally, computing devices play a significant role in performing these tasks. As computing devices have evolved, so have these management tasks; in the process becoming increasingly more complex. In the 80’s, the PC was the primary tool by which users managed information. This continued in the 90’s with the advent of the World Wide Web providing users with access to previously unheard of amounts of new information. In the last ten years, single PCs have been replaced by sets of devices, many of which are mobile. This evolution has lead to changes in information management; it has become more difficult.

Difficulties arise from several on-going trends in the computing world that are affecting how users manage information. The first of these trends is the decreasing cost, and increasing size of storage. This is providing users with access to larger amounts of space to store information. This space is subsequently being used and exhausted, as is evident by the continuing market for larger storage disks. The second trend is that devices are becoming increasingly connected, allowing for both increased access to external resources and interactions with remote entities. Third, due to the decreasing cost and size of computing devices, the number of devices per user is increasing. It is quite common today for the average user to own, or have access to, several different personal computers throughout a given day (at home or at work) and to have several smaller devices (such as PDA or phones) in their personal procession. Each of these is capable of storing and accessing information. Finally, computing devices have become
pervasive in our society.

When viewed together, these trends create a challenging scenario for managing information. Despite improvements in connectivity, users often take advantage of increases in storage to replicate portions of their working environments locally. In doing this, users have increased responsiveness during connected operation and ensure availability during disconnected operation at the expense of manually tracking and maintaining potentially large numbers of replicas. Further, accessing and storing large amounts of information exacerbates the difficulty many users experience in recalling details concerning information (i.e., where is file “report.txt” stored). This is even further complicated when information resides across multiple devices. Also, while network connectivity is improving, it is not universal. Mobile devices may go offline for a number of reasons (i.e., lack of signals, power conservation, or security concerns) making disconnected operation and changes in online membership common.

Finally, individual users are rarely online in isolation; the sharing of information has become a social activity. Large scale file sharing networks [31, 43] are quite popular. Many popular online services (i.e., YouTube [81], Flickr [27], Myspace [53]) cater to the desire to share certain types of information. Most instant messaging have features to allow the pair-wise exchange of files. The sharing of information is becoming an increasingly complex activity and can involve groups, or federations, of users.

We believe that small to medium size (10s to 100s of nodes) federated communities represent the most common and interesting targets for studying how to improve information management. Aside from several well-known P2P communities, which arguably have a very simple model of information sharing, no large scale systems have been deployed or are actively in use. We feel that smaller communities provide a more suitable environment for fostering complex interactions and sharing patterns. This could be a result of participants knowing each other personally or that the information being shared is very specific and thereby only attracts a limited interest group (i.e., files being shared at a workplace or amongst members of a family).

In such federated environments, a user’s computing environment is sufficiently fragmented to produce three management deficiencies. First, as the same information may
be potentially named differently across devices (i.e., a file system structure on a machine running a Windows operating system and on a Linux-based operating system differ), there is no consistent manner for accessing and storing information. Second, replica management requires users to manually reason about the properties of information (i.e., its version and the flow of updates) and its location to ensure its availability. Finally, searching for information in a distributed computing environment is difficult as search tools are often device specific. Further, these tools rely on content-centric queries, ignoring the large amount of structure and metadata stored within file systems that could be used to improve the ranking of relevant results.

1.2 Thesis Structure

In this thesis, we present the Wayfinder file system which aims to simplify the management of information in federate systems by providing the tools that will enable users to reason about the data and its properties while ignoring the physical complexities of the underlying environment. While this goal is neither new or unique to our work, our contribution lies in addressing it in a federated system of personal devices centered around information sharing. As such, we assume that devices may reside in separate administrative domains and so may act with relative autonomy; including leaving the community at any time. Further, we hold that personal devices should, as their primary function, prioritize the needs of their respective owner over that of the community. However, engaging in collaboration and sharing on a communal level remains beneficial. Thus, the salient feature of this thesis is providing users with a single available view of their entire computing environment while accounting for the autonomous behavior of nodes and balancing the individual needs of users on their personal devices with that of the communal good.

We address the above mentioned deficiencies through three synergistic abstractions. These abstractions simplify the management burden by requiring users to reason only about the data and its properties while ignoring the underlying physical complexities of the system. The abstractions are: 1) a global namespace that is uniformly accessible across connected and disconnected operation – ensuring a consistent naming of
information, 2) a user-centric automatic availability management component to ensure continuous access to information based on data-centric availability policies – removing manual overhead of replica placement and management, and 3) a multi-dimensional fuzzy search framework that significantly improves relevance ranking – allowing expressive search queries over a distributed computing environment. Their combination allows a computing environment to appear as a single coherent and available resource to a user.

As part of this thesis, we will show that these abstractions can be built using a weakly consistent distributed query-based object store [16] that provides only probabilistic guarantees. With this object store, remotely stored information is retrieved by posing appropriate queries. It is a further goal of this thesis to demonstrate that this model of storage can be employed in a federate system for the purpose of supporting a variety of useful abstractions.

The thesis consists of four main parts as follows.

**Federated File System**

We begin by addressing how to improve the storage and organization of information in federated systems. Motivation for various approaches can be found in the example of the World Wide Web. The success of web search engines such as Yahoo [79] and Google [32] demonstrate the usefulness of search technologies for finding information. The usefulness of these tools, however, is dependent on having a set of useful terms with which to search. In contrast, several online services exist, such as DMOZ [23] or Yahoo groups [80], that classify information in large manually constructed namespaces. These allow users to browser unfamiliar information by utilizing a pre-constructed ontology; alleviating the need to construct a query. The web is therefore an example where both browsing and searching are useful for the successful management of large volumes of information.

Similar circumstances to the Web exists in a federated community where a majority of the shared content will be residing on devices belonging to other users and possibly be unfamiliar, providing motivation to leverage both search and browsing to organize
information. In particular, we investigate creating a collaborative organizational structure that will provide a consistent browsing experience regardless of which devices a user is on. Further, given the large amounts of information present, this structure will leverage search technologies to automatically place new information into relevant portions of the structure.

We propose two mechanisms as part of the Wayfinder file system to address these issues. The first mechanism is a unified globally accessible namespace that allows device-independent browsing of all data in a connected community. Our approach is novel in that the namespace is constructed by merging directories with the same name, but located on different devices, into a single directory viewable across all the devices. This produces a single coherent structure that presents a shared view of all data stored across any set of connected nodes. By using this merge approach, the global namespace is able to expand and contract smoothly on node arrivals and departures. Critically, this permits the namespace to be uniformly accessible across connected and disconnected operation of devices. At the extreme, disconnected operation simply means that the namespace will include only the local hoard. Additionally, the community may, at any point, split into multiple connected subsets, each with its own shared namespace, and later rejoin to recreate the entire global namespace.

Such a namespace reduces data management overheads by removing the need for users to explicitly reason about what replica resides on which device. Finally, in naturally encompassing partitioned and disconnected operation users are provided with some continuity in how their data may be accessed irrespective of their connected state.

The namespace, however, presents a fundamental trade-off; allow users to continue accessing information regardless of their connected state versus the consistency of accessed information. Partitioned and disconnected operation imposes limitations on any consistency of replicated information. The novelty of our work is that we present a unified model that explores this trade-off across the full range of connected states a user may experience; complete disconnection, partially connected, and fully connected. This ensures a browsing experience that will always be similar, albeit the amount of available information will change.
The second mechanism is search queries that are embedded persistently into the global namespace in the form of Semantic Directories [30, 33]. These are directories that have associated content queries. The queries are evaluated over the shared content of the federated system and the directory is populated by files that are returned as results. When re-evaluated periodically, these directories allow a portion of the global namespace to become an active organization structure by automatically creating file bindings for new incoming content. Our work explores providing this functionality in a federated settings.

An evaluation of a prototype implementation shows that our method of dynamic namespace construction does incur some overhead. To minimize this, we show how to leverage a storage paradigm from the Peer-to-Peer community towards constructing a communal cache, specifically we use a light-weight Distributed Hash Table [67] (DHT).

We feel that given the benefits of the system, that the observed overheads are tolerable. In fact, we have found the device-independent naming of files provided by the global namespace to be quite useful. This feature assisted in greatly simplifying the designs presents in the later chapters of this dissertation. However, we also found that various file system operations, such as deleting files and directories, become more complicated.

**Automatic Availability Management**

We then turn to addressing issues concerning the availability of information in federated information systems. Increasingly, users are accessing their computing environments from multiple personal devices. Among these devices, users maintain multiple copies of data to increase responsiveness during connected operation and availability during disconnected (or low-bandwidth) operation. Users also create replicas of important information stored remotely to guarantee continued personal availability; a sort of personal back-up. Furthermore, replicas of files may be created, either manually or automatically, to ensure that they remain online for the benefit of the federated community. Each of these scenarios requires the creation of replicas to improve availability, albeit the placement and number of replicas may vary. Each fundamentally raises the overheads of data management as users are required to create and track data replicas.
across numerous devices and often under different policies.

In an ideal situation, we would replicate content across these devices so that any shared content would be accessible from any device at anytime with high probability, regardless of the device’s connection status at the time of an access. This device and connection transparency would allow users to avoid the need to reason about the placement of data replicas on specific devices.

This ideal situation is impractical. We instead propose a novel unified availability model which differentiates between three types of availability; online, offline, and ownership availability. The first represents a communal availability (i.e., being able to access information when connected) while the latter two are forms of personal availability (i.e., ensuring that information can be accessed by a user on their devices). We argue that in addressing these three availabilities, we can reasonably approximate the ideal situation by ensuring availability across periods of connected, temporary disconnected, and permanent disconnection operation.

We also propose that in a federated system composed of personal devices, that a replication algorithm should allow devices to prioritize the availability needs of their owners over the needs of the community. The primary use of a personal device is to store content that the user cares about (i.e., content he is likely access in the near future) and content that the user owns (i.e., content he would want in the case of permanent disconnection). Failure to store this basic information may cause problems and result in the user leaving the community out of frustration. Nevertheless, it is beneficial for devices to collaborate to maintain high online availability for all shared content because this allows all users to find new content of interest to them as well as easy access to content that they have not used in a long time.

We present the design of a single replication algorithm that achieves all three types of availability while prioritizing the needs of individual users on their personal devices. We present an implementation of this algorithm as part of the Wayfinder file system. Coupling our replication algorithm with the Wayfinder global namespace removes the need for users to reason about the placement of replicas for both locating data and ensuring data availability. The burden of data management for each user is thus reduced
to reasoning about what availability properties he desires for specific portions of the
global namespace.

Our evaluation shows that with a sufficient amount of communal storage, the repli-
cation algorithm is able to efficiently achieve any specified availability targets. However,
if space is sufficiently constrained, nodes enter a non-cooperative mode of operation in
which they prioritize the needs of their owners before that of the community. The results
is that personal availability targets are meet, while attaining any degree of communal
availability is a mere side-effect.

**Single Node Multi-dimensional Search**

We then consider methods for improving search techniques. Despite this disserta-
tion’s focus on federated systems, we first focus on addressing this topic in a single node
setting, addressing deficiencies of many contemporary search tools. Applying this work
to a federated setting is next.

There is an explosion in the amount of data users access and store. Beyond the
content of a file, file systems provide potentially useful metadata that can assist users
in searching. There is a need for powerful search tools to access often complex data
in a simple and efficient way. Numerous third-party search tools have been developed
to perform keyword searches and locate personal information stored in personal infor-
mation management systems such as the commercial file system search tools Google
Desktop [29] and Spotlight [66]. However, these tools usually index text content, allow-
ing for some *ranking* on the textual part of the query—similar to what has been done
in document search in the Information Retrieval (IR) community—but only consider
structure (e.g., file directory) and metadata (e.g., date, file type) as *filtering* conditions.

We argue that this approach is insufficient in many query scenarios and believe that
allowing flexible conditions on structure and metadata can significantly increase the
quality and usefulness of search results in many search scenarios. The challenge is then
to adequately score the search results by taking into account flexibility in the textual
component *together* with some flexibility in the structural and metadata components
of the query. Once an adequate scoring mechanism is chosen, efficient algorithms to
identify the best query results, *without considering all the data in the system*, are also
needed.

To this end, we present a novel approach that allows users to provide fuzzy conditions on three query dimensions: content, metadata, and structure. We describe individual $IDF$-based scoring approaches for each of these dimensions (scoring of the structural dimension was done outside the context of this thesis and we present only a brief overview) and present a unified scoring framework which aggregates the individual dimensions scores to produce a single relevance score per file.

We perform an evaluation using a real-world data set. We show that our approach is able to provide relevant results for query situations that are difficult for contemporary search tools. We demonstrate that our $IDF$-based scoring approach provides a meaningful distribution of scores that captures the specificity of each dimension. We also show that our multi-dimensional score aggregation technique preserves the properties of individual dimension scores and has the potential to significantly improve ranking accuracy.

**Multi-dimensional Federated Search**

To complete our work in developing improved search techniques, we turn to applying our multi-dimensional search framework to a federated setting. In this setting, our approach to fuzzy multi-dimensional score evaluation is complicated by the required content and metadata information being distributed across the community and the lack of any centralized indexes.

It is our goal to allow a user to search the shared content of a federated system in the same manner that they would perform a local search on a single node. We seek an approach that can be integrated with all of the previously outlined work and design goals (i.e., file system and availability model). This offers a clear management benefit when coupled with the global namespace of the Wayfinder file system; users will have an improved method for locating relevant information remote nodes (or just within their own device set) while not needing to concern themselves with the physical location of the actual data.

We attempt to employ a standard approach to distributed query evaluation. Given a query $Q$, we first choose a set of candidate nodes that may contain data relevant to $Q$. 
$Q$ is then sent to this set and evaluated locally. Any results are returned and merged to produce the final set of relevant answers. The difficulty lies in the computation of the relevance scores during query evaluation. No single node is likely to have a complete snapshot of the shared content and, by association, all of the necessary scoring information. Thereby, locally independent query evaluation across nodes is likely to results in variety of scores, possibly for the same data.

Further, the computation of any IDF-based relevance scores is complicated by the existence of distributed replicas. In the Wayfinder file system, replicas can be created either explicitly through common file system operations or implicitly by the action of our automatic availability framework. In either case, this can result in an uneven number of replicas for individual files, resulting in relevance scores being skewed. Any scoring approach must either account for these replicas or be tolerant of them. In this thesis, we have opted for the latter approach by treating each replica as a separate file. We hypothesized that this will have minimal effect on the quality of query results. Our evaluation confirms this.

We propose an approach where each queryable node maintains a global index of approximate scoring information. More specifically, each node individually computes a summary of its local scoring information in a replica-oblivious manner; that is without information to discern if files are accounted more then once across several nodes. These summaries are then communicated, collected, and aggregated throughout the community. Query evaluation then utilizes this global index when computing relevance scores. Intuitively, if all nodes have the same global index, any compute scores should then be comparable thereby allowing distributed results to be merged meaningfully.

Our work investigates approaches to efficiently compute the necessary indexes and to quantify the impact that skews in file replication may have on the results of queries. For the latter aspect, our evaluation consists, in part, of a comparison between evaluating queries in a federated and single node environments. The results demonstrated that using global indexes alters the final results only slightly, with changes being concentrated among the lower ranked files. Further, we demonstrated that a simple encoding and aggregation techniques can significantly reduce the communication overhead of
exchanging summary information without greatly affecting results.

1.3 Contributions

The contributions of this thesis are as follows:

- We studied the viability of using a loosely consistent query-based data store for building abstractions and tools for managing information in a federated environment.

- In Chapter 3, we present a design for a federated file system that has among its features a global unified namespace constructed dynamically by merging the information stored on individual devices. This namespace provides location independent naming of files and ensures continuity in how information is accessed by providing a uniform view of information across changes in a device’s connectivity status.

- In Chapter 4, we present a design for a three part availability model to capture the availability needs of a community and its individual users. We implemented and evaluated a single replication algorithm that achieves this model while prioritizing the specific needs of users on their personal devices.

- In Chapter 5, we present a design and evaluation of a unified multi-dimensional scoring framework for a single node. This framework allows users to provide flexible query conditions on several dimensions of information relating to files.

- In Chapter 6, we present a design and evaluation of an extension to our multi-dimension scoring framework to allow distributed query evaluation over the shared content of a federated community.

- We implemented and evaluated all of our ideas and designs in a prototype implementation of a federated file system called Wayfinder.
Chapter 2

Background and Related Work

In this chapter, we begin by presenting a background discussion of federated systems and a federated toolkit that was used in our work. We then conclude with a discussion of related work in areas relevant to this thesis.

2.1 Federated Systems

Generally speaking, federated systems are collections of distributed computing devices located across administrative domains. Participation within these systems provides access to resources, and/or information, being shared through the collaborative efforts of the individual member devices. This access is often provided through a unified abstraction (i.e., a global namespace in the case of a federated file system) which attempts to hide the complexities of the underlying distributed environment.

Federated systems are distributed systems with the following characteristics. First, \textit{each device contributes local resources} to be used by the system. These resources are shared by integrating them as part of a globally unified abstraction which is accessible to all devices. This is in contrast to a system in which devices obey a strict server-client model (e.g., an NFS file system) or systems in which resources are partitioned (e.g., multi-tiered web-server).

Second, the members of a federated system \textit{are relatively autonomous} with regards to their actions and local resources. Devices may reside in separate administrative domains, be geographically distributed, or may simply not be very dependent on one another. In this regard, these systems distinguish themselves from tightly coupled distributed systems (i.e distributed databases).
Third, interactions between individual devices are governed by a set of communal protocols. A device’s participation in the system often requires adherence to these protocols which govern per-device behavior and inter-device communication.

Finally, the dynamics of federated systems are influenced by external factors. External incentives for participation in the system (e.g., the desire to share information or job requirements) encourage a majority of the users to allow their devices to remain actively involved. This helps avoid widespread “free-loading” which is one form of a common dilemma encountered in many P2P systems known as the “Tragedy of the Commons”. This dilemma argues that a shared, yet limited, resources can be destroyed if all users act of their own self-interests, even if its destruction is not beneficial in the long term. This is avoided through external relationships (e.g., friendships and occupational) that may influence the resource dynamics by imposing policies or rules not articulated in any protocol. For example, users participating in federated file systems may agree on a structure for a shared global namespace. While not needed for proper operations, the alignment of these external factors typically help to improve the overall user experience.

In this thesis, we are specifically targeting federated systems that allow sharing of information within a community of users. It is our belief that the dynamics of sharing information on a large scale in this setting is poorly understood. Aside from several well-known peer-to-peer (P2P) communities (both past and present) no large scale systems have been deployed or are actively being used by the general public. Arguably, the reasons for the large size of these P2P systems (i.e., tens of thousands of users) is the simple data model (publish and download), users’ tolerance of minimal guarantees (both for performance and the availability of information), and the nature of the information being shared (i.e., illegally distributed content). These factors align themselves to create an environment in which popular content is heavily replicated and easy to find. Rare content requires more effort to locate but the reward of obtaining a copy for free is often sufficient incentive. We believe such environments are poorly suited for users with complex information management needs.

Instead, we specifically focus on communities whose sizes range from tens to several
hundred devices. It is our belief that information exchanges in these smaller communities are likely to be more complex and diverse, and hence more interesting. Examples of such communities may include:

1. Social groups to share information among family and friends.
2. Company infra-structure for sharing information between employees.
3. A single user desiring to share information among their personal devices.

In addition to the community size, we make several assumptions about our target environments that affect the design choices presented in later chapters. First, we assume that a community’s membership is relatively stable over the short-term. We do anticipate changes in membership, but given our target size and the motivations for forming communities, it seems unlikely that large changes will occur in a short period of time. Second, devices may change their connectivity status at any time and users may continue to use a device regardless of its connectivity state. Finally, we expect that the average user will have multiple personal devices and may use them all to participate in a federated system; the behavior and information on these devices will exhibit strong correlations.

2.2 PlanetP

In this section we present the PlanetP toolkit [16] which provides the foundation for the construction of our federated communities. We present only the features of PlanetP that are central to the design choices made in this thesis. To this end, PlanetP provides four useful abstractions; a gossiping-based communication module [16, 21], a communal membership directory, an data storage system and distributed query processing engine, and a lightweight active Distributed Hash Table (DHT). Previous work [16] has shown that these abstractions are sufficient for building useful federated systems based on weakly consistent global state. We will discuss each abstraction in turn.

PlanetP uses a gossip-based communication protocol [16] to replicate and maintain shared state (see below) across all members of a community. This protocol includes
both anti-entropy, where two devices perform a complete information exchange, and
rumoring, where a device push out new information. While the choice of the communication partners is random, gossip-based communication has theoretical properties that allow information to reach a large portion of a community in a relatively short amount of time.

To maintain a community, PlanetP supports a shared communal membership directory known as the Global Directory. This structure contains information about every member of a PlanetP community. Each instance of PlanetP maintains a local version of the Global Directory which is updated with new information as needed. For example, suppose Device_A detects that Device_B has come online. Device_A would update its local global directory to show Device_B’s status as being online. The above mentioned gossip-based communication is then used to keep all the local instances of the Global Directory consistent. Continuing with our example, all the other members of the community would eventually learn that Device_B is online.

PlanetP’s data storage system provides the abstraction of a distributed table. In particular, this distributed table stores bindings of the form \{k_1, k_2, ..., k_n\} \rightarrow s, where \(k_i\) is a text key, \(s\) is a textual fragment, and we say that \(\text{keys}(s) = \{k_1, k_2, ..., k_n\}\). Retrieval of stored bindings is done in a device-independent manner by specifying queries over the distributed table. These queries are comprised of text keys combined using three operators, and (\&), or (\lor), and without (\-). For example, a query (“cat” \& “dog” \- “bird”) would retrieve the set \(\{o \mid (\{\text{cat, dog}\} \subseteq \text{keys}(o)) \land (\{\text{bird}\} \not\subseteq \text{keys}(o))\}\).

When a binding \{k_1, k_2, ..., k_n\} \rightarrow s is stored at a particular node, PlanetP actually stores the information in a persistent local table. Associated with this table, PlanetP maintains a summary table that stores a collection of summaries other nodes. Each stored summary is obtained from a remote node and represents information about the bindings stored locally on that node (i.e., the contents of the local table belong to its instance of the distributed table). These summaries assist in identifying nodes that may contain a bindings for a particular key. More specifically, given a key \(k\), the summary table can be used to determine the set of nodes that have at least one published binding for a string \(s\), where \(k \in \text{keys}(s)\). This table is replicated across the community and
Figure 2.1: Layout of storage system in PlanetP. Shown is the distributed table and the gossip-based communication component for a single running instance. The latter component ensures that any updated information is received and shared with the rest of the community.

kept loosely synchronize using PlanetP’s gossiping-based communication.

Given these two tables (i.e., local and summary), (Figure 2.1), PlanetP answers a query over the distributed table by first using the summary table to identify the subset of remote nodes that contain keys relevant to the query and then communicating the query to these nodes. Each target node then independently evaluate the query against their local tables and return all matching strings to the querier.

The individual node summaries used in the summary table are implemented as Bloom Filters [7]. A Bloom Filter consists of a finite sized bit vector and a set of hashing functions whose ranges map to locations in the bit vector. Values are stored by hashing them to locations in the bit vector using the set of hashing functions. The bits at these locations are then set. Searching a Bloom Filter for a value proceeds similarly except the bits are checked to see if they are already set. If all the necessary bits are set, then a value may have been stored in the Bloom Filter. This process can generate a result that is a false positive but never one that is a false negative. Bloom
Filters have mathematical underpinnings that allow the probability of this occurring to be controlled by altering the length of the bit vector. When used as described above, this representation provides a compact summary of the keys in the local table with an underlying theoretical model that can be used to guide trade-offs between accuracy and space.

To complement the gossip-based persistent data store, PlanetP also implements an active unreliable Distributed Hash Table (DHT). This DHT is active in that stored objects can execute on the hosting node to alter their stored state but unreliable in that objects are not replicated and so can be arbitrarily lost if nodes leave (or fail) without redistributing their portions of the DHT’s content. There are two expected use of this DHT: a rendezvous point for reducing the number of potential conflicts as concurrent operations will attempt to communicate with the same DHT object and caching of any soft state to enhance performance.

To support the work presented in this thesis, we have made several modifications to the original design of the PlanetP toolkit as presented above. In our work, we encode
and store complex information using a structural representation (i.e., XML). Queries must account for this structure in their conditions. The storage and query model of PlanetP are insufficient for this. To this end, our changes extend PlanetP’s storage and query capabilities by allowing additional storage structures and more complex query evaluation (Figure 2.2). First, we expanded the storage system to allow additional user-defined distributed tables. Each additional table independently behaves as described above. Internally, these user-defined tables may use one or more persistent data structures to store information internally. Query evaluation proceeds as before with the query being evaluated against each defined table separately.

Second, we altered the interface of PlanetP to accept self-contained XML documents rather than the previously discussed tuples. To account for different types of XML documents, we create several different global tables to store them. When published, a document is parsed by PlanetP and, based on the existence of predefined XML tags, stored in a table specifically designed for it. For example, an XML document containing the tag “<ContentSummary>” will be sent to a global table tailored for storing content information. This mapping of XML document types to tables is defined when the table is created. It is the responsibility of the storing table to store the XML encoded information persistently and return it when queried for. Note, the XML encoding is required only when transferring information to and from the table and may not reflect how the information is actually stored.

Finally, given the alterations in the data model and storage framework, we altered the query language to support a simplified version of the XQuery query language [77]. It is simplified in that the we only support the portion of XQuery that allows us to express

---

```
FOR $i in /Entity[FileSysMetadata/extension = 'pdf']
    FOR $j in /Entity[ContentSummary/WordInfo/Term = 'time' OR ContentSummary/WordInfo/Term = 'machine']
    WHERE $i/fileID = $j/fileID
RETURN $i/FileSysMetadata/@id
```

Figure 2.3: Example XPath query to retrieve information from PlanetP.
independent predicates (i.e., FOR Statements), perform a join operation over the results (i.e., WHERE Statement), and finally return a result (i.e., RETURN statements). Figure 2.3 presents an example query exhibiting all of these aspects. The XPath expression associated with each FOR clause defines a particular table in PlanetP over which the given predicate is evaluated. For example, the first expression in Figure 2.3 contains the predicate \texttt{extension} = 'pdf' and the path /Entity/FileSysMetadata. This will result in a search of the PlanetP table storing file system metadata (presented in Chapter 3) for all files having an extensions of “pdf”.

### 2.3 Distributed File Systems

Wayfinder is a distributed file system; it allows for the managing and locating of files on remote devices through a file system abstraction which hides the underlying distributed nature of the environment. In this regard, the Wayfinder file system is related to a wide range of existing cluster-based and distributed file systems research.

Cluster-based systems are comprised of homogeneous nodes within a single administrative domain. As a result of this environment, cluster-based systems such as XFS [3], Network File System (NFS), and Frangipani [73] make strong assumptions concerning the integrity and integration of their nodes. These systems cannot tolerate a network partitions and all assume low latencies for network communication. Wayfinder’s target environment invalidates may of these assumptions by assuming that devices can span multiple administrative domains or be geographically distributed.

Given this degree of decentralized and heterogeneity, the most relevant work to this thesis with respect to file systems is from the Peer-to-Peer community. This community has developed several data sharing system such as CFS [18], Pasta [51], Ivy [52], and Pond [61]. These systems are build using fault-tolerant distributed hash tables (DHT) [17, 62, 67, 84] that employ a key-based routing scheme based on consistent hashing [41]. Wayfinder distinguishes itself from these works in two major ways; the target environment and the underlying storage model. Wayfinder’s target environment is small to medium size (10’s to 100’s) communities while the target environments of
the DHT-based systems are large communities (thousands to millions of nodes). As a result many of our design choices relating to scale differ.

Furthermore, Wayfinder utilizes a dual-arrangement for storing information in which file content and file metadata are stored separately. Specifically, the local storage of a device stores shared content information (in the form of whole files) and a locally running instance of PlanetP stores any file system metadata. Remote devices then use PlanetP to access the shared metadata to learn of, and retrieve, any locally stored content. We employ this dual approach for several reasons. 1) storing files in their entirety at locations where they are used simplifies dealing with disconnected operation. A file has to be available in its entirety to be useful offline. 2) Separating the storage of file content from metadata ensures a user has continued access to files outside of PlanetP as files are stored in a usable form in the local file system. This may be necessary if a PlanetP community is dissolved or a user no longer wishes to participate. This approach is distinct from the above mentioned P2P systems which store both file content (often at the level of individual blocks) and metadata in the DHT. A user must therefore be connected to the DHT to access even locally stored information and the usage of local storage is governed solely by the key-based routing scheme.

Two particularly relevant projects to our work are Farsite [1] and Pangaea [63]. Both projects build a federated file system over communities of nodes. Both systems, however, target a much larger community size than Wayfinder.

Farsite is a general read/write server-less file system for a corporate environment that serves to replace the traditional centralized file server. It is assumed that this corporate environment is comprised of nodes that are relatively homogeneous both in their resources and availability. Farsite utilizes idle resources across desktop machines together with Byzantine-fault tolerant protocols and encryption techniques to ensure both consistency and security. The Byzantine-fault tolerant protocols allow Farsite to implement a much stricter consistency model than what is possible in Wayfinder given our support for disconnected operation and target environment. However, this benefit comes at the cost of a more complicated design.

Pangaea [63] is a wide-area file system designed for large communities of users
spanning the globe. This work attempts to minimize the use of the wide area network (i.e., the network economy) during file system operations by considering the physical proximity of nodes.

Total Recall [6] is a project focused on providing availability for data in P2P systems. Part of this work involved the construction of a distributed file system that used the Total Recall availability architecture. We present a more detail comparison of this work below in our discussion on availability.

2.4 Global Directory Structure

A key aspect of the Wayfinder file system is the construction of a shared global namespace, an idea which has been explored by several previous projects. Most distributed file systems mentioned above maintain a global namespace [1, 18, 51, 52, 63] for their users. In contrast, Wayfinder’s contribution is performing this construction in a distributed setting by merging the local namespaces of individual devices. A similar approach is used in the Federated File System [68] which recursively merged the local namespaces on a set of cluster nodes in a tightly integrated and highly available cluster environment.

In contrast to our approach of using the global namespace as a communal resource for managing information that is collaboratively organized, several projects have explored using more personalized organizational structures; in particular the Prospero file systems [54], the Jade File System [59], and the Pasta storage system [51]. In all of these cases, the work focuses on providing users with personalized views of information. More specifically, Prospero and Jade allow users to link together distributed resources that are accessible over the network into a single personalized namespace while Pasta permits independently manipulate of directories in the global namespace to create a personalized directory structure.

2.5 Consistency Model

Numerous projects have experimented with weak consistency models in the context of file systems [36,44,57,63]. In this section, we discuss several that share similarities with
our work, namely Bayou [57], Ficus [36], and Pangaea [63].

The Bayou [57] system supports data sharing among a collection of mobile devices in which nodes have less than ideal connectivity. Access to information is provided via a complete replica of a shared database stored on each individual node. Changes to any copy of this shared database is communicated to other nodes using a form of gossiping-based communication. To ensure agreement among all nodes as to the ordering of updates, Bayou designates one server as a primary commit server. Wayfinder differs from Bayou in that nodes are not required to retain complete copies of the shared state and our method for reconciling updates does not require a primary commit server.

The consistency model of Pangea and Ficus are very similar to that of Wayfinder. Both support a single-copy availability model but differ primarily in how the eventual consistency of information is achieved. For this Pangea [63] maintains an abstraction called a replica set. This set defines the locations of replicas for a particular file, or directory, and are used to construct per-replica connected graphs. Given an update, the links of these graphs form a multi-cast tree to quickly propagate diff information to all replicas. Alternatively, Ficus relies on periodically running several community-wide two-phase algorithms to ensure the all replicas are brought up-to-date. These approaches are in contrast to the continuous push/pull model employ in Wayfinder. Similar to Wayfinder, both Ficus and Pangea rely on version vectors to detect concurrent updates.

2.6 Disconnected Operation

In our support for disconnection operation we are similar to the Ficus File System [36], Coda File System [44], and the Pangaea file system [63]. In comparing these works to our own, we will refer to the three stages a node must pass through as it transitions between connected and disconnected states. These stages are information hoarding, actual disconnection, and re-integration.

In the case of Coda, our work differs in the approach to re-integration. Specifically, Coda requires each user to explicitly specify the set of files to retain on a per-machine
basis. This is similar to a user specifying offline availability targets in our availability model. Once disconnected, a device has access to any files that were hoarded in this manner. However, unlike Wayfinder, upon reconnection a Coda device must synchronize any changes to its local storage with a server node for them to be considered as committed.

In Ficus, objects are organized into volumes that are replicated across nodes. During disconnected operation, access is limited to the content of any replicas that exist within these locally replicated volumes [45]. It is not clear if user-defined hoarding is possible as this would require user-level control of volume placement and content. For reconciliation, Ficus relies on several communal two-phase algorithms to ensure that all updates are considered and committed.

Wayfinder also resembles Ficus in the overhead associated with creating replicas. In Coda, the creation of replicas on devices amounts to storing a cached copy. The creation or removal of these copies requires little or no administrative overhead on the part of the system since any necessary state processing is confined to the client. In Ficus, however, there are no cached replicas. Each replica is maintained and monitored by the overall system and their creation and removal requires updating system state [45]. The same is true for Wayfinder as all replicas are monitored and have published state.

In the Pangaea File System there is no method for a user to predefine which content should be hoarded on a device. Instead the contents of a node’s local store at the time of disconnection is determined by the replication requirements of the system and a user’s explicit activity. Upon reconnection, changes are propagated using the previously mentioned per replica graphs.

The Seer project [46] attempted to improve the usefulness of hoarding algorithms by leveraging semantic distance and clustering algorithms for prefetching content. This work is complementary to our own in that we can leverage their approaches to improve Wayfinder’s hoarding algorithm (currently based on LRU) for improved offline availability.
2.7 Content-based addressing

Content search is a useful tool for finding information in file systems [48, 49] and we explore its use as an organization tool by allowing queries to be embedded persistently in a global namespace as semantic directories. These directories are populated by files deemed relevant to the query. The original idea of semantic directories was introduced by the Semantic File System [30]. This work was later extended by the HAC File System [33] by allowing semantic directories to be embedded into an existing hierarchical directory structures forming a hybrid namespace. Wayfinder has further extends this line of work by allowing a construction of a similar hybrid namespace in a distributed environment.

2.8 Personal Information Search

The Wayfinder file system contributes to a growing body of work that investigates personal information search. Much of this work has focused on a single device/user settings. The SEMEX [10] and Haystack [42] systems allow users to specify semantic associations between pieces of data. These associations can be leveraged to improve information organization and the locating of relevant information during searches. The work done by Soules et. al [65] attempts to identify related information specifically based on the context in which it was accessed. This context information can be used at query time to locate additional relevant information.

Other works [22, 78] propose generic data models capable of storing and retrieving heterogeneous and evolving data. The Nebula [9] project explores designing a file system using an object-oriented database system as the underlying storage mechanism.

These works are aimed at creating, identifying, or accessing additional information beyond what is supported by traditional file systems. The Wayfinder file system focuses on using data that is already present in the file system and returns results based solely on the conditions of a user-provided query. While Wayfinder does deviate from traditional file systems in how metadata is stored and managed, our approach is not as generic as those mentioned above.
There has also been a recent surge in projects attempting to improve search capabilities specifically in the desktop environment [13, 29, 66]. These projects provide search capabilities over content and then employ other pieces of information, such as size, date, or file type, as filtering conditions. We argue in later chapters that this approach is insufficient in many search scenarios and addressed this deficiency with our work.

2.9 Distributed Information Search

Numerous works have explored the improvement and designing of search technologies in a distributed environment. Among these are distributed IR Systems such as GLOSS [34] and PlanetP [16]. The latter of which is the basis for the toolkit upon which the Wayfinder file system is built. Others explore search technologies in the context of P2P systems [5, 12, 38, 55, 60, 69–71, 82, 83]. These works explored methods for building distributed online indexes with efficient techniques for query evaluation. Their primary focus is on providing content search, whereas our work in improving search techniques explores the use of multiple query dimensions. Further, to reduce storage and communication costs, we explore methods for reducing the overall size of the global index through compression and aggregation. This is done in a manner which aims to minimize the impact on the quality of results. Of the mentioned works, several [69, 70] have explored the use of Information Retrieval techniques to improve search. Various other projects have also explored the application of Top-K [26] evaluation techniques in a distributed setting [4, 11, 50, 74].

In this thesis, we explore the design of a framework for evaluating multi-dimensional queries in a federated system. Query evaluation is performed in a manner that allows results to be scored and ranked using a global approximation of the necessary statistics. The closest work to this approach is pSearch [70]. This project investigates the design of a scalable P2P IR system using DHTs and IR algorithms. As with our approach, this framework requires maintaining global statistics for the purpose of scoring. However, contrary to our approach of computing the necessary information on each node by gathering and aggregating individual node summaries, pSearch pre-computes and maintains statistics based on samples that are representative of the potential document
2.10 Availability

The availability work presented in this thesis builds on a previous effort by Cuenca et al. [15]. This earlier work considered the task of ensuring the continuous availability of information in highly dynamic environments and proposed a replication algorithm in which devices make autonomous replication decisions based on a small amount of loosely consistent state. We have extended this work to support multiple devices per user and dynamic content. Specifically, our extensions include support for additional types of availability, permitting changes to the hoards of individual devices (i.e., adding or removing files), enabling the loose coordinate of actions for multiple devices belonging to a single user, and accommodating mutable shared file content (i.e., writing files).

Related to our work are many of the above mentioned P2P file systems [18, 28, 51, 52, 61], each of which takes measures to ensure the availability of information. The main difference between these efforts and our own is that they consider only online availability whereas Wayfinder supports a unified availability model.

Two systems that are particularly relevant are Total Recall [6] and Farsite [1]. With respect to availability, Wayfinder differs from these projects primarily in how availability is monitored and in replica placement. In this regard, Wayfinder’s approach consists of continuously monitoring the availability levels for both files and nodes and performing replica placement randomly. Also, as with the previously mentioned P2P file systems, both only consider online availability.

Of the two, Total Recall is the most related to Wayfinder. Total Recall is contemporary work to our own that also explores the problem of maintaining online availability of files for P2P content sharing systems through replication. The degree of replication required is determined using two types of predictions; a short-term availability estimation that accounts for transient node departures and a long-term prediction to account for non-transient node failures. This dual approach is different from that employed by Wayfinder (i.e., continuous monitoring and random placement). Both Wayfinder and
Total Recall assume communities with devices of heterogeneous availability.

Since Wayfinder replies on continuous monitoring, failures may be detected and responded to quickly. If the failures are transient, any additional replicas created as a result of this detection may be unnecessary in the future, resulting in over-replication. For permanent failures, this method results in a faster time to replace any lost replicas.

Recall that the purpose of the Farsite project is to replace a centralized file server with distributed resources. To this end, Farsite attempts a stronger consistency model than Wayfinder. Specifically, directory information is replicated across nodes running a Byzantine-fault tolerant protocol to ensure agreement. Furthermore, Farsite attempts to achieve an even distribution of availability for all files by replicating all files equally and giving considerable thought to where (i.e., which nodes) the replicas are placed. This is possible given that Farsite targets relatively large and stable environments (i.e., corporate and academic) with availability higher than that of most Internet hosts [1].

In contrast, Wayfinder’s approach (i.e., continuously monitoring and random placement) and our target environment (nodes with varying degrees of availability) can result in files having significantly different replica counts. Furthermore, given Wayfinder’s champion nodes, a file is only replicated if there are resources (i.e., a user and his devices) who are interested in the file. Thereby, Wayfinder does not ensure the availability of all files, but rather only those that have online advocates.

Pangea [63] maintains availability through the per file/directory replica sets mentioned above. The membership of these sets define a minimal replication factor for files/directories. A subset of each set is defined to be Golden replicas and can not be deleted. Furthermore, as mentioned earlier, the availability of latest version of a file is increased by propagating changes through these per-replica graphs built over each replica set. Such per-replica graph construction would be difficult during periods of partial connectivity which are supported in Wayfinder.
2.11 Cooperative Caching

For several aspects of its design, Wayfinder employs whole file replication to provide improved access to information. Each device maintains a region of local storage to store these replicas for local use. The placement of files into this region is governed in part by a user actions and the various running sub-systems of the file system (See Chapter 4). Files can be evicted from these local stores at any time (with certain restrictions on certain nodes) as needed. This region behaves essentially as a local file cache with information begin retained by the community so long as at least one cached copy persists.

In this regard, the Wayfinder file system is similar to work done on Cooperative Caching [19] which leverages resources on remote machines to store data. The difference in our work are several. Cooperative caching systems assume LAN networks (i.e., high-speed and low-latency) and share memory resources to store fixed size blocks of data. These systems use various algorithms to track replicas and ensure write-consistency. In contrast, the devices in a wayfinder file systems share disk storage and may be distributed over great distances (both in terms of network and geographically). Further information is shared at the granularity of whole file and we assume a weak consistency model for writes. Finally, replicas are tracked through small amount of weakly consistent state that is shared amongst the community.
Chapter 3

Federated File System

3.1 Overview

In this chapter, we present the design of the Wayfinder federated file system and evaluate the performance of a prototype implementation. Wayfinder enables a federate community to persistently store and share data using two complementary abstractions: a global namespace and persistent search queries. We shall see in this chapter that these two abstractions allow information to be located, and organized, by both name (i.e. name-based addressing) and content (i.e., content-based addressing).

To store the files to be shared via the file system, we assume that each device provides a portion of its local storage system. This space is called the device’s local hoard and consists of the directory structure and files containing all the information a device wants to share. Given a set of connected devices, $S$, we construct the global namespace by overlaying and merging the directory structures found in the individual hoards as shown in Figure 3.1. This merged namespace is then accessible to every device in $S$.

We adopt the above merging approach for two reasons. First, it provides users with a consistent browsing (i.e., name-based addressing of files) experience by ensuring that any file can be named in a device independent manner. Secondly, a merging approach naturally encompasses connected, partitioned, and disconnected operation as shown in Figure 3.1. At the two extremes of connectivity, a device will either have access to the entire namespace or be limited to its local hoard. During partitioned operation a community may be split into multiple connected subsets. Each subset will independently construct an isolated namespace from its member’s hoards. This approach ensures that
Figure 3.1: Wayfinder dynamically constructs a shared namespace across any set of connected devices by merging their local hoards. This figure shows 5 nodes originally being connected so that the shared namespace is the merged view of hoards H1 through H5. When the community is partitioned into 3 connected subsets, Wayfinder maintains a merged view for each subset. When the subsets reconnect, Wayfinder dynamically re-merges the shared namespace. Shaded boxes indicated connected partitions.

with changes in connectivity, the amount of accessible content may change but the manner in which users browse and access files does not; any reachable file remains so through the same name.

To complement the above name-based approach, we have implemented content-based addressing and organizing by implementing search capabilities which include the use of persistent queries that can be embedded in the namespace via semantic directories [30,33]. A semantic directory is a search query that is embedded persistently in the namespace in the guise of a normal directory. The content of such a directory is the set of relevant files returned when evaluating the associated query. In this manner, semantic directories allow the automatic name binding of files based on their content and other attributes. For example, a paper dealing with federated file systems will be automatically placed in a semantic directory with the query “Federated, File, Systems” based solely on its content. Given that these queries are persistent, it is possible for users to preserve “good” queries and share them through the global namespace.
A semantic directory is periodically reevaluated to reflect changes in the shared data set. This re-evaluation turns the file system namespace into an active organizational tool, allowing it to automatically classify new information entering the community. Further, similar to the HAC file system [33], Wayfinder allows users to refine the query results of a semantic directory through direct manipulation of the directory’s content as an alternative to manipulating the original query until the exact set of desired files are returned.

In the next section, we outline several scenarios that illustrate Wayfinder’s benefit in simplifying a user’s management role. We then discuss Wayfinder’s design and prototype implementation. In particular, we will show that an important aspect of our design is that its abstractions are all implemented as queries against our federated toolkit PlanetP (Section 2.2). Therefore, the primary difference in implementing either the name-based or the content-based addressing is the manner in which the appropriate query is formulated. We also describe how Wayfinder utilizes PlanetP’s light-weight distributed hash table (DHT) as a caching infrastructure to make this query-based design efficient.

3.2 Example Usage

In this section, we present two scenarios that demonstrate the usefulness of the abstractions outlined in Section 3.1. We believe these scenarios to be representative of common situations encountered in the workplace.

3.2.1 Usage Example: Wiki

A Wiki is a collaborative web-based tool for maintaining a set of shared web pages. Users can independently access, upload, and edit information within these shared pages. While a wiki provide a useful tool for collaboration, it does presents several information management difficulties.

First, information stored within a wiki can only be accessed and manipulated
through wiki-specific tools and interfaces, requiring users to learn and use them. Second, users must explicitly partition their data between two separate logical information domains (i.e., their local file system and the online wiki). Unless users make exclusive use of only a wiki, this partitioning often requires at least a portion of a user’s the information to be replicated across the shared web-pages and their local file system. Third, in dealing with separate information domains, users are forced to use disjoint tools for searching and managing information (i.e., the search of the online wiki can not also search the local file system and vice-versa). Finally, as a wiki is web-based, it requires continuous network connectivity to access.

As an alternative approach, the web pages contained in a wiki can be maintained as a shared file system using Wayfinder. This shared file system can be integrated directly into a user’s local file system (We will show later (Section 3.8) that our prototype implementation can be mounted within the namespace of existing file systems). Users can then access any shared information as local files while continuing to use their standard set of applications (i.e., vi, emacs, Frontpage) to manipulate them. Wayfinder will maintain the illusion of a shared workspace by automatically replicating shared information and updates to other users as necessary. Finally, if a user still desires to view information in a web-based format, a locally running web server can be used.

This scenario demonstrates how a user might leverage Wayfinder as an alternative to managing shared information across multiple logical information domains. By using Wayfinder’s abstractions, this task can be simplified to reasoning about a single domain (i.e., the local file system) while still retaining the features (i.e., publishing to interested parties) necessary for collaboration.

3.2.2 Usage Example: Shared Search Structure

We now present a scenario illustrating how a small research group might use Wayfinder to collaboratively manage information across several users and devices. This scenario is motivated by a research group’s effort to share the efforts of one student’s work in researching file-systems. We will refer to the student in question as Alice and assume that she has a locally running instance of Wayfinder mounted in her local file system
We begin our example with Alice finding eighteen papers pertaining to the topic of “file systems” that she considers relevant for the group. She places these files into her local directory structure under a Wayfinder directory `/shared/Alice/papers/worthKeeping`.

Having found the papers, Alice is now faced with the organizational task of placing the papers (i.e., as files) into the namespace in a manner that is conducive to the remainder of the group finding them should they browse the namespace. We will show in this scenario that this can be accomplished by creating several semantic directories, examining the results, and making changes as necessary.

Continuing our example, Alice next creates a semantic directory ”file systems” within the directory `/shared/group/papers`. This semantic directory will become logically populated by all the files in her original set. We shall see later that a query represented by a semantic directory is a boolean expression over the terms associated with the semantic directories in the pathname; in this case, it is a logical conjunction of the terms “file, systems”. In addition, Wayfinder adds several files matching the query “file, systems” from another student’s desktop and from a professor’s laptop in the directory `/shared/profs/Bob/needsReading`. At this point, we can observe that Wayfinder allows Alice to not only organize her own set of papers, but also to include relevant papers from other users.

Alice then further refines the file-system area by creating an additional semantic directory `/shared/group/papers/file systems/replication` (in this case the query terms corresponding to the new directory are “file”, “systems”, and “replication”), which finds all the files containing the word replication from the set of papers in `/shared/group/papers/file systems`. In our example, this prunes the original set of papers to ten. She refines the query again by creating a further semantic directory `/shared/group/papers/file systems/replication/optimistic` to find papers concerning optimistic replication. The resulting directory might contain papers describing the Farsite [1], Ficus [36], and Pangaea [63] projects.

Alice next creates a semantic directory with the path `/group/papers/file systems/ficus` intending to find any papers primarily dealing with
the Ficus File System [36]. Wayfinder returns a directory with seven files. However, she decides only two are relevant because the rest mention Ficus only in the references. She deletes the unwanted files from the semantic directory, which results in Wayfinder excluding those files as results in the semantic directory. These deletions are localized to the semantic directories; the files in their original locations are unchanged.

We close our example with a professor needing to select a paper for the group’s seminar. He begins his search by browsing the /shared/group/papers/ directory on his laptop. Because of the merged view he sees the file system directory created by Alice. With further browsing, the professor locates the directory /shared/group/papers/file system/replication/optimistic. Based on the three papers in this directory, the professor names the discussion topic for the week “optimistic replication in file systems” and selects one of the papers. The location of this paper is then e-mailed to the entire group. As there is a single global namespace for all users and devices, the emailed location can be resolved by all the users.

While our example is simple, it demonstrates several possible uses of Wayfinder. First, it allows a single user to share a meaningful organization of content through a shared directory structure. Second, it allows automatic publication of the information by placing it in a namespace. Third, our example shows that content-based lookup is useful to organize a set of documents given that users may not be aware of the location of all relevant files. Fourth, it shows the merit of sharing the resulting search-based organization structure. In this case, the choice of keywords and query results was reused by the professor to organize his seminar. Finally, manual query refinement, which in example involved pruning results, is necessary as semantic directories can become populated with an excess of information.

3.3 File System Design

In this section, we describe the design of the Wayfinder file system, focusing specifically on details of storage and the implementation of files and directories. Recall that each device in a Wayfinder community manages an area of its local file system, known as the
hoard (Figure 3.2). The hoard stores the content (in files) and the namespace that a
device is sharing; both of which are a subset of what is present in the global namespace.
Each device also runs an instance of PlanetP that is used to store and search for file
system metadata; both local and remote.

For each file, or directory, \( f \) in a device’s local hoard we encapsulate the aggregate
state of \( f \) in an abstraction called a waynode. A waynode is a data structure storing a
file’s state which may include metadata, content summary, and structural properties.
Within a Wayfinder community, a single file (or directory) may have multiple replicas
residing on different devices. Each of these replicas is represented by its own distinct
waynode.

To optimize the process of storing a waynode, we allow its state to be partitioned
into disjoint segments, each of which summarizes one aspect of the file’s state (i.e., meta-
data, content summary, or structure information). Each segment is stored in PlanetP,
specifically in a global table maintained as part of PlanetP’s storage framework. The
design of these tables is detailed in Section 3.7. When passed to PlanetP for storage,
each waynode segment is encoded as an XML document.

Individual devices only maintain waynodes for files/directories in their local hoards.
To construct any global state, a node must retrieve the localized information stored
across remote nodes by queries to PlanetP. Returned results are then processed by the
querying node to form the global state, or view, of either a file or directory. Subse-
quent re-constructions are necessary to learn of remote changes to the state of a file or
directory.

This query-based model offers two benefits: (1) As we assume that a sizeable portion
of our target environment will consist of mobile devices, we anticipate a corresponding
amount of churn in online membership as these devices are used and turned off. There
is a fundamental trade-off between keeping a single view of an object’s state and al-
lowing each device to determine the state for itself (i.e., by querying for it). Given a
directory, \( d \), in the global namespace and \( \text{state}(d) \) representing all state related relating
to \( d \), requiring a single persistent copy of \( \text{state}(d) \) mandates that it be stored some-
where. This limits the use of \( d \) to devices with access to \( \text{state}(d) \) and should \( \text{state}(d) \)
becomes inaccessible, either because the hosting device is inaccessible or the accessing device is disconnected, then any available portions of the namespace dependent on $d$ is likewise inaccessible. (2) Finally, the query-based model allows an easy transition to disconnected and partition operation, as queries are evaluated only on connected nodes and so accurately reflect the available information.

In the remainder of this section we will describe the implementation of file and directories.

### 3.3.1 Files

Each Wayfinder file is defined by a globally unique identifier that is determined when the file is created. For a file to be present in the global namespace it must be located in the local hoard of at least one device in the community.

A file waynode contains the metadata and content summary of its respective replica. The metadata information consists of file attributes such as the file ID, version, a URL
(our prototype implementation uses a simple HTTP web-server to transfer files) indicating where the replica can be retrieved and other information commonly maintained by standard file systems (e.g., the information contained within an inode in the Unix file system). The content summary is information about a replica’s content. At a minimum, this information can be used (i.e., by PlanetP) to perform relevance ranking (i.e., $TF \cdot IDF$) for content queries. For storage, this information is separated into a metadata and content segments. Examples of the XML encoding of these segments is given in Figures 3.3 and 3.4.

A file’s location in the local hoard is a mirror of its location in the global namespace. Assuming the root of a local hoard is at the path `/WFRoot`, then a file stored in the
global namespace at /path/name would be stored locally at /WFRoot/path/name. Any access to a file's content requires a local replica. If an access is attempted and no replica exists, it is automatically retrieved.

For example, suppose that the hoard of a device $n$ is located at /WFRoot in $n$'s local file system. If a file /workspace/foobar.txt is opened on device $n$, Wayfinder first retrieves the ID for foobar.txt from the metadata associated with /workspace (see below) and queries PlanetP for the set of waynodes representing all replicas of foobar.txt (specifically the segments encoding the metadata information). From this set, Wayfinder computes the latest version and individual locations of any replica. Then, if $n$ does not have a local copy of foobar.txt, Wayfinder retrieves a copy, stores it at /WFRoot/workspace/foobar.txt, creates the a local waynode for this replica, and stores it in PlanetP. The new waynode contains the file's unique ID and has location information (i.e., URL) to reflect its presence $n$'s hoard (Figure 3.3). On the other hand, if $n$ has an old version of foobar.txt, Wayfinder updates the local state (both file and metadata) to the newer version. Finally, after a local and updated replica is present, Wayfinder completes the open operation. File creation works similarly except that Wayfinder generates a new file ID for the newly created file.

For operations writing a file replica, Wayfinder enforces open/close semantics. When a written replica is closed, the replica’s version number is incremented and the metadata segment of its waynode is updated. The replica’s content is then scheduled to be indexed in the background. Once the indexing is completed, Wayfinder updates any stored content information by updating the content segment of the replica’s waynode.
3.3.2 Directories

Recall that we construct the global namespace by merging the contents of locally hoarded directories with the same name. To simply this process, we ensure that any merge-able directories have the same unique global identifier; namely their full path-name.

A directory waynode contains metadata information and the name-binding information for all files/directories the local replica contains. The metadata information is similar to that of files, although smaller as some information is not needed (i.e., versions, size). The name-bindings represent the “content” of the directory. On a given node \( n \) and a directory in the local hoard, \( d \), for each file \( f \) that is child of \( d \), we store the parent-child binding \( d \rightarrow f \).

For storage, we partition this waynode into a metadata segment and a set of separate structural segments; one segment for each parent-child binding. The parent-child bindings are established during the creation of the a file/directory. By keeping this information as separate bindings, the creation process involves simply adding more structural segments to the necessary tables rather then updating existing ones. As with files, all segments are encoded as XML documents before being stored in PlanetP. An example of the encoding for a structural segment is given in Figure 3.5.

To determine the set of bindings for \( d \) in the global namespace, the appropriate query is place to retrieve all of the waynodes that contain as \( d \) as a parent value (e.g., any structural segments of any waynodes matching \( d \rightarrow * \)). Each device only stores information for the local parent-child bindings present in its hoard. It possible for a device to have only a portion of the global state for a directory present locally; thus we say that directories are partially replicated.

As mentioned earlier, every access to a file’s content requires a local replica and the namespace of the local hoard mirrors that of the global namespace. As result, the replication of a file \( f \) will naturally result in the replication of all directories in \( f \)’s path name.
3.4 Caching

A common operation in Wayfinder is the construction of file and directory views in response to a user’s action (i.e., the user performs an “ls” operation). Recall that underlying the construction of a view is a query to retrieve any necessary state, either local or remote. Performance would be unacceptable if such an operation required contacting a majority of the community for each traversal. Thus, we make extensive use of caching. In particular, Wayfinder caches a processed version of the query’s result both locally and in PlanetP’s DHT. Both copies are retained for a pre-defined amount of time; 10 seconds for the local cache and 15 minutes for the DHT-based cache. The latter state makes results available to the community as a whole and, as we will present shortly, is proactively maintained, allowing for a longer caching time. We examine the performance of a directory traversal in the presence and absence of caching in Section 3.8.

With the possible existence of pre-computed state for directories existing in the DHT, a lookup for a directory, $d$, on node $n$, proceeds as follows: 1) $n$ checks the DHT, using the identifier of $d$ as a key, for any pre-existing state. If found, the state is retrieved and used. 2) If no state is found, $n$ retrieves the set of all of the waynodes related to $d$, $W(d)$. 3) $W(d)$ is processed (i.e., removing duplicates, parsing information, and summarizing information) to produce $state(d)$. 4) $state(d)$ is cache locally and used to complete the lookup of $d$. 5) Additionally, the tuple $<d, state(d)>$ is then stored in the DHT with “d” being used as the key. A similar caching process exists for files.

Cached entries in the DHT are maintained as active objects that can receive and process information regarding updates. Any device adding, deleting, or editing a file represented by a cached entry, updates the cached entry to reflect the change. With the exception of the initial publishing of cached information, a node is only responsible for correctly publishing its portion of the global state. While cache views are continuously updated in this manner, they are periodically discarded to remove any stale data (e.g., from nodes that have left the community).

In summary, information concerning files and directories is made public in two
ways using PlanetP; either by publishing in the PlanetP’s global table or in the DHT. Through the DHT, it is possible for nodes to be made aware of changes before the global table has had an opportunity to fully propagated similar information.

### 3.5 Consistency Model

Wayfinder supports partitioned operation by allowing continued operation even when the sharing community is partitioned. Operating in such scenarios inherently introduces inconsistencies into the file system when files are manipulated concurrently by devices not in direct communication with one another. Inconsistencies can even occur during connected operation as Wayfinder does not attempt to coordinate operations using a single rendezvous point. This results in an inherent delay in learning about changes on other devices in the system. The caching model described in section 3.4 attempts to minimize any windows of inconsistency but is not required for normal operation. For these reasons, Wayfinder exports a weak consistency model, similar to that of Bayou [57] and Coda [47], for both directories and files.

This model allows for two types of observable inconsistencies. The first is a content conflict resulting from concurrent writes to multiple replicas of the same file. The second is name conflict resulting from concurrent creations of files with the same name bindings. An example of the latter would occur if two replicas of a directory on separate devices each contain a file with the same name but have different global identifiers. When constructing the global namespace, the merged directory will have two different files, each with the same name, resulting in a name collision.

Due to our weak consistency model both type of inconsistencies are possible during either connected, partitioned, or disconnected operation. In the remainder of this section, we will outline our consistency model as it applies to both files and directories in greater detail.
3.5.1 Files

For files, Wayfinder supports a single-replica availability model [36] in which access to a single replica is sufficient to access and manipulate a file. Upon accessing a non-local file, $f$, at a device $n$, Wayfinder will attempt to locate the latest version of $f$ and download it to the hoard of $n$. If $f$ is already stored locally, $n$ may try to determine whether the local copy of $f$ is out-of-date with respect to other online replicas and update it accordingly. Any file system operations on $f$ at $n$ are then performed on the local replica.

Under partitioned operation, this model can lead to users seeing stale data if recent writes occurred outside of $n$’s currently connected partition. Fortunately, previous studies have shown that the incidents of write-sharing among files in multi-user environments [44] tends to be a rare occurrence.

To address write conflicts, Wayfinder associates a history vector with each file replica. A history vector is an ordered sequence of tokens for uniquely identifying successive changes that have been applied to a file. Each change is identified by a monotonically increasing version number and the identifier of the editing device. For example, the vector $[(1, x), (2, y), (3, x)]$ would indicate a file that was edited three times on two different devices, $x$ and $y$, with no content conflicts. Changes to a file extend the vector (i.e., $[(1, x), (2, y), (3, x), (4, x)]$).

Potential content conflicts are detected when updating a local replica and learning of another replica with a conflicting history vector, indicating concurrent edits. Detection occurs when tokens within two vectors for different replicas of the same file have the same version number but different device identifiers. For example, in the vectors $[(1, x), (2, x), (3, x), (4, x)]$ and $[(1, x), (2, x), (3, x), (4, y)]$, we observe a concurrent edit after the third updates. In dealing with these conflicts, we pessimistically assume that any concurrent write indicates a conflict within the file. This is done even if the areas affected by the updates does not overlap as they may alter the file in semantically different ways rendering the content meaningless.

Given two conflicting replicas, Wayfinder attempts to determine which is the most
of up-to-date by identifying the one with the longest sequence of applied updates. Intuitively, this property represents the largest non-conflicting set of changes to a replica. Wayfinder then automatically reconciles the state of the second replica to that of the most up-to-date.

During this process, Wayfinder leverages the history vectors of each replica to determine which is the most up-to-date. Specifically, given two replicas of the same file with conflicting history vectors, we identify the *dominating* vector. The dominating history vector is defined as having the largest update count on the final token. For example, given $[(1,x), (2, x), (3, x), (4, x)]$ and $[(1,x), (2, x), (3, y)]$, the first vector would be dominating. Note, this is a poor measure of how much a file’s content may have changed as a single committed update could completely rewrite a file’s content. If the final vector entries are the same (this could occur with partitioned operation) or the numbers are tied, the dominating vector is chosen in a deterministic manner (i.e., lexicographical ordering). Once chosen, the conflict is resolved by updating the *dominated* replica’s content to that of the *dominating* replica.

With automatic reconciliation it is still possible to obtain a version of file that is semantically meaningless (i.e., the content is no longer useful to the user). For such situations, Wayfinder allows users to manually review the results of the automatic reconciliation process and create a new version of the file. We facilitate this by extending the information maintained in the history vector to include all the updates ever considered during reconciliation (see below) while also retaining copies of conflicting replicas outside of the global namespace. These copies can be used by users, or applications, to devise a more useful conflict resolution. For example, a tool with access to this
information could assist users in resolving any conflicts, construct a new version of the
file, and then commit this as a new update. The conflicting copies are retained for a
fixed amount of time (possibly several weeks) after which they are simply deleted.

Under this model, a replica’s history vector (Figure 3.6) can be divided into two
disjoint sets of updates; those applied and those in conflict. The first token associated
with each update number (i.e., 1, 2, 3, and 4 in Figure 3.6) represents the applied update
for that version number with conflicting updates listed subsequently. For example, given
the vector in Figure 3.6, the set of applied updates is \{(1, N_A), (2, N_B), (3, N_A), (4,
N_C)\} while the set \{[(2, N_B), (2, N_C), (3, N_B)]\} reflects past conflicts.

When determining the dominance of two history vectors, only the applied updates
are considered. After a conflict is resolved, the complete history vectors of both replicas
are merged to reflect both the change in content and the presence of the conflict. For
example, given two vectors for replicas of the same file, \[((1, x), (2, x), (3, x), (4, x), (5,
x))] for replica \(R_1\) and \[((1, x), (2, x), (3, a), (4, a))] for \(R_2\), \(R_1\) would be considered the
dominating replica and after reconciliation, both would have a vector of \[((1, x), (2, x),
(3, x), (3, a), (4, x), (4, a), (5, x))]\). This deterministic process allows devices to resolve
conflicts without resorting to a voting or distributed consensus protocol.

To reduce the likelihood of conflicts occurring, information stored in the PlanetP
DHT (Section 3.4) can allow a device to determine the state of a file with a single lookup
and also provides a on-line rendezvous point for file information.

### 3.5.2 Directories

Accessing a directory 
\path\dir in the global namespace from a device \(n\) requires either
the construction, or retrieval (from the DHT), of the associated directory view. Part
of the state associated with this view is set of the name-bindings for files contained
in \path\dir. These bindings are gathered from the online devices within \(n\’\)s partition
when the view is created. Therefore, a user can only access directory \path\dir if a
device in this partition has a replica of \path\dir.

The creation of a new name-binding in \path\dir can be done at any time. If a device
has a local replica of the directory, then the name-binding is added to the existing state.
If no local replica exists, a local replica of /path/dir is created and the name-binding is then added to it resulting state. In either case, the directory replica can then be merged with other partial online replicas of /path/dir.

As with files, a user may see inconsistent information when accessing a directory. These inconsistencies may include seeing name-bindings that do not exist, name-bindings that have been deleted, or name-bindings that are in conflict. The latter are detected during the construction of the directory view and are dealt with by a renaming process. For example, two different files having the same name, f, in directory d on device x and y will be renamed to /d/f−x and /d/f−y in the global namespace. Any attempt to access a file through a renamed binding will result in a user notification so that a permanent rebinding can be affected.

To delete a name-binding /path/f, Wayfinder unlinks path/f in the local hoard, removes f from the cached entry of /path in the DHT, and publishes a temporary delete notification to PlanetP. Whenever a device accesses /path, it see the set of delete notification associated with /path and deletes any local replicas as needed. To limit their accumulation, delete notifications are discarded after an expiration period currently set to one month. This expiration may allow a device that was offline for longer than this period to bring back a copy of a deleted file when it comes back on-line.

Directory deletion poses unique problems in Wayfinder as a directory should be deleted only if it is empty of all name-bindings. The merging approach to constructing directory views makes this particularly difficult as it is impossible to ensure that a directory is complete empty as several name-bindings may be absent or offline. To deal with this, Wayfinder pursues a lazy and localized approach; local directories are automatically deleted once they are empty. Thereby, a directory is only fully deleted from the global namespace after all devices have deleted all local replicas contained within that directory and subsequently the local directory replicas have been automatically deleted as well. Intuitively, this is akin to garbage collecting unused portions of the local directory structure in the hoards of individual devices.

However, this method of deleting a directory can be interrupted. Assume a device n is attempting to delete a directory, /path, and delete notifications for all containing files
has been published. As these notification are being applied to the replicas throughout
the community, another device, $m$, can create a new name-binding $/path/g$. This
binding would prevent the directory’s complete removal as $/path$ can not be deleted
on $m$. We deem this behavior acceptable because the original deletion attempt was
conducted with the original set of files in mind. If another user start using a deleted
directory again, this is akin to the user re-creating the directory for a different purpose
after it is deleted on a single device system.

To permit the existence of empty directories, Wayfinder automatically includes the
creation of a placeholder file as part of the directory creation. Intuitively, this file acts
as a flag to prevent the automatic removal of a directory.

3.6 Security Model

The Wayfinder file system is designed to work on devices spanning multiple adminis-
trative domains. It is therefore impossible to ensure the integrity of all devices. We
do, however, assume that a majority are well-behaved. We pursue a security model
that does not to prevent malicious behavior but rather limit its impact on the proper
working of the community at large. This model is part of a preliminary design and has
not been implemented.

Recall that Wayfinder supports a single copy availability model and any operations
involving a file’s content requires a local copy be present. As part of our design, we
do not restrict the downloading of replicas by any devices and so Wayfinder does not
enforce any form of control for read access. Instead, we rely on users encrypting the
contents of file using standard encryption techniques.

Furthermore, given a local replica of an available file, it is impossible to prevent a
user from subsequently altering a file once it is downloaded. It is equally impossible
to prevent the publishing of malicious update information. Therefore, Wayfinder does
not attempt to prevent malicious users from either of these actions, but rather tries
to stem the effects of their actions from spreading through the community. This is
accomplished by imposing write access control at each device independently through
the detection of improper versions of files (i.e., files having history vectors that contain disallowed updates) and preventing them from being presented to the user. Essentially, files with malicious updates are ignored by properly functioning devices.

To implement this model, each file is regarded as being owned by a public/private key pair. An access control list (ACL) that stores the public keys of principals (i.e., devices) that can write a file is encoded as part of a file’s metadata waynode (Section 3.3). The ACL is digitally signed by the owning key pair with the public key being embedded in the file’s globally unique ID. Files can have multiple owners by sharing its private key.

Once write permission has been given to a principle, it cannot be revoked. A file’s owner can only add to a file’s ACL, never delete from it. Otherwise, it would be impossible to tell whether writes from a user with revoked access rights preceded or followed the actual revocation. This is particularly relevant if a device makes valid changes to a file while disconnected and has its write permissions revoked during this period. As a result, write revocation is implemented by creating a new file with the same content, a modified ACL, and same file ID but with different keys. Wayfinder is able to distinguish between identical files with different keys to prevent name collisions.

Intuitively, the unique identifier of the file becomes a time-stamp for the ACL while the delete mechanism for files (Section 3.5) ensures that older versions of the ACL are eventually removed. It is up to the user accessing a file to verify that it is owned by the proper principal since multiple owners may compete for a single name within the shared namespace.

When a file is modified, its content is signed by the modifying principal’s private key. Devices receiving knowledge of a change must first verify its origin using this signature. Once verified, the new version is checked against the current ACL of the file. If both are correctly signed and the write is permitted, the new version of the file is considered as valid by Wayfinder; otherwise it is ignored.

A subtle complication arises in our system when considering indirect sharing of updated files. To protect the integrity of files, the history vector and content hash must be signed by one of the principals with write permission. Otherwise, a malicious device
can apply disallowed updates to a file without updating the file’s metadata. This *invalid* version of the file can then propagated to innocent devices as a properly protected file. This means that there can exist two version of a file; an “un-official” version in which all legal updates have been considered but the resulting content has not been validated and an “official” version in which all updates and content have been verified by one of the write principals. For efficiency reasons, Wayfinder allows devices to interchange non-official updated versions so that not all devices are required to repeat the work of validating changes while waiting for a write principal to sign a new version. The users of these collaborative devices, however, must establish a trust relation outside of Wayfinder’s write control model.

### 3.7 Storage Tables

Throughout this chapter we have demonstrate how Wayfinder’s abstractions are implemented as queries against the underlying PlanetP toolkit. To store the necessary information to support these abstractions we use three separate global tables that are part of the PlanetP storage system; specifically a content table, a metadata table, and a structural table. We will discuss each table in turn, presenting a description of its implementation and the summary information it generates.

#### 3.7.1 Metadata Table

The metadata table stores the typical metadata (i.e., size, dates, owner, etc) associated with a single file/directory which is encoded by the metadata portion of a file/directory’s waynode. Each entry in this table is associated with a particular file. Note, in most current file system, this information is typically dispersed throughout the file system (i.e., stored within the inodes), making it difficult to find files that match specific metadata query conditions without actually traversing the entire system.

The summary information for this table consists of a Bloom Filter that encodes the set of unique file IDs associated with every file; essentially one value per row of the table. With this information being shared, it is possible for a node to ascertain which
remote nodes contain information for a particular file/directory.

### 3.7.2 Structural Table

The structural table stores information concerning the directory structure of a device’s local hoard, specifically the information encoded in the structural portion of waynodes for the local directories. This table is implemented using a set of data structures to store the necessary name-bindings. In particular, given a file $f$ in a directory $d$, the parent-to-child binding ($f \rightarrow d$) and its reverse ($d \rightarrow f$) are stored. The latter binding is kept to make file deletes more efficient.

There is no summary information for this table as remote nodes can determine the location of directory replicas using the summary of the Metadata table (see above).

### 3.7.3 Content Table

The content table stores information concerning the content of files in a device’s local hoard which is necessary to return ranked results for any content queries (i.e., those associated with semantic directories). Any file that is stored by in the Wayfinder file system is subjected to having its content parsed. For a file $F$, this process results in a set $S = \{(k_1, n_1), (k_2, n_2), \ldots, (k_i, n_i)\}$ where $i$ is the number of unique terms in $F$ and $(k_j, n_j)$ represents the $j^{th}$ unique term with a term frequency of $n_j$ in $F$. If a file’s content can not be parsed (e.g., binary files) then the $S = \{\emptyset\}$. During parsing, terms are stemmed and stop words are removed.

As with the structural table, this table is implemented as a set of data structures. For each file, three types of information are stored: 1) a term → file mappings, for every $k_i$ in $F$ the binding $k_i \rightarrow F$ is stored, 2) the total number of terms for each file, $F \rightarrow \sum_{i=1}^{m} n_i$, and 3) a reverse mapping of the form $F \rightarrow \{k_1, k_2, \ldots, k_n\}$. The first two types are used for content search and ranking while the latter is for simplifying the deletion of a file’s content at the cost of using additional storage. If storage space is constrained, the reverse mapping can be removed and a lazy traversal of the term → file mappings used to delete a file’s content.

The summary information for this table is a Bloom Filter that encodes the set of
unique terms in this table. With this information being shared, it is possible for a node to ascertain which remote nodes contain information relevant for a particular term in a content query.

### 3.8 Performance

Having described the design of Wayfinder in detail, we now turn to evaluating its performance during standard file system operations and robustness using a prototype implementation.

#### 3.8.1 Experimental Setup

Our prototype is written in Java and uses a modified version of the JNFSD server [40] to export its services as a locally mounted user-level NFS system. All reported results were obtained on a cluster of PCs where each node is equipped with a 64-bit 2.8 GHz hyper-threaded Intel Xeon processor, 2 GB of memory, and a 10K RPM 70 GB SCSI disk. All nodes ran the Linux 2.6.18 kernel and Sun’s Java 1.5.0 JVM. The cluster is interconnected by a 100Mb/s Ethernet switch.

All information stored in the underlying instance of PlanetP is done so persistently using the BerkeleyDB database [64]. In several experiments, we forgo this persistency by using in-memory data structures to show the overhead of storing data persistently.

Each Wayfinder node caches file and directory views retrieved from the DHT in local memory for 10 seconds. When Wayfinder is used by communities connected over the Internet, this caching reduces the impact of communication over the WAN. Note that this caching is similar to caching done by the Linux NFS client (3–30 seconds), although Linux has a more sophisticated policy of when to disregard the cache. Unless specified, all nodes are participants in the PlanetP DHT.

#### 3.8.2 Micro Benchmark

Table 3.1 shows the results of a micro-benchmark experiment in which we measured the time taken for various NFS operation as a node enters a community, creates a set of
files and directories, edits the files, and concludes by deleting them. In the single-node case, all requests are satisfied by local operations and so represent the best possible times for a Wayfinder client. The multi-node case simulates the scenario when a node enters an existing community and all information is located on remote nodes thereby requiring network communication. This experiment was performed on a 12 node system in which the node running the benchmark did not participate in the DHT. The latter point ensures that all access to the DHT would require remote communication. We do this to ensure that the observed behavior is similar to what would be encountered in a larger community where a DHT access would most likely not be satisfied locally.

In looking at the single node case, we observe that operations that require storing new metadata, such as Create and Mkdir, incur longer overheads resulting from storing information in PlanetP. The time for Mkdir is larger than that of the Create as this time includes the overhead for creating the actual directory and the required hidden file (Section 3.5). Without this file, a Mkdir operation would take less time than a Create because it does not require the storing of content information.

Further can we observe that any operation that require communication with the DHT to access cached state, with the exception of the NFS Lookup and Getattr, takes longer in the multi-node setting. When Lookup operations require learning about remote files via waynodes, the results are cached for a small amount of time (a few
seconds) to improve performance of repeated Lookups. For Getattr the required file is always cached by previous operations.

Accessing the DHT to store information does incur visible overheads. As we shall see in later (Section 3.8.4), the benefit of the DHT is in improving the retrieval of information. For this, utilizing the DHT means the difference between possibly accessing a single node storing needed data in memory or contacting a large number of nodes where the needed data is stored persistently on disk.

Finally, as expected there is a significant cost in storing data persistently. The choice of using the BerkeleyDB for persistency was one made for its simplicity and ease of use. The BerkeleyDB stores all information as arrays of bytes. Our Java-based prototype uses Java serialization to convert between these arrays and various Java objects and primitive data types. Such conversions are performed when storing or retrieving information from the database. As an optimization, we have implemented a cache to store frequently accessed data. A persistent data store tailored to the data requirements of Wayfinder that could avoid this process would improve access times.

3.8.3 Macro Benchmark

Table 3.2 shows the running time for the Modified Andrew Benchmark [37] for Linux NFS, the unmodified JNFSD, and Wayfinder in various configurations. The benchmark consists of five phases executed by a single client: (1) create a directory structure, (2) copy a set of files into the directory structure, (3) stat each file, (4) grep through the files, and (5) compile the files. In all cases, the NFS server and client ran on the same machine for comparison against when Wayfinder is running on a single node. The column titled “Wayfinder: 1 Node” indicates the experiment being run in a single node community. “Wayfinder: 12 Nodes” reflects performance for the scenario where the community is sufficiently large so that each access to the DHT requires a message exchange. For this scenario, we present the times when using both persistent storage and in-memory data structures.

Observe that Wayfinder imposes little overhead compared to the JNFSD when the workload is not entirely comprised of file system operations. In particular, Wayfinder
Table 3.2: Results of the Modified Andrew Benchmark using the Linux NFS, original JNFSD and the JNFSD linked with Wayfinder running in isolation and in a community of 12 nodes.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Linux NFS</th>
<th>JNFSD</th>
<th>Wayfinder: 1 Node in-memory</th>
<th>Wayfinder: 12 Nodes persistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.01</td>
<td>0.02</td>
<td>0.20</td>
<td>0.53</td>
</tr>
<tr>
<td>2</td>
<td>0.12</td>
<td>0.45</td>
<td>1.23</td>
<td>2.04</td>
</tr>
<tr>
<td>3</td>
<td>0.21</td>
<td>0.26</td>
<td>0.30</td>
<td>0.42</td>
</tr>
<tr>
<td>4</td>
<td>0.22</td>
<td>0.25</td>
<td>0.28</td>
<td>0.29</td>
</tr>
<tr>
<td>5</td>
<td>1.58</td>
<td>2.60</td>
<td>2.85</td>
<td>3.16</td>
</tr>
<tr>
<td>Total</td>
<td>2.12</td>
<td>3.58</td>
<td>4.86</td>
<td>6.44</td>
</tr>
</tbody>
</table>

imposes insignificant overheads for phases 4 and 5, when the client is grepping and compiling, respectively.

Currently to determine whether or not a write operation has actually changed a file, Wayfinder creates a snapshot of the file’s content when it is opened for writing. This snapshot is used for comparison when the file is flushed. This process, in part, contributed to the higher performance penalty imposed in Phases 1 and 2. Additionally, Phases 1 and 2 require the synchronously flushing of waynodes from the local caches and force a remote DHT update of the any cached entries. This overhead can be observed when comparing the fourth and fifth column in Table 3.2, specifically for phases 1 and 2. Phase 3 benefits from the cache footprint resulting from phase 2 and so imposes only a modest amount of overhead.

In the sixth column of Table 3.2, we observe the worst case scenario in which all DHT access require remote communication and the accessing, or changing, of file system metadata requires accessing persistent storage. In particular, we see that Phases 1, 2, and 5 have significant overheads resulting from file creations. The cost of accessing persistently stored data (i.e., in the BerkeleyDB) surpasses any communication overheads.

We thus conclude that while Wayfinder does impose visible overheads on basic file system operations. These overheads are acceptable given that the prototype is a largely un-tuned Java program. We also observe that the Andrew Benchmark gives the worst
case scenario for Wayfinder: all operations are performed at a single client and so gives no measure of Wayfinder’s effectiveness for collaborative workloads.

### 3.8.4 Scalability and Robustness

We now show the advantage of Wayfinder’s dual nature for sharing information; using gossiping for robustness to failures and caching in the DHT for scalable performance. In this experiment we measure the time required for a single node to perform a complete traversal of an artificial namespace, e.g., doing an “ls -R”. The scanned namespace is a complete trinary directory tree of depth 3 at the root of the namespace, giving a total of 41 directories with each directory containing a single file. We ensure that all accesses to the PlanetP DHT require remote communication by disallowing the scanning node from being a participant in the DHT. We again do this to ensure that the observed behavior is similar to what would be encountered in a larger community where a DHT access would most likely not find the required state locally. We also ensure that sufficient time passes between each scan to negate any benefits we may receive from the local caching of view objects. However, we do allow the caching of objects retrieved from persistent storage. We have already observed in the previous section that accessing persistent storage incurs significant overheads. The absence of this caching would further increase the observed times in this experiment as DHT objects are stored in memory and manual re-creation of views may require accessing information stored persistently on remote nodes. In this experiment, we instead focus on the increased cost of communication. Finally, each node hoards the entire namespace so that in the absence of cached state, communication with every node is required to determine the current state of each file.

Figure 3.7(a) plots the time required for this experiment as we increase the size of the community as an average of five runs. As the namespace is replicated universally, increasing the community size increases the amount of distributed state that needs to be retrieved. As expected, the scan time without caching in the DHT grows linearly with community size since computing each directory view requires contacting all nodes. With caching, however, the scan time remains relatively constant as the cost of accessing cached DHT entries remains unchanged.
Figure 3.7: (a) Time required to scan a namespace plotted against community size. (b) Scan time in the presence of DHT failures. X-axis indicates a sequence of successive scans with solid vertical lines denoting a scheduled failure point. The label associated with each solid line indicates the number of nodes involved. Dashed lines represent the minimum and maximum times across twelve samples.

Figure 3.7(b) shows Wayfinder’s robustness to loss of data cached in the DHT. In this experiment, we run a sequence of twenty scans over a similar namespace as before and at four points simulate multiple node crashing in the DHT by discarding all DHT content on selected nodes. The average results of twelve runs of this experiment is show with the vertical bars at each data point delimiting the range of observed times.

We observe a rise in the scan time immediately proceeding each DHT failure indicating that some directory and file state had to be reconstructed manually by the scanning node. The subsequent return to the original scan time shows that this reconstructed state was successfully re-inserted into the DHT. The range of values after each failure reflect how much information was discarded as a result of the DHT failure.

### 3.8.5 Storage Overhead

For storage costs, our prototype incurs the cost of storing the tables defined in Section 3.7 persistently. These costs are described in detail for real life data sets in Section 5.4.8.
3.9 Summary

In this chapter we presented both the design and a prototype implementation of the Wayfinder federated file system. This file system allows a community of users to share and store information in a collaborative setting through two complementary abstractions: a single unified global namespace constructed dynamically from the available information of individual users and search queries that can be persistently embedded into the namespace in the form of Semantic Directories. These abstractions allow users to address information by both name and content.

With regard to these abstractions, we detailed the a method of constructing the namespace based on a merging principle. This method of construction allows the namespace to dynamically represent the available content while also providing effective support for both partitioned and disconnected operations. Furthermore, we showed that the combination of this global namespace and semantic directories provides a active search structure that can be shared among users.

This chapter detailed Wayfinder’s storage model in which all file system metadata is stored in a loosely consistent federated data store provided by the PlanetP toolkit. We described the method by which all of Wayfinder’s abstractions are implemented by posing the appropriate queries to this underlying storage system. We also demonstrated the usefulness of using a light-weight DHT for improving performance.

We have demonstrated through experimentation that while Wayfinder does impose some overhead for common file system operations, it may be reasonable given the benefits the system provide.
Chapter 4

Automatic Availability Management

In this chapter, we will consider the problem of automatic data replication in a federated system used for sharing information. In an ideal situation when sharing information in a federated community, we would replicate any shared content across all devices, ensuring that any shared content would be accessible from any device at anytime, regardless of the device’s connection status at the time of an access. This device and connection transparency would allow users to forgo reasoning about the placement of data replicas on specific devices. Furthermore, the effect of a user’s permanent disconnection would be tempered as his personal devices would contain a complete snapshot to access any content. Such permanent disconnection can arise for a number of reasons, including catastrophic failure of the federated system, dissolution of the federation, or the user simply leaving permanently. Mitigating this effect removes the need for a user to hoard content external to the federated system to ensure availability in case of such a disconnection.

This ideal situation is impractical. We anticipate that for most communities the amount of information being shared will be considerably larger than the storage of any single device. Each device will likely contain only a subset of the aggregated shared content of the community making the guarantees of the ideal model (i.e., access any file at any time) impossible.

We instead propose an intuitive 3-part availability model that approximates the ideal. Each user should be able to (1) access any file from any device connected to the federated system with high probability—this corresponds to the traditional definition of availability, which we will call online availability, (2) access files within a working set from any device within a set of personal devices when operating in disconnected
mode with high probability—we call this offline availability, and (3) access files that he owns from his personal devices with high probability, even if he becomes permanently disconnected from the community—we call this ownership availability.

Our model departs from the ideal in two ways. First, during disconnected operation, the user is limited to working from one of his personal devices and can only access files local to that device. With high probability, these local files should include his working set. This deviation is both reasonable and intuitive given that users do typically constrain disconnected operation to their personal devices rather than arbitrary devices in the federated system. Further, when operating in disconnected mode, the user is limited to the resources available on his local device and thus cannot expect to have access to all shared content. Having a well-defined working set automatically placed on his personal devices should go a long way toward meeting a user’s need for disconnected operation with minimal data management overheads [44]. The second deviation occurs when a user becomes permanently disconnected from the community. In this case, his devices may only contain the subset of files that he owns rather than a snapshot of the entire system. Again, this difference is reasonable and intuitive given the finite resources of a user’s device set and that the user is unlikely to be interested in the content of the entire federated system.

In the remainder of this chapter we discuss and present a supporting replication algorithm for this model. Specifically, we demonstrate how this algorithm explicitly attempts to balance the personal needs of an individual user (i.e., ownership and offline availability) with the needs of the community (online availability) while allowing nodes to retain a high degree of autonomy in their actions. We conclude this chapter by discussing a prototype implementation of our replication algorithm as part of the Wayfinder file system and present an evaluation.

### 4.1 Replication Algorithm

In this section, we describe an automatic replication algorithm devised to support the above mentioned availability model.
A key intuition behind our replication algorithm is that devices belonging to a user should prioritize offline and ownership availability for their user over online availability for the community. This is because the primary use of a personal device is to store content that the user cares about; i.e., content in the user’s working set, which he will likely access in the near future, and content that the user owns, which he would want in the case of permanent disconnection. However, it is beneficial for devices to collaborate to maintain high online availability for all shared content because this allows all users to find new content of interest to them as well as ensure easy access to content that they have not used in a long time. In our algorithm, devices selfishly use their local storage to store files in their owners’ working set (the set of files a user may want to access in the near future) and ownership set (the set of all files a user will want to retain a copy of). However, excess storage across the federated system is used to collaboratively ensure high online availability for all shared content. Server-like devices that do not belong to any single user can also be added to the system to ensure that sufficient communal storage is available to maintain high online availability.

Critically, our replication algorithm explicitly considers the impact of the selfish hoarding actions of individual devices on the online availability of shared content. This means that hot content, i.e., content that has been recently accessed by many users, typically does not need to be replicated for online availability. On the other hand, as content becomes cold, the algorithm will ensure that sufficient replicas remain in the system to maintain a target online availability level.

Our approach relies only on replication decisions being made autonomously by individual devices using only a small amount of loosely synchronized global state. This state is shared as part of PlanetP’s global directory and consists, in part, of an approximate replica-to-device mapping and a per device average availability measure. With this information, our replication algorithm can differentiate between the state of various files and ensure that they are neither over- nor under-replicated. Further, replication decisions are made locally and autonomously. This autonomy is important because devices in a federated system may join and leave the online system unpredictably and may have low online availability (e.g., laptops that are often turned off). Devices can
Figure 4.1: (a) Basic and (b) modified replication algorithms for a user $u$ and his device set. Files in $\text{Own}_u$ must be stored in the PSpace of at least one device in $\text{DS}_u$ (or just the champion (b)). Files in $\text{WS}_u$ are stored in the PSpace of all devices in $\text{DS}_u$. Each device in $\text{DS}_u$ (or just the champion (b)) then pushes files in $\text{Own}_u$ to the CSpace of other devices in the community.

also arbitrarily leave the system permanently.

More specifically, our algorithm mandates that personal devices belonging to a user $u$ collaborate among themselves to maintain offline and ownership availability for $u$’s working set and ownership set, respectively, and for all devices in the federated system to collaborate to maintain the online availability of all shared content. Toward this goal (See Figure 4.1), devices belonging to a user $u$ would: (1) hoard replicas of files in $u$’s working and ownership sets, and (2) push replicas of files owned by $u$ throughout the system to achieve a target online availability level.$^1$ Assuming that each file is owned by at least one user, then the push component of our algorithm serves to maintain online availability for all files. However, because devices prioritize offline and ownership availability over online availability, the target online availability is only achievable when there is sufficient excess storage space.

In the remainder of this section, we first introduce some notations and several important assumptions. We then describe the various aspects of our algorithm in detail.

$^1$Although we talk about replicating files because files are a well-understood content encapsulating abstraction, our replication algorithm should be applicable to arbitrary data objects.
4.1.1 Terminology and Assumptions

With respect to notation, let $WS_u$ denote the working set of a user $u$, $Own_u$ denote $u$’s ownership set, and $DS_u$ be the set of all personal devices belonging to $u$. Also, let us divide the local storage of each device into two logical regions called personal space (PSpace), which is used to achieve offline and ownership availability for the device’s owner, and communal space (CSpace), which is used to achieve the communal online availability (Figure 4.1). As shall be seen, the boundary between PSpace and CSpace is determined dynamically, with devices selfishly extending PSpace and shrinking CSpace as needed to maintain offline and ownership availability for their owners.

To support the availability model introduced in Section 4.1, we assume that up to two tags can be associated with each file $f$ for each user $u$: (1) $\langle u, f, OffA, t \rangle$, specifying that $u$ wants high offline availability for $f$ until the expiration time $t$, and (2) $\langle u, f, OwnA \rangle$, indicating that $u$ wants high ownership availability for $f$. The set $\{f | \exists \langle u, f, OffA, t \rangle, where t > the current time\}$ then defines $u$’s working set at any point in time while the set $\{f | \exists \langle u, f, OwnA \rangle\}$ defines $u$’s ownership set. As shall be seen in Section 4.2, in an implementation, tag inheritance, e.g., tagging a directory in a federated file system and specifying that files and subdirectories inside that directory should inherit the tags, and automatic system tagging, e.g., automatic tagging of files recently accessed by a user for high offline availability, makes this tagging scheme practical. As the tagging is done per user, it is possible for multiple users to own the same file. In this context ownership only implies that user wants to retain a copy of the file to guard against permanent disconnection and not having any special rights to the file compared to other users.

The key point with respect to the model is that users are asked to explicitly reason about the availability properties for files within the system’s namespace; that is, users locate files using their names and then attach, modify, or remove the desired availability tags. Users do not, however, have to reason about the replication and placement of files to achieve the specified availability properties. This latter aspect is the responsibility of the system running the replication algorithm discussed in Section 4.1.2. We shall
see later (Section 4.3) that the global namespace provided by the Wayfinder file system (See Chapter 3) simplifies this locating of files by name in a federated system.

We also assume that: (1) for each user \( u \), \( DS_u \) contains at least 1 device that is online most of the time—that is, this device has high online availability with respect to the federated system; (2) given a file \( f \), each device can inexpensively determine the latest version of \( f \) and the approximate location of all replicas of \( f \) as well as all replicas of a specific version of \( f \); (3) each device can inexpensively track the online availability of all other devices in the federated system; and (4) devices in \( DS_u \) can inexpensively track \( WS_u \) and \( Own_u \). In Section 4.2, we will show how these assumptions can be supported in an actual system.

### 4.1.2 Replication

The replication strategy for each user \( u \) is then to: (1) replicate each file in \( Own_u \) in the PSpace of at least one device in \( DS_u \); (2) replicate each file in \( WS_u \) in the PSpace of all devices in \( DS_u \); and (3) replicate each file in \( Own_u \) in the CSpace of devices throughout the system as needed to achieve a communal online availability target \( TOAC \). The first component ensures that the user will have at least one copy of each file that he owns in the case of permanent disconnection. The second component ensures that the user will have access to his working set during disconnected operation, regardless of which personal device he is using. Finally, the last component ensures online availability for all shared content when there is sufficient space.

To simplify the implementation of the above strategy, we introduce the notion of champion devices. Each user \( u \) must designate at least one champion device \( C_u \) from \( DS_u \)—typically the per-user highly available device assumed above. Ideally, \( C_u \) also has plentiful storage and processing capacity. The role of \( C_u \) is then to maintain ownership availability for \( u \) and to shoulder's \( u \) portion of ensuring online availability for all shared content (Figure 4.1). (Note that while we describe the algorithm as if there is a single champion per user, each user can in fact have multiple champions without introducing added complexity.) The remainder of the devices in \( DS_u \) are only concerned with maintaining offline availability for \( u \).
Further assume for the moment that $C_u$ has sufficient capacity to store all files in $Own_u$. Then, $C_u$ would monitor $Own_u$ and download any new member of this set to its PSpace. If its local storage is full, it will evict enough files from its CSpace to accommodate the new file. Eviction is described in Section 4.1.3. $C_u$ would also periodically, every $T_r$ time units, randomly select a file $f$ from $Own_u$ with lower online availability than $TOA_C$ and push a replica of $f$ to a randomly selected peer that does not yet store $f$.

Simultaneously, each non-champion device in $DS_u$ monitors $WS_u$ and downloads any new member of this set to its PSpace. If its local storage is full, then the device will evict enough files from its CSpace to accommodate the new file. If $WS_u$ becomes larger than the device’s local storage capacity, then files in $WS_u$ are evicted in order of expiration time, nearest to furthest in the future (i.e., LRU ordering).

If $C_u$ cannot hold all files in $Own_u$, then it must ask one or more peer devices in $DS_u$ to hold some of the files (in their PSpaces) on its behalf. This is a “golden” copy with respect to ownership availability and thus cannot be dropped by the peer device without the consent of $C_u$. Further, the peer device must become the champion for maintaining the online availability target for the subset of $Own_u$ that it is storing. This is the only instance in our algorithm when two devices must explicitly coordinate. Note that if a user has multiple champion devices, it is quite easy for these devices to coordinate the partitioning of files in $Own_u$ among themselves; they are highly available and so can easily run a standard commit protocol. This ensures that the size of a user’s ownership set is not limited to the storage capacity of a single machine.

If $C_u$ has plentiful storage, it can also monitor and hoard files in $WS_u$ in the case that $u$ ever needs to use it in the disconnected mode.

All devices may receive push requests from peers in the system to increase the online availability of under-replicated files. When a device receives such a request, it accepts and stores the replica in its CSpace if it has sufficient free space. Otherwise, it can either reject the request or evict from its CSpace to free up space.
4.1.3 Eviction

Eviction is a two-part process: (1) migrating files from PSpace to CSpace, and (2) evicting files from CSpace. Specifically, each device migrates each file $f$ in its PSpace to its CSpace whenever $f$ is no longer a member of $Own_u \cup WS_u$. A non-champion device holding a golden copy of a file in $Own_u$ but not in $WS_u$ can also migrate that file to its CSpace after $C_u$ negotiates to take back the responsibility for the golden copy.

When evicting files from CSpace, each device should evict the files with the highest online availability. If each device deterministically evicts the most over-replicated files, however, then multiple devices running the same algorithm autonomously may simultaneously victimize the same set of files, leading to drastic changes in the files' availability. Thus, we instead use a weighted random selection process, where files with higher availability have higher chances of being selected for eviction.

This availability-conscious eviction policy is implemented as follows. Periodically, each device computes the average number of nines in the availability of all files in its CSpace. We use the number of nines rather than the availability itself because it linearizes the differences between availability values, i.e., the difference between 0.9 and 0.99 (1 and 2 nines respectively) is the same as that between 0.99 and 0.999 (2 and 3 nines respectively). If a push request requires eviction, then the request would be rejected if the availability of the file to be replicated is more than a threshold percentage (we use 10%) above the computed average availability of local files. This prevents the acceptance of a replica that will likely be evicted the next time a replication request is received by the target device. Otherwise, lottery scheduling is used to affect a weighted random selection of victims where over-replicated files are heavily penalized for their excess availability, making it highly probable that a replica of an over-replicated file will be evicted.

In particular, a set of tickets is divided into two subsets with the ratio 80:20. Each replica in CSpace is assigned an equal share of the smaller subset. In addition, replicas with availability above 10% of the average are given a portion of the larger subset. The amount given to each of these replica is proportional the difference of the average local
file availability and the availability of the respective replica.

The intuitions behind our eviction policy are as follows. First, we reject the incoming replica if it will simply become a target for eviction the next time a replication request is received by the target device. Without this condition, we will simply be shifting replicas around without much effect. Our threshold for this outright rejection may seem rather low (i.e., 10% above the computed average of local files); at some cost of bandwidth, if we were less aggressive at rejecting replicas, perhaps over time, the system can reach a better configuration. However, we learned in previous work that while this threshold affects bandwidth usage, it does not significantly affect the overall replication process [15]. Next, we penalize over-replicated files heavily for the number of nines in their availability, making it highly probable that a replica of an over-replicated file will be evicted.

4.1.4 Updates

Thus far, we have described our replication algorithm as if files are immutable. When a file is updated, however, we must ensure that the update is propagated to maintain the availability of the latest version. (We are, of course, assuming that the file system itself does not ensure that an update is applied to all existing replicas.) If a file is updated on a non-champion device, then the device would push the new version of the file to the champion as soon as possible. Further, the device cannot evict the file until the new version reaches the champion. As shall be seen, when a system automatically tags recently accessed files to be in the accessing users’ working sets, the accessing devices would naturally avoid evicting these files for some time to ensure offline availability.

When an updated file reaches the champion $C_u$, or if the update was performed on $C_u$, there are two possible cases: (1) the file is owned by $u$, and (2) the file is not owned by $u$. In the first case, $C_u$ uses the standard periodic pushing process described earlier to push the new version of the modified file. One complication is that $C_u$ is faced with the problem of computing the availability of the latest version as opposed to that of old versions of the file. We address this problem by distinguishing between file online availability and version online availability, where file availability is computed by
considering all replicas of the file, regardless of the version, and version availability is computed by considering only replicas of a particular version. The champion then uses the version availability of the latest version of a file when it is considering whether a file still needs to be pushed for increased online availability. Eviction remains based on file online availability. This ensures that replicas of out-of-date versions will eventually be flushed from the system since they inflate file online availability but are not maintained by any champion.

In the second case, $C_u$ becomes a temporary champion for the updated file and pushes the new version to achieve $TOA_C$ for it. $C_u$ stores a replica of this version in its PSpace until it stops championing that file, at which time the replica would be migrated to its CSpace. All champion devices periodically look for updates to files in their users’ Own sets and download the new versions. Once the true champion of an updated file has downloaded the new version, it takes over the responsibility of maintaining the online availability of that version. (Ownership availability is already ensured by the fact that the champion downloaded the new version.) Note that this hand-off is implicit in that the temporary champion will push the new version of the updated file for a period of time, after which it quits under the assumption that the true champion has found the update.

When pushing an updated file, the champion will preferentially select a device that already has a replica of a previous version of the file by giving these devices more weight in its random selection of replication targets. This limits the storage devoted to replicas of out-of-date versions (although, even without this bias, out-of-date versions would eventually be flushed from the system as explained above).

To further ensure high online and offline availability for the latest version of a file in the presence of remote changes, each champion actively monitors files in the working set of its user looking for outstanding updates. If found, these updates are applied to the local replica.
4.1.5 Estimating Online Availability

Assuming that devices going online and offline are independent events, the availability of a file \( f \) can be computed as

\[
A(f) = 1 - \prod_{d \in D(f)} (1 - A_d),
\]

where \( D(f) \) is the set of devices that contain replicas of \( f \) and \( A_d \) is the availability of device \( d \). The effect on the availability of a file when increasing the number of replicas on devices with similar availability is shown in Figure 4.2. Recall that \( D(f) \) can be computed inexpensively according to assumption (2) in Section 4.1.1 and the availability of all devices are known according to assumption (3). The availability of a particular version \( f_v \) of file \( f \) can be computed similarly using the set of devices that contain replicas of \( f_v \).

4.1.6 Algorithm Implications

We now turn to discussing several important implications of the replication algorithm. First, it is quite easy to extend our algorithm to support per-file online availability targets, as opposed to a single common target. As space becomes constrained, however, our eviction algorithm will essentially place a cap on the maximum online availability that is achievable.

Second, while users do not have to reason about replicas and their placement, they
do need to be aware of the amount of storage available on their personal devices vs. the files that they want to own. As a user owns more files, more storage on his personal devices will be devoted to maintaining ownership availability, which may directly reduce offline availability for the user and online availability for the community as a whole. As already mentioned, a user can easily aggregate the resources of multiple champions to store and maintain the ownership availability of a large ownership set.

Third, it is easy to add devices not belonging to any user, e.g., server-like machines, to the system to provide communal space for maintaining online availability. These devices could passively provide additional storage (i.e., CSpace) to the community or actively take ownership of a portion of the shared content to help monitor and maintain online availability. Thus, our replication algorithm extends quite naturally to hybrid environments containing both personal devices and shared servers.

Finally, note that our replication algorithm, as presented, is not concerned with durability. By specifying a high ownership availability, users can ensure that a file will be replicated on at least one of their devices. If the user’s champion device (which is tasked with maintaining the ownership availability) has sufficient availability, then this file may only have one replica in the system. In this scenario, the file would not survive the single failure of this champion device.

Our model can be extended to consider durability as a factor during replica placement by allowing users to associate persistency ratings with devices in their device set. These rating can be derived from a variety of factors including the quality of the hardware, how the machine is administered, and the type of software installed. In turn, these rating can be used to define durability rules that are evaluated during the replication and eviction process. During their evaluation, a file can be deemed sufficiently persistent or require additional replicas. An example of a simple rule might be, “Ensure that all files in Ownu are located on at least two of my devices or on a device having a “Back Up” rating; indicating the presence of an external back-up system.”
4.2 Implementation

As indicated earlier we implemented a prototype of our replication algorithm as part of the Wayfinder file system. In this section we will present relevant aspects of the implementation. As we shall see, we employ the global tables in PlanetP presented in Chapter 3 but expand the summary information, specifically for the Metadata table.

4.2.1 Locating Replicas

For each replica of a file \( f \) having a unique ID \( f_{ID} \) and version \( v \), we store the terms \( f_{ID} \) (already required by the file system) and the additional term \( f_{ID}.v \) as part of the summary information of the Metadata table. A device can then locate replicas of \( f \) by querying PlanetP for the set of devices containing the term \( f_{ID} \) in their respective tables. Similarly, a device can determine the locations of a particular version of \( f \). Note that both these computations only involve consulting the summary information in the Metadata table (Section 3.7) and so does not require any network communication.

4.2.2 Maintaining Node Groups

For each user, \( u \), having a set of devices, \( D \), we maintain a group identifier, \( ID_u \). Devices belonging to \( D \) adds the term \( ID_u \) to their summary information of the Metadata table. The champion node of \( D \) adds the additional term “Champion.ID\(_u\)” to the summary of the Metadata table. Similar to when locating files, remote devices can determine the membership and the champion of a particular device set by querying PlanetP for the appropriate set of devices.

4.2.3 Tracking Device Online Availability

Each device in a Wayfinder system tracks when it is connected to the system (online) and when it is not (offline) and computes its availability as \( P(online) = \frac{\text{online time}}{	ext{online time} + \text{offline time}} \). Each device publishes its own availability as a property in the PlanetP membership directory.

While the above definition of online availability is quite standard, in a federated
system such as Wayfinder where any subset of devices can form a working subsystem,
the question arises of what constitutes being connected to the system. Under the
assumption that any Wayfinder system is likely to have at least a small number of
highly available devices if it is to provide reasonable data availability, we define a core
set that contains one or more of these highly available devices for each file system.
Then, a device is defined to be connected to the file system if it can connect to at least
one device in the core set.

4.2.4 File Tagging

To support our availability mode, Wayfinder allows users to specify tags for directories
that should be inherited by descendant files and sub-directories. In essence, this allows
the user to specify an availability directive for an entire portion of the namespace with
just one explicit tag. In addition, Wayfinder can optionally tag files automatically for
each user $u$ as follows: (1) whenever $u$ accesses a file $f$, $f$ is tagged with $\langle u, OffA, t \rangle$,
where $t$ is the current time plus a user-specified drop period; and (2) whenever $u$ creates
a new file $f$, $f$ is tagged with $\langle u, OwnA \rangle$. This tagging information is maintained in
the locally stored waynode as part of a replica’s metadata information.

4.2.5 Maintaining Offline and Ownership Availability

Each device in $DS_u$ maintains copies of $u$’s working and ownership sets and loosely syn-
chronize them with its peers in $DS_u$ using PlanetP’s gossiping service. (Non-champion
devices only have to maintain their additions to the ownership set.) Whenever a device
learns of an addition to the working set, it downloads the new file to its hoard as part
of PSpace. The champion device does the same thing for the ownership set.

By maintaining the working set, devices learn about changes made to files within
$DS_u$. Changes made outside of $DS_u$, however, may go unnoticed. To learn of such
remote changes, each champion device maintains a finite collection, $M$, of recently ac-
cessed files to periodically query for remote changes. Files are automatically placed
into $M$ when they become part of a user’s working set. Assuming there may be some
locality in file updates, we use a weighted multi-chance lottery to choose update candidates. Specifically, the lottery uses tickets assigned to each file on its initial placement into $M$. When a file $f$ is chosen, a user’s device attempts to determine if any outstanding updates to $f$ exists and if so, updates the local replica of $f$. If the update is not successful, the file’s ticket count is halved for future lotteries. This continues until the ticket count drops below one at which point the file is removed completely from consideration. If successful, the ticket count for the file is reset to the amount given at insertion.

An interesting consequence of tag inheritance is that a user may become the owner of a file that he did not create. An example of this situation arises when a user is interested in maintaining a copy of all content in a directory, regardless of who first introduces that content. Thus, to properly maintain the ownership set, each champion device periodically traverses all portions of the namespace tagged for ownership by its owner. In fact, the champion does not maintain the full ownership set; rather, it only maintains the set of roots defining the portions of the namespace that it must traverse to compute the ownership set.

### 4.2.6 Maintaining Online Availability

Recall that each champion device $C_u$ must periodically choose a file from $Own_u$ that has not achieved an online availability of $TOA_C$ and push a replica of that file out to the system. As $C_u$ traverses the namespace to compute the ownership set, it computes the current version online availability of the latest version of each member of $Own_u$ and inserts it into a pool of candidates if the file is under-replicated. The champion then periodically selects a file from this pool, recomputes its version availability if a threshold time period has passed since the availability was last computed, pushes a replica if still necessary, and removes the file from the pool if it has now achieved its availability target. The pool has finite size. If the pool fills up, candidates are removed using a random selection similar to that described in Section 4.1.3 but weighted by the version availability instead of the file availability.
4.2.7 Eviction

A finite pool of candidates for eviction is maintained on each device similar to the pool on each champion for replication. Each device continuously traverses its local hoard, filling the pool with potential candidates for removal. Files are considered candidates if they are in CSpace. When needed, eviction victims are then chosen from the pool using the method described in Section 4.1.3.

4.2.8 Updates

A new version of a file is created as the results of a write operation. Whenever a device learns of an update to a file in its PSpace, it downloads the new version of the file to replace the older version. To reduce the overhead of detecting such file updates in our implementation, only the champions look for updates to files in their owners’ working sets. When a champion learns of an update, it downloads the new version of the file and then gossips the existence of the update to its peer devices in the device group. Each peer devices may then attempt to update its own local copy as needed.

4.3 Evaluation

We will now explore the performance of the replication algorithm presented in this chapter. We have implemented the replication algorithm as part of the Wayfinder file system. Wayfinder provides a natural hosting platform for our model as it provides a global namespace that ensures uniform naming of information across connected and disconnected operation. Coupling our replication algorithm with a global namespace removes the need for users to reason about the physical placement of replicas for both locating data and ensuring data availability. The burden of data management for each user is thus reduced to reasoning about what availability properties he desires for specific portions of the global namespace. Note that the latter is an already existing and necessary part of participating in a federated system unless the user’s devices have sufficient resources to hoard all shared content.

We evaluate our prototype implementation using a micro-benchmark as well as a
macro-benchmark derived from traces of a online wiki site. The macro-benchmark is particularly relevant because wikis are designed to allow communities of users to collaboratively maintain shared sets of web pages.

4.3.1 Methodology

Our study centers around two benchmarks; a micro-benchmark that injects bursts of file creations and a macro-benchmark derived from a read/write trace of a wiki. The former studies the efficiency of our randomized replication algorithm while the latter studies our algorithm’s behavior under a real, albeit condensed, workload.

Our study focuses only on Wayfinder’s maintenance of online and ownership availability. While the full replication algorithm is always executed, we do not measure offline availability as we do not have sufficient data on disconnected access patterns. Further, our contribution is a combined availability model that includes hoarding as a component, rather than any novel hoarding algorithm for disconnected operation. Other efforts [46] have explored this latter problem and our replication algorithm can directly leverage their algorithms if desired. As mentioned, our algorithm maintains an LRU working set for simplicity.

Our benchmarks are run in two different environments designed to represent two different styles of federated communities.

Corporate (CO). This community represents what we might see in a standard corporate or university environment, where each employee is assigned a desktop. The goal of this community is to provide high online availability for all content. This community is parameterized using data from Bolosky et al.’s [8] study of a large corporate environment. Specifically, we set each node in a cluster of 12 nodes to have 80% availability with an average uptime of 272 minutes. (The average uptime was shortened by a factor of 11 from that reported by Bolosky et al. because our benchmark is a compressed trace designed to reduce experimental time.) Each node belongs to a distinct device set. This is akin to each user only owning a single node in the community. All files in the system are tagged as owned by a single node, which represents the infrastructural resource devoted to maintaining availability for all shared content. In essence, this node
is the champion for the entire community. We could have also chosen to have all nodes own all files. However, this would have put the algorithm in the best possible case of fully concurrent downloads by all nodes.

**Heterogeneous Workgroup (HW).** This community represents an extended, more mobile environment, where each user may have a desktop at work, a laptop for mobile computing, and a home machine. This community is quite representative of our research lab and is parameterize using measurements obtained from our lab. Specifically, the community contains 3 device sets belonging to 3 distinct users. Each device set has a work desktop with 80% availability, a home desktop with 50% availability, and a laptop with 32% availability. The community also has 3 additional server-like nodes that do not belong to any user but rather provide resources for the entire community. Each of these server-like nodes has 95% availability. The average node uptimes were set to 387 min., 225 min., 60 min., and 55 min. for nodes with 95%, 80%, 50%, and 32% availability, respectively. Files were partitioned into 3 non-overlapping ownership sets, one per distinct user.

**Experimental Platform.** All reported results were obtained on a cluster of PCs, where each node was equipped with a 64-bit 2.8 GHz hyper-threaded Intel Xeon processor, 2 GB of memory, and a 10K RPM 70 GB SCSI disk. All nodes ran the Linux 2.6.18 kernel and Sun’s Java 1.5.0 JVM. The cluster is interconnected by a 100Mb/s Ethernet switch. The Wayfinder prototype was configured with a 1 second PlanetP gossiping interval and a 3 seconds inter-push time for replication for online availability, 10 seconds local caching of directory and file views.

### 4.4 Online Availability for Creation and Write Bursts

Two key aspects to any replication scheme is how events that require the replicating of data can be detected and once detected, how quickly until the system reaches a stable configuration again. The speed at which such events are detected and the replication speed in Wayfinder is influenced by two process; the traversal speed of over the namespace to find replication candidates and the rate of replication.
We study these two aspects by measuring the time required by our prototype to achieve a stable configuration stable configuration, i.e., one in which all files have achieved $TOA_C$, after a single node has injects a burst of new data into the system.

Specifically, we use a benchmark that first creates $N_i$ files and allows the system to reach a stable configuration. The creation of these initial files is designed to evaluate whether the time to reach a stable configuration depends on the number of existing files as well as the size of a creation burst. Then, the benchmark creates an additional $N_b$ files and measures the time required for the system to achieve $TOA_C$ for these files. All created files are small, on the order of several hundred bytes, and so the actual time to create a physical replica is negligible.

Table 4.1 shows the results for the file creation bursts when the above benchmark is run on a cluster of 12 nodes with $TOA_C = 0.999$. In both the CO and HW environments, the bursts were performed on a champion device. During the experiment, node arrival to and departure from the online system were driven by exponential arrival processes based on the mean times given above. The times presented were averaged over five runs of the benchmark.

Observe that the time required for Wayfinder to achieve the online availability target for all newly created files is roughly linear to $N_b$ but independent of $N_i$. This independence from the number of existing files arises from the “pool of candidates” implementation described in Section 4.1.2 and is quite important because Wayfinder systems may contain very large numbers of files (e.g., millions). The linear dependence on the burst size shows that Wayfinder’s randomized selection process is (almost) as

<table>
<thead>
<tr>
<th>No. background files ($N_i$)</th>
<th>Time to achieve $TOA_C$ in CO (sec)</th>
<th>Time to achieve $TOA_C$ in HW (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Burst Size ($N_b$)</td>
<td>Burst Size ($N_b$)</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>100</td>
<td>138</td>
<td>266</td>
</tr>
<tr>
<td>200</td>
<td>137</td>
<td>269</td>
</tr>
<tr>
<td>400</td>
<td>145</td>
<td>276</td>
</tr>
</tbody>
</table>

Table 4.1: The median time required for Wayfinder to achieve the target online availability $TOA_C$ for a group of newly created files.
efficient as the use of a predetermined deterministic schedule.

Observe also that Wayfinder reaches stability faster in HW than in CO. This is because the ownership of the created files in HW was partitioned among the three device sets—they were created in different directories, each of which was owned by a different user with tag inheritance turned on—and so the new files were pushed by all three champions as opposed to the single champion in CO.

Variance in convergence time is due largely to nodes altering their connectivity during the benchmark. In the event of an inconsistent view of the community’s online membership, attempts to create replicas fail if they are directed to nodes that have recently gone offline. Also in case of the champion going offline, the replication process may stall entirely.

4.5 Online Availability for a Collaborative Workload

We now explore Wayfinder’s behavior using a trace of a collaborative workload. In particular, we derive a benchmark based on data collected from an online wiki site, specifically for the PlanetLab Project [58]. The PlanetLab is an endeavor targeted at building and deploying an open platform for running planetary-scale services. The documentation for this project is maintained in an online wiki website (http://www.planetlab.org). Through this wiki, the PlanetLab community collaboratively provides evolving documentation for the system. The wiki software used by PlanetLab is much like a version control system in that it preserves the history of a given page’s evolution. This makes it ideally suited as a source for a collaborative workload.

We were able to obtain information about file creations (one file per web page) and updates, when they occurred, and who performed them. When we collected our trace on September 27, 2005, the wiki web contained 457 distinct files with 2800 distinct versions. The updates spanned a period of several years and were performed by 153 distinct users.

We created a benchmark from the above data as follows. First, we chose a period of 134 days during which the wiki was particular active, containing 532 updates spanning
128 files and derived an update stream. This update stream was then compressed by reducing each inter-write interval that was longer than 15 seconds down to a random time between 5–15 seconds. This led to a trace that runs for approximately 90 minutes. The trace requires (in its worst case) approximately 2.35 MB with a final size of 1.84 MB.

Second, to actually run the benchmark, we needed to map the updates in the trace to authoring nodes in the federated system. Since we are not evaluating offline availability, all accesses needed to be mapped to nodes that are online at the time of the access. We ensured this by first mapping accesses of each user to a node in the community—typically, this meant that multiple users were assigned to each node—and then constructing the online and offline behavior of each node around the accesses assigned to that node. In particular, for each node, we generate a time line with arrival to and departures from the online community by choosing a sequence of online/offline intervals (as defined by the availability and average uptime of the node). If an access would occur during an offline interval, then we discard the tail of the sequence starting at the online interval before the offending offline interval. We then regenerate the tail and try to grow the time line until we have a legal sequence for the trace. As the interval lengths were generated using an exponential distribution, the resulting time line represents one possible sequence consistent with the probabilistic behavior of the node. Each node’s behavior corresponded to one possible sequence of arrival to and departure from the online system according to an exponential arrival process with the appropriate mean.

Third, we added in read traffic by injecting reads to random files in the web during nodes’ uptimes according to an exponential arrival process with a mean inter-arrival time of 5 seconds. Finally, to avoid name conflicts, which are difficult to handle in an automated trace-driven experiment, we set the benchmark to pre-create all files at the beginning of the experiment with an initial size of zero.

We carefully logged all relevant actions of each Wayfinder instant in order to reconstruct the placement of file replicas and the availability of each file at any instant in the trace.
Figure 4.3: Average file online availability, average version availability (for the latest version of each file at each instant in time), and minimum availability plotted as functions of experiment time for (a) CO with infinite space, (b) HW with infinite space, (c) CO with constrained space, and (d) HW with constrained space.

We then ran the above benchmark in several different environments: CO with infinite storage, CO with constrained storage (each non-champion node was given about 35% of the final hoard size), HW with infinite storage, and HW with constrained storage (champs given approximately 80% of final hoard size, devices given 19% of final size, and servers given approximately 11% of the hoard size) The shared servers were given very little space on account of the high availability and to cause contention for their storage.

The eviction threshold, the hoard size to which nodes attempt to reclaim space to during the eviction process was set to 70% of the local hoard. $TOA_C$ was again set to 0.999.
Figure 4.4: Division of files between PSpace and CSpace for (a) CO with infinite space, (b) HW with infinite space, (c) CO with constrained space, and (d) HW with constrained space.
Figures 4.3 and 4.4 show some of the results. We make several observations based on these results. First, when there is sufficient storage space, the system consistently achieves and maintains the target online availability for all files. Figures 4.3(a-b) show that Wayfinder takes some time to achieve the target online availability for all files, particularly when there is only one champion as in the CO environment, because of the creation burst at the beginning of the benchmark. In CO, the champion has to push all files created at the beginning of the benchmark as well as updates that occur before it can achieve $TOA_C$ for the files. However, once the minimum file availability reaches the target online availability, it never drops below this target again. In fact, the average file online availability is greater than the target as the benchmark continues to run. This is because of the hoarding of files in the working set for offline availability; as files are read or written on a device, they become members in the user’s working set and are hoarded on that device. Since we have infinite storage, no file is ever discarded from any device’s local storage.

Second, when there is insufficient storage space, the system approaches a non-cooperative configuration where devices selfishly use their local storage to achieve offline and ownership availability for their owners. Observe that there is significant separation between the minimum and average file availability in Figure 4.3(c-d) as space becomes constrained. This is because devices are not cooperating to maintain the target online availability for shared content. Rather, online availability is just a consequence of devices hoarding content for offline and ownership availability. This is shown clearly in Figures 4.4(a-b) and (c-d), where CSpace is much smaller in the space constrained case (c-d) than in the case with infinite space (a-b) and nodes attempt to reclaim space through the removal of files in CSpace.

Additionally observe that in Figure 4.4(b), the number of files in PSpace is reduced with respect to the scenario in Figure 4.4(d). This reduction is a results of nodes evicting additional files from their working set when the set of candidate files in CSpace is exhausted. Despite this eviction, in all scenarios the target ownership availability is ensured.

Third, our unified availability model allows the system to account for the selfish
behaviors of devices in attempting to achieve the best possible online availability for all shared content. Observe that even though CSpace is much more constrained in Figure 4.4(c-d) than in (a-b), the average file availability is still not far from the target of 3 nines. This is because replicas of files that are relatively over-replicated by selfish hoarding are preferentially evicted from CSpace. Also, because each file is owned by at least one device, typically a champion, the minimum file online availability remains at least at the availability of the champion—in this case 90% (or 1 nine).

Finally, the system efficiently maintains the target online availability for all files, despite a stream of updates that requires re-replication of the modified files. The nature of our trace ensures that a continuous stream of writes produces files with changing versions. In Figure 4.3, we observe a noticeable difference throughout the trace between the average file availability and the average version availability. The lower version availability is a side effect of the replicas being out-of-date with respect to recent changes. As a general rule, in the presence of updates, high availability of a file need not imply a high version availability for the same file.

Wayfinder attempts to address this by targeting replication requests for files to existing replicas if the respective replica availability is already sufficiently high. The replication request then becomes translated locally to a request for updating the file’s content, thereby improving the version availability.

If the modified files are part of a user’s working set then changes are communicated
rapidly to a user’s devices prompting a quick update of replicas. This behavior is clearly observable in Figure 4.5 which depicts the version availability of a single file in the HW environment during the experiment. We see clearly that with every write operation, the version availability of the file drops. The magnitude of the drop reflects the availability of the device performing the write as this will become the new availability of the latest version. This is followed by the system rapidly regaining the target online availability for the new version.

Comparing the size of PSpace and CSpace (Figure 4.4), we see that PSpace is larger in the HW community than for the CO Community when space is not restricted. Recall that PSpace is comprised of files in the ownership set or working set of a node. In the HW, the growth seen in PSpace is the result of replicating the working set across devices in a given device set. As the devices sets in CO are completely disjoint, this additional replication does not occur.

4.6 Ownership Availability for a Collaborative Workload

To examine the ownership availability, we chose to monitor a single user’s activity with respect to a representative file. This file was chosen because it accessed throughout the trace and was done so by several different users. During the experiment, we track both the placement and the version of the selected file among a user’s devices set.

Figure 4.6 plots the ownership availability of our representative file against experiment time as the wiki benchmark is executed in the HW environment with infinite storage (a) and constrained storage (b). Ownership availability is expressed as the number of up-to-date file replicas in the owning user’s device set. This count can drop to zero if the file is updated by a device outside of the owning user’s device set. This can be observed occurring several times in Figure 4.6. However, in each case at least one device in the device set notices the update within a short time so that there is almost always at least one up-to-date replica in the device set. This notification can be a result of targeted replication or the champion node’s monitoring of files in the Ownership set.
Figure 4.6: Ownership availability of a representative file vs. experiment time for the HW environment with (a) infinite space and (b) constrained space. Figures plot the number of replicas contained within a user’s device set and how many of these replicas are up-to-date given changes performed by the user (Group Access) and remote changes (Remote Access).

Figure 4.6(b) plots the ownership availability when the benchmark is run in the HW environment with constrained space. In this case, we see that only one replica of the representative file is kept unless that file is accessed by the user and so becomes part of the working set, resulting in replicas being created on all online devices. These new replicas are then rapidly discarded from the non-champion devices as other files are accessed.

4.7 CPU and Network Utilization

Our replication algorithm relies on a number of periodic processes—e.g., pushing under-replicated files, looking for and downloading updates to files in the user’s working and ownership sets, etc.—that must run continuously in an implementation. Thus, we measured the CPU utilization during the execution of the wiki benchmark to evaluate the CPU overhead imposed by these periodic processes. The results are summarized in Table 4.2. For all tested environments, including the one with constrained storage space, the average CPU utilization was below 3.8 % on a champion device and less than 2.5% on a non-champion device. More specifically for a champion devices the CPU utilization was relatively constant between 3.68% to 3.72% while a non-champion
Table 4.2: Percentage of CPU utilization while running Wiki Trace in both the HW and CO environments with different restrictions on the size of the local hoard.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>CO</th>
<th>HW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Infinite  Space</td>
<td>Constrained</td>
</tr>
<tr>
<td>Champion</td>
<td>3.68</td>
<td>3.69</td>
</tr>
<tr>
<td>Non-Champion</td>
<td>2.44</td>
<td>2.34</td>
</tr>
</tbody>
</table>

Table 4.3: Count of operations performed by replication algorithm during the Wiki Trace. Replication Request is the actual sending of a replication request from a champion. Download Attempt is the number of actual file downloads that occurred as a result of either an access or replication request. File Eviction is the number of performed replica evictions.

<table>
<thead>
<tr>
<th>Operations</th>
<th>CO</th>
<th>CO</th>
<th>HW</th>
<th>HW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Infinite</td>
<td>Constrained</td>
<td>Infinite</td>
<td>Constrained</td>
</tr>
<tr>
<td>Replication Request</td>
<td>1631</td>
<td>1629</td>
<td>4266</td>
<td>4291</td>
</tr>
<tr>
<td>Download Attempts</td>
<td>1916</td>
<td>2143</td>
<td>2684</td>
<td>3252</td>
</tr>
<tr>
<td>File Evictions</td>
<td>0</td>
<td>1022</td>
<td>0</td>
<td>1256</td>
</tr>
</tbody>
</table>

To provide an alternative measure of the amount of work done by our replication algorithm, Table 4.3 presents the counts of various replication related activities that were performed during the wiki benchmark for various device types. We see an expected increase in the number of file downloads and evictions as the storage space in the community is reduced. Note that the number of replication requests stays constant across configurations for the same community. In our implementation, the replication process is ongoing and performed at a constant pace regardless of the community.

We also measured the write bandwidth needed while running the wiki benchmark. The entire data set, i.e., a single copy of the final version of each file is approximately 1.8 MB in size. The mean file size is 14.7 KB with a median size of 5.6 KB;

The results of this analysis are presented in Table 4.4. Shown is the total amount of
### Table 4.4: Breakdown of the amount of data written during the wiki experiment for different environments. Included is the total amount of information written (in MBs), the average write bandwidth (in KB/s) for devices and a breakdown (in MBs) of the total bandwidth into replication related activities. Content is the retrieval of remote file content. Search is the amount of information exchanged when during distributed query processing in PlanetP. Replication is the amount of information sent to issue replication requests.

<table>
<thead>
<tr>
<th>Community</th>
<th>Write Bandwidth</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Infinite Space</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ch.</td>
<td>NCh.</td>
</tr>
<tr>
<td>Node Type</td>
<td>Content</td>
<td>Search</td>
</tr>
<tr>
<td></td>
<td>16.3</td>
<td>11.8</td>
</tr>
<tr>
<td></td>
<td>144.0</td>
<td>899.3</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>207.5</td>
<td>1103.5</td>
</tr>
<tr>
<td></td>
<td>39</td>
<td>19</td>
</tr>
</tbody>
</table>

Information (in MBs) sent by each node type (champion and non-champion) for each evaluated community (CO-infinite, CO-constrained, HW-infinite, and HW-constrained), bandwidth broken down by replication related activities, and the average amount of write bandwidth used by each node. Note that the measurements for the categories of content and search represent an upper-bound as these include activities not explicitly related to replication. For example, user-level file system requests, such as file read and writes, may trigger the search for and downloading of files; both of which would be accounted for in these categories.

From this data, we observe that the bandwidth requirements for any node ranges between 10 K to 40 KB with champion nodes requiring twice the bandwidth of non-champion nodes. We can make several additional observations.

In the constrained environments, the amount of data required to process a distributed query (search category) is reduced. This saving is a result of replicas being constantly evicted and thereby fewer nodes needing to be contacted at query time.

The amount of information downloaded is a poor indicator of the work being done by the replication algorithm. Observe that for the content category in the CO environment,
the amount of information sent decreases when comparing the CO-infinite environment to the CO-constrained environment.

In CO-constrained, file availabilities change more frequently on account of replica being evicted. Given their larger number, smaller files are more likely to be candidates for evicted and then subsequent re-replicated. The overall effect is that larger files are replicated less frequently in CO-constrained because there are always smaller files with availability less than $TOA_C$, thus lowering the bandwidth requirement for replicating content. This observation is supported by the data in Table 4.3 which indeed shows a higher incidents of downloads and evictions for CO-constrained. We determined that the average file download size for CO-constrained was 15.1 KB while for CO-infinite it was 23.3 KB; supporting that larger files are transferred more often in the latter case.

In the HW environment this drop in content bandwidth is not present. We attribute this to the faster rate of replication given the existence of 3 champions devices and device sets actively attempting to maintain offline availability. However, it should be noted that this measurement for HW-finite may be misleading. We have determined that only 50% of the written information is actually read on the receiving devices. Recall that in our prototype implementation, we use a simple web server (Section 3.3) to publish local files for downloading by other instances of Wayfinder. This web-server uses a simple HTTP protocol and may send information that is unused by the receiver. Specifically this can occur in a constrained environment in which the receiver may abort a download upon learning the size of a replica (if there is an metadata inconsistency, the expect size and actual size may differ) after the actual download has begun.

Note that in our experiment, various settings are elevated to account for the aggressive compression of the actual wiki trace that was used to obtain the benchmark. For example, we set the gossiping interval for PlanetP to 1 second. In a more typical setting, the gossiping interval is likely to be larger and so the bandwidth usage should be significantly lower.
4.8 Summary

In this chapter, we presented a novel unified availability model targeted to content sharing federated systems. Such federated systems are typically comprised of a number of users collaborating to share data, each of whom may be using a distinct set of personal devices. To accommodate this environment, our model differentiates between three types of availability; online, offline, and ownership availability. Based on these distinctions, we outlined a user-centric availability model that tries to make data available to users across periods of connected and disconnected operation. This model also helps users to preserve data they care about in case they become permanently disconnected from the federated system. Consequently, the model removes the need for users to explicitly manage data replicas and to hoard data external to the federated system.

We have outlined the design of a single replication algorithm that achieves all three types of availability. This algorithm allows devices to selfishly prioritize ownership and offline availability for their owners over online availability for the community. However, the algorithm explicitly accounts for the impact of devices’ selfish hoarding actions on online availability in order to minimize the space required to achieve a target online availability level for all shared content. Further, this algorithm is based on autonomous actions from devices in the community, allowing the system to tolerate the fact that devices in a federated system is not under centralized control and so may have unpredictable prolonged periods of disconnection or even leave the system permanently.

We have presented a practical implementation of the proposed algorithm as part of the Wayfinder file system. Our presented evaluation involved modeling several types of information sharing communities and utilizing a collaborative trace obtained from an online wiki. Our results show that this implementation efficiently achieves our design goals.
Chapter 5

Single Node Multi-Dimensional Search

5.1 Introduction

We now turn our attention to the search technologies underlying the content-addressing portion of the Wayfinder file system. In particular, we explore how to improve the current textual search that underlies most of today’s search engines. In a previous chapter (Section 2.8), we presented several contemporary search tools that produce results through a combination of ranking and filtering techniques. When evaluating a query with multiple predicates, these tools determine results by first applying a ranking algorithm to a textual part (often a content predicate) of the query and then using the other dimensions (date, size, location) as filtering conditions.

As an example, consider a user that has saved personal information on a computing device. In addition to data explicitly added by a user, some of the data available on the device may come from external sources (such as other users in a network setting) and therefore may not be familiar to the user. Alongside this data, the system may store (a potentially large amount of) metadata information (e.g., access time, file type), as well as some navigational structure information (e.g., directory structure).

In such a scenario, it is possible to ask the query:

“Find a pdf file created on March 21, 2007 that contains the words ‘proposal draft.’ ”

Keyword-based search tools answer this query by returning all files of type pdf created on 03/21/2007 (filtering conditions) that have content similar to “proposal draft” (ranking expression), ranked based on how close the content matches the text “proposal draft” using some underlying text scoring mechanism. Contemporary tools would miss any relevant file that do not strictly adhere to the date and file type filtering
conditions; for example, *tex* documents created on 03/19/2007 that contain the text “proposal draft” would not be considered. The only recourse left to the user is to use filtering conditions based on ranges (i.e., all dates in March 2007). The fear of not finding the desired information, however, leads users to construct ranges which are overly broad and so may not be very useful in filtering the ranked results.

In this chapter, we present the design and implementation of a multi-dimensional fuzzy search framework for a single node setting. In the next chapter, we will expand this design to a federated environment. Our single node framework has two advantages over contemporary tools. First, objects that are approximate matches to any query conditions (i.e., not just content) will continue to be considered as results. Second, an object’s relevance will be determined by an aggregate numerical value that conveys how well the object matches the individual dimension-specific predicates of the query. These, in turn, will not be a simple binary score, as would be the case with filtering conditions.

It is our claim that these features can significantly increase the quality and usefulness of search results in many search scenarios. For instance, in the previous example, the user might not remember the exact creation date of the file of interest—or may not be the original creator of the file—but remembers that it was created around 03/21/2007. Similarly, the user might be primarily interested in files of type *.pdf* but might also want to consider relevant files of different but related types (e.g., *.tex*, or *.txt*). In this case, the date and file type conditions should not only be considered for filtering purposes but should also be part of the ranking conditions of the queries.

While the work presented in this chapter is applicable to a wide range of information management scenarios, we focus specifically on file systems. That is, we consider the granularity of the search results to be a single file in a larger file system, whether it be on a single node or distributed. For this reason, we explore queries over three dimensions of information relevant in most file systems; content, metadata, and structure.

Our scoring approach is based on Inverse Document Frequency (IDF) scores. That is, the magnitude of an object’s relevance score for a query predicate is inversely proportional the number of matching documents. We will describe a scoring approaches
for each of the above mentioned dimensions and present a framework for unifying the
dimensional scores for multi-dimensional queries. The final result will be a single rele-
vance score for each file returned as a result. These scores are computed in a manner
that allows for approximate matches in all queried dimensions. Furthermore, we em-
ploy a top-K algorithm to score top ranking files without considering all the data in
the system.

In the next section, we will present the details of our multi-dimensional scoring
framework. This will be followed by a description our system’s overall architecture and
the algorithms used in aggregating scores and returning the best answers to a query.
We then present the details and evaluation of a single node prototype implementation.

5.2 A Unified Multi-dimensional Scoring Framework

In this section, we present our unified framework for assigning scores to files based on
how closely they match individual query dimension conditions. We distinguish three
scoring dimensions: content for conditions on the textual content of the files, metadata
for conditions on the system information related to the files, and structure for conditions
on the directory path to access the file.

Our scoring strategy is based on an $IDF$-based interpretation of scores for each
dimension. Traditionally, the $IDF$ score of a document for a keyword in the IR world is
a function of how many documents contain the keyword [76]. The content $IDF$ scoring
strategy has been widely adopted in IR systems as it considers the data distribution to
assign scores. We extend this idea to each of our search dimension and assign a score to a
file in a query dimension based on how many files match the query dimension condition.
The unification aspect of our scoring framework comes from this $IDF$-based scoring
approach; for each dimension, the score of a file is a function of the document frequency.
The multi-dimensional score of the file is then a combination of the individual dimension
scores. It is our belief that using a unified $IDF$ framework allows us to meaningfully
combine scores on several orthogonal dimensions to provide a single result, as we will
show experimentally in Section 5.4.
We first give a brief overview of our query model (Section 5.2.1), we then present our $IDF$-based scoring strategies for each dimensions: content (Section 5.2.2), metadata (Section 5.2.3), and structure (Section 5.2.4). Finally, we show how we aggregate scores across dimension in Section 5.2.5.

### 5.2.1 Query Model

To perform multi-dimensional queries, we need a query language that can express metadata and structure conditions in addition to content searches. For this, we use the simplified version of XQuery [77] supported by PlanetP (Section 2.2) as our query language. The path structure in each query condition indicates the type of data (i.e., content, metadata, or structure) being queried. For example, query conditions containing the component “FileSysMetadata” in their paths indicate queries over metadata. The query from our earlier example would be expressed as follows:

```xml
FOR $i$ in /File[FileSysMetadata/FileDate = '03/21/07']
  FOR $j$ IN /File[ContentSummary/WordInfo/Term = 'proposal'
    AND ContentSummary/WordInfo/Term = 'draft']
  FOR $m$ IN /File[FileSysMetadata/FileType = 'pdf']
  WHERE $i/@fileID = $j/@fileID AND $i/@fileID = $m/@fileID
  RETURN $i/fileName
```

An answer to a query is a file that is relevant to one or more of the query conditions. Internal flags are used to specify whether only exact matches are allowed (filtering conditions) or whether approximate matches are considered (ranking conditions). If approximate matches are allowed, a score is assigned for each query condition based on how closely the answer (file) matches the condition (exact matches for filtering conditions have a score of 1).

In remainder of this section, we discuss our strategies for scoring approximate matches for query conditions in each dimension.
5.2.2 Scoring Content

We use standard indexing structures and scoring mechanisms from the IR literature [76] for query conditions involving text content. Specifically, we assign scores using a standard $TF \cdot IDF$ scoring function. Relaxation is already an integral part of $TF \cdot IDF$ since it scores files that contain only a subset of the terms as well as those containing all terms in the content query condition.

**Definition 1 (Content $TF \cdot IDF$ Score of a File)** For a given keyword query, $Q$, consisting of the terms $t_1, t_2, \ldots, t_n$, the content score of a file $F$, with respect to $Q$ is computed as:

$$score_{Content}(Q, F) = \sum_{i=1}^{n} (IDF_{t_i} \cdot TF_{t_i,F}) \sqrt{|F|}$$

with

$$TF_{t,F} = 1 + \log(F_t) \quad IDF_t = \log(1 + \frac{N}{N_t})$$

where $|F|$ is the total number of terms in the file, $F_t$ is the number of times the term $t$ appears in file $F$, $N_t$ is the number of files containing the term $t$, and $N$ is the total number of files.

Note that to stay consistent with traditional IR system, in this dimension we consider the $TF$ score of a match as well as its $IDF$ score. This stays in the spirit of our overall $IDF$-based framework, as the $TF$ score is only used to give additional weight information on the quality of the matches.

5.2.3 Scoring Metadata

Quite often systems that store information (i.e., file systems, dataspaces, and personal information systems) store metadata information alongside files. Such metadata may include file sizes, file owners, and various file timestamps (e.g., date created and date last modified). File extensions can also hint at the corresponding file types. Users often want to enhance their query with metadata conditions (e.g., file was accessed last week, file is a pdf document), but may not accurately remember the exact metadata values
for which they are looking. Therefore, allowing for some approximation in metadata conditions is desirable.

In this section we discuss our scoring strategies for metadata information. We develop the concept of metadata relaxation to score and retrieve approximate matches to metadata query conditions.

5.2.3.1 DAG Representation of Metadata Relaxations

We use a DAG indexing structure to support the scoring of relaxations on metadata conditions. In particular, we construct a DAG-based index for each type of searchable metadata to support both the retrieval and the scoring of results. In these indexes, possible metadata values that files can have are stored in the leaves of the DAG. Internal nodes of the DAG then represents progressive hierarchical generalizations of their children.

For example, Figures 5.1 and 5.2 represents (subgraphs of) the DAGs associated with file types (Figure 5.1(a)), file dates (Figure 5.1(b)), and file sizes (Figure 5.2).
Figure 5.2: Fragment of the indexing DAG for file sizes. The DAG spans the range of file size up to 8 MB. Portions of the DAG have been removed to simplify the presentation. Shown are the branches associated with files size of 16 bytes and 512 KB. Dashed lines represent locations where several nodes have been removed for space considerations. Highlighted nodes indicate the sequence of relaxations for a file size query “512 KB”.

For the file type DAG, each leaf represents a specific file type (e.g., .doc and .pdf) and contains a count as well as references to all files of that type. Each internal node represents a more general file type that is a union of the types of its children (e.g., Media is the union of Video, Image, and Music types) and thus a relaxation of its descendants. Correspondingly, each internal node contains the sum of the file counts of its descendant leaves. Note that the count maintained at each internal node is thus guaranteed to be greater than or equal to the count at any of its children.

Similarly, in the DAG for file dates, individual timestamps are represented at the leaf node level and internal nodes represent larger time periods spanning those represented by their descendants. The length of the represented time periods are based on, or on multiples of, actual calendric units (i.e., days, months, and years). As seen in Figure 5.1(b), DAG construction is based on the containment properties of these periods (i.e., hours are contained within days and days within months). Additionally, we observe
that there internal nodes whose periods do not overlap (i.e., months, days, years) and those which do overlap (i.e., 3 month intervals). At query time, this overlap allows the use of relaxations in which the query time is relatively centered. For example, given a timestamp for February 15th, the relaxation interval for 3 Months would consider all timestamps in the months of January, February, and March with the queried timestamp falling in the middle month. Similarly, a timestamp for March 15th, would consider timestamps in the months of February, March, and April with March being the middle month. This results in multiple paths existing from the root of the DAG to a given leaf node. The reverse, however, is not true, ensuring that for a given timestamp the sequence of relaxations periods is deterministic.

A similar DAG is constructed to represent file sizes (Figure 5.2). As most file systems contain a greater number of smaller files than larger files, the distribution of file sizes generally mirrors an exponential distribution with a long tail. In constructing relaxations intervals for this type of metadata it is desirable for each interval to match approximately the same number of files (or at least match a small number) to provide a good distribution of scores. This can be accomplished by allowing smaller files sizes to be represented by relaxation intervals of shorter length while larger files sizes be represented by intervals of longer length. Figure 5.2 is a sub-graph of a DAG for file sizes constructed in this manner showing two leaf nodes representing file sizes of 16 bytes and 512 KB. Each node is labeled with the range of values it represents. As before, each successive relaxation interval (traveling from leaf to root) represents a larger range of values. The difference in the ranges of two successive intervals is proportional to their sizes; a shorter interval has a smaller increase to reach the next interval. Intuitively, small increases for intervals representing larger file sizes are unlikely to result in many additional matches given their expected distribution, hence larger increases are employed.

To expand a DAG node, the range of each internal node, as shown by the node labelled “[0,1 MB)”, is sub divided into four sub-ranges whose lengths increase exponentially. When this division is not possible (e.g., with node labeled “[16 B, 18 B)”) a division based on individual bytes is used. This manner of division produces a skewed DAG in which the left size (representing small file sizes) has a greater depth than the
right size (representing larger file sizes). Given that we anticipate a greater number of smaller files than larger ones, the higher density of file sizes will be divided among a larger number of DAG nodes.

5.2.3.2 Scoring Metadata Relaxations

As mentioned, the metadata DAG indexes are used for scoring. Our IDF-based framework requires the score of a file to depend on how many files match a given relaxation of a query condition. Given a specific metadata condition, the path from the matching leaf to the root of the DAG index for a metadata type represents all of the approximations that we can score for that condition. For instance, the query condition `FileType='pdf'` in our earlier example (Section 5.1) would exactly match the leaf `.pdf` in the DAG example of Figure 5.1(a). The leaf’s ancestor nodes, `Documents` and `All Files`, represent categories for approximate matches.

Continuing the example, our IDF-based scoring approach then scores matching files as follows. Files of type `.pdf` would have the highest score as they are exact matches to the query condition. Files of type `Document`, other than type `.pdf`, (e.g., `.doc` and `Code`) would be assigned a lower score. Finally, files of type `All Files`, other than type `Document`, which would consist of all remaining files in the system, would be assigned yet a lower score. The latter two assigned scores are for the two approximations of the query condition as they are assigned to files that do not match the query condition exactly.

**Definition 2 (Metadata Condition Score of a File)**  For a given metadata query, $Q$, consisting of a target value $v_Q$ for a metadata condition $C$, the metadata score of a file $F$, with corresponding metadata value $v_F$, with respect to $Q$ is computed as:

$$
\text{score}_{\text{MetaData}}(Q, F, C) = \frac{\log(N)}{\log(\text{fileDesc}(\text{commonAnc}(v_Q, v_F)))}
$$

where $N$ is the total number of files, $\text{commonAnc}(x, y)$ returns the closest common ancestor of nodes $x$ and $y$, and $\text{fileDesc}(x)$ returns the files that can be reached through the descendants of node $x$ in the metadata DAG. The score is normalized by $\log(N)$ so
that a single perfect match would have the highest possible score of 1. The most relaxed matches to the condition \( C \) will have a score of 0 as their closest common ancestor with \( v_Q \) is the root node of the DAG which contains all \( N \) files as its descendants.

Intuitively, we find the closest common ancestor of \( v_Q \) and \( v_F \) in the metadata DAG, and count the number of files that can be reached through descendants of this common ancestor. The higher this number is, the lower the score of \( F \) for \( Q \) will be as many other files share the same level of approximation with \( Q \) as \( F \).

### 5.2.3.3 Aggregating Metadata Scores

For queries involving multiple metadata conditions (e.g., our example query, with a condition on date and a condition on filetype) the individual condition scores have to be aggregated to produce a unified metadata score.

We aggregate individual metadata scores by considering both the query and the document as vectors of dimension \( n \), where \( n \) is the number of individual metadata conditions \( C_1, ..., C_n \). The document vector \( \vec{V}_F \) consists of the individual \( \text{score}_{\text{MetaData}}(Q, F, C_i) \) (\( 1 \leq i \leq n \)). The query vector \( \vec{V}_Q \) has value 1 (exact match) for each dimension. The unified metadata score is then the normalized length of the projection of the document vector on the query vector.

**Definition 3 (Metadata Score of a File)** For a given metadata query \( Q \) with corresponding query vector \( \vec{V}_Q \), consisting of several metadata value conditions \( C_1, ..., C_n \), the metadata score of a file \( F \) with corresponding document vector \( \vec{V}_F \), with respect to \( Q \) is computed as:

\[
\text{score}_{\text{MetaData}}(Q, F) = \frac{\vec{V}_F \cdot \vec{V}_Q}{|\vec{V}_Q|}
\]

Note that if only one metadata condition \( C \) is present in \( Q \), then \( \text{score}_{\text{MetaData}}(Q, F) = \text{score}_{\text{MetaData}}(Q, F, C) \).

### 5.2.4 Scoring Structure

Most users organize their files into a hierarchical directory structure for navigation. In addition, the structure within a document can be seen as an extension of the directory
path structure and used for more complex query searches [22]. However, users are notoriously bad at remembering where they stored a particular file or how the files are structured [14]. When a user searches for a file using structure information such as directory path information, the query is likely to be incorrect, as users often confuse or misremember the order of the directories, their relationships, or their labels. However, it is common that users do correctly remember some portion of the path whether it be a prefix or several (possibly non-consecutive and out-of-order) directory names. Therefore, allowing for a method of approximation that leverages any correct information in an otherwise incorrect (when taken as a whole) path is desirable.

Previous work [56, 75] has explored possible scoring strategies for the structure information of files. While very relevant to this section, this work was done outside the context of this thesis. We will review the approach presented in this work in so far as it is relevant to presenting a unified scoring approach. We will begin by outlining the structural relaxations required to handle the specific needs of user searches in a file system. We will then briefly describe the DAG used to compute and score structural relaxations.

Assuming that structure query conditions are given as pathnames, these relaxations are:

- **Edge Generalization** is used to relax a parent-child relationship to an ancestor-descendant relationship. For example, applying edge generalization to /a/b would result in /a//b.

- **Path Extension** is used to extend a path $P$ such that all files within the directory subtree rooted at $P$ can be considered as answers. For example, applying path extension to /a/b would result in /a/b//*. 

- **Node Deletion** is used to drop a node from a path. For example, applying node deletion on $b$ from /a/b/c would result in /a//c. Note that the edge between $a$ and $c$ is generalized. This is to preserve a necessary containment property [56, 75].

- **Node Inversion** is used to permute nodes within a path. For example, applying
node inversion on $b$ and $c$ from $/a/b/c$ would result in $/a/(b/c)$, allowing for both
the original query condition as well as $/a/c/b$.

These above relaxations can be applied to the original query condition as well as
relaxed versions of the original condition. We then say that a file *matches* a (possibly
relaxed) query condition if all structural relationships between the condition’s com-
ponents are preserved in the file’s parent directory. For example, the file $/a/b/c/f$
matches the condition $/a//c$ because the parent-child relationship between $/$ and $a$
and the ancestor-descendant relationship between $a$ and $c$ are preserved in the directory
$/a/b/c$.

![Figure 5.3: The structure DAG for the structural query condition Personal/Ebooks/JackLondon. Solid lines represent parent-child relationships. Dotted lines represent ancestor-descendant relationships, with intermediate nodes removed for simplicity of presentation.](image)

### 5.2.4.1 DAG Representation of Structure Relaxations

As proposed in [2], a DAG is used to represent all possible structural relaxation of
a path query condition. The DAG structure is used not only to compute and store
score information but also for query processing, as it allows us to incrementally access
increasingly relaxed answers during query processing (Section 5.3). Figure 5.3 shows
an example relaxation DAG, along with example *IDF* scores, for the structure query.
condition Personal/Ebooks/JackLondon. At the root of this DAG is a node representing the exact query condition itself, with each non-root node representing a relaxed form. Note that this structure DAG is a query specific in that is a representation of the query and all possible relaxed forms of that query given the above relaxation operations. This is in contrast to the indexing metadata DAGs in Figure 5.1. which are an actual index of data in the file systems. For each DAG node, we compute an IDF score. Matches for the exact query P/E/J have a score of 1, while matches to increasingly relaxed versions of the query, as we go down the DAG, have lower scores, with matches to the most general relaxation of P/E/J: //∗ having a score of 0. Algorithms exist for the efficient construction and evaluation of these DAGs [56, 75].

5.2.5 Aggregating Multi-dimensional Scores

A strength of our scoring framework is that all dimensions are scored using a similar IDF metric, which takes into account the number of files that match a particular query condition (or relaxation of that condition). This unified framework allows us to meaningfully aggregate scores across different query dimensions.

The individual dimension scores are aggregated to produce the final score of a file for a query. We use a vector projection for the aggregation of multi-dimension scores, similar to the one we used for aggregating individual metadata condition scores (Section 5.2.3.3). We build a 3-dimension file vector $\vec{V}_F$, which consists of the (normalized) three dimension (content, metadata, and structure) scores. For a query $Q$ and a file $F$, we have:

$$
\vec{V}_F = (\text{score}_{\text{Content}}(Q,F), \text{score}_{\text{MetaData}}(Q,F), \text{score}_{\text{Structure}}(Q,F))
$$

The query vector $\vec{V}_Q$ has value 1 (exact match) for each dimension. The file multi-dimensional score is the normalized length of the projection of the document vector on the query vector.
Definition 4 (Query Score of a File) For a given query, $Q$ with corresponding query vector $\vec{V}_Q$, the score of a file $F$ with corresponding document vector $\vec{V}_F$, with respect to $Q$ is computed as:

$$score(Q, F) = \frac{\vec{V}_F \cdot \vec{V}_Q}{|\vec{V}_Q|}$$

Note that our aggregation assigns the same importance to each dimension in the query. We could easily incorporate weights in our aggregation function to give more importance to one or more dimension.

5.3 Implementation

In this section we present details of our implementation, focusing specifically on the design of the various dimensional indexes and efficient query processing.

5.3.1 Indexing Structures

We have implemented our approach as part of the PlanetP toolkit (Section 2.2). In doing so we have extended the metadata and structural tables described in Chapter 3.7 with several data structures and added a top-$k$ query processing algorithm to efficiently find the top $k$ relevant results. In the remainder of this section, we will present relevant design details pertaining to both of these aspects of our implementation.

5.3.1.1 Metadata Index

We have extended the Metadata table with five query indexes to support efficient searches on the size, type, date-of-creation, last-modified-time, and last-accessed time attributes. Each of these query indexes is used to obtain the necessary scoring information need for our DAG-based approach.

For query indexes storing discrete values (i.e file type), the index structure employed is a predefined statically constructed DAG similar to the one presented in Figure 5.1 for file types. As references to files are stored at the appropriate leaf nodes and the path from a leaf node to the root represents the progressive relaxations steps of a specific file type, this static DAG is used both for storage and scoring.
Query indexes for continuous values (i.e. size, time-stamps) use quad-trees [20] for storage. Given a large range of values, quad-trees can efficiently store data by employing a dynamically constructed tree in which each node represents a finite non-overlapping sub-range. Nodes are recursively divided into smaller ranges if the amount of data they are storing exceeds a predefined threshold. Thus, a sparse sub-range may be represented by very few (possibly even one) node(s). The tree structure and the assignment of non-overlapping ranges to nodes permits efficient querying of data for a given range query.

At query time the required scoring information is determined as follows: 1) Given a query value, \( x \), determine the structure of the relaxation DAG for \( x \). We need not create the entire DAG (as seen in Figure 5.1(b)) but rather only the path from the leaf node to the root. This path represents all the relaxations that will be considered for the query value of \( x \) during evaluation. We have opt to compute this path at query time rather then keep an entire pre-computed DAG in memory. 2) Given the relaxation DAG (or relevant path), evaluate the sub-ranges associated with each node against the quad-tree to determine the number of matching files associated with the respective relaxation. For example, in Figure 5.1(b) for the nodes labeled “Day”, the quad-tree would be queried for all times that fall within the 24 hours period containing the query value. 3) Given the file counts, compute the scores for each DAG node as discussed in Section 5.2.3.

### 5.3.1.2 Structural Index

Numerous additional data structures were added to the structural table to support the construction and efficient evaluation of the structural DAGs presented in Chapter 5.2.4. A detailed discussion of these can be found in Wang et al. [56, 75].

### 5.3.2 Efficient Query Processing

Our query model (Section 5.2) supports ranked retrieval of answers based on how closely they match a query. Previous work on top-\( k \) query processing has shown that evaluating all possible matches to return the \( k \) best answers is prohibitively expensive. In our scenario, this would lead to scoring and ranking every single file in the system for each
query. Several top-\textit{k} query processing techniques have been proposed in the Database community in recent years. We have adapted the Threshold Algorithm (TA) \cite{26} to our scenario. TA uses a threshold condition to avoid evaluating all possible matches to a query, instead focusing on identifying the \textit{k} best answers.

\textit{TA} takes as input several sorted lists, each containing the system’s objects (files in our scenario) sorted in descending order according to their relevance scores for a particular attribute (dimension in our scenario) and dynamically accesses the sorted lists until the threshold condition is met to find the \textit{k} best answers without evaluating all possible matches to a query. In our adaptation, TA uses the aggregation approach described in Section 5.2.5 to compute the unified relevance score for each candidate object that it considers.

\textit{TA} relies on \textit{sorted} and \textit{random} accesses to retrieve individual attribute scores. Sorted accesses, that is, accesses to the sorted lists mentioned above, require the files to be returned in descending order of their scores for a particular dimension. Random accesses require the computation of a score for a particular dimension for any given file. Random accesses occur when \textit{TA} chooses a file from a particular list corresponding to some dimension and then needs the scores for the file in all the other dimensions to compute its unified score. To use \textit{TA} in our scenario, our indexing structures and algorithms need to support both sorted and random access for each of the three dimensions.

Finally, recall that our scoring approach requires a single score be computed for each queried dimension. In the case of content and metadata, several query conditions may be involved in each computation. For content this occurs when querying for multiple keywords and for metadata, querying over several different types of metadata in the same query. To compute the final score, individual dimension scores must be computed first. To make this process efficient, we employ a two tiered top-\textit{k} framework. Each dimension executes the above \textit{TA} top-\textit{k} algorithm separately to compute its top-\textit{k} results. These results are then aggregated using a concurrently running \textit{TA} top-\textit{k} algorithm across all queried dimensions.
5.4 Experimental Evaluation

In this section, we will present our the experimentally validation of our $IDF$-based scoring approach and evaluate the potential for the corresponding multi-dimensional fuzzy search approach to improve relevance ranking. We also report on query performance to show that multi-dimensional search is practical to use. This evaluation will be done in a single node environment and will be used for comparison in the next chapter where we investigate multi-dimensional fuzzy search in a federated environment.

5.4.1 Experimental Setting

5.4.1.1 Platform

All experiments were performed using the Wayfinder file system. Experiments were run on a PC with a 64-bit hyper-threaded 2.8 GHz Intel Xeon processor, 2 GB of memory, and a 10K RPM 70 GB SCSI disk, running the Linux 2.6.16 kernel and Sun’s Java 1.5.0 JVM.

5.4.1.2 Data Set

As noted in [22], there is a lack of synthetic data sets and benchmarks to evaluate search over personal information management systems; therefore we used a real user data set comprised of (a representative subset of) files and directories from the working environment of a user in our lab. This data set contained 24,927 files in 2,339 directories. 24% of this data set were multi-media files (e.g., music and pictures), 17% document files (e.g., pdf, text, and MS Office), 14% email messages,\(^1\) and 12% source code files. The average directory depth was 3.4 with the longest being 9. On average, directories contained 11.6 sub-directories and files, with the largest—a folder containing emails—containing 1013. Wayfinder extracted 347,448 unique stemmed content terms.\(^2\) File modification dates spanned 10 years. 75% of the files were smaller than 177 KB, and 95% of the files were smaller than 4.1 MB.

---

\(^1\)Email messages are stored in the Maildir format in which each email is stored in a separate file.

\(^2\)Content was extracted from MP3 music files using their ID3 tags.
5.4.1.3 Query Set

For several experiments, we require a large number of queries to evaluate our system. In the absence of a meaningful benchmark, we create a set of synthetic queries for this evaluation. These queries are designed to include a large variety of query conditions combined in different ways across three or four dimensions. The query specifically targets files from two content categories: emails and documents. In the document category, we specifically target files belonging to the sub-categories of ebooks, code files, and academic papers. For each of the four categories (i.e., emails, ebooks, code, and papers), we generate 50 random queries. This diverse set of queries allow us to explore the performance of our system across the parameter space that should include most real-world search scenarios.

The construction of these queries has been described in previous work [75] and we review it here. It has been noted [72] that individuals often know exactly what they are looking for when they execute searches for an email message, a file, or even a Web page. Thus, each of our query targets a specific file \( f \), and so is built using (relaxations of) \( f \)'s attributes. The query conditions are formed as follows.

- **Content:** Each condition has 2 to 4 terms chosen randomly from the top 50 terms (based on TF values) in \( f \)'s content.

- **Metadata:** Each date (last modified) is randomly chosen from a small range (\( \pm 7 \) days to represent cases where users are searching for files they recently worked on) or a large range (\( \pm 3 \) months to represent cases where users are searching for files that they have not worked on for a while and so only vaguely remember the last modified times) around \( f \)'s actual last modified date. Each file type (extension) is randomly chosen from \( .txt \) or \( .pdf \) for a document; otherwise, it is the correct file type. For file sizes, three ranges based on the queried values are constructed; one that spans the range of numbers with the same order of magnitude (i.e., tens, thousands, and millions), one that spans numbers that are one order of magnitude less, and the last for numbers which are one order of magnitude greater. Of these three, one is chosen randomly and then middle number of that range is computed.
and use.

- **Structure**: Each condition is randomly chosen from: (a) the correct path \( p \), (b) one random word was dropped from \( p \), (c) two adjacent words in \( p \) are swapped, and (d) one word in \( p \) was misspelled.

### 5.4.2 Behaviors of Scoring Functions

We start by studying the behaviors of our scoring functions. Figure 5.4 plots the scores of all relevant files as a function of their ranking for several 1-dimensional queries targeting three different dimensions as follows. Figure 5.4(a) plots the normalized scores for four content queries that contain from one to seven keywords. For each query, the scores of matching files were normalized against the highest score since content scoring is based on \( TF \cdot IDF \) as opposed to just \( IDF \). Figure 5.4(b) plots the scores for three
structure queries. The first query is a standard path query that corresponds to a path that exists in the directory tree (of the form /a/b/c/d), the second is the original path query after deleting a node (/a/b/d), and the third is the original path query after permuting two nodes (/a/b/d/c). We choose these queries to exhibit common user mistakes in querying structure. Finally, Figure 5.4(c) plots the scores for three metadata queries for three different file sizes.

First and foremost, we observe that all scoring functions have similar behaviors. In all cases, the relevance scores monotonically decrease (by design) as files become less similar to the query conditions. Most critically, all scoring functions allow a large number of files that do not exactly match the query conditions to be scored and ranked, providing the desired flexibility over filtering. This is particularly important when a query condition such as a non-existing directory is provided in the query; in such cases, filtering would not consider any file in the system as being relevant to the query.

On the other hand, there are several interesting differences between the scoring function for content and the scoring functions for the other dimensions. In particular, the scoring functions for structure and metadata (Figures 5.4(b-c)) are noticeably plateau-shaped because of our DAG-based approach to computing IDF. In this approach, each relaxation step is likely to bring a set of files that are deemed to be equally similar to the query condition. For instance, in Figure 5.4(c), each plateau corresponds to a discrete relaxation interval to which a range of file sizes has been mapped.

Plateaus in the scoring function, where many files are assigned the same score, can make a query dimension less useful for ranking. For metadata, we can arbitrarily smooth the scoring function as long as files do not have exactly the same attributes (e.g., same size) by considering increasingly smaller relaxation intervals. In contrast, this is not possible to do for structure, as the relaxation intervals are defined by the matching directories and the number of files inside each directory, which is under user control. (Users can aid the search engine by using sparser directory structures.)

The content scoring function also differ from the other two in its sharp drop from the top several ranked results to the 10th-100th ranked results and its non-zero scoring of a much smaller subset of the files in the system. These differences do not seem
fundamental, however. For example, if we choose a set of content terms that appear in most files, then the content scoring function would likely look more like the other two.

Thus, despite the differences, we conclude that the data in Figure 5.4 makes a strong case for combining relevance scores from orthogonal query dimensions using our framework as it places these dimensions into a common setting to be compared. The differences mentioned likely provide opportunities to explore more complex score aggregation approaches in future work.

5.4.3 Scores and Rankings for Approximate Answers

We now show how scoring (and the corresponding ranking) is affected by inaccurate query conditions. For this purpose, we choose a target file that is an exact match for the query conditions of a particular query, resulting in a high score and rank. We then modify the target file so that it is progressively farther away from the query conditions; that is, the file will only match increasingly relaxed approximations of the query conditions. While relaxing the target file, we alter several other files as needed to ensure that any global statistics used in scoring computations are kept constant. This ensures that scores for files unrelated to our relaxation process are unaffected, providing a stable background for interpreting the changing score (and rank) of the target file.

Figures 5.5(a-d) plot the score and rank, respectively, of the target file for two representative 2-dimensional queries covering the four dimensions of content, structure, size metadata, and (last modified) date metadata. In the content dimension, we relax the file by progressively removing occurrences of the query term from within the file. In the structure dimension, we relax the file by progressively moving the file up the directory tree (representing simple relaxations steps) from its original location. In the size metadata dimension, we relax the file by progressively decreasing its size (We also relax in the opposite direction but do not plot the results because they are not significantly different). In the date metadata dimension, we relax the file by progressively moving its date backward from the query condition.
Figure 5.5: Score (a-b) and rank (c-d) of a target file returned as a result of a constant 2-dimensional query as the file is relaxed away from the query conditions across the two query dimensions. Score and rank of the same file target file plotted for a content-only query (e) and a structure-only query (f).
It should be noted that in Figure 5.5, the target file is not the only exact match to the query condition in the structure, size, and date dimensions, leading to a relevance score of less than 1 in each of these dimensions even before relaxation of the file. Also, the target file is not returned as the top result to the content-only query leaving again a score of less than 1. This arises from the fact that our data set has several small files that contain a subset of the query terms. As our $TF \cdot IDF$ scores are normalized by file lengths, these smaller files achieve higher scores than the target file, which is a novel containing over 100,000 terms.

Figures 5.5(a-b) show that the combined scoring functions for a multi-dimensional query behave as expected; that is, they preserve the trends of the 1-dimensional scoring functions, decreasing as the file is relaxed away from the query conditions in either dimension; we plot the score and rank of the target file as it is relaxed against 1-dimensional content and structure queries in Figures 5.5(e-f) for comparison purposes.

More interestingly, we observe that providing query conditions for other dimensions in addition to content, even when the provided query values are somewhat inaccurate, can significantly improve ranking accuracy. For example, when the target file contains only 5 of the 7 terms (-2t) in the content query, its rank drops to around 50 (Figure 5.5(e)). When we provide an approximate structural value, the parent directory of the directory containing the target file, the ranking jumps to close to 10 (Figure 5.5(c)). Similarly, if we provide information on the file’s size and date, even when the size is 8% off and the date is incorrect by 1 month, the target file is ranked 32nd (Figure 5.5(d)). Interestingly, sometimes inaccurate query conditions on one dimension do not affect ranking, as is shown in (Figure 5.5(d)) where a slight approximation in the query condition of one dimension, provided the other dimension is exact, still results in a rank of 1 since few exact matches to each individual dimension exist in the data set.

Of course, providing incorrect query values can hurt ranking as well. For example, providing an ancestor directory two level up pulls the rank of the target file down to around 100, even when the file contains all 7 query terms in the content dimension. Keep in mind, however, that providing incorrect non-content query values to current filtering approaches may prevent the target file from being ranked at all. Thus, in
Table 5.1: The rank of a target file—the novel Sea Wolf by Jack London—returned by a set of related queries. In the presence of ties in the relevance scores, the highest rank the target file could have is given. The queried dimensions include Content, Type (Metadata), Date (Metadata), and Structural. The initial content query Q1 provides the set $C$ containing the 4 query terms \{jack, london, sea, wolf\}. Structural values are abbreviated. The complete path of our target file is `/Personal/Ebooks/Novels/JackLondon/`. Queries which contain a “*” in the first column represent those in which the target file would not be considered as a relevant answer given today’s typical filtering approach.

<table>
<thead>
<tr>
<th>Query</th>
<th>Content</th>
<th>Type</th>
<th>Date</th>
<th>Structure</th>
<th>Rank</th>
<th>Comments on Relaxation from Query Q1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>49</td>
<td>Base Query</td>
</tr>
<tr>
<td>Q2</td>
<td>C</td>
<td>.txt</td>
<td>26 Feb 07 16:08</td>
<td>/p/e/n/j</td>
<td>1</td>
<td>Correct Values (all dim.)</td>
</tr>
<tr>
<td>Q3</td>
<td>C</td>
<td>.txt</td>
<td></td>
<td></td>
<td>6</td>
<td>Correct Value</td>
</tr>
<tr>
<td>Q4</td>
<td>C</td>
<td>.pdf</td>
<td></td>
<td></td>
<td>1026</td>
<td>Incorrect Value</td>
</tr>
<tr>
<td>Q5</td>
<td>C</td>
<td>.doc</td>
<td></td>
<td></td>
<td>45</td>
<td>Incorrect Value</td>
</tr>
<tr>
<td>Q6</td>
<td>C</td>
<td>Docs.</td>
<td></td>
<td></td>
<td>21</td>
<td>Relaxed Range</td>
</tr>
<tr>
<td>Q7</td>
<td>C</td>
<td></td>
<td>26 Feb 07</td>
<td></td>
<td>5</td>
<td>Relaxed Range (Day)</td>
</tr>
<tr>
<td>Q8</td>
<td>C</td>
<td></td>
<td>25-28 Feb 07</td>
<td></td>
<td>5</td>
<td>Relaxed Range (Week of month)</td>
</tr>
<tr>
<td>Q9</td>
<td>C</td>
<td></td>
<td>Feb 07</td>
<td></td>
<td>7</td>
<td>Relaxed Range (Month)</td>
</tr>
<tr>
<td>Q10</td>
<td>C</td>
<td></td>
<td>27 Feb 07 16:08</td>
<td></td>
<td>9</td>
<td>Incorrect Value (off by 1 day)</td>
</tr>
<tr>
<td>Q11</td>
<td>C</td>
<td></td>
<td>19 Feb 07 16:08</td>
<td></td>
<td>14</td>
<td>Incorrect Value (off by 1 week)</td>
</tr>
<tr>
<td>Q12</td>
<td>C</td>
<td></td>
<td>26 Mar 07 16:08</td>
<td></td>
<td>150</td>
<td>Incorrect Value (off by 1 month)</td>
</tr>
<tr>
<td>Q13</td>
<td>C</td>
<td></td>
<td></td>
<td>/p/e/n/j</td>
<td>3</td>
<td>Correct Path</td>
</tr>
<tr>
<td>Q14</td>
<td>C</td>
<td></td>
<td></td>
<td>/p/e</td>
<td>13</td>
<td>Prefix of Correct Path</td>
</tr>
<tr>
<td>Q15</td>
<td>C</td>
<td></td>
<td></td>
<td>/j/e</td>
<td>3</td>
<td>Incorrect Order/Correct Names</td>
</tr>
<tr>
<td>Q16</td>
<td>C</td>
<td></td>
<td></td>
<td>/p/e/n/h</td>
<td>11</td>
<td>Incorrect Path</td>
</tr>
<tr>
<td>Q17</td>
<td>C</td>
<td>Docs.</td>
<td>Feb 07</td>
<td>/p/e/n</td>
<td>3</td>
<td>Relaxed Range (all Dim.)</td>
</tr>
<tr>
<td>Q18</td>
<td>C</td>
<td>.pdf</td>
<td>19 Feb 07 16:08</td>
<td>/j/e</td>
<td>36</td>
<td>Incorrect Values</td>
</tr>
<tr>
<td>Q19</td>
<td>C</td>
<td>.pdf</td>
<td>19 Feb 07 16:08</td>
<td>/j/e</td>
<td>2</td>
<td>Incorrect Values (all Dim.)</td>
</tr>
</tbody>
</table>

5.4.4 Impact of Flexible Multi-Dimensional Search

In the last section, we demonstrated the general trends of multi-dimensional scoring to validate that our combined scoring function behaves as desired. We also argued that providing fuzzy query conditions in non-content dimensions has the potential to significantly improve scoring (and thus ranking) accuracy. In this section, we explore this latter potential and compare our approach against current filtering approaches in more detail.

In this study, we initially construct a content-only query intended to retrieve a specific target file and then expand this query along several other dimensions. We start
with a base content-only query because content-only queries are the standard search interface in many real world systems. For each query, we consider the ranking of the target file by our approach together with whether the target file would be ranked at all by today’s typical filtering approaches on non-content query conditions.

Table 5.1 summarizes the results of our study. The target file is the novel Sea Wolf by Jack London and the set of query content terms, $C$, in our initial content-only query, Q1, contains the four terms sea, wolf, jack, and london. While the query is quite reasonable, the terms are generic enough that they appear in many files, leading to a ranking of 49 for the target file. Query Q2 augments Q1 with the exact matching values for file type, date, and containing directory. This brings the rank of the target file to 1. Of course, this result by itself is not meaningful because it is unlikely that the user will remember the file attributes with such precision.

In the remainder of the table, we explore what happens when we modify the query in two different ways for the non-content dimensions: (1) instead of the precise correct value we provide a range around the precise value, and (2) we provide an incorrect value. The results are quite promising. For example, in query Q15, just getting a couple of components correct in the directory name—note that the components are given in an incorrect order—brings the ranking up to 3. Providing an incorrect directory that shares a common prefix with the correct directory brings the ranking to 11 (query Q16). In contrast, if such directories were given as filtering conditions, the target file would be considered irrelevant to the query and not ranked at all.

Similar results can be seen for most other queries marked with a “*” in the first column (indicating that the target file would not be found using a filtering-only approach). Two exceptions include queries Q4 and Q12. In these queries, the incorrect values for the other dimensions reduce the ranking of the target file below that achievable with only content. For query Q4, this decreased ranking is because there are many pdf documents that achieve a higher metadata score than the target file. Similarly, for

---

3We do not consider incorrect ranges, that is, a range that does not include the value of the target file, because the results are similar to incorrect values; filtering would not rank the target file and in our approach, while the scores change, the ranking does not change significantly.
query Q12, many files with dates closer to the query condition achieve higher metadata scores. Given that the ranking in these two cases are 1026 and 150, our approach is not meaningfully different from filtering since users are unlikely to look that far down a ranking list.

Using ranges also give promising results although our approach is unlikely to outperform filtering (when the matching value of the file attribute is included in the range so that the file is not filtered). Intuitively, however, we believe it is easier to provide an approximate query condition and allowing the search engine to rank all files based on their similarity to the condition than it is to guess at the correct filtering range, which may require overfitting or increasing the range in several query iterations.

Based on the above results, we conclude that our approach of providing flexible query conditions for non-content search dimensions has the potential to considerably improve search accuracy over current filtering approaches. Future work could involve validating this potential in more extensive user studies.

5.4.5 Impact of Multi-dimensional Scoring on Results

To complement the last section, where we studied the ranking of a single target file with respect to a set of related queries, we now consider the impact of our scoring approach on the entire set of top-k files returned in answer to a query. Specifically, we compare the query results for several multi-dimensional queries with those of a content-only query. To measure the impact of our techniques, we use the minimized Spearman’s rho as described in [25]. The standard Spearman’s rho (\( \rho \)) measures the distance between \( l_1 \) and \( l_2 \), two permutations of the same list. The minimized Spearman’s rho (\( \rho_{min} \)) is an adaptation of the standard Spearman’s rho to top-k lists, which may not overlap. We normalize the minimized Spearman’s rho between -1 and 1, where a score of -1 means that objects in the two top-k lists are disjoint, and a score of 1 means the two lists are identical:

\[
\rho_{min} = 1 - \frac{6 \sum d_i^2}{k(k+1)(2k+1)}
\]
Figure 5.6: $\rho_{\min}$ value for various multi-dimensional queries as a function of $k$.

where $k$ is the number of results returned, $d_i$ is the difference in rank between each object that appears in $l_1$ or $l_2$; an object that does not appear in one of the list is considered to have a rank of $k + 1$ in that list.

Figure 5.6 shows the $\rho_{\min}$ values for various multi-dimensional queries as a function of $k$. We use two different queries, A and B, to which we add dimension conditions. For Query B we see that the addition of either metadata or structure conditions has only a slight effect on the overall results (indicated by the respective lines staying above 0.5). The combination of both, however, results in significant changes to the set of results. In contrast, for Query A the addition of the metadata dimension provides us with a spearman score ranging from 1.0 to -0.4 indicating that as we increase $k$ the results change significantly. This indicates that the set of files relevant to the content condition of the query and the meta-data condition are quite different. The addition of the structural condition lessens this trend.

Our results show that the multi-dimensional scoring modifies the top-$k$ results with the impact being the most visible for smaller values of $k$. We have also shown that the degree of change is dependent on the conditions with which the query is extended. Set of conditions whose relevant files are similar will result in very little movement, or introduction of new results, into the final top-$k$ files.
Table 5.2: Query performance for various single- and multi-dimensional queries for both in-memory indexes/tables and persistent indexes/tables. $k$ is set to 20.

### 5.4.6 Query Performance

Until now our evaluation has focused on measuring the impact of our unified scoring framework on query results. In this section, we now turn to evaluating its performance. For this evaluation, we utilize a set of queries randomly generated queries (Section 5.4.1.3). Fifty queries were generated for each of the four content categories (including sub-categories).

We have implemented several top-$k$ query optimization techniques to speed up query evaluation (Section 5.3.2). Our techniques ensure that the correct top $k$ answers for a query, according to our unified scoring framework, are returned to the user.

Table 5.2 shows the query performance of several single- and multi-dimensional queries using both completely in-memory indexes/tables and persistent indexes/tables. Recall that our persistent indexes/tables were implemented using the BerkeleyDB [64] via its Java API. Each reported number is an average of all the query evaluations for the respective configuration.

Immediately noticeable are the larger times for both content and structure when compared to either size and date. The large content times result from our current unoptimized index/table design. Processing a query for each term currently requires the retrieval of the entire list of all files that contain that term. For structure, the larger times result from constructing and evaluating the structural DAG at run time.
Figure 5.7: (a) The mean time for queries targeting email and documents plotted as a function of the data set size. (b) CDF of the total query time for all queries targeting email and documents. $k$ is set to 20.

The increases in time between the in-memory and persistent index/table stems from the need to read data from the persistent data store (implemented using BerkeleyDB). We have implemented several simple caching mechanisms to minimize these accesses. We believe, however, significant opportunities for optimization remain.

The difference in times between one dimension and the multi-dimensional searches is largely due to the overhead of top-$k$ processing. While it may be cheap to access the top results for a single dimension using sorted indexes, a multi-dimensional search may require to accessing files (via more expensive random accesses) that have low scores in one or more dimensions.

Future work could investigate additional methods to further improve performance. Among these are more aggressive caching techniques to further minimize access to disk, adjustments to our top-$k$ algorithm that will reduce it computational cost. Our results show reasonable query response times. Since these measurements were taken in an early, mostly unoptimized prototype, we believe our fuzzy multi-dimensional scoring approach is practical for implementation in real systems.
5.4.7 System Scalability

We believe that our experimental data set is sufficiently large that our performance results apply directly to personal information management systems. Nevertheless, we briefly study the scalability of our system to assess its potential to handle very large personal data sets. We again use a set of randomly generated queries described earlier. Figure 5.7(a) plots average query times against data set size and Figure 5.7(b) presents the CDF of the total query times for these queries for different data set sizes. These result show that query performance scales linearly with data set size but with a relatively flat slope (e.g., increase of only 0.1 seconds in mean query processing time when the data set doubles in size). Further, analysis shows that the linear growth is directly attributable to our unoptimized implementation of the top-k algorithm; score evaluation times remain relatively constant vs. data set size. This result is quite promising because there are many known optimizations that we can apply to improve the performance and scalability of the top-k algorithm.

Along a different dimension, in Figure 5.8 we present measurement of query performance for increasing values of k. Results show that our approach scales very well with k. For example, the 90th percentile processing time (i.e., the time within which 90% of the queries completed) only increased from 0.95 seconds for k = 10 to 1.38 seconds for k = 50 to 2.28 seconds for k = 200. Average and median query processing times followed the same trend.
5.4.8 Storage Cost

To show that our approach is practical with respect to space (i.e., storage cost), we now report on the cumulative size of our indexes and tables for storing the data set used in this evaluation. In total, our indexes/tables require 273 MB of storage to store the data set defined in Section 5.4.1.2. This is less than 2% of the data set size (14.4 GB). Data for the metadata dimension accounts for approximately for 6% of the total storage with 17.5 MB, of which 11.2 MB is accounted for by the metadata table and 6.3 MB by the query indexes (Section 5.3.1). Storage costs are dominated by the content table, which accounts for about 92% of the total space with 252 MB. The indexes/tables are so compact compared to the data set because of the large sound (music) and video (movie) files. As future data sets will be increasingly media rich, we expect that our indexes/tables will continue to require a relatively insignificant amount of storage.

5.4.9 Comparison of Third Party Search Tools

In Section 5.1, we discussed the existence of several tools providing advanced search functionality on users' desktops. As our evaluation has presently focused on a single device setting, we attempted a comparative evaluation of Wayfinder with several of these tools; specifically Copernic Desktop Search (CDS) [13] (Version 2.1.1), Google Desktop (GD) [29] (version 5.10707), and Spotlight (SL) [66]. In each instance we supplied the application with the data set presented in Section 5.4.1 and provide sufficient time to index. We then manually ran a set of queries and compared the results.

Our findings were ambiguous at best. All three applications were sufficiently different to Wayfinder, and with each other, that comparing results in a meaningful manner is difficult. Each application dealt differently with key tasks such as parsing, ranking, and result presentation. For example, all three applications (as well as Wayfinder) disagree on the set of file types to parse: GD does not seem to parse latex files while SL does, and CDS does so only after the indexing settings are manually altered. Additionally, while Wayfinder, SL, and CDS consider the entire file when parsing, GD indexes only the first 5000 terms in a document.
With regards to ranking of results, only Wayfinder, SP, and GD allow this functionality. SP, however, will only use ranking to determine the single most relevant result and in GD the ranking of results must be explicitly selected as it is not the default behavior. Finally, the difference in scoring/ranking and parsing further complicates any comparison. As expected, however, we were able to validate the results of Section 5.4.4 by locating specific target files with inaccurate queries using Wayfinder whereas the filtering-based search tools failed.

5.5 Summary

Contemporary search tools use a combination of ranking and filtering to evaluate multi-predicate queries provided by users. These tools apply ranking algorithms to the content predicate of a query while any remaining predicates are used as filtering conditions. We have argued in this chapter that this approach is insufficient in many search scenarios.

In this chapter, we have presented a unified scoring framework for multi-dimensional queries over personal information file systems. We proposed individual IDF-based scoring approaches for several dimensions of data; namely content, metadata, and structure. In particular, we have defined a methodology for constructing progressive relaxations for each of these dimensions that may be used to locate relevant files that are not exact matches to a query. We have shown how to use these relaxations at query time to assign relevance scores based on how many files match a particular query condition, or a relaxation of it. We then presented a method of for aggregating individual dimension scores to produce a single unified relevance score for each file.

We presented an implementation and evaluation our scoring framework as part of the Wayfinder file system. Our evaluation has shown that our IDF-based scoring approach provides a meaningful distribution of scores that captures the specificity of each dimension. Additionally, we have shown that our multi-dimensional score aggregation technique preserves the properties of individual dimension scores and has the potential to significantly improve ranking accuracy. We reported on the impact of our multi-dimensional scoring on query answers, on query performance, results scalability, and
data set scalability.
Chapter 6

Federated Multi-dimensional Search

In this chapter, we present an extension to our single node multi-dimensional search framework to allow searches over the shared content of a federated community. Our goal is to allow a user to leverage the framework of the previous chapter for querying content throughout a federated system. At the same time, the method of querying should be identical to a single node environment and with comparable results. We present an approach that is compatible with all the abstractions and designed considerations previously outlined in this thesis.

Distributed query evaluation can be discussed in the context of three distinct stages: 1) Node Selection - determining the set of nodes that may contain information relevant to a query's evaluation, 2) Query Evaluation - the actual evaluation of the query on the set of chosen nodes, and 3) Result Merging - aggregating the results received from queried nodes and presenting the final set of results to the user. Each of these stages is complicated by our target environment and previous design choices made for the Wayfinder file system.

Recall that in a Wayfinder community, it is unlikely that any single node will have a complete snapshot of content being shared by a community. Instead, a node will contain a partial snapshot whose content is determined by various Wayfinder operations (automatic availability management) or user activities (file system). This non-standardized placement of data complicates the selection of nodes to be considered during query evaluation as relevant information may reside anywhere. Similarly, no single node will contain a complete collection of the necessary scoring statistics needed for computing relevance in a communal context. Independent query evaluation that relies on using only local information may therefore lead to different scores, possibly for the same data,
complicating the final merging of results.

In an ideal situation, we would maintain aggregated scoring information in a reliable communal online data structure. Any scoring information would be computed over all of a community’s content (accounting for replicas) and be accessible to any node wishing to compute relevance scores (in our case IDF-based scores). Given a query \( Q \), the simplest form of query evaluation would then be to send \( Q \) to all nodes, have \( Q \) evaluated locally while using the global scoring information to compute relevance scores, and then return the results to the querying node. As scores would be computed using the same state, the process of merging remote results to produce the final set of answers is greatly simplified.

This ideal situation is difficult to achieve. Maintaining and ensuring the availability of global data structures is difficult and expensive given the connectivity properties of our target environment.

Instead, to limit the impact of dynamic membership and avoid maintaining a large amount of online information, we propose an approach that considers only a small subset of a community during query evaluation and leverages globally replicated state. More specifically, we define a set of nodes \( N_C \) whose combined local content is (with high probability) a complete snapshot of a community’s content. During query evaluation, queries are forward to only this set of nodes. The coverage property of \( N_C \) ensures that all information in the community is considered during evaluation.

Given \( N_C \), we must then ensure that remotely computed scores are comparable. For this, we allow each node in \( N_C \) to individually collect and maintain the necessary global scoring statistics to perform relevance scoring. This is done separately for each distributed table that is used in evaluating a query locally (i.e., content, metadata, and structural). Each device in \( N_C \) constructs an index summary specific for each global table maintained by PlanetP. Each summary contains the information used during local query evaluation to compute our IDF-based relevance scores. These summaries are propagated to other nodes in \( N_C \) using PlanetP’s gossip-based communication. Each receiving node independently aggregates (details given below) the index summaries related to a particular distributed table to form a Global File Count (GFC) index for
that respective table. During query evaluation, computation of IDF-based relevance scores utilizes these GFC indexes rather than the local indexes/tables.

Intuitively, a GFC index captures information about the communal files counts relative to the values stored in a distributed table in PlanetP. As each node in $N_C$ will maintain an independent instance of these indexes and they are kept consistent through gossiping, individual nodes will have similar information for computing IDF-based relevance scores. Therefore, disregarding any inconsistencies that might arise from gossiping delays, independently computed IDF scores will be comparable. The independent copies also allow scoring despite changes in the connectivity of nodes in $N_C$.

The federated evaluation of a query then proceeds as follows. A querying node sends a query to all the nodes in the set $N_C$. Each node in $N_C$ evaluates the query locally utilizing its local instance of the GFC indexes to compute relevance scores. The computed results are returned to the querying node where they are merged and presented to the user.

In the remainder of this chapter, we present a discussion detailing our design of a GFC index in greater detail. We explore various methods of compressing index information using both lossy and lossless techniques to allow for a practical implementation. We then present our approach for federated query evaluation using these indexes and then conclude with an evaluation of our approach.

6.1 Design

In this section we will review the design of the GFC indexes and the distributed evaluation of a query in greater detail.

6.1.1 Global Indexes

A Global File Count (GFC) index is built for each distributed table used during local query evaluation by each node in $N_C$. While a GFC index maintains global state and utilizes summary information, they are different from the distributed tables of PlanetP;
\[ \text{IDF} = 1 + \log_2(N/T) \]

\[ N = \text{Total number of doc.} \]

\[ T = \text{Number of matching doc.} \]

Figure 6.1: (a) Equation for computing Inverse Document Frequency (IDF) and (b) a plot of this function as the value of \( T \) increases with a value of 1000 for \( N \).

they are used for retrieving scoring information and not retrieving remote information. A distributed table of PlanetP use summary information to establish a term → peer mapping. This mapping is used for determining which nodes should be contacted to retrieve information pertaining to a given query. In the case of the GFC, the summaries are used to construct a term → file count mapping. When accessed, the GFC returns the file count stored in this mapping; this requires no remote communication.

The summaries used in constructing a GFC index are defined as follows. For a table, \( I \), on device \( m \), the summary of \( I \) is of the form \( [(k_{m1}, C_{m1}), (k_{m2}, C_{m2}), \ldots, (k_{mn}, C_{mn})] \) where \( k_{mi} \) is the \( i \)th unique table key in \( I \) on device \( m \), \( C_{mj} \) is the number of files having value \( k_{mi} \) specifically on device \( m \), and \( n \) is the total number of unique keys in \( I \) on \( m \).

Index construction involves the collecting and merging of these per-device summaries for the same distributed table across nodes in \( N_C \). Merging is done by finding the common keys among the summaries and adding the corresponding file counts. The end results is a mapping \( k_j \rightarrow C_j \) where \( k_j \) is the \( j \)th unique key across all summaries and \( C_j \) is the accumulated count of all the file counts associated with \( k_j \) in the collected summaries.

This method of index construction maintains no information concerning individual files and so ignores the impact of skewed file replication (that is, some files may be
more heavily replicated than others). We believe this is acceptable for several reasons. First, discerning identical file replicas within the summaries would require a significant increase in the amount of state that must be represented. Second, given our availability model and method of constructing the global namespace, any file that is persistent, or reasonably available, is at any time likely to have at multiple replicas. As the equation for computing IDF scores is logarithmic (Figure 6.1(a)), large variations in scores will occur primarily if the single device document count is quite small (i.e., less than 10) and the aggregated replica count is significantly larger (Figure 6.1(b)). Therefore, we hypothesize that while the number of replicas for a file will vary, there will be minimal changes in the final relevance ranking.

For file sizes and dates, the set of keys maintained in the GFC index (i.e., $k_{mi}$) are the unique values obtained from the Metadata table of nodes in $N_C$ for the respective property. Given such a GFC index, it is a simple task to determine the approximate number of files matching a queried value, or a range of values (in the case of relaxations). For file types, the keys stored are in the GFC index are the set of tracked types. For content, the keys stored in the GFC index are unique content terms stored in the Content table. For structure, the keys are paths stored in the Structural table. The file count information associated with each of the required summaries can be easily determined using the existing tables presented in previous chapters.

6.1.2 Query Evaluation

We now turn to discussing the process of federated query evaluation in more detail. We will address each of the three previously mentioned stages in turn.

6.1.2.1 Node Selection

Recall from Chapter 4, we assume a federated community in which each user’s device set contains a champion device. These nodes have the role of ensuring offline and ownership availability for their users and have sufficient storage and availability to accomplish the task. To this end, independent of the rest of the community, the set of champion nodes in a community should contain close to a complete snapshot of a community’s shared
content. Furthermore, their number relative to the community size should be small.

For this reason, we designate $N_C$ as the set of champions nodes in the community. To further simplify our design, we currently consider the entire set as candidates for each query. The communication cost for querying is then proportional to the number of users in the systems, as we assume each user retains the use of at least one champion. This latter simplification may result in considering nodes that are unlikely to produce relevant results. Future work can explore approaches to optimize the choice of candidate nodes.

6.1.2.2 Query Evaluation

Upon receipt of a remote query, a node evaluates the query over the content of its local hoard using the local indexes/tables; just as if the query were issued locally. However, rather then using local file count information when computing IDF-based relevance scores, the GFC indexes are used. As these indexes are replicated across nodes, nodes will produce the same score for replicated files.

A query is evaluated until the $k$ most locally relevant results are found. This final set is returned to the querying node. Included with the returned results is the complete scoring information for each file which includes the un-normalized dimensions scores. This information will be needed in the next stage.

6.1.2.3 Result Merging

Once the results have been collected from the queried nodes, the results are merged to produce the final $k$ relevant answers to the query. This task is greatly simplified because the actual returned relevance scores are comparable.

Recall that during query evaluation, individual dimensions are normalized by the highest possible score (i.e., an exact match). In the case of content, this normalizing is done with respect to the highest content score. As this score may differ across nodes, we must re-normalize all of the received results before they can be merged. In particular, we determine the necessary factor by considering the un-normalized scores for all returned results and then proceed by adjusting the scores uniformly. With the
final scores computed, the results are sorted, duplicates removed, and the top \( k \) are chosen as the final result.

It can arise in practice that inconsistencies may exist in the GFC indexes on different nodes (i.e., as the result of gossiping delays). Such inconsistencies can result in the same file receiving different scores on different nodes. We will show in Section 6.3.2.6 that we expect that the effects of such inconsistencies will cause only minor variations. Should this situation arise in practice, we retain the higher score for the file.

### 6.1.3 Partitioned Federated Search

Until now, our approach to evaluating federated queries has required access to the set of champion nodes for a community. However, such a degree of connectivity may not always be possible given Wayfinder’s support for disconnected and partially connected operation. Should a node, or group of nodes, become disconnected from all the champions in a given community, they lose access to the pre-computed GFC indexes and content of these champions nodes.

During complete disconnection, federated searches are reduced to local searches and so the global indexes are not needed. When partially connected, the amount of accessible content is naturally limited to the local hoards of devices within the connected partition. For this, we employ a two-phase process for query evaluation. Assume a set \( S \) of connected nodes and a user formulated query \( Q \). In the initial phase, \( Q \) is sent to all nodes in \( S \). Each node determines the local file count information that is needed to score local files relevant to \( Q \). This information is returned to the querying node where it is aggregated to create the necessary GFC index state required to evaluate the query. During the second stage, \( Q \) is again sent to all nodes in \( S \) along with the query specific index information. Each node then evaluates \( Q \) as described above using the manually computed index information when appropriate. Results are then returned and merged as before.
6.2 Implementation

We have implemented a prototype of the distribute querying framework presented in this chapter as part of the Wayfinder file system. In this section we will discuss relevant aspects of our implementation.

6.2.1 Maintaining Global Indices

The construction of a GFC index is initiated by explicitly contacting remote champion nodes to request their respective summaries. To simplify our design, after an index is constructed, we retain only the aggregated state. Changes to the local state of a champion node is affected in the GFC indexes by propagating update information using the gossip-based communication protocol of PlanetP. To detect missing, or out-of-order, changes, each update is given a monotonically increasing version number.

Since a GFC index stores only aggregated information, we can not rely on PlanetP to maintain its consistency (as is the case with the distributed tables) as this would require keeping significant state for each individual node in $N_C$. Instead, we maintain only the version for the last seen update for each node. In the event that a missing update is detected, a complete synchronization with the necessary champion node is performed to retrieve the missing state.

Situations may arises that can not be dealt with by merely using updates. These may include the permanent departure of a user (and their champion node) or if update information is permanently lost. In these situations, it is impossible to reason about the correct state of the index and it is discarded and rebuilt. For example: Given two node, $n_1$ and $n_2$, with $n_1$ generating the summary $S_1 = [(10, 1), (20, 1)]$ for file sizes and $n_2$ generating $S_2 = [(10, 5), (25, 1)]$, the GFC index would have the aggregated state $S_{agg} = [(10, 6), (20, 1), (25, 1)]$. Should node $n_1$ permanently depart from the community, it is not possible to remove $S_1$ from $S_{agg}$ without actually knowing $S_1$.

If we retained the individual node summary information, such discards may be avoided at the expense of managing the additional state. However, we believe that such events will occur infrequently in our target environment and that this expense is
### Data Set Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>26K</th>
<th>User-Set</th>
<th>Server-Set</th>
<th>Fed-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Files</td>
<td>24,927</td>
<td>287,819</td>
<td>876,579</td>
<td>18,537</td>
</tr>
<tr>
<td>Number of Directories</td>
<td>2,339</td>
<td>21,798</td>
<td>92,819</td>
<td>1,701</td>
</tr>
<tr>
<td>Total</td>
<td>27,266</td>
<td>309,617</td>
<td>969,398</td>
<td>20238</td>
</tr>
</tbody>
</table>

### Content Information

<table>
<thead>
<tr>
<th></th>
<th>202,794</th>
<th>4,696,351</th>
<th>NA</th>
<th>223,857</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid Term Count</td>
<td>202,794</td>
<td>4,696,351</td>
<td>NA</td>
<td>223,857</td>
</tr>
<tr>
<td>Average Term Length (char)</td>
<td>8.69</td>
<td>13.27</td>
<td>NA</td>
<td>8.75</td>
</tr>
</tbody>
</table>

### Metadata Information

<table>
<thead>
<tr>
<th></th>
<th>15,201</th>
<th>32,475</th>
<th>72,824</th>
<th>10,125</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique File Sizes</td>
<td>15,201</td>
<td>32,475</td>
<td>72,824</td>
<td>10,125</td>
</tr>
<tr>
<td>Unique Modification Times</td>
<td>12,258</td>
<td>43,182</td>
<td>155,783</td>
<td>7,947</td>
</tr>
</tbody>
</table>

### Structure Information

<table>
<thead>
<tr>
<th></th>
<th>45.6</th>
<th>91.7</th>
<th>73.1</th>
<th>56.93</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Path Length (char)</td>
<td>45.6</td>
<td>91.7</td>
<td>73.1</td>
<td>56.93</td>
</tr>
<tr>
<td>Average Number of Path Components</td>
<td>4.7</td>
<td>7.3</td>
<td>7.2</td>
<td>5.5</td>
</tr>
<tr>
<td>Average Path Name Length (char)</td>
<td>9.7</td>
<td>10.8</td>
<td>18.2</td>
<td>9.1</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of the data sets used in our evaluation. Presented are summary statistics for metadata, content, and structural information.

not warranted.

### 6.2.2 Summary Sizes

The size of a summary used to construct a GFC index and the size of the final index itself is proportional to the number of unique keys in the summarized tables on nodes in $N_C$. We expect these numbers to be quite large. As summaries are shared, for large local tables this may result in a prohibitive communication cost. In this section we consider several techniques for compressing these summaries and the GFC indexes. We begin by first presenting several data sets that we will use in our discussion and subsequent evaluation.

#### 6.2.2.1 Data Sets

We use several data sets in our discussion and the subsequent evaluation (Section 6.3). The characteristics of these data sets are summarized in Table 6.1. For content terms we only consider alphanumeric terms with a length of no more than 256 characters and at most four numeric characters [76]. Any other terms are unlikely to be used by users
for searching.

The 26K data set was used in the evaluation of the previous chapter. To review, it includes a representative snapshot of a single user’s personal files and contains both personal and work related content. In this chapter, we re-visit this data set for comparison.

The User-Set represents a complete snapshot of a single user’s files that reside on a research lab’s file server. This data set is intended to represent a single user’s work environment and contains files relating to coding, papers, logs, and executables.

The Server-Set is a complete snapshot of all the metadata and structural information for all the users on our research lab’s file server. For privacy reasons, this snapshot does not contain any content information. The Server-Set and User-Set are obtained from the same file server, albeit on different dates. This data set represents a larger snapshot containing information from 28 distinct users.

Finally, the Fed-Search represents a collection of shared files that a single user might store on his personal devices. This collection was constructed by aggregating files from real users on our lab’s file server that would be of interest to a single user. These include files relating to papers and code being worked on collaboratively, pictures of events, and HTML pages. Overall, files can be sub-divided into three broad categories; personal files, files belonging to other users which are of interest, and files being used in collaboration with other users. This classification of files is done primarily on the basis of directories with all the immediate children of a directory falling entirely into one category. This data set is used in our evaluation to model multiple users sharing information for the perspective of what might interest a single user.

6.2.2.2 Lossless Summary Compression

Recall that an index summary is comprised of a set of tuples where each tuple contains a unique key value and the number of files having that key as the value for a particular attribute. Given the number of unique values that are presented in the data sets outlined in Table 6.1, the corresponding summaries can be quite large. For this reason, we seek methods for compressing them to reduce the communication costs.
Example of Encoding Schemes for Index/Table Summaries

<table>
<thead>
<tr>
<th>(a)</th>
<th>Original Summary:</th>
<th>[(1, 32), (10, 3), (32, 14), (128, 7)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b)</td>
<td>Run-Length Encoding:</td>
<td>[(1, 32), (9, 3), (22, 14), (96, 63)]</td>
</tr>
<tr>
<td>(c.1)</td>
<td>Byte Aligned Encoding:</td>
<td>[(0000 0001, 0010 0000), (0000 1001, 0000 0011), (0001 0110, 0000 1110), (0100 0000 0110 0000, 0011 1111)]</td>
</tr>
<tr>
<td>(c.2)</td>
<td>Gamma Encoding:</td>
<td>[[(0, 111 1100 0000), (111 0001, 101), (1 1110 0110, 111 0110), (1 1111 1010 0000, 111 1101 1111)]]</td>
</tr>
</tbody>
</table>

Figure 6.2: The three stages of compressing an index/table summary (a) using a run-length encoding (b) on index/table values followed by either a (c.1) byte-aligned encoding scheme or (c.2) gamma encoding scheme. For the byte-aligned encoding, the underlined bits indicate (in binary) how many bytes are used.

In the case where the table keys are numeric (i.e., for file sizes and dates), these tuples can be sorted by the key values, making a run-length encoding applicable [35]. That is, rather than encode each table key separately, we instead encoded the difference between two adjoining table keys (See Figure 6.2(b)). This can reduce the range of possible values and provide additional opportunities for further compression.

Among the compression techniques we consider are the use of either a byte-aligned or gamma encoding. Both methods attempt to improve on a naive encoding of 4 bytes per integer value (assuming a 32 bit architecture). Byte-aligned encoding is a fixed length encoding that minimizes the number of bytes needed to store a value. For a value $x$, the two left-most bits of $x$’s encoding are reserved for indicating the minimum number of bytes needed to encode $x$. The value of $x$ is then encoded in binary and appended to these bits (See Figure 6.2(c.1)). Using this encoding, a value can be stored in either 1, 2, 3, or 4 bytes [35]. Gamma encoding is a variable length compression technique that uses a family of universal codes [24]. In this scheme (See Figure 6.2(c.2)), an integer value $x$ is represented with $2\lfloor \log_2(x) \rfloor + 1$ bits. The first $\lfloor \log_2(x) \rfloor$ bits are a unary representation of $\lfloor \log_2(x) \rfloor$ (that is the value 3 is represented as “111” and the value 5 as “11111”). This is followed by a single “0” bit and ends with the binary representation of $x - 2^{\lfloor \log_2(x) \rfloor}$ (requiring $\lfloor \log_2(2) \rfloor$ bits) [35]. This scheme is particular effective at
Table 6.1: Table showing the number of hashed values that experienced collision when encoding structure and content values using an MD5 hash and truncating it to various sizes.

For textual values (i.e., content terms and pathnames), we employ a simple hashing scheme. Given a string value $S$ we compute its MD5 hash and truncate the resulting 16 byte binary number to the required number of bytes. We see in Table 6.1 that the length of a content term is greater than eight. If we assume a naive string encoding of 1 byte per character, then any hash length of less than eight will result in a space savings.

Table 6.2 shows the number of collisions that occur when using this hash encoding scheme for the structure and content information in several of our experimental data sets. We see that for structure, a 4 byte truncated hash is sufficient to avoid any collisions while for content this results in collision for 0.06% of the terms.

During query evaluation, structural relaxations are matched against individual components within the paths present in the local hoard. For example, the path /a/b/c must
match the path queries //b/c or /a//c. Any encoding of structural information must retain this granularity of matching. To accomplish this, we assign each directory in the namespace a unique integer identifier. We then proceed to encode each directory name using a 4 byte hash. The entire link structure is then encoded using this information. More specifically, for each directory we encode its integer identifier, the hash of its name, the identifier of its parent, and the number of files present in that directory. The latter number is the file count information. If we assume the number of directories is less than $2^{16}$ and no directory holds more than $2^{16}$ files, then the total encoding cost for each directory is 10 bytes.

In Table 6.3 we present the cost of encoding the various dimensions in our data sets using combinations of these compression techniques. As we use a run-length encoding, we expect that a gamma encoding will fare better than a byte-aligned encoding because of the extra bits being wasted when encoding smaller numbers\(^1\).

We see in Table 6.3 that using a gamma encoding for both the key value and the file count requires the least amount of space for the User-Set and Server-Set datasets. For the 26K Data set, using a byte encoding for the key value is more beneficial for file dates. We attribute this to the range of values present in the data set. The 26K has a significantly small number of files and so the key values are more likely to be spread out. In contrast the User-Set and Server-Set have a sufficiently larger number of files to fill in any holes in the range of possible values, allowing for the run-length encoding to produce smaller offsets.

We also observe that regardless of the encoding used for the key values, there is always improvement in using a gamma encoding for the file counts. The file counts are generally being very small numbers on account of the summaries storing the exact values in the index/table. In the next section, we explore what happens when this is not the case.

\(^1\)There are larger numbers for which a gamma encoding will results in a better degree of compression than a byte-aligned encoding. This occurs when the extra bits in a byte-aligned encoding are not actually used for encoding a value. For the range of values which can be stored for by 2 bytes using a byte-aligned encoding (i.e., 0 to 16383), a gamma encoding will be more efficient for only for 2% of the range. For 1 byte, this percentage is 24%. Therefore, for smaller numbers, a gamma-encoding is often best.
Table 6.3: The space requirements for encoding the summaries of the data sets outlined in Table 6.1 under various schemes. Naive refers to using 4 bytes per integer value and a tuple (size, value) for any strings. Hash encodes textural information as a hash value of the specified size. Byte refers to byte-aligned encoding and Gamma to a gamma encoding. For rows identified by two encodings (i.e., Byte, Gamma), the first is the encoding of the key value and the second the corresponding file count for each tuple in the summary. All versions use a run-length encoding on the key values.

### 6.2.2.3 Lossy Summary Compression

In discussing the above compression techniques, we assumed that only exact information is used and communicated. For date and size, this requires distinguishing between values at the granularity of seconds and bytes, respectively. We believe it unlikely that users will remember attributes of files at this granularity and therefore investigate decreasing the fidelity of the data stored in the global indexes to improve compression.

We proceed by normalizing each key value in a table by a predefined aggregation factor when computing the summary information. If multiple values normalize to the
Table 6.4: The number of unique keys for (a) modification times and (b) file sizes that results from aggregating the unique values of the respective tables into increasing larger ranges.

<table>
<thead>
<tr>
<th>Bin Size (in bytes)</th>
<th>Unique Sizes User-Set</th>
<th>Server-Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32475</td>
<td>72824</td>
</tr>
<tr>
<td>2</td>
<td>25016</td>
<td>55077</td>
</tr>
<tr>
<td>4</td>
<td>19131</td>
<td>41866</td>
</tr>
<tr>
<td>8</td>
<td>14647</td>
<td>31985</td>
</tr>
<tr>
<td>16</td>
<td>11431</td>
<td>24529</td>
</tr>
<tr>
<td>32</td>
<td>9025</td>
<td>18765</td>
</tr>
<tr>
<td>64</td>
<td>7140</td>
<td>14261</td>
</tr>
<tr>
<td>128</td>
<td>5647</td>
<td>10926</td>
</tr>
<tr>
<td>256</td>
<td>4446</td>
<td>8487</td>
</tr>
<tr>
<td>512</td>
<td>3510</td>
<td>6448</td>
</tr>
<tr>
<td>1024</td>
<td>2779</td>
<td>4976</td>
</tr>
<tr>
<td><strong>Total File Cnt</strong></td>
<td><strong>309798</strong></td>
<td><strong>971152</strong></td>
</tr>
</tbody>
</table>

Table 6.4 shows the effect of this aggregation on the number of unique table keys in two of our data sets. For both date (Figure 6.4(a)) and size (Figure 6.4(b)), a significant reduction in the number of unique values can be observed with only a slight increase in the granularity of the values being stored. Based on Table 6.4, when aggregating table information we employ a 1 hour and 32 byte aggregation factors for file date and size, respectively. This reduces the number of unique terms by an order of magnitude and we believe are sufficiently small to not impede how a user would formulate a query.

In Table 6.5, we presented the results of encoding an aggregated version of the table for three of our data sets. This table presents only results for encoding file date and file size as these are the two dimensions that can be aggregated. We also include the
Encoding Overheads for Data Sets with Aggregation

<table>
<thead>
<tr>
<th>Date Encoding (in KB)</th>
<th>26K</th>
<th>User-Set</th>
<th>Server-Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive</td>
<td>32.00 (66.6%)</td>
<td>44.38 (86.8%)</td>
<td>145.38 (88.1%)</td>
</tr>
<tr>
<td>Byte, Gamma</td>
<td>5.63 (75.0%)</td>
<td>8.30 (88.2%)</td>
<td>28.98 (88.4%)</td>
</tr>
<tr>
<td>Byte, Byte</td>
<td>8.31 (73.8%)</td>
<td>11.60 (88.0%)</td>
<td>37.75 (89.2%)</td>
</tr>
<tr>
<td>Gamma, Gamma</td>
<td>3.80 (84.5%)</td>
<td>5.88 (88.8%)</td>
<td>18.37 (91.1%)</td>
</tr>
<tr>
<td>Gamma, Byte</td>
<td>6.48 (80.8%)</td>
<td>9.19 (88.4%)</td>
<td>27.15 (91.1%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size Encoding (in KB)</th>
<th>26K</th>
<th>User-Set</th>
<th>Server-Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive</td>
<td>54.33 (54.3%)</td>
<td>70.51 (72.2%)</td>
<td>146.60 (74.2%)</td>
</tr>
<tr>
<td>Byte, Gamma</td>
<td>10.10 (54.8%)</td>
<td>13.71 (71.9%)</td>
<td>29.05 (74.3%)</td>
</tr>
<tr>
<td>Byte, Byte</td>
<td>15.07 (55.8%)</td>
<td>19.39 (71.9%)</td>
<td>39.82 (74.1%)</td>
</tr>
<tr>
<td>Gamma, Gamma</td>
<td>8.20 (60.0%)</td>
<td>10.59 (72.1%)</td>
<td>22.16 (74.5%)</td>
</tr>
<tr>
<td>Gamma, Byte</td>
<td>13.17 (59.1%)</td>
<td>16.27 (72.0%)</td>
<td>32.93 (74.3%)</td>
</tr>
</tbody>
</table>

Table 6.5: The number of bytes needed to encode the date and file size information for several data sets after tables values have been aggregated. The percentage after each number is the percentage of reduction achieved compared to the encodings given in Table 6.3.

percentage of reduction that we achieve compared with a non-aggregated table. These results show that with aggregation a significant saving of 54% to 91% is possible. Also, despite changing the distribution of values through aggregation, the \((\text{Gamma}, \text{Gamma})\) encoding is still best.

To understand the effect that this aggregation will have on our scoring approach, we examine how various levels of aggregation may alter the computed scores for a dimension. In Figure 6.3, we present the distribution of scores (Similar to Figure 5.4) when querying for a file size of 1 KB when the table has been aggregated using different factors.

In this figure we observe two possible behaviors. The first is shown by a large aggregation factor (1 KB). In this, several levels of scoring associated with the small relaxation intervals disappear and the scored results receive a lower score. The larger aggregation factor has essentially removed the ability to distinguish among smaller relaxation levels when computing scores. In this case, any interval less than 1 KB.
Second, is the reduced score for the small aggregation factor (32 bytes) while matching the same number of files as the non-aggregated scoring (Base). This is unusual because lower scores should be associated with a greater number of file matches and so the initial plateau should be larger horizontally. This behavior is explained in Figure 6.4. During query evaluation, relevant files are found locally via the relaxation intervals for the query value. For example, in Figure 6.4 interval R1 will match sizes S1 and S2. For purposes of local scoring, the size of this interval would be deemed as 2. However, when using the GFC indexes the range of the relaxation interval is mapped to the aggregated intervals used to compress the table’s state. In this case, interval R1 falls within the range of the second aggregation interval (i.e., 32-64). Locally, this interval accounts for sizes S1, S2, S3. Thus, two files match the relaxation interval R1 but are scored on the basis of matching three.

6.3 Experimental Evaluation

In this section, we experimentally evaluate our federated scoring approach using our prototype implementation. With this evaluation, we attempt to determine how our approach to federate query evaluation compares with a non-federated approach. We will also gauge the cost of maintaining the GFC indexes over time.
Figure 6.4: A pictorial representation of how scores are assigned when Global IDF Summary indexes are aggregated. Shown is a sequence of file sizes along a range (middle), the span of several relaxation steps over these files sizes (top), and the aggregation intervals for a factor of 32 bytes. The labels S1, S2, S3 explicitly identify three specific sizes.

6.3.1 Experimental Setting

6.3.1.1 Platform

All experiments were performed using the Wayfinder file system. Experiments were run on a cluster of PC where each node was equipped with a 64-bit hyper-threaded 2.8 GHz Intel Xeon processor, 2 GB of memory, and a 10K RPM 70 GB SCSI disk, running the Linux 2.6.16 kernel and Sun’s Java 1.5.0 JVM. The cluster is interconnected by a 100Mb/s Ethernet switch.

6.3.2 Federated Query Evaluation

In this section we will attempt to quantify the effect that any inaccuracies that result from using the GFC indexes have on query evaluation. We compare the results of performing a query in a federated setting to an evaluation in a single node setting. To do this we create a federated system to represent devices belonging to multiple users. We use the Fed-Search data set and publish it among the devices based on a pre-defined mapping of directories to users. We then allow our replication algorithm (See Chapter 4) to achieve the necessary target availabilities for these files. At this point, we construct a set of random queries and evaluate them in this federated setting. For comparison, we publish the Fed-Search data set on a single node and evaluate the
same set of randomly generated queries. In the remainder of this section we present further details of this set-up and conclude with a presentation and discussion of the results.

6.3.2.1 Federated System

As our search algorithm relies primarily on the champion nodes for query evaluation, we attempt to model a system consisting mostly of champion nodes. Since we are attempting to model the effects of federated search from the perspective of a single user, we also model at least a single user’s complete device set. Our federated community for our experiments consists of 12 separate devices. Four devices are designated as belonging to a single user’s device set; of these devices one is a champion. We will refer to this particular user as Bob in our discussion. The remaining devices are designated as belonging to other individual users; they are the champions of their respective users. All champions are given 90% availability and all non-champion nodes 80%. We do not limit the size of the local hoards.

Each user (thereby each champion) is assigned a set of namespace tags for files in the Fed-Search data set that they will maintain according to our availability model. Recall that this data set is constructed through the aggregation of files belonging to multiple users. The tagging reflects this ownership and the user’s collaborative interests. Several directories are shared by multiple users and some by a single user. As we shall discuss below, this will introduce a skew in the number of file replicas when our availability algorithm is run. Bob’s device set contains a complete snapshot of the entire data set as it is his files being shared.

6.3.2.2 Data Replication

Files are replicated according to Wayfinder’s availability model. Each champion node will attempt to maintain the online, offline, and ownership availability for any files specified in its assigned tags. We set the target online availability to 99.9% (i.e., $TOA_C$ for device $C$).

To introduce skew in the file replication, we purposely tagged the namespace to allow
files to fall into three distinct categories after the replication processes has reached a stable configuration; those that have a few replicas \((Rep_{Low})\), those that have a high number of replicas \((Rep_{high})\) and those falling in-between \((Rep_{med})\). Files in \(Rep_{high}\) are considered to have a replica count in excess of what is needed (as a result of the introduced skew) to reach \(TOAC\).

Note, that if a file is replicated on each of the devices in Bob’s device set, that this is sufficient to achieve \(TOAC\). Intuitively, we expect the files in \(Rep_{Low}\) to be files belonging to Bob’s working set or only files that Bob is interested in. We consider these files to have low replication factor not on account of the number of replicas but rather the impact on the GFC indexes. Despite being replicated on several devices, files stored only within Bob’s device set will be accounted for only once in the GFC as only file on champions are considered. The highly replicated files are those tagged by numerous champions and so will have more replicas to achieve ownership availability.

### 6.3.2.3 Queries

As in the previous chapter, our evaluation used a sets of synthetic queries (Section 5.4.1.3). Queries are generated for each of the aforementioned replication categories. To account for the characteristics of different groups of files, we further sub-divide \(Rep_{med}\) into small categories. Specifically, this sub-division is \(Rep_{medpic}\), \(Rep_{medpubs}\), and \(Rep_{medcode}\) for files relating to pictures, publications, and code. The number of queries generate for each category is given in Table 6.6. These queries were generated automatically. Some of these created queries were meaningless and subsequently discarded. For example for the category “MedPic”, there were very few useful “Content Only” queries because only a small number of picture files contained any content information. This resulted

<table>
<thead>
<tr>
<th>Query Counts</th>
<th>All</th>
<th>Low</th>
<th>High</th>
<th>MedPic</th>
<th>MedPub</th>
<th>MedCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Dimensions</td>
<td>433 (+1)</td>
<td>88</td>
<td>93</td>
<td>97</td>
<td>72</td>
<td>83 (+1)</td>
</tr>
<tr>
<td>Content Only</td>
<td>346 (-2)</td>
<td>100</td>
<td>45 (-2)</td>
<td>7</td>
<td>97</td>
<td>97</td>
</tr>
</tbody>
</table>

Table 6.6: The number of valid queries generated for each category of files. Shown are the number for exact queries. If the number of relaxed queries differ, the difference is given in parenthesis.
in the observable differences in the query counts.

As before, each query targets a specific file \( f \) in our data set. Using \( f \), we construct two sets of queries; the first consists of querying for \( f \)’s attributes with exact values and the second using relaxations of the attribute values of \( f \). The former represents queries where the user has accurate knowledge of \( f \) while the latter assumes inaccurate knowledge. For the latter class of queries, attribute values are relaxed in the manner described in Section 5.4.1.3

### 6.3.2.4 Federated Query Performance

To evaluate our approach to federated query evaluation we begin by comparing the results achieved when evaluating our set of queries over our federated environment (FE) against an evaluation in our single-node environment (SE). In FE, scoring will be accomplished using the GFC indexes while in SE scoring will employ only local information. We compute several metrics over these results.

The first metric attempts to measure how similar the results for FE are to those of SE by comparing the \textit{amount of overlap} in the final ranked results. For each rank in the final result of a query evaluated in FE, we use the following equation:

\[
\text{Result Overlap} = \frac{|\{\text{Single Node Search Docs.}\} \cap \{\text{Federated Search Docs.}\}|}{|\{\text{Federated Search Docs.}\}|}
\]

Scores can be in the range of 0 to 1, with 1 indicating perfect overlap.

The second metric is the \textit{missed rank} which measures the cumulative number of misses (files present in the results of an SE evaluation but not corresponding FE evaluation) according to their original rank position when evaluated in SE. This metric is intended to show which portion of the ranked results in SE are most affected by our federated scoring approach. The final cumulative count for each rank is normalized by the total number of queries in each category.

The results for comparing the evaluation of our randomly generated queries in SE and FE for exact queries using these metrics is shown in Figure 6.5 and for relaxed queries in Figure 6.6.
Figure 6.5: (a-b) The Result Overlap and (c-f) normalized cumulative ranks of any missed files when comparing the set of exact queries evaluated on a single node (SE) to an evaluation in a federated setting (FE). For the cumulative miss rate, every fifth rank is denoted by a darkened bar.
In Figure 6.5(a,c,d), we observe that for Content-Only queries the result overlap is quite good with only slight degradation after rank 14. The least amount of overlap occurs in the queries targeting files in $Rep_{Low}$. This is a result of the global file counts for content terms not changing uniformly. A low replication factor for a file does not necessarily equate to a low global file count for its content terms as they may also be present in other highly replicated files. This results in observed situations where the scores associate with terms altered their relative importance towards a document’s final relevance score. Such changes typically result in small score changes that allows files which already close in score to switch positions. In some cases, larger jumps can be observed. This is supported by Figure 6.5(c,d) which shows that files which fell out of the ranking were concentrated to the end of the list of results where slight rank inversions make the difference between inclusion or exclusion in the final set of ranked results.

The results for multi-dimensional queries (Figure 6.5(b,e,f)) show similar trends as the content-only queries but in general there is even less overlap, especially for queries targeting files in $Rep_{Low}$. This is due largely to the structural dimension. Files in $Rep_{Low}$ reside in the same directories within the shared namespace (recall that replication behavior is defined by namespace tagging). Therefore, the IDF scores of these directories will increase as the perceived overall number of files increases for other directories during the construction of the GFC indexes. This manifests itself in boosting the structural score significantly in queries searching these under-replicated directories. Similar behavior is possible in the content dimension, although it is rare because scores are based on $TF \cdot IDF$. In such cases, large changes to IDF are tempered by the TF component.

In Figure 6.6, we present the results for relaxed queries for multi-dimensional queries. We do not show results for Content-Only queries as we cannot “relax” content terms and so the results would be similar to those seen in Figure 6.5(a,c,d). The performance for this scenario is notably worse with the queries for files in $Rep_{Low}$ again faring the most poorly. The reason for this lies in the “inaccurate” query values of the relaxed queries. These values match other files which they are close to, resulting in a greater
Figure 6.6: (a) The Result Overlap and (b) normalized cumulative ranks of any missed files when comparing the set of relaxed queries evaluated on a single node (SE) to an evaluation in a federated setting (FE). For the cumulative miss rate, every fifth rank is denoted by a darkened bar.
variety of results. However, in this greater variety there are less files that match all dimensions well, resulting in lower overall scores. This coupled with the previously mentioned scoring boost observed in the structural dimension leads to a greater number of rank changes. As before, we can see in Figure 6.6(b,c) that most of the missed results are located at the end of the returned results.

Recall that each query used in our evaluation is constructed with a particular target file in mind. Until now we have only considered the set of query results as a whole and ignored this target. In Table 6.7, we analyze how the ranks of the target files for our queries are affected when evaluated in FE.

In Table 6.7(a) we see that the results for the set of exact queries shows very little difference as most ranks are unchanged. Since we employ the exact values along multiple dimensions, the target file is often ranked first despite the inaccuracies introduced by the GFC indexes.

Table 6.7(b) presents the results for the set of relaxed queries. In contrast to Table 6.7(a) there are several noticeable differences. First is the larger number of queries that did not find the intended target file within the top 20 results. This is a result of the value relaxation matching other files more closely than the target files. Second, for the queries targeting files in $Rep_{Low}$, around 25% see an improvement in their ranking caused by the previously mentioned increase in structural scores. However, overall when the target file is found, its rank remains unchanged.

Finally, in Figure 6.7 we present the results of comparing the results of query evaluation in SE and FE when using an aggregated version of the GFC indexes. In this case, aggregation was done with a 1 hour and 32 byte factors for date and size respectively. The results is almost identical to Figure 6.5(b) reaffirming that using a small degree of aggregation has little effect on query result while providing a large savings in encoding size. Table 6.7(c) shows the changes in the rank of the target file for the various query categories in this setting. Again the results are almost identical to previous results (Table 6.7(a)). There are several cases when the results do worsen because the initial relaxation levels are lost on account of the aggregation.
Figure 6.7: Result Overlap when comparing the results of query evaluation on a single node (SE) to the results of evaluation in a federated setting (FE) in which the global indexes have been aggregated. Aggregation is done based on a small configuration and queries used are exact.

6.3.2.5 Stability

To determine whether our results and trends are stable as \( k \) changes, in Figure 6.8 we present the Result Overlap for both content-only queries and multi-dimensional queries as we vary the value of \( k \). We observe that the previously discussed trends are stable as \( k \) increases.
Figure 6.8: Result Overlap for (a) content-only and (b) multi-dimensional when comparing the results of query evaluation on a single node (SE) to the results of evaluation in a federated setting (FE) for various values of $k$. 
<table>
<thead>
<tr>
<th>Effect on Rank</th>
<th>Query Categories</th>
<th>Low</th>
<th>High</th>
<th>MedPics</th>
<th>MedPub</th>
<th>MedCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Worsened</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Unchanged</td>
<td></td>
<td>88</td>
<td>92</td>
<td>96</td>
<td>71</td>
<td>80</td>
</tr>
<tr>
<td>Not Found</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total Query Count</strong></td>
<td></td>
<td>88</td>
<td>93</td>
<td>97</td>
<td>72</td>
<td>83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effect on Rank</th>
<th>Query Categories</th>
<th>Low</th>
<th>High</th>
<th>MedPics</th>
<th>MedPub</th>
<th>MedCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved</td>
<td></td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Worsened</td>
<td></td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Unchanged</td>
<td></td>
<td>48</td>
<td>44</td>
<td>37</td>
<td>66</td>
<td>77</td>
</tr>
<tr>
<td>Not Found</td>
<td></td>
<td>9</td>
<td>46</td>
<td>60</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total Query Count</strong></td>
<td></td>
<td>88</td>
<td>93</td>
<td>97</td>
<td>72</td>
<td>84</td>
</tr>
</tbody>
</table>

Table 6.7: Summary of the changes in rank of the target files used in the construction of our set of queries when comparing a single node (SE) query evaluation to that of a federated system (FE) broken down by query categories. Shown is the behavior for the set of exact queries (a), relaxed queries (b), and exact queries using an aggregated index (c). Improved represents that the target file attained a higher rank, for Worsened it attained a lower rank, for Unchanged the rank was the same, and Not Found represents that the target file was not found in either setting. For these queries, K was set to 20.
6.3.2.6 Global Index Maintenance

A critical part of our approach in constructing the GFC indexes consists of ensuring that the state being used on remote nodes is up-to-date. While the initial state transfer can not be avoided, subsequent changes to indexes can utilize diff information. The cost of sending this information is related to how much and how often information changes. To examine this cost we collected several traces for a single user from our lab’s file server. These traces consist of daily snapshots of the file structure, file metadata, and (in part) file content. These snapshots allow us to determine how much index state would change daily should the user’s files be published in a Wayfinder community.

More specifically we collected two traces. The first trace consists of all of the above mentioned information for a duration of 19 days. The second trace consists of only the metadata and structural snapshots for a period of 126 days for the same user. The time frames of the traces do not overlap.

For each trace, we computed the state of the GFC indexes and track the number of index entries that change. The results are plotted in Figures 6.9 and 6.10.

Figure 6.9 shows that the changes to the state of the global indexes can be bursty
across all dimensions. In Figure 6.9(b) we see that a large number of changes in files (through creation or deletion) may not require changing a corresponding amount of state in the global indexes. This is because the GFC indexes track file counts for table keys and not the files themselves. For example, if we remove 1000 files all of which have a file size of 1024 bytes, only the value associated with the file size of 1024 will be affected.

In Figure 6.10, we see that the behavior observed in Figure 6.9(b) continues in the longer trace. It should be noted that these traces were collected from a user’s home directory and so reflect changes that result from his working environment. These include running experiments and compiling large amounts of code and so may reflect more information than a user would actually wish to share in a collaborative setting. For example, the spike in the number of files created in Figure 6.10(a) around day 93 is a result of running numerous experiments that produced over 110,000 output files. However, in most cases the number of changes reflect only a small portion of the entire index. The median number of keys that changes on a given day throughout the trace ranges from 0.5% (for structure) to 1.0% (for dates) of the respective indexes. The mean values are slightly higher, ranging from 0.7% to 1.8%. Given this information
we believe that it is possible to batch communications for changes. Furthermore, given the number of changes and the encodings techniques discussed earlier, the amount of information transferred should be small.

### 6.4 Summary

We have argued in this thesis that contemporary computing environments are no longer restricted to a single devices. In this chapter, we presented an extension of the framework and methodology outlined in Chapter 5 to allow multi-dimensional searches across the shared content of an entire federated community. This extension is compatible with the abstractions and the goals of previous chapters.

This chapter presented an approach to distributed query evaluation that is based on computing relevance scores using a global approximation of the necessary scoring information. This approximation is built by each node generating summaries of its local scoring information and communicating this information. These local summaries are collected and aggregated to form a global approximation of the necessary scoring information. The individual scoring summaries are computed in a replica-oblivious manner; that is information needed to account for replicas is not included and so replicas across nodes are treated as distinct files. We hypothesized that this will have minimal effect on the quality of query results.

The distributed evaluation of query then takes the form of 1) sending the query to various nodes to be evaluated, 2) evaluating the query locally while using the approximated scoring information to compute relevance scores, and 3) returning the results to the querying node to merge them and produce the final set of results. The final step is greatly simplified since scores are computed with the same approximate global score information and are therefore comparable.

In this chapter, we presented an evaluation of our approach based on a prototype implementation as part of the Wayfinder file system and an investigation using several data sets and file-server traces. Our prototype evaluation showed that while ranking is affected by replication when evaluating distributed queries, the effect is small. The
evaluation of the data sets show that significant reductions in the cost of communicating the score summaries can be achieved through simple aggregations and encoding techniques. The evaluation of the traces demonstrated that the rate and size of updates to any shared scoring state is not prohibitive.
Chapter 7

Conclusions

In this dissertation, we have argued that computing environments are becoming increasingly complex. Various computing trends are producing devices that are increasingly smaller, more powerful, and cheaper than their predecessors. In turn, these devices are providing users with new ways and increased flexibility in accessing and sharing information. However, this improvement comes at the cost of complicating most management tasks in federated information systems.

Complications arise from the requirement that users be aware of, and reason about, the underlying distributed nature of these complex systems when locating, accessing, or searching for information. In this dissertation, we addressed these difficulties in the context of file systems, specifically a federated file system which we have designed called Wayfinder.

To simply the management of information, we proposed a set of three synergistic abstractions that hide the physical aspects of the federated system from the user. When performing a management task, these abstractions allow a user to reason about the desired properties of the data while ignoring the complexities of the underlying system.

In our investigation, we considered small to medium-sized federated communities constructed for collaboration and information sharing. These systems contain multiple users, each of which may be represented by several devices. The memberships of such communities is expected to be stable but nodes may exhibit a high degree of churn in their connectivity.

The first abstraction was a single unified global namespace with the purpose of providing a consistent view of shared information across computing devices. The construction
of this namespaces involves the recursive merging of the namespaces of individual devices. We demonstrated that this manner of construction provides a namespace that reflects the currently available information and allows devices to see information in a consistent manner whether they are in a connected, partially connected, or disconnected state. Furthermore, as the namespace is seen by all users, its construction and design is a collaborative effort, benefiting the entire community.

To complement this namespace, we further allowed the use of queries that could be persistently embedded into the namespace in the form of Semantic directories. The queries associated with each directory are evaluated over the content of the federated system and the directory is populated by files that are returned as result. When re-evaluated periodically, these directories allow a portion of the global namespace to become an active organization structure, automatically creating file bindings for new incoming content. The combination of the global namespace and semantic directories allows files to be addressed by either content or name.

We presented a design and prototype implementation of this namespace as part of Wayfinder file system. We showed that our approach to constructing the global namespace does impose some overhead compared to local file system operation but we believe it is tolerable given the benefits of the system. We found the location-independent naming afforded by the namespace to be very useful and it assisted us in simplifying many of our later designs. However, it complicated other various aspects of our design, such as directory deletion.

The second abstraction was a user-centric unified automatic availability model. The goal of this abstraction was to ensure the continued access to information despite the expected churn in node connectivity in our target community. More specifically, a user should be able to, with high probability, access any shared information from any of their devices, at any time. To improve file availability, we employed the traditional method of creating additional replicas. However, we hold the tenet that a computing device should prioritize the needs of its owner before the needs of the community.

To accommodate this, we proposed a novel unified availability model which differentiates between three types of availability; online, offline, and ownership availability.
Based on these distinctions, we outlined a user-centric availability model that tries to make data available to users across periods of connected and disconnected operation. This model also helps users to preserve data they care about in case they become permanently disconnected from the federated system. Consequently, the model removes the need for users to explicitly manage data replicas and to hoard data external to the federated system.

We designed a single replication algorithm that achieves all three types of availability as part of the Wayfinder file system. This algorithm allows devices to selfishly prioritize ownership and offline availability for their owners over online availability for the community. The algorithm explicitly accounted for the impact of devices’ selfish hoarding actions on online availability in order to minimize the space required to achieve a target online availability level for all shared content. Further, this algorithm was based on autonomous actions from devices in the community, allowing the system to tolerate the fact that devices in a federated system are not under centralized control and so may have unpredictable prolonged periods of disconnection or even leave the system permanently.

Our evaluation showed that when a federated community has sufficient space, our algorithm can efficiently achieve its availability goals. However, when space becomes constrained, nodes enter a non-cooperative configuration where any communal availability is a side-effect of trying to achieve the availability goals of their respective owners.

The third abstraction involved an investigation into methods for improving of search techniques. In this work, we addressed the limitation of many contemporary search tools that use a combination of ranking and filtering to evaluate multi-predicate queries. These tools apply ranking algorithms to the content predicate of a query while the remaining predicates are used as filtering conditions.

We began our investigation by presenting a unified scoring framework for multi-dimensional queries over personal information file systems, specifically in a single node setting. We proposed individual $IDF$-based scoring approaches for several dimensions of data; namely content, metadata, and structure. In particular, we defined a methodology for constructing progressive relaxations for each of these dimensions that may be used
to locate relevant files that are not exact matches to a query. We showed how to use these relaxations at query time to assign relevance scores based on how many files match a particular query condition, or a relaxation of it. We then presented a method for aggregating individual dimension scores to produce a single unified relevance score for each file.

Further, we presented an implementation and evaluation of this framework as part of the Wayfinder file system. Our evaluation showed that our IDF-based scoring approach provides a meaningful distribution of scores that captures the specificity of each dimension. We further demonstrated that our multi-dimensional score aggregation technique preserves the properties of individual dimension scores and has the potential to significantly improve ranking accuracy. We reported on the impact of our multi-dimensional scoring on query answers, query performance, results scalability, and data set scalability.

We then expanded the multi-dimensional search framework to allow query evaluation in a distributed setting. Specifically, we integrated our search framework with PlanetP’s distributed query engine.

We proposed an approach to distributed query evaluation that is based on computing relevance scores using an approximation of global scoring information. This approximation is constructed by a process in which each node individually computed summaries of its local scoring information. These summaries are then communicated, collected, and aggregated to build global indexes used for scoring. The summaries are computed in a replica-oblivious manner; that is, replicas across nodes are treated as distinct files. We hypothesized that this would have minimal effect on the quality of query results.

The distributed evaluation of a query then takes the form of 1) send the query to various nodes to be evaluated, 2) evaluate the query locally while using the global indexes to compute relevance scores, and 3) return the results to the querying node and merged them to produce the final set of results. The final step is greatly simplified since scores are computed with the same global index information and are therefore comparable.
Our evaluation attempted to quantify the effect of replication on search results given our approach. We found that predicting the behavior of query results based on the amount of file replication is difficult as a file’s replication factor does not necessarily equate to the replication factor of its attributes. However, in general we observed that the effect on query results was small. Furthermore, we studied several techniques to reduce the bandwidth costs of implementing our global index scheme.

In conclusion, it was the goal of this thesis to provide users with the means of viewing their computing environments as a single accessible unified resource. In this way, we reduce the management burden by allowing users to access, search, and manipulate all of their information at all times, regardless of which device they are using and to do so in a unified manner. Our contribution was pursuing this goal in the context of a federate system composed of multiple users and their personal devices.

We articulated several management deficiencies that arise in federated computing environments and designed abstractions to address each. In doing this we pursued two specific goals in our designs. First, each abstraction was built using a weakly consistent distributed query-based object store designed for our target environment. This showed the viability of the storage model presented by PlanetP [16] for constructing useful services in dynamic federated systems.

Secondly, each abstraction was required to be useful regardless of the connected state of the device being used. This was done to allow a user to continue working at all times and proved to be both a useful and necessary requirement in simplifying a user’s management roles. However, this was achieved at the expense of ensuring any form of strong consistency for information. This thesis explored this trade-off across the range of connected states a user might encounter.

An important question raised by this work and not addressed in this thesis is how much consistency a user can, or is willing, to tolerate in their computing environment. While this thesis did present methods for maintaining a weak level of consistency, it is not clear how the behavior of a user would be altered if they were made aware of any inconsistencies. In our system, such situations could arise for a number of reasons; write conflicts on files, a user trying to search for information that has disappeared, or
inconsistencies among a user’s devices because needed information did not disseminate fast enough. Addressing this issue would require detailed user studies.

Finally in this work, we have focused primarily on information stored in file systems. While this will most likely account for large portion of the information stored in a user’s computing environment, it is not all of it. Additional information can be stored in a variety of different formats or locations. These may include various device-specific databases or online services (such as MySpace [53] or Flickr [27]). Such information must be accessible to any abstractions that seek to provide a unified view of a user’s complete computing environment. Further work is required to address issues resulting from these other sources of information.
References


Vita

Christopher R. Peery

1999 B.S. in Computer Science, Richard Stockton College of New Jersey, Pomona, New Jersey, USA

2003 M.S. in Computer Science, Rutgers University, New Brunswick, New Jersey, USA

2009 Ph.D. in Computer Science, Rutgers University, New Brunswick, New Jersey, USA

Selected Publications


the International Conference on Date Engineering (ICDE - Demo Track), April 2008.