Context-Based File Systems and Spatial Query Applications

by

Ji Zhang

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 5, 2013

Keywords: Context-Based, File System, Voronoi Diagram

Copyright 2013 by Ji Zhang

Approved by

Xiao Qin, Associate Professor of Computer Science and Software Engineering
Wei-Shinn Ku, Associate Professor of Computer Science and Software Engineering
Sanjeev Baskiyar, Associate Professor of Computer Science and Software Engineering
Saad Biaz, Associate Professor of Computer Science and Software Engineering

Abstract

This dissertation presents studies related to I/O techniques in data-intensive computation and advanced solutions of spatial queries. There is a lack of general mechanisms for integrating multiple file system techniques and, therefore, the dissertation first illustrates a framework for Context-Based File Systems (CBFSs), which simplifies the development of context-based file systems and applications. Unlike existing informed-based context-aware systems, the framework provides a unifying informed-based mechanism that abstracts context-specific solutions as views, thereby allowing applications to make view selections according to their behaviours. The framework can not only eliminate overheads induced by traditional context analysis, but also simplify the interactions between file systems and applications. In addition, offloading a portion of a program to an active storage is another way to improve I/O performance in a cluster by significantly reducing data traffic. In the offloading study, we design a general offloading framework or ORCA that enables programmers without I/O offloading experience to complete the offloading development.

In the second part of this dissertation, we propose two novel spatial queries, multi-criteria optimal location query and keyword-spatial query. In our approaches, Voronoi diagram techniques are utilized for efficiently answering the queries. Besides two intuitive approaches, we explore two advanced solutions, Real Region as Boundary (RRB) and Minimum Bounding Rectangle as Boundary (MBRB), which are based on our proposed Overlapping Voronoi Diagram (OVD) model. High complexity of Voronoi diagram overlap computation in RRB motivates us to reduce costs of the overlap operation by replacing real boundaries of Voronoi diagrams with their Minimum Bounding Rectangles (MBR). Moreover, we employ a filter-and-refine strategy in an evaluation system for the keyword-spatial query. The system is comprised of Keyword Constraint Filter (KCF), Keyword and Spatial Refinement (KSR),

and a ranker. KCF calculates the keyword relevancy of spatial objects, and KSR refines intermediate results by considering both spatial and keyword constraints. The extensive experimental results show that the queries can be efficiently and effectively evaluated by the proposed solutions.

Acknowledgments

This dissertation would not have been completed without invaluable guidance, help and experience sharing from the people who constantly support and encourage me during my study at Auburn University.

First and foremost, my most sincere and deepest gratitude goes to my advisor, Dr. Xiao Qin, for his great efforts and trust in my work. I will never forget his extensive knowledge in the field of computer science and inexhaustible enthusiasm for research, which keeps inspiring and driving me to accomplish my research. When working on the paper of Context-Based File Systems (CBFSs), he gave me numerous valuable advices and suggestions, including setting up accurate motivations behind the research, building a multiple context model, and a concrete design that introduces a creative concept, view, to file systems.

I am also tremendously grateful to be advised by Dr. Wei-Shinn Ku. His solid understanding in spatial queries, unlimited patience in answering my questions and meticulous working style impressed me in our discussion and meetings. His insightful comments and suggestions helped and enlightened me with literature reviews, appropriate topic targeting, idea extension and demonstration, and experimental validation.

I owe my gratitude to Dr. Min-Te Sun, whose creative thinking and constructive criticism helped me considerably improve the way how the idea of the Overlapping Voronoi Diagram (OVD) model is extended and organized in our paper. His extensive experience and strong understanding in mathematics boosts my confidence to present mathematical details, such as property analysis and proofs, in the dissertation.

I would like to thank my committee members, Dr. Sanjeev Baskiyar and Dr. Saad Biaz, who reviewed my proposal and dissertation. They gave me a number of valuable suggestions, by which my dissertation had been substantially improved. I am equally grateful to Dr.

Fa Foster Dai, who gave me helpful comments and suggestions on my dissertation as the university reader.

Working with our research group is fantastic. I would like to thank Xiaojun Ruan, Zhiyang Ding, Shu Yin, James Majors, Yun Tian, Jiong Xie, Yixian Yang, Maen Al Assaf, Xunfei Jiang, Ajit Chavan, Tausif Muzaffar, Sanjay Kulkarni and Yuanqi Chen who helped me with paper writing, experimental result collection and group discussions. I would like to thank all the professors and students in the Department of Computer Science and Software Engineering, who create and maintain an excellent atmosphere for study and research.

Finally and most importantly, the endless love and support from my family is the most powerful strength that keeps me fighting for my research. My mother Yingxiang Hu, my father Lixin Zhang and my wife Xunfei Jiang always stay with me, cheering for achievement and overcoming all difficulties.

To my parents

and Xunfei Jiang

Table of Contents

Abstract . . . ii

Acknowledgments . . . iv

List of Figures . . . xii

List of Tables . . . xvi

1 Introduction . . . 1

1.1 Motivation of the Framework for Context-Based File Systems . . . 2

1.2 Motivation of the Offloading Framework . . . 4

1.3 Multi-Criteria Optimal Location Queries . . . 6

1.4 Spatial Keyword Queries on Networks . . . 8

1.5 Dissertation Organization . . . 10

2 Related Work . . . 11

2.1 Context-Based File Systems . . . 11

2.1.1 File Systems . . . 11

2.1.2 Context-Aware Systems . . . 12

2.1.3 View-Enhanced Systems . . . 13

2.2 Offloading Techniques . . . 14

2.3 Multi-Criteria Optimal Location Queries . . . 16

2.3.1 Reverse Nearest Neighbor Queries . . . 16

2.3.2 Optimal Location Queries . . . 17

2.4 Spatial Keyword Queries . . . 18

2.4.1 k Nearest Neighbor Queries . . . 18

2.4.2 Text Retrieval . . . 19

2.4.3 Spatial Keyword Queries . . . 20

3 Frog: a Framework for Context-Based File Systems 22
3.1 Context-Based File Systems 23
3.1.1 Basic Concepts 23
3.1.2 Context-Aware vs. Context-Based File Systems 24
3.1.3 The Frog Framework for Context-Based File Systems 25
3.2 Design 26
3.2.1 Overview 27
3.2.2 Operations 29
3.2.3 Three Frog-based CBFS Designs 32
3.2.4 Interactions with Applications 33
3.2.5 Application Comparisons 35
3.2.6 Overheads 37
3.3 Case Studies 42
3.3.1 The BAVFS File System 43
3.3.2 The BHVFS File System 45
3.4 Evaluations 46
3.4.1 File Creation/Deletion Rate 48
3.4.2 BAVFS 49
3.4.3 BHVFS 50
3.5 Discussions 52
3.5.1 Kernel Implementation 52
3.5.2 Work in a Distributed Environment 54
3.5.3 Optimization for View-unaware Applications 55
3.5.4 Context Exposure 57
4 ORCA: An Offloading Framework for I/O-Intensive Applications on Clusters . . 59
4.1 The ORCA I/O-Offloading Framework 60
4.1.1 System Architecture 60

4.1.2 Structure of Applications in ORCA . . . 61
4.2 Design Issues . . . 63
4.2.1 Data-Intensive Module Identification . . . 63
4.2.2 Offloading a Program . . . 63
4.2.3 Controlling an Execution Path . . . 65
4.2.4 Data Sharing among Storage and Computing Nodes . . . 65
4.3 Implementation Details . . . 66
4.3.1 Configuration . . . 67
4.3.2 Workflow of an Application in ORCA . . . 67
4.3.3 Offloading APIs . . . 69
4.3.4 Sharing Data . . . 71
4.4 Evaluations . . . 72
4.4.1 Experimental Testbed for ORCA . . . 72
4.4.2 Benchmark Applications . . . 74
4.4.3 PostgreSQL: A case study . . . 76
4.5 Experimental Results . . . 78
4.5.1 Overall Performance Evaluation . . . 78
4.5.2 Network Load Evaluation . . . 81
4.5.3 CPU Usage Evaluation . . . 85
4.6 Experience . . . 88
4.6.1 Offloading Module Identification . . . 88
4.6.2 Data Sharing . . . 89
5 MOLQ: Multi-Criteria Optimal Location Query with Overlapping Voronoi Diagrams . . . 91
5.1 Preliminaries . . . 92
5.1.1 Definitions . . . 92
5.1.2 Voronoi Diagram . . . 94

5.1.3 Fermat-Weber Point . . . 94
5.2 Basic Algorithms . . . 96
5.2.1 Sequential Scan Object Combinations . . . 96
5.2.2 Sequential Scan Locations . . . 97
5.3 The OVD Model . . . 98
5.3.1 An OVD Example . . . 98
5.3.2 Overlapped Voronoi Diagram Definition . . . 99
5.3.3 Algebraic Structure of MOVD . . . 103
5.4 Design . . . 106
5.4.1 Framework of the MOVD-based Solution . . . 106
5.4.2 The RRB Approach . . . 107
5.4.3 The MBRB Approach . . . 111
5.4.4 A Cost-Bound approach in *Optimizer* . . . 114
5.5 Experimental Validation . . . 116
5.5.1 Data Sets . . . 117
5.5.2 Cost-Bound Approach Evaluation . . . 117
5.5.3 Overlapping Two Voronoi Diagrams . . . 118
5.5.4 Overlapping Multiple Voronoi Diagrams . . . 119
5.5.5 MOLQ Evaluation . . . 120
6 Efficient Evaluation of Spatial Keyword Queries on Spatial Networks . . . 122
6.1 Query Type Definition and Background . . . 122
6.1.1 Foundation . . . 123
6.1.2 Spatial Keyword Ranker . . . 126
6.1.3 Spatial Keyword **k**NN Queries . . . 126
6.1.4 Spatial Keyword Range Queries . . . 127
6.1.5 Network Voronoi Diagram . . . 127
6.2 System Design . . . 130

6.2.1 Framework of Query Evaluation 130
6.2.2 Keyword Constraint Filter 131
6.2.3 The Network Expansion-Based SKkNN Query Algorithm 134
6.2.4 The Voronoi Diagram-Based SKkNN Query Algorithm 137
6.2.5 The Spatial Keyword Range Query Algorithm 140
6.3 Experimental Validation 141
6.3.1 Data Sets 142
6.3.2 Data Set Size Experiment 143
6.3.3 Number of Keywords Experiment 144
6.3.4 Number of k Experiment 145
6.3.5 Query Range Experiment 146
6.3.6 Page Access Experiment 147
7 Conclusion and Future Work 149
7.1 Framework for Context-Based File Systems 149
7.2 Offloading Framework 150
7.3 MOLQ Evaluation 151
7.4 Spatial Keyword Query 152
Bibliography 153

List of Figures

1.1 An example of residential location selection. The object weights are indicated as $<w^t, w^o>$. 6

1.2 A sample spatial network of hotels close to an airport. 9

3.1 A comparison of context-aware systems. 24

3.2 The framework of a Frog-based CBFS. 26

3.3 Overview of a Frog-based CBFS. 27

3.4 Frog-based CBFS designs. 32

3.5 Interactions between processes and Frog. 34

3.6 Examples of view-aware applications. 35

3.7 Application modification comparisons. 36

3.8 Two consistency mechanisms. 39

3.9 File locking among views. 40

3.10 Structure of BAVFS. 44

3.11 Name processing in two views. 45

3.12 The structure of BHVFS. 46

3.13 File creation/deletion. 48

3.14 Random and sequential read evaluation in BAVFS. 50

3.15 Random read and write evaluation in BHVFS. 51

3.16 VED and views allocation. 53

3.17 Two block mappings. 54

4.1 The architecture of commodity clusters, where a number of nodes are connected with each other through interconnects. We focus on clusters enhanced with active storage nodes that have computing capability. 60

4.2 An offloading domain is a logic processing unit, in which a pair of computing and offloading modules are coordinated. I/O-bound modules are assigned to and executed on storage nodes; CPU-bound modules are handled by computing nodes. ORCA overlaps the executions of CPU-bound and I/O-bound modules. . 61

4.3 The execution flow of a data-intensive application running in the ORCA offloading framework. 68

4.4 A simple example of offload_call 71

4.5 The execution flow of official PostgreSQL 77

4.6 The execution flow of offloading PostgreSQL in ORCA. The computing node handles the parser, rule system, and optimizer; the executor is offloaded to the storage node. 78

4.7 ORCA-based applications vs. Official applications. Execution times of the five real-world benchmark applications running on the homogeneous cluster (i.e., the first testbed). 79

4.8 ORCA-based applications vs. Official applications. Execution times of the five real-world benchmark applications running on the heterogeneous cluster (i.e., the second testbed). 82

4.9 Network load imposed by both official and ORCA-based PostgreSQL accessing different databases on the homogeneous cluster (i.e., the first testbed). 83

4.10 Network load imposed by the four real-world applications and their ORCA-based counterparts accessing 800 MB datasets on the homogeneous cluster (i.e., the first testbed). 84

4.11 Network load imposed by both official and ORCA-based PostgreSQL accessing different databases on the heterogeneous cluster (i.e., the second testbed). . . . 85

4.12 Network load imposed by the four real-world applications and their ORCA-based counterparts accessing the 800 MB datasets on the heterogeneous cluster (i.e., the second testbed). 86

4.13 CPU load imposed by the five real-world ORCA-based applications in the storage nodes of the homogeneous cluster (i.e., the first testbed). 87

4.14 CPU load imposed by the five real-world ORCA-based applications in the storage nodes of the heterogeneous cluster (i.e., the second testbed). 87

5.1 Ordinary Voronoi diagrams and OVDs. 99

5.2 The Framework of the MOVD-based solution. 106

5.3 Overlapping two MOVDs. 108

5.4 Weighted Voronoi diagrams (the numbers indicate weights). 112

5.5 Data structures. 112

5.6 Optimal locations. 114

5.7 The CB approach evaluation. 118

5.8 Execution time. 118

5.9 Number of OVRs. 118

5.10 Memory consumption. 118

5.11 Varying number of object types. 119

5.12 Three object types. 121

5.13 Four object types. 121

6.1 An example spatial network. 123

6.2 Voronoi diagram examples. 128

6.3 Framework of the spatial keyword query evaluation system. 131

6.4 Inverted index structure. 132

6.5 An example of KeywordMatch. 133

6.6 A VDkNN query example. 140

6.7 Execution times of NEkNN and VDkNN queries as a function of data set size. . 143

6.8 Execution times of NEkNN and VDkNN queries as a function of number of keywords. 145

6.9 Execution times of NEkNN and VDkNN queries as a function of number of k. . 146

6.10 Execution time of SKR queries as a function of query range. 146

6.11 Page Access evaluation with different data set sizes. 148

List of Tables

4.1 The ORCA Offloading Programming Interface 70

4.2 Hardware and Software Configurations 73

4.3 Configuration of the two Testbeds for ORCA 73

4.4 Real-World Benchmark Applications 75

5.1 Symbolic Notations. 92

6.1 A sample data set of hotels. 123

6.2 Symbolic notations. 125

6.3 Simulator configurations. 141

6.4 Data sets employed in experiments. 142

6.5 Default values of parameters used in experiments. 143

Chapter 1

Introduction

With rapid growth of data volume, methods of efficiently processing large amount of data have been reported in the past decade. In order to overcome the challenging problem, various advanced I/O techniques and approaches have been designed to alleviate the I/O bottleneck. Fast file system [80] and log-structure file system [99] increase the throughput of read and write operations by applying update-in-place and update-out-of-place strategies, respectively. I/O prefetching [90] and buffering [86] techniques are proposed for further performance improvement by conducting I/O behaviour analysis and prediction. Offloading techniques that offload programs instead of transferring huge data over network are designed to avoid network traffic in a distributed computing environment or save energy on mobile devices.

In addition to I/O techniques, investigating spatial query applications is another focused field in this dissertation. The spatial queries are not only data-intensive applications that evaluate the queries on large data sets, but also real world applications that have impacts on our lives. For example, top k nearest neighbor query provides us with the best candidates, such as restaurants, hotels or museums, based on customized criteria [107]. Range query returns all qualified candidates within a particular area.

This dissertation consists of two parts. The first part contains the studies related to I/O techniques; the second part presents the advanced solutions of novel spatial queries. This chapter is organized as follows. Sections 1.1 and 1.2 elaborate motivations of our frameworks for context-based file systems and offloading applications. Sections 1.3 and 1.4 illustrate the problem statements and motivations of novel spatial queries with simple examples. Finally, section 1.5 outlines the dissertation organization.

1.1 Motivation of the Framework for Context-Based File Systems

Context-aware computing allows modern storage systems to adjust I/O schemes and mechanisms according to specific contexts (*e.g.*, read-intensive contexts). Taking context information into account enables storage systems to dynamically optimize I/O performance, energy efficiency, etc. For instance, quFiles create multiple physical representations of files for data-specific contexts [122]. Prefetching can be optimized (*e.g.,* adjusting the number of prefetched blocks) by identifying and modeling future access patterns in a real-time manner [111]. When it comes to energy saving, a server may have two operating contexts: a full-utilization mode during daytime and an energy-saving mode during nights [73]. To reduce Solid-State Drive (SSD) maintenance costs, an HDD-SSD hybrid file system may have two contexts: a performance critical context and an SSD lifetime-extending context [112]. Both contexts can be implemented on SSDs and Hard-Disk Drives (HDDs), respectively.

Context-aware techniques have been investigated in previous studies. However, the following four problems make the existing context-aware techniques impractical in file systems. First, the context-aware techniques are too complicated to be implemented. Various heuristic or informed approaches have been proposed to identify contexts. Appropriately selecting an approach for a context-aware file system requires extensive experience. Second, the existing informed-based approaches expose different interfaces to applications. Migrating them from one approach to another is difficult. Third, a context of a file system can be divided into several finer-grained contexts, which make it easy to optimize I/O operations. Last, the existing approaches suffer from the back-end analysis, which is likely to incur a significant overhead during I/O operations in file systems.

In this study, we propose a framework, **Frog**, for **Context-Based File Systems (CBFSs)** that encapsulates a number of solutions, each of which is dedicatedly designed for a particular context. The context-specific solutions are abstracted as views in the internal representation of Frog-based CBFSs. Since the views are independent of each other, a new context can be supported in the CBFSs by creating a new view.

Frog is a unifying informed-based framework for statically mapping contexts to appropriate solutions in CBFSs. The framework simplifies the interfaces between file systems and applications. Rather than propagating hints through a dedicated interface (*i.e.*, *ioctl* [90] or newly developed *ctx_pread* [111]), the framework allows applications to choose contexts by inserting view names in file path strings. Without knowledge of extra I/O interfaces, application developers are able to write view-aware codes.

Frog does not dynamically identify any context based on hints, thereby avoiding overheads of the back-end analysis and shortening I/O response times. The contexts maintained in Frog-based CBFSs are configured off-line; view-aware applications are in charge of context selections. The performance of programs running in the CBFSs are automatically optimized by appropriate context-specific solutions implemented in form of views. Frog takes care of details of integrating multiple solutions, keeping consistency among views, and avoiding resource contention (see Section 3.2.6). This allows system developers to focus on context-specific solutions without really paying any attention to the integration issues.

Frog is compatible with existing non-view-aware applications. The compatibility is enforced by setting a default view for non-view-aware applications that generally do not represent any specific context. Moreover, the default view is an important abstraction, on which the consistency and locking mechanism rely.

Frog offers three approaches to integrating multiple context-specific solutions by considering metadata and physical data management. In the shared-data approach, physical data are shared among views, whereas in the shared-nothing approach, each view maintains a replica of files. The third approach is a hybrid in the sense that it is a combination of the shared-data and shared-nothing approaches. We also illustrate flexible interactions between Frog and applications by two concrete examples. Applications can access a file through different views based on their contexts. In addition, an application can update a file through a view while another application can read the file through another view later.

We implement two prototypes for the shared-data and shared-nothing approaches, respectively. The first prototype is the Bi-context Archiving Virtual File System (BAVFS) and the second prototype is the Bi-context Hybrid Virtual File System (BHVFS). BAVFS separates the contexts of sequential and random reads, and applies aggressive and conservative prefetching in each view. BHVFS manages two views for both read-intensive context and write-intensive context. Specifically, the *update-in-place* (*e.g.,* fast file system [80]) and *update-out-of-place* (*e.g.,* log-structured file system [99]) strategies are used for each context.

1.2 Motivation of the Offloading Framework

Although offloading techniques have been applied to a wide range of computing platforms (*e.g.* parallel file systems [38] [93] and object-based storage [36]), there is lack of a general offloading framework tailored for I/O intensive applications running on clusters. Moreover, none of existing works pay any attention on offloading application development from developers' perspective. Based on our experience, writing an appropriate offloading program is difficult and time-consuming. In this chapter, we propose a new offloading framework called ORCA to map I/O-intensive code to a cluster that consists of computing and storage nodes.

The following two factors motivate us to develop the ORCA offloading framework:

- heavy network traffic is imposed by transferring data from storage to computing nodes in clusters, and
- writing an offloading program without any general framework is difficult.

Due to the nature of I/O intensive applications, heavy network traffic is caused by retrieving massive amount of data between computing and storage nodes in clusters. During the data staging phase, data to be processed by applications running on computing nodes must be loaded from storage nodes through interconnections. Transferring huge amount of data can slow down the performance of the applications. This I/O problem becomes

even worse for clusters using the Ethernet, where all nodes share network bandwidth in the clusters.

The second motivation driving us is that studies of offloading development are missing. Even for an experienced developer, a number of issues related to the offloading development are difficult to solve, including appropriately designing offloading programs, accurately deciding I/O-bound modules of programs, controlling execution paths and efficiently sharing data.

Our goal is to address the above two issues by developing the ORCA framework to automatically offload I/O-bound modules of an application to active storage nodes in a cluster. The offloading framework deals with configurations, execution-path control, offloading executable code, and data sharing. Our framework coupled with an application programming interface (API) and a run-time system enables programmers without any I/O offloading experience to easily write new I/O-intensive code or extend existing code running efficiently on clusters.

The main contributions of this work are:

- We describe the ORCA framework centered around an offloading API and a run-time system (see Sec. 4.2).
- We discuss the implementation details, including the issues of configurations, programming interface, and data sharing (see Sec. 4.3).
- We develop a testbed to evaluate real-world applications in our run-time system (see Sec. 4.4).
- We present experimental results to show that both homogeneous and heterogeneous clusters powered by ORCA experience reduced amount of network bandwidth used to transfer data among computing and storage nodes (see Sec. 4.5).

Online resources. The source code and documentation of our I/O offloading framework are freely available at `ftp://ftp.eng.auburn.edu/pub/jzz0014/offloading/library/`

1.3 Multi-Criteria Optimal Location Queries

Numerous spatial queries, including nearest neighbor and reverse nearest neighbor queries, had been extensively studied; however, there are still spatial queries applied in real applications, such as location decision making, that cannot be efficiently addressed by existing spatial query types. A typical example is making residential location decisions, such as finding home locations that would maximize residential satisfaction, which is a critical part of community planning and development [85]. In order to attract more customers, an optimal location would be selected based on a comprehensive consideration of a number of factors, such as transportation accessibility (the ease of reaching a bus or subway station), the distance to an elementary school, or the distance to a supermarket where residents can purchase food and living necessities.

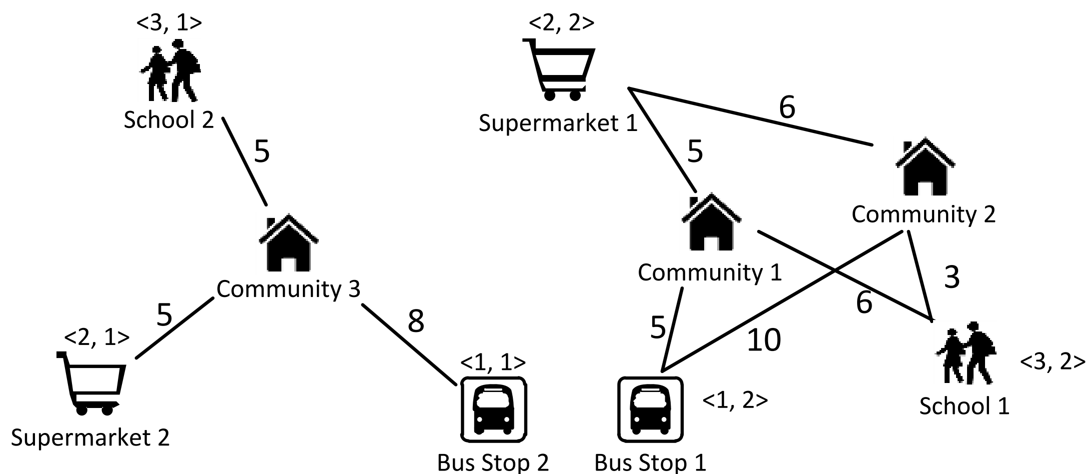


Figure 1.1: An example of residential location selection. The object weights are indicated as $<w^t, w^o>$.

Fig. 3.6 displays a simple example of residential location selection in a city. We assume that there are two schools, two bus stops, and two supermarkets in the city. Their locations are indicated by symbols. The figure also shows three potential community locations. Lines connect communities to their closest bus stop, school, and supermarket, respectively. The numbers on the lines indicate the distance between two locations. If we assume that the

optimal location for a new community is the place that minimizes the total distance to its closest school, bus stop, and supermarket, the best location is *Community* 1, the total distance (16) of which is shorter than that of *Community* 2 (19) or *Community* 3 (18).

Tradeoffs of multiple factors are actually considered in real residential location selection [126]. The importance of schools, bus stops, and supermarkets varies greatly among different people. For example, some people may prefer living near a school because it is convenient to drive their children to school. In addition, objects of a particular type are considered differently. When selecting a school, the ones that provide higher quality programs are more attractive than others. In order to take these differences into consideration, a *type weight* w^t and *object weight* w^o are associated with each object. With the weights $<w^t, w^o>$ indicated in the figure, the best choice is *Community* 3 (33) if the weighted distance of a community and an object is calculated as the product of distance and the two weights. Instead of associating a single weight with an object, *type weight* and *object weight* are set individually in the example because various weight functions are allowed to be applied to the query. This will be described in Section 5.1. In the example, a multiplicatively-based weight function is applied to both *type weight* and *object weight*.

In order to efficiently answer the query, we propose an advanced solution that utilizes Overlapped Voronoi Diagram (OVD) model and Fermat-Weber techniques. The OVD model comprehends location information and object weights w^o of spatial objects by overlapping the diagrams generated from the objects. With the OVD model, the closest objects of different types to a particular location can be efficiently retrieved. Fermat-Weber techniques are used for finding the optimal location of given objects.

In addition, due to a surprisingly large number of Fermat-Weber problems generated in our solution, we propose a cost-bound iterative solution that is able to significantly reduce the computation complexity of the original iterative solution [127]. The contributions of this study are summarized below:

1. We identify a novel query type that is able to find optimal locations comprehensively considering multiple criteria.

2. We build an OVD model, analyze its properties systematically, and create an algebraic structure of its overlap operations.

3. Based on the OVD model, We propose a Real Region as Boundary (RRB) solution that is able to efficiently evaluate the novel query.

4. In the proposed Minimum Bounding Rectangle as Boundary (MBRB) solution, we optimize the overlap operation by avoiding overlapping region calculations.

5. Facing large amount of Fermat-Weber problems, we propose a cost-bound iterative solution that is able to significantly reduce the computation complexity of the original iterative solution.

6. We evaluate the performance of the proposed solutions through extensive experiments with real-world data sets.

1.4 Spatial Keyword Queries on Networks

A Spatial Keyword (SK) query is an approach of searching qualified spatial objects by considering both the query requester's location and user specified keywords. Taking both spatial and keyword requirements into account, the goal of a spatial keyword query is to efficiently find results that satisfy all the conditions. However, all existing solutions for SK queries are designed based on Euclidean distance [40, 53, 139, 136], which is not realistic since most users move on spatial networks. Moreover, all current approaches of SK queries are limited to finding objects that fully match the given keywords. Nevertheless, the objects with fully matched keywords could be *far away* from the query point. In this research, we design novel SK query techniques based on spatial networks. In addition, we take both fully and partially matched query results into account in the process of keyword searching.

This new SK query mechanism enables users to not only retrieve qualified results on spatial networks, but also obtain partially matched objects when there are not enough fully matched results *in the vicinity* of the requester.

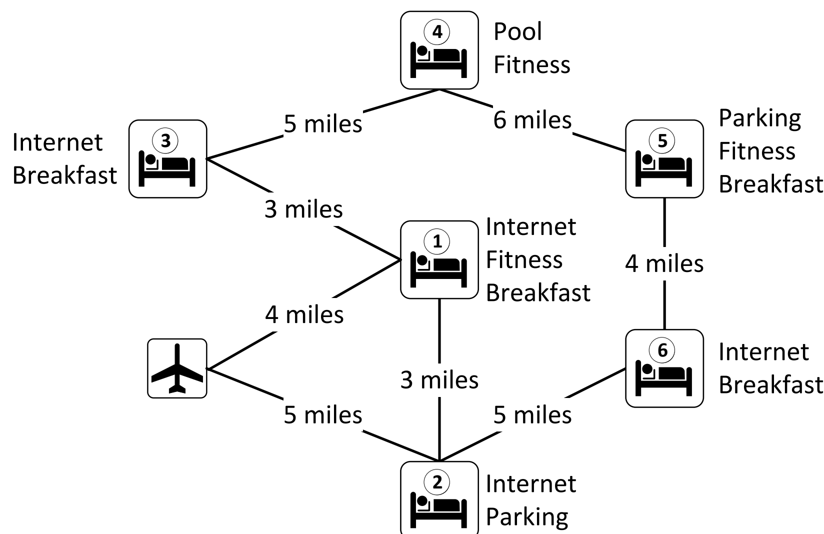


Figure 1.2: A sample spatial network of hotels close to an airport.

Figure 1.2 illustrates an example: a tourist who flies to Atlanta may want to search for two hotels which provide both “Internet" and “Breakfast" amenities and have the shortest driving distance to the Atlanta airport. In addition, the tourist may also search for all the hotels which are within 10 miles of the airport and provide the two amenities in order to compare the hotels' reviews and prices. For retrieving the qualified hotels, the tourist will launch a Spatial Keyword k Nearest Neighbor (SKkNN) query with ranking parameters for the first search, and the query results are hotels 1 and 3. A Spatial Keyword Range (SKR) query will be executed for the second inquiry, and the answers are hotels 1, 3, and 6. In this chapter, we focus on solving the two aforementioned spatial query types by devising three novel solutions which employ the inverted index technique, shortest path search algorithms, and network Voronoi diagrams. Particularly, the inverted index is used to maintain

the relationships between spatial objects and their attached keywords for quickly retrieving spatial objects whose features match the given keywords. In addition, we propose both a network expansion-based approach and a Voronoi diagram-based approach to efficiently answer SKkNN queries on spatial networks. The contributions of this study are as follows:

1. We provide formal definitions of spatial keyword kNN and range queries on spatial networks.

2. We develop two novel approaches for efficiently processing SKkNN query and one approach for SKR query on spatial networks.

3. Our SKkNN solution is able to return partially matched query results based on the output of the spatial keyword ranker.

4. We evaluate the performance of the proposed SKkNN and SKR algorithms through extensive experiments with both real-world and synthetic data sets.

1.5 Dissertation Organization

This dissertation is organized as follows. In Chapter 2, related work reported in the literature is briefly reviewed. Chapter 3 introduces the concept of context-based file systems, and illustrates the design of a framework, Frog, that is able to simplify the development of context-based file systems. In Chapter 4, we develop an offloading framework, ORCA, that is able to not only significantly reduce the network traffic by offloading programs over network, but also help developers to deal with complex issues involved in offloading development. In Chapter 5, we first propose an Overlapping Voronoi Diagram (OVD) model. Then, two advanced solutions based on the model are introduced and investigated. In Chapter 6, two advanced solutions, network expansion-based and Voronoi diagram-based approaches, are proposed and evaluated. Finally, Chapter 7 summarizes the contributions of this dissertation and comments on future directions on the research.

Chapter 2

Related Work

As Chapter 1 mentioned, a variety of I/O techniques and spatial queries have been extensively studied in the past. This chapter briefly reviews existing approaches that most relevant to file system design, offloading techniques, and spatial queries.

2.1 Context-Based File Systems

2.1.1 File Systems

Two completely different update strategies were implemented in file systems: *update-in-place* and *update-out-of-place*. An *update-in-place* file system (*e.g.*, FFS [80], IBM's JFS [8], SGI's XFS [10], ReiserFS [9] and Ext 4 [78]) usually commits an update at a place where data is stored. On the other hand, an *update-out-of-place* file system (*e.g.*, LFS [99]) appends updates in a log file, which is reorganized by a separate cleaner program in a batch manner.

Hybrid file systems integrate multiple I/O techniques in a single file system. Identifying the characteristics of a file system workload, MFS separates data into control, data, and log storage [82]. The control and data storage manage data that has not been recently modified. The log storage is designed for the crash recovery purpose. DualFS is a journaling file system that separates metadata from data using partitions or devices [91]. *Update-in-place* and *update-out-of-place* strategies are applied to the metadata and data partitions, respectively.

HyLog is a hybrid file system that incorporates the FFS and LFS strategies [124]. To avoid the cleaning overhead of LFS, HyLog divides disk space into fixed-size segments. Hot segments - containing hot pages - are organized in LFS to achieve high write performance.

Cold segments - containing cold pages - are managed in FFS. When hot pages become cold, HyLog moves the cold pages to the cold partition to avoid extra cleaning overhead. The nature of pages, regardless of hot or cold, is determined by a separating algorithm.

Similar to HyLog, hFS is a hybrid file system combining FFS and LFS [138]. In hFS, files are classified into two categories: large and small files. Large files are stored in FFS to offer competitive read performance by avoiding fragmentations incurred by small files. Metadata and small files are organized in the LFS fashion because read operations accessing small amounts of data are likely to respond by retrieving one or two blocks.

Unlike the existing file systems, our Frog-based CBFS allows applications to interact with file systems. The CBFS exhibits not only file information but also a number of contexts in which the context-specific solutions are applied. View-aware applications are able to benefit from the context-specific solutions by appropriately selecting views.

The stackable file systems were proposed as an easier way of implementing and extending file systems [100] [134] [135]. By using unionfs [133], files in several directories appearing to be merged are actually managed separately. Our design is different from these systems in the sense that Frog-based CBFS combines multiple context-specific solutions by duplicating either a part of or an entire file system. In light of contexts, applications can pick views that offer optimization solutions to meet the needs of the applications.

2.1.2 Context-Aware Systems

Context-aware computing provides ample opportunities to improve the performance of adaptive systems. For example, in a context-aware mobile computing system, context-sensitive data (*e.g.*, locations and users' activities) can be simply collected by sensors. Based on the context-sensitive information, mobile devices can identify contexts and adjust device behaviors accordingly [69][109].

Collecting context-sensitive data is non-trivial at storage server ends. A context-aware prefetching study was conducted at storage servers, in which access patterns can be identified

via hints provided by applications through dedicated programming interfaces [111]. Rather than relying on hints, our Frog-based CBFS handles context selections made by applications that are aware of future I/O access needs. The CBFS can avoid the overheads of context analysis on storage servers.

A number of context-aware middle-ware systems were proposed to make context-aware computing possible [30][50][98]. Our approach is different from these systems in that we focus on context-based file systems, in which view-aware applications can achieve context-specific I/O optimizations.

2.1.3 View-Enhanced Systems

Conceptual view models have been widely used in software design. Along with multiple views, data can be logically presented in a diverse manner. A typical example is a relational database management system, in which views - constructed on top of hierarchical structures - are available for applications [95]. Gehani *et al.* proposed a file system interface for an object-oriented database [47] where objects can be accessed through both file manipulations and database operations.

MVSS introduces a multi-view model for active storage systems to provide flexible services by dynamically generating in-memory views. A set of views are mounted at distinct points in a file system namespace [76]. Unlike MVSS, which implements multiple views at the storage system level, Frog supports multiple views at the file system level, where it takes care of the details of integrating multiple solutions. Frog handles metadata and physical data management, consistency issues, and resource contention, which can be applied to either a native file system in the kernel or a storage system in a distributed environment. Moreover, Frog statically duplicates metadata and physical data across views; thereby avoiding the overhead of dynamic methods (*e.g.*, making decisions about when and what types of views are created).

The issues of context-aware adaptation were addressed in quFiles [122]. In particular, various contexts in quFiles are accessed in the form of views, in which an optimal policy is selected for a specific I/O context. To provide flexible and transparent services, quFiles dynamically or statically creates views. One view is set as the default view. Compared with quFiles, Frog diversifies not only physical representations but also metadata management as well as I/O operations. This salient feature of Frog makes it possible to integrate multiple I/O optimizations (*e.g.*, prefetching) for particular contexts. More importantly, we show how to make use of Frog to implement feasible designs of context-based file systems.

2.2 Offloading Techniques

The concept of *active disks* was proposed by Acharya *et al.* [16]. In their active disk architecture, processing power and memory are deployed into individual disks, to which a portion of application computation can be offloaded by using a stream-based programming model. Their simulation results show that significant performance improvement can be achieved by reducing data traffic. Riedel *et al.* designed a similar system that moves an application's processing to disk drives. In addition, they developed an analytical model, evaluating a wide range of applications that may benefit from the active storage [96]. Keeton *et al* presented an "intelligent" disk (IDISKs) architecture for decision support database servers [63].

Lim *et al.* designed an active-disk-based file system (ADFS), in which application-specific operations can be executed by disk processors [75]. When a file is loaded, only results processed by an application are returned. Moreover, the ADFS file system also offloads a part of file system functionalities (*e.g. lookup*) to active disks. This approach is able to significantly reduce the workload of central file management. Chiu *et al.* presented a distributed architecture that utilizes smart disks equipped with processing power, on-disk memory, and network interfaces [29]. A set of representative I/O-intensive workloads

are evaluated on the architecture. Their experimental results suggest that the distributed architecture outperforms partially distributed and centralized systems.

The idea of active storage was implemented in the Lustre file system by Felix *et al.* [38]. Afterwards, piernas *et al* proposed an active storage for Lustre in the user space [93]. Both approaches reduce data movements and improve computing capability. Compared with the kernel-based implementation, the user-space approach is faster, more flexible and readily deployable. In addition, motivated by requirements of specific applications, piernas *et al.* designed and evaluated an efficient way to manage complex striped files in active storage [92].

Du designed an intelligence storage system that combines active disks and object-based storage device (OSD) [36]. Du's study mainly focused on fundamental changes in existing storage systems; he proposed a number of future research directions in the realm of OSD-based storage. Son *et al.* investigated an active storage in the context of parallel file systems [110]. Based on the analysis of parallel applications, they designed an enhanced programming interface that enables application codes to embed in the parallel file system. Moreover, their approach also provides server-to-server communication for reduction and aggregation.

Although the offloading techniques have been extensively explored in the aforementioned studies, our approach differs from the above solutions with respect to the following three issues. First, recognizing that there is a lack of generic offloading framework, we propose a general offloading programming model that can be applied to either sequential or parallel applications (e.g., multi-thread and multi-process programs). We introduce the concept of offloading domains to represent computation consequences. Server-to-server communications proposed by Son *et al.* [110] can be converted into domain-to-domain communications. Second, developing offloading-oriented applications is non-trivial from programmers' perspective. In this study, we address a number of critical implementation issues raised in the development of offloading-oriented programs. These issues, which are crucial to program designs and performance, include I/O-bound module identification, execution path control,

and data sharing in an offloading domain. Last, developing offloading-oriented programs in C language is time consuming due to the complexity of the programming language and; therefore, we propose an approach that is able to share dynamic and static data (e.g., codes).

2.3 Multi-Criteria Optimal Location Queries

2.3.1 Reverse Nearest Neighbor Queries

Korn and Muthukrishnan [66] proposed the influence set notion based on reverse nearest neighbor (RNN) queries. They presented a precomputation-based approach for solving RNN queries and an R-tree based method (RNN-tree) for large data sets. In order to decrease index maintenance costs in [66], Yang and Lin [131] presented the Rdnn-tree which combines the R-tree with the RNN-tree and leads to significant savings in dynamically maintaining the index structure. The solutions in [66, 131] can be employed to evaluate both the monochromatic RNN query and the bichromatic RNN query; however, these precomputation-based techniques incur extra maintenance costs for data updates. Therefore, several solutions without precomputation were proposed. For discovering influence sets in dynamic environments, Stanoi *et al.* [113] presented techniques to process bichromatic RNN queries without precomputation. The design is to dynamically construct the influence region of a given query point q where the influence region is defined as a polygon in space which encloses all RNNs of q. For the monochromatic RNN query, Tao *et al.* [117] developed algorithms for evaluating RkNN with arbitrary values of k on dynamic multidimensional data sets by utilizing a data-partitioning index. The algorithms were later extended to support continuous RkNN search [118], which returns the RkNN results for every point on a line segment.

There are some other works related to RNN query evaluation. Retrieving RNN aggregations (such as COUNT or MAX DISTANCE) over data streams was introduced in [67]. Yiu *et al.* [132] proposed pruning-based methods to find RNNs in large graphs. The algorithms for efficient RNN search in generic metric spaces were presented in [119]. The techniques

require no detailed representations of objects and can be applied as long as the similarity between two objects can be computed and the similarity metric satisfies the triangle inequality. Cheema *et al.* [27] studied the problem of continuous monitoring of reverse k nearest neighbors queries in Euclidean space as well as in spatial networks. While the aforementioned approaches work well for R(k)NN queries, they cannot be utilized directly to evaluate the unique query type studied in this chapter due to the fundamental differences between query definitions.

2.3.2 Optimal Location Queries

One group of optimal location queries (OLQ) are defined with an optimization function which maximizes the influence of a facility. Given a set of sites, a set of weighted objects, and a spatial region Q, the optimal-location query defined in [37] returns a location in Q with maximum influence based on the L_1 distance where the influence of a location is the total weight of its RNNs. Xia *et al.* [129] proposed pruning techniques based on a metric named $minExistDNN$ to retrieve the top-t most influential sites according to the total weights of their RNNs inside a given spatial region Q. The Optimal Location Selection (OLS) search was introduced in [46], which retrieves target objects in a target object set D_B that are outside a spatial region R but have maximal optimality with a given data object set D_A and a critical distance d_c. Here, the optimality of a target object $b \in D_B$ located outside R is defined as the number of the data objects from D_A that are inside R and meanwhile have their distances to b not exceeding d_c.

Another group of location optimization queries are defined with a different optimization function which minimizes the average distance between a client and the nearest facility. Zhang *et al.* [137] proposed the Min-Dist Optimal Location Query (MDOLQ). Given a set S of sites, a set O of weighted objects, and a spatial region Q, MDOLQ returns a location for building a new site in Q, which minimizes the average distance from each object to its closest site according to the L_1 distance. They provide a progressive algorithm that quickly suggests

a location, tells the maximum error it may have, and continuously refines the result. When the algorithm finishes, the exact answer can be found. Because user movements are usually confined to underlying spatial networks in practice, Xiao *et al.* [130] extended OLQ to support queries on road networks. They design a unified framework that addresses three variants of optimal location queries. By observing that users can only choose from some candidate locations to build a new facility in many real applications, Qi *et al.* [94] introduced the Min-dist Location Selection Query (MLSQ) based on the studies in [137, 130]. Given a set of clients and a set of existing facilities, MLSQ finds a location from a given set of potential locations for establishing a new facility where the average distance between a client and her nearest facility is minimized. MND, a method, for efficiently solving MLSQ, employs a single value to describe a region that encloses the nearest existing facilities of a group of clients, is presented in [94] . However, these studies differ from the proposed query type in definition and optimization functions. Consequently, we cannot use them for answering our novel query type.

2.4 Spatial Keyword Queries

2.4.1 k Nearest Neighbor Queries

In spatial databases, k nearest neighbor (kNN) and range queries are fundamental query types. These two types of spatial queries have been extensively studied and applied in various geographic information system (GIS) applications. By employing the R-tree [51, 21] based indices, depth first search (DFS) [101] and best first search (BFS) [55] have been the prevalent branch-and-bound algorithms for processing nearest neighbor queries in Euclidean spaces. However, neither DFS nor BFS can be applied to spatial networks for answering kNN queries.

For answering spatial queries on road networks, Papadias *et al.* [87] developed a Euclidean restriction and a network expansion framework to efficiently prune the search space. Based on the proposed frameworks, solutions for nearest neighbor queries are designed in the context of spatial network databases. In addition, a network Voronoi diagram-based

solution for kNN searches in spatial network databases is presented in [65] by partitioning a large network to small Voronoi regions and pre-computing distances both within and across the regions. Moreover, Jensen *et al.* [59] proposed a data model and definitions of abstract functionality needed for moving kNN queries in road networks and designed corresponding solutions. Because most Dijkstra's algorithm based kNN solutions have been shown to be efficient only for short distances, Hu *et al.* [56] proposed an efficient index, distance signature, for distance computation and query processing over long distances. Their technique discretizes the distances between objects and network nodes into categories and then encodes these categories to accelerate kNN search process. Furthermore, in order to speed up kNN searches, Samet *et al.* [102] designed an algorithm to explore the entire network by pre-computing the shortest paths between all the vertices in the network and employing a shortest path quadtree to capture spatial coherence. With the algorithm, the shortest paths between all possible vertices can be computed only once to answer various kNN queries on a given spatial network. Nevertheless, all the aforementioned techniques mainly focused on the distance metric. They did not consider text description (keywords) of spatial objects in their query evaluation processes.

2.4.2 Text Retrieval

Text retrieval is another important topic related to spatial keyword queries. There are two main indexing techniques, inverted files and signature files, widely utilized in text retrieval systems. According to experiments made by Zobel *et al.* [141, 140], signature files require a much larger space to store index structures, and are more expensive to construct and update than inverted files. In addition, Baeza-Yates and Ribeiro-Neto [20] also stated that inverted files outperform signature files in most cases.

Although these aforesaid methods perform quite well in text retrieval applications, none of them can efficiently process spatial keyword queries. In other words, it is impractical to answer spatial keyword queries by simply employing approaches introduced in this or the

previous subsection. An effective way to handle spatial keyword queries is to combine the two groups of techniques as discussed in the following subsection.

2.4.3 Spatial Keyword Queries

As local search services become more and more popular, many solutions [25, 31, 40, 53, 28, 139, 136] have been developed to evaluate spatial keyword queries by integrating index techniques previously used in spatial queries and text search.

Location-based web search is studied by Zhou *et al.* [139] to find web pages related to a spatial region. They described three different hybrid indexing structures of integrating inverted files and R*-trees together. According to their experiments, the best scheme is to build an inverted index on the top of R*-trees. In other words, the algorithm first sets up an inverted index for all keywords, and then creates an R*-tree for each keyword. This method performs well in spatial keyword queries in their experiments; however, its maintenance cost is high. When an object insertion or deletion occurs, the solution has to update the R*-trees of all the keywords of the object. Cong *et al.* [31] illustrated a hybrid index structure, the IR-tree, which is a combination of an R-tree and inverted files to process location-aware text retrieval and provide k best candidates according to a rank system. They also proposed the DIR-tree and the CIR-tree, two extensions of the IR-tree, which take both minimizing areas of enclosing rectangles and maximizing text similarities into account during construction procedures. Recently, Cary *et al.* [25] proposed an efficient approach of answering top-k spatial boolean queries. They combined an R-tree with an inverted index to search the k best candidates which satisfy a group of boolean constraints. However, with their method, only candidates which completely meet boolean constraints will be found. The ones merely matching part of the constraints will simply be discarded because of strict constraints or an error input by mistake.

Felipe *et al.* [40] developed a novel index, IR^2-Tree which integrates an R-tree and signature files together, to answer top-k spatial keyword queries. They record signature

information in each node of R-trees in order to decide whether there is any object which satisfies both spatial and keyword constraints simultaneously. However, the size of space for storing signatures in each node is decided before IR^2-Tree construction. Once the IR^2-Tree has been built, it is impossible to enlarge the space unless reconstructed. If the number of keywords grows quickly, a system will spend a lot of time on repeatedly rebuilding IR^2-Tree. Hariharan *et al.* [53] proposed an indexing mechanism, *KR*-tree*, which combines an R*-tree and an inverted index. The difference between their solution and [40] is that they only store related keywords in each node of an R*-tree in order to avoid merging operations to find candidates containing all keywords. Consequently, the number of keywords that appear in each node varies. However, such a complicated indexing technique has a high maintenance cost as well. If an object with new keywords is inserted, the method not only has to add new keywords to corresponding nodes from leaf to root of the R*-tree, but also update the inverted index (*KR*-tree* List).

Although there are a number of previous studies on spatial keyword queries, the solutions can only evaluate queries in Euclidean spaces. This limitation is due to the adoption of the R-tree (or its variants), which cannot index spatial objects based on network distances, into their hybrid index structures. In addition, it is infeasible to provide partial results with existing solutions, which consider both spatial and keyword constraints simultaneously.

Chapter 3

Frog: a Framework for Context-Based File Systems

This chapter presents the **Frog** framework for **Context-Based File Systems (CBFSs)** that aim at simplifying the development of context-based file systems and applications. Unlike existing informed-based context-aware systems, Frog is a unifying informed-based framework that abstracts context-specific solutions as views, thereby allowing applications to make view selections according to application behaviors. The framework can not only eliminate overheads induced by traditional context analysis, but also simplify the interactions between the context-based file systems and applications. Rather than propagating data through solution-specific interfaces, views in Frog can be selected by inserting their names in file path strings. With Frog in place, programmers can migrate an applications from one solution to another by switching among views rather than changing programming interfaces. Since the data consistency and resource protection are automatically enforced by the framework, file system developers can focus their attention on the context-specific solutions.

We implement two prototypes to demonstrate the strengths and overheads of our design. The Bi-context Archiving Virtual File System (BAVFS) utilizes conservative and aggressive prefetching for the contexts of random and sequential reads. The Bi-context Hybrid Virtual File System (BHVFS) combines the *update-in-place* and *update-out-of-place* solutions for read-intensive and write-intensive contexts. Our experimental results show that the benefits of Frog-based CBFSs outweigh the overheads introduced by integrating multiple context-specific solutions.

This chapter is organized as follows. Section 3.1 introduces basic concepts of the context-based file system, and compares it with context-aware systems. Section 3.2 illustrates the

design of the context-based file system and the Frog framework. The implementation details of the two prototypes are described in section 3.3. The experimental results are shown in section 3.4. We discuss several interesting topics related to our design in section 3.5.

3.1 Context-Based File Systems

3.1.1 Basic Concepts

An **I/O Context** is a set of interrelated conditions, under which I/O operations are performed. A file system may leverage access-pattern information exploited from an I/O context (i.e., conditions) to improve I/O performance. For instance, data can be prefetched before being required by applications in a sequential-read context.

An **I/O Context-Specific Solution** is a set of techniques or mechanisms deployed to maximize benefits (e.g., performance and energy efficiency) of I/O operations in a given context. Regularly, a context-specific solution that fits one context may be inadequate for another context. For example, a file system may benefit from a prefetching solution in a sequential-read context, whereas the solution can cause performance degradation in a random-read context due to retrieving unused data.

A **Context-Based File System (CBFS)** is a file system that encapsulates an array of I/O context-specific solutions. A CBFS exposes the solutions to and receives context selections from applications. When a context is selected by an application, I/O operations will be processed by the context-specific solution of CBFS.

A **Context-Based Application** is a program that is able to identify its contexts, perceive contexts provided by systems, and select the best context leading to good performance. Differing from context-aware applications, context-based applications can interact with context-based systems to select contexts based on both I/O operations and available contexts offered by the systems.

3.1.2 Context-Aware vs. Context-Based File Systems

In this subsection, we show differences between context-based file systems and context-aware file systems. Context-aware file systems can be roughly divided into two categories: heuristic-based and informed-based systems. Fig. 3.1 illustrates that three types of file systems encapsulate a set of context-specific solutions. The methods of context identification and solution selection are completely different among the three approaches.

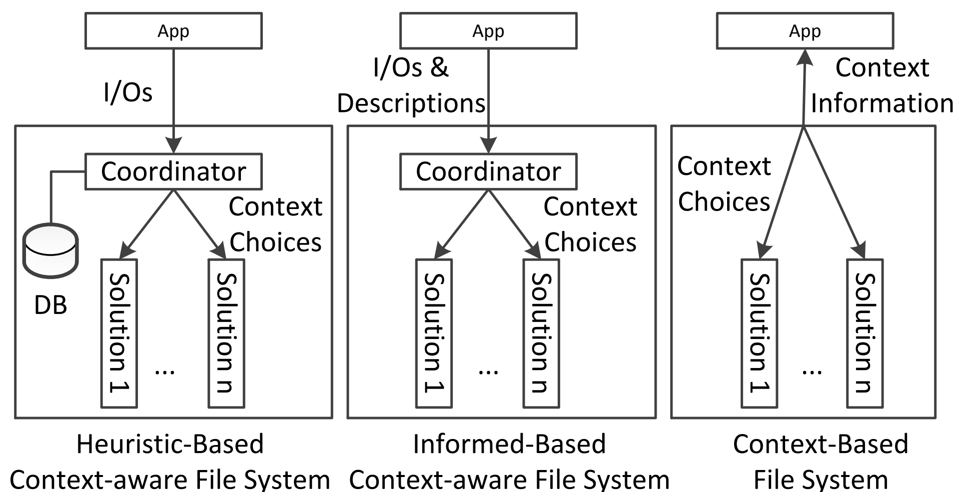


Figure 3.1: A comparison of context-aware systems.

In a heuristic-based file system, a coordinator keeps track of system-wide I/O operations in an internal database. By analyzing historical I/O accesses, the coordinator identifies a context for current I/O operations. The advantage of heuristic-based systems is that context processing is transparent to applications. Any application can be supported by heuristic-based systems without modifying the application's source code. Heuristic-based systems have three major drawbacks. First, maintaining I/O records increases the complexity of system design, and incurs extra costs on behavior analysis. Second, accuracy of context selections relies heavily on the analysis. Because applications may exhibit different I/O patterns, recording and analyzing their behaviors may have negative impacts on context identifications. Third, applications with random behaviors (e.g., triggered by events) produce noises to the context identification process.

In an informed-based file system, a coordinator receives I/O operations with their description (e.g., represented by data structures) through dedicated interfaces. The description is transformed into context choices based on particular rules. The informed-based systems do not suffer from the overhead of back-end analysis; accuracy of context identification depends on applications. However, the systems have to incorporate additional interfaces to collect extra information from applications. Currently, the interfaces are abstracted in various ways. The lack of consistent information propagation prevents the informed approach from widespread deployment.

Context-based file systems, or CBFSs, aim to overcome the aforementioned problems in context-aware file systems. CBFS can be envisioned as a special informed-based system without relying on context analysis. The context selections are propagated from applications. In addition, CBFS provides unifying interfaces between file systems and context-based applications, meaning that the applications can easily and freely migrate among context-based file systems.

3.1.3 The Frog Framework for Context-Based File Systems

Encapsulating multiple context-specific solutions in a file system is challenging. The methods of organizing metadata and physical data in the solutions may be different, implying that a potential conflict may allow only one method to be applied due to the fact that most systems have only one copy of metadata and physical data. If CBFS maintains multiply copies of data, each of which is managed by solutions individually, other issues are raised by data duplications. Take the consistency issue for example; once data handled by a solution has been updated, other solutions have to change their managed data accordingly.

To simplify context-based file system design, we propose a general framework, Frog, that provides (1) a unifying interaction mechanism between systems and applications, and (2) a solution encapsulation mechanism. Fig. 3.2 shows that the context-specific solutions are abstracted as views in Frog. Frog is implemented as an intermediate layer between

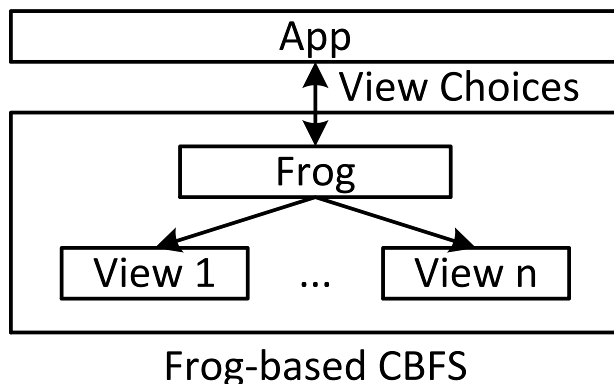


Figure 3.2: The framework of a Frog-based CBFS.

applications and views. Frog and applications exchange the view information through a unifying interface at runtime. Propagating a view from applications to Frog indicates that the corresponding context is selected by the applications. From the perspective of application development, the unifying interface significantly reduces the costs of solution migrations, replacing an existing solution with a new one so that only the view information needs to change.

Importantly, Frog makes it easy to create individual spaces for context-specific solutions, each of which maintains data in its own space. Separating data spaces can allow a solution to be plugged in as if there were only one solution in the file system. Frog is in charge of solution maintenance issues such as consistency. Due to the independence among solutions, one or all of the solutions can have their own heuristic-based methods, e.g., tracking and analyzing I/O behaviors in specific contexts. In Section 3.5.3, we present a special case in which heuristic-based methods are utilized to boost the I/O performance of view-unaware applications.

3.2 Design

In Frog, we creatively introduce the view concept into file systems. Three essential attributes, file metadata management, physical data management and I/O operations, are encapsulated in views. In the subsection, we highlight the overview of Frog as well as new

concepts by illustrating Frog-based CBFSs. Next, due to the diversity of physical data management, three Frog-based CBFS designs are proposed. Finally, we discuss three types of overheads in Frog-based CBFSs.

Since our study mainly focuses on designing a framework for CBFSs, we do not demonstrate context-specific solutions in details. In real CBFS development, existing solutions can be selected to be implemented in views. We will show two concrete prototypes as CBFS examples in the next subsection.

3.2.1 Overview

Fig. 3.3 shows the hierarchical structure in the metadata space of Frog. Directories, views, and files are three types of abstractions represented by white, gray, and black boxes, respectively. These directories and files are called **View-Enhanced Directories (VEDs)** and **View-Enhanced Files (VEFs)**. To concisely and accurately illustrate our design, we focus on the differences between Frog-based CBFSs and traditional file systems. We do not discuss the design principles of Frog-based CBFSs that are similar to those of the traditional file systems.

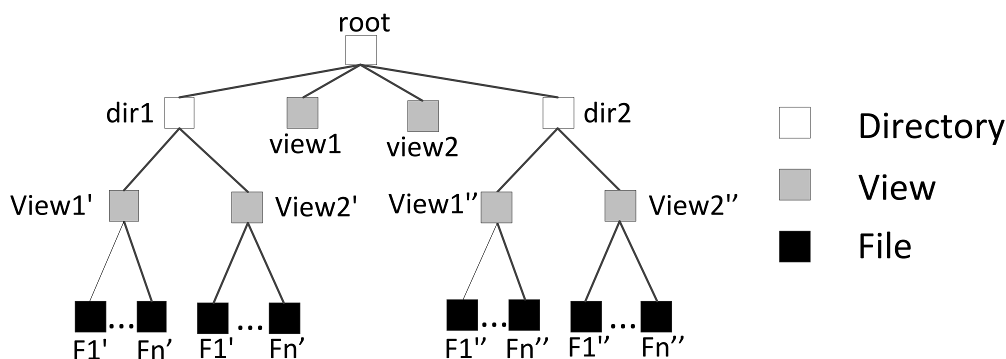


Figure 3.3: Overview of a Frog-based CBFS.

A Frog-based CBFS is organized in a tree structure, the root of which is an instance of VED. A VED manages other VEDs and a set of views, but does not directly contain any VEF. The number of views in a VED is configured when the CBFS is developed. Once the

CBFS is mounted to a point of the naming space, the number of views cannot be modified (dynamic configuration will be considered in future work). Note that views under a VED are the instances of different view types. Fig. 3.3 illustrates an example of two view types configured in a Frog-based CBFS. *view1*, *view1'* and *view1"* are three instances of a view type, and *view2*, *view2'* and *view2"* are instances of another view type. In the root node, *view1* and *view2* are two instances of different types. Throughout this section, views are referred to as view instances if the views are not explicitly indicated as view types.

Views are an intermediate layer between VEDs and VEFs. A view is a container that organizes only VEFs rather than VEDs or other views. Thus, views do not contribute to the hierarchy construction. The views under a VED expose logically identical interfaces and metadata of VEFs (*e.g.*, file names and file sizes) to applications; however, the implementation details (*e.g.*, managing VEFs in a B-tree or a hash table) of views may vary.

VEFs are the basic units that refer to physical data on devices. In traditional file systems, a file is a logically unique unit that has only one copy of metadata. However, Frog maintains multiple metadata copies (VEFs) for a logical file. VEFs are always the leaf nodes in the tree structure.

It is worth noting that two special cases of Frog-based CBFSs are the Single-context File System (SFS) and Bi-context File System (BFS). SFS is configured with one view type. Only one view instance is created in VEDs. SFS degrades to a traditional file system that does not duplicate any metadata of logical files. The only difference between SFS and the traditional file system is that SFS separates the file metadata and encapsulates them in views. Unlike SFS, BFS is the simplest form of Frog-based CBFSs that maintains duplication. Compared with other Frog-based CBFSs with more than two views, BFS incurs the minimum overhead in data duplication management.

Fig. 3.3 shows an example of BFS, where the root VED has two VEDs (*i.e.*, *dir1* and *dir2*) and two views (*i.e.*, *view1* and *view2*). These two views do not have any VEF. *dir1* has two views (*i.e.*, *view1'* and *view2'*), each of which contains a number of VEFs. The file

information exposed from *view1'* and *view2'* is identical. *dir2* has organization similar to *dir1*.

3.2.2 Operations

Now we discuss the operations of VEDs, views, and VEFs at the file system level. Let us emphasize on new features supported by Frog. In order to show compatibility with existing applications, we set views of a particular type in VEDs as default views. The default views play a key role in consistency maintenance and locking mechanism design.

Frog-based CBFS Initialization/Finalization

Let us start the operation description with the initialization/finalization processes in Frog-based CBFSs when the file systems are mounted/unmounted at a point in the naming space. Before deploying a Frog-based CBFS, a format program should be executed to construct a super block, set up metadata and data partitions, and create a root VED and its sub-views. The format program builds an empty file system, the information of which is stored on a disk. When the CBFS is mounted, the root VED and sub-views are loaded into main memory. Then, the CBFS is ready to serve I/O requests. When the CBFS is unmounted, all metadata should be permanently serialized to disks.

VED Operations

VED operations supported by Frog-based CBFSs are directory creation, deletion, and listing. These operations differing from those of traditional file systems are that Frog must consider views in addition to directories. In the creation process, a new VED is created and appended to a children list of its parent. Views under the new VED are also generated. Hence, by default, an empty VED has two special directories ("." and "..") and a number of empty views. Views are not created under "." and ".." directories.

In traditional file systems, only empty directories are allowed to be deleted. Frog follows this principle; a VED can be deleted if it does not contain any other VEDs (except for "." and "..") and all sub-views are empty. In the process of VED deletion, its sub-views are automatically removed.

The directory listing operation (through *readdir* system call) will be discussed in the default view subsection (see Section 3.2.2), where we show how to make trade-off between supporting compatibility for existing applications and exposing views to view-aware applications.

View Operations

In Frog, view supports creation, deletion and listing operations. The listing operation can be individually issued, but the creation and deletion have to be committed by VED creation and deletion operations. In the view creation, a new view that maintains an empty VEF list is created. The relationship between the new view and its parent VED is determined by the VED creation operation. A deletion operation can only delete empty views that do not have any VEFs. When a listing operation (through *readdir* system call) is issued on a view, the names of VEFs in the view will be returned.

The names of views are pre-determined by their types. For example, in Fig. 3.3, *view1*, *view1'* and *view1"* are instances of a view type. These views can share an identical name (*e.g.*, "view1") without any name conflicts because they are in different VEDs. If the names of view types are different, views in a VED cannot conflict with each other as well; however, similar to "." and ".." directories, the view names pollute the naming space in VEDs, so that the VED names cannot be the same as the view names. The name conflicts will be detected before a VED is created. The operation will be denied if any name conflict exists. The conflicts cannot occur during view creation since there is no sibling-VED existed when a view is created.

VEF Operations

Due to metadata duplication across views, creating/deleting a file in a view must trigger the view synchronization in order to keep metadata consistent. This synchronization process takes place in any operations, in which the metadata will be modified.

Moreover, Frog must consider synchronization issues of concurrent VEF accesses, which significantly differs from tradition file systems. For example, two files may be concurrently created by applications through two views. In another example, one file may be concurrently modified by applications through two views. Without concurrent controls, the metadata consistency will be violated. To simplify our design, the consistency problem is solved by applying a lock mechanism. Two locks are set for VEF creation/deletion and read/write operations, respectively. The creation/deletion lock, setting in the parent VED, has to be acquired before any VEF is created/deleted. The read/write lock mechanism will discussed in Section 3.2.6.

Default View Configuration

In our design, we classify applications into two categories: view-aware and non-view-aware applications. Obviously, existing applications are non-view-aware. The challenge of supporting both types of applications is that views are visible in view-aware application and invisible in non-view-aware ones. To show compatibility with non-view-aware applications, we set up view instances of a particular type as **Default Views** in VEDs, which is used when view names are missing from file paths. We describe the following typical scenarios to demonstrate how the default views work.

File paths of two types of applications are different. In Fig. 3.3, VEF *F1'* under *dir1* is referred to as the path "/root/dir1/view1'/F1'" in view-aware applications, and "/root/dir1/F1'" in non-view-aware applications. The difference between these two file paths is that the view name is visible in the path string in view-aware applications.

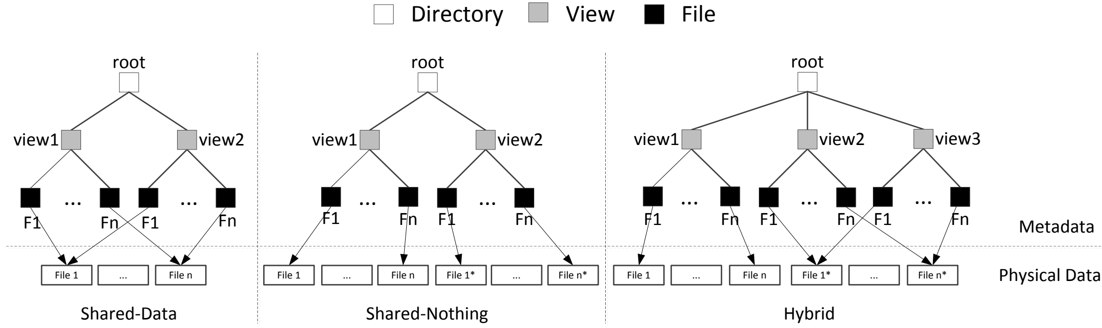


Figure 3.4: Frog-based CBFS designs.

The default views are used when view names are missing from file paths. For example, when the *open* system call is issued on "/root/dir1/F1'", the last entity in the path string must be the file name *F1'*, and the next to the file name can be either a VED or a view. This can be decided by the name because view names are pre-determined. If the entity is a VED, the default view name (either *view1'* or *view2'*) will be inserted to form a view-aware path.

The semantics of directory listing are different between view-aware and non-view-aware applications. When listing a VED, view-aware applications expect to retrieve the names of sub-VEDs and sub-views; whereas non-view-aware applications obtains the names of sub-directories and files. In case that existing applications cannot be modified, our design adapts view-aware applications to non-view-aware semantics, so that the view names will not be exposed in directory listing. Another reason we compromise the view-aware semantics is that similar to "." and ".." directories, the view names are pre-determined. It is reasonable to assume that view names are known as a priori in applications.

3.2.3 Three Frog-based CBFS Designs

Metadata management has been discussed in the previous subsections. If we take into account physical data organizations, there are three potential designs of CBFSs (see Fig. 3.4). Note that other forms of CBFSs might be derived from these three designs.

Shared-Data CBFS shares one copy of physical data among multiple views. *F1*s in *view1* and *view2* refer to the address of *File1* stored on disks. Shared-data CBFS simplifies the physical data management by sharing data among views. Shared-data CBFS is applicable to the integration of context-specific solutions that are not conflicting in physical data management. BAVFS - a good example of shared-data CBFS - integrates aggressive and conservative prefetching in two views. Theoretically, these two prefetching techniques do not interfere with physical data management. Instead, the techniques address the issues of read operations (*e.g.*, how much data is prefetched each time) by duplicating metadata.

Shared-Nothing CBFS advocates a data duplication approach, where each copy is managed by a view; thus, VEFs in views refer to the addresses of data in separate copies. A typical example of shared-nothing CBFS is BHVFS, which maintains data replicas for both *update-in-place* and *update-out-of-place* strategies that conflict with each other in data organization. Compared with shared-data CBFS, shared-nothing CBFS has to pay extra overheads to physical data synchronization among views.

Hybrid CBFS is a combination of shared-data and shared-nothing CBFSs. In a hybrid CBFS, some views share data whereas others separately manage duplicated data. An example of hybrid CBFS is to combine *update-in-place* and *update-out-of-place* strategies with prefetching techniques, in which *update-out-of-place* is applied in *view1*, and *update-in-place* in *view2* and *view3*. In addition, conservative prefetching is applied in *view2* for random reads; aggressive prefetching is used in *view3* for sequential reads.

3.2.4 Interactions with Applications

Besides the diversity of Frog-based CBFS designs, Frog is incredibly flexible to support a wide range of applications. Thanks to the availability of all views in Frog, one view or multiple views can be chosen by applications. In the current design, views (or contexts) are pre-determined, Frog does not expose the view information by a dedicated interface. Instead, applications provide their view selection by inserting the view names in the file path

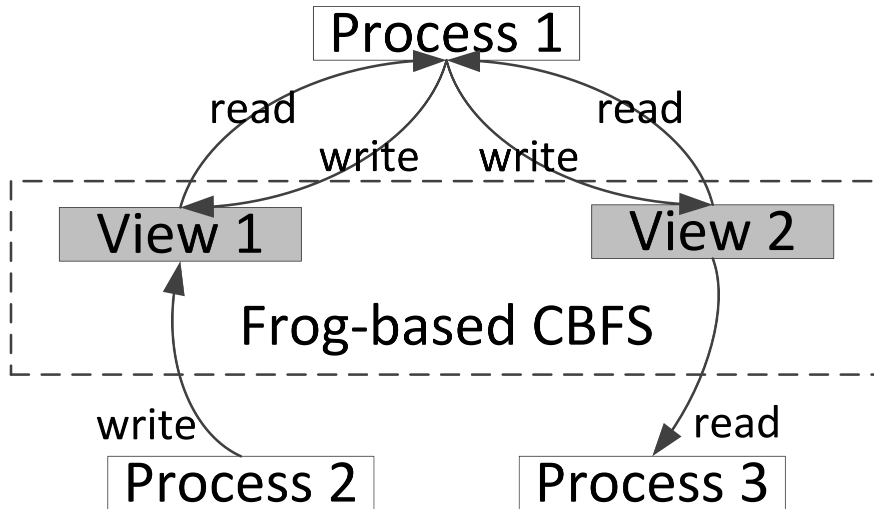


Figure 3.5: Interactions between processes and Frog.

string. Fig. 3.5 displays two typical scenarios of interactions between processes and a Frog-based CBFS. On top of the figure, process 1 concurrently communicates with both views; in particular, it would choose the best view for an operation. For example, if the *update-out-of-place* strategy is employed in *view1* and *update-in-place* in *view2*, the process can commit writes through *view1* and retrieve data using *view2* to achieve better performance. Fig. 3.5 shows another example at the bottom. The data written by process 2 through *view1* can be retrieved by process 3 through *view2*.

The concrete examples of view-aware applications are shown in Fig. 3.6. *foo* writes "hello world" in a given file, and *bar* reads a string from a given file (see the top two functions in Fig. 3.6). In the first case at bottom left, an application processes two files from two views (*i.e.*, update file1 from *view1* and read file2 from *view2*). The application can switch from one view to another at runtime. In the second case at bottom right, an application updates data in a file and another application later retrieves the data from the file. These two applications can be supported by two different views.

```
foo (char* path){
   char* str = “hello world”;
   FILE* fh = fopen(path, “w”);
   fwrite(str, strlen(str)+1, 1, fh);
   fclose(fh);
}

Application 1:
void main (){
  foo (“.../view1/file1”);
   ...
  bar(“.../view2/file2”);
}
```

```
bar(char* path){
   char buf[20];
   FILE* fh = fopen(path, “r”);
   fread(buf, sizeof(buf), 1, fh);
   fclose(fh);
}

Application 2:
void main(){
   foo(“.../view1/file1”);
}
Application 3:
void main(){
   bar(“.../view2/file1”);
}
```

Figure 3.6: Examples of view-aware applications.

3.2.5 Application Comparisons

Recall that context-based file systems are different from context-aware file systems (see Section 3.1.2). Now we further illustrate their differences by comparing applications running in these systems. Apparently, the applications running on heuristic-based systems are context-unaware applications, which do not identify and disclose context information. On the other hand, applications running on context-aware and context-based systems are context-aware. In contrast to context-aware applications, context-based applications can identify contexts based on a combination of their behaviors and contexts perceived from systems. Such ability offers context-based applications a powerful adaptability that enables the applications to perform well in diverse and changeable systems.

The ease of context-specific solution migration estimated by the efforts paid on solution replacement is different among the three types of applications. Because heuristic-based applications are transparent to solutions, no application modification is necessary when applications are migrated from one solution to another; however, the other two types of applications are aware of solution replacements. Thanks to a unifying interface provided by Frog,

context-based applications only change view names, whereas context-aware applications may have to shift to a new interface applied by the new solution.

Fig. 3.7 illustrates a concrete example where an application must be modified in case of a solution replacement. In this example, a prefetching solution proposed by Patterson *et al.* [90] has been employed in the file system. For some reasons (e.g., hardware updates such as replacing HDDs with SSDs), the solution needs to be replaced by another prefetching solution (for example, [111]). In this case, both the file system and application have to be modified accordingly.

~~code~~ Codes related to the old solution

code Codes related to the new solution

```
context-aware-app()
{
char buf[20];
int fd = open("/tmp/file", ...);
//initialize context parameter
structure context ctx;
...
ctx_action(CTX_BEG);
//initialize hints
HINT_STRU hints;
hints.filename = "/tmp/file";
hints.pattern = SEQ_READs;
...

ioctl (fd, REQ, &hints);
read(fd, buf, sizeof(buf));
ctx_pread(ctx, buf);
ctx_action(CTX_ENG);
close(fd);
}
```

```
context-based-app()
{
char buf[20];
int fd = open("/tmp/view1/file", ...);
int fd = open("/tmp/view2/file", ...);
read(fd, buf, sizeof(buf));
close(fd);
}
```

Figure 3.7: Application modification comparisons.

An intuitive modification method is to remove codes related to the old solution, and to write new codes for the new solution. The old codes are indicated by strike-through texts;

the new codes are highlighted by gray background. In the context-aware applications (on the left-hand side in Fig. 3.7), a total of 6 lines of codes are removed and 5 lines are newly written, which is more than 2-line changes in the context-based applications (on the right-hand side in Fig. 3.7). Please note that changes of real-world context-aware applications are more complex than those demonstrated in this example.

3.2.6 Overheads

Due to data duplication, Frog introduces three types of overheads that are data consistency, resource contention and data duplication.

Data Consistency

In Frog, views are managed in separate spaces. Once an update is issued in a view, the update will be synchronized with other views in order to keep all of the views consistent.

Due to replicated metadata, all three designs mentioned earlier have to address the issue of metadata synchronization. The more views configured in Frog, the higher the consistency maintenance overhead is. Importantly, the overhead of physical data synchronization differs significantly among the designs. Shared-data CBFSs do not conduct any physical data synchronization, whereas shared-nothing CBFSs have to synchronize all updates among the views. Hybrid CBFSs can be anywhere between shared-data and shared-nothing CBFSs. Files are normally significantly larger than their metadata [17] [116]; thus, data consistency overhead is dominated by physical data synchronization if there exists one.

Moreover, we also consider crash recovery issues that Frog can automatically reconcile among views; even some of them are in inconsistent states. We propose two mechanisms for metadata protection. *Centralized Journaling* encapsulates the journaling processing in the framework as shown in Fig. 3.8. The Frog will create a separate area that tracks metadata updates in a *journal* file [106]. The updates will be written to disk before committing them to the main file system. If the file system crashes during the processing, the *journal* file will

bring the system to a consistent state. We can adopt an existing journaling method in Frog. For physical data updates, we can simply apply ordered update mechanism that data blocks will be committed to disk before the metadata [78].

When committing metadata updates to the main file system, the updates will be committed to one view. Other views will cache the updates in memory until the disk is not busy. In this way, the file system has at least one view in consistent state, and the costs of synchronization is hidden in back end. In addition, the parent directory should maintain a bit map to indicate that which view is in consistent state. When the system recovers from a crash, file information will be retrieved from the view in consistent state. The consistent view can be either statically (*e.g.* the default view) or dynamically selected by a particular criteria.

Centralized Journaling reduces the difficulty of system development by providing a unifying journaling mechanism. Developers can focus on the functionalities of the context-specific solutions. On the other hand, the flexibility of system design is compromised. All metadata updates have to be journaled, which prevents from integrating a non-journaled solution (probably for performance consideration). In addition, the entire file system is running under a journaling mechanism. Other metadata protection methods, such as soft update [79], log-structure file systems [99] and copy-on-write [1] [23], can not be utilized in the file system.

The other mechanism is *Distributed Journaling* that provides developers with flexible journaling choices. Distributed Journaling mechanism separates file updates from directory updates. The directory updates are journaled by Frog as centralized journaling mechanism does; however, all file updates are directly forwarded to views, which protect metadata individually. During the update distribution, one view is selected as a consistent view, and commits the updates to disk. Other views cache the updates in memory. In each view, the view-specific solution selects the metadata, even physical data, protection mechanisms.

Under distributed journaling mechanism, any metadata protection technique can be used in views. However, the metadata consistency is actually protected by the consistent

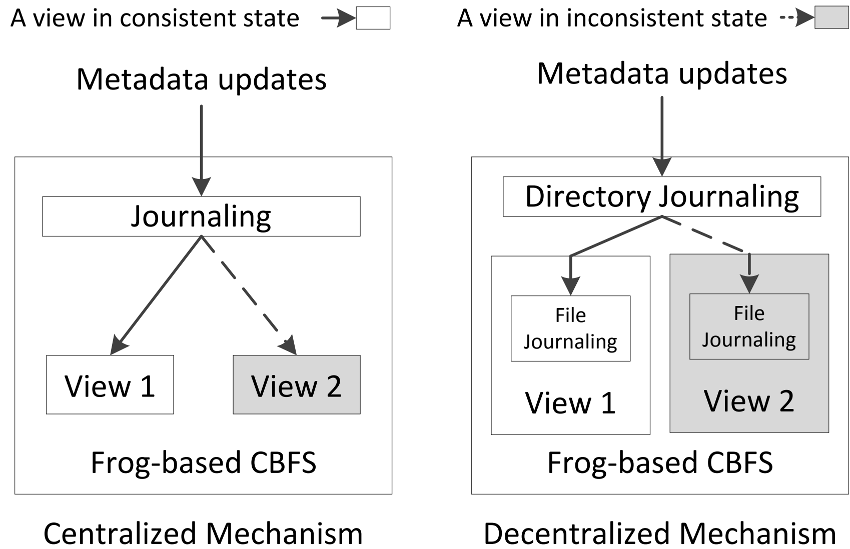


Figure 3.8: Two consistency mechanisms.

view. If the view is dynamically selected, the consistency is actually protected by the weakest mechanism among views. The bit map in the directory has to be updated after view updates, which will be processed by the directory journal in Frog. If a crash occurs between directory and file updates, the system will recover from a consistent view indicated by the bit map in the last update, which means that the latest file update is lost.

Resource Contention

Resource contention in traditional file systems takes place when two concurrent requests (*e.g.*, a read and write) are issued on a file. A file locking mechanism can be utilized to solve this problem. Before issuing any I/O operation, applications must successfully acquire a lock. We have to carefully address the contention problem in the context-based file system because two requests are allowed to be issued on a logical file from two views by view-aware applications.

In a traditional file system, the locking mechanism is carried out in two steps. First, applications retrieve a file descriptor by opening a file. Then, a file lock operation is issued through *flock* or *fcntl* on the file descriptor. Frog follows the two steps. When opening a

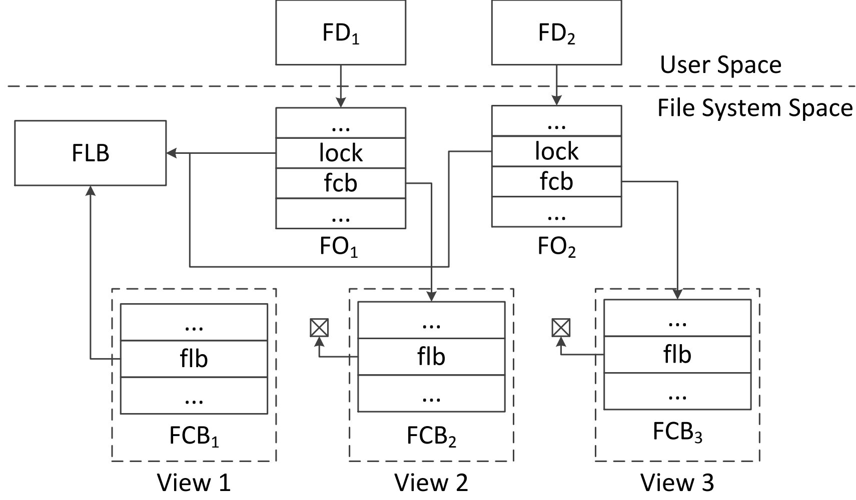


Figure 3.9: File locking among views.

file, Frog will create a **FO** (File Object) that interacts with applications. The **FD**s (File Descriptor) held by applications refers to the FOs, shown in Fig. 3.9. The FO has two key fields: *lock* refers to a **FLB** (File Locking Block) that manages the lock associated with the logical file. *fcb* points to **FCB** (File Control Block) (*e.g.,* an inode) in the tree structure. Only the *flb* field of the FCB in the default view (*e.g.,* view 1) refers to the FLB. The *flb* of the FCB in other views are always NULL. The details of the lock operation is described in Algorithm 1. The fundamental idea is that only one lock is created for a logically unique file. When a locking operation is issued, FCB in the default view is loaded into memory.

Before an FLB is associated with a logic file, the *lock* fields in FOs and *flb* fields in FCBs are NULL. When a lock operation is issued, Frog checks the associated FO at first. If it refers to a FLB, the locking request will be committed on the FLB. Otherwise, the *flb* field of the corresponding FCB in the default view will be checked. If a FLB has been created, the *lock* field of the FO will refer to the FLB. Otherwise, a new FLB will be created, and the locking operation will be issued on the newly created FLB.

Algorithm 1 flock(fd, op)

```
1.  Get fo by fd
2.  if fo.lock != NULL then
3.      lock (fo.lock, op)
4.      return SUCCESS
5.  end if
6.  Find fcb in the default view, which manages a logically unique file with fo.fcb
7.  if fcb.flb == NULL then
8.      Create a new file locking block lb
9.      fcb.flb = lb
10. end if
11. fo.lock = fcb.flb
12. lock (fo.lock, op)
13. return SUCCESS
```

Duplications

Frog requires extra disk and memory space to accommodate metadata duplications. Increasing the number of views in the file system can drive the disk space overhead up. The space overhead for physical data duplication greatly varies among designs. As shown in Fig. 3.4, the shared-data CBFS does not have any physical data replicas; on the other hand, the number of data replicas maintained by the shared-nothing CBFS is equal to the number of views.

The reason that we create metadata replicas in all three designs and do not carry out any de-duplication technique is that the size of metadata is relatively small compared to the overall capacity of file systems. According to the study that the mean and median fullness (ratio of usage to capacity) of file systems are between 40% to 50% [17]. Disk space is available to metadata replicas in most cases. In addition, disk space is seemingly wasted for physical data duplications; however, data reliability of file systems is improved by the virtue of data replicas. When one data copy is lost due to bad sectors, the data can be retrieved from other copies. Benefiting from flexibility of system design, data replicas can be distributed across multiple disks or servers to further boost system reliability. Typical

storage systems that offer data replicas include RAID-1 (data is mirrored) [89], GFS [48] and HDFS [108] (3 copies are created by default).

3.3 Case Studies

We conduct two case studies to evaluate the effectiveness, generality, and flexibility of the Frog framework. In the first case study, we outline the implementation issues of BAVFS - the shared-data design of a context-based file system that applies a dual-mode (*i.e.*, aggressive and conservative) prefetching for the context of sequential and random small reads. In the second case study, we focus on BHVFS - the shared-nothing design that separates read-intensive contexts from write-intensive ones, in each of which either the *update-in-place* or *update-out-of-place* strategy is employed to improve I/O performance.

The prototypes of BAVFS and BHVFS are implemented on the top of the FUSE file system [115], which is popular in the file systems research community [57][114][125]. Directories are used to emulate views in a hierarchy structure. Our prototypes enforce the differences between the two abstractions. We adopt the distributed journaling mechanism without directory journaling in the framework of both prototypes because the underlying file systems (Ext 4 in our experiments) take care of metadata protection. The operation interfaces implemented in the prototypes include: *open*, *create*, *unlink*, *read*, *write*, *release*, *mkdir*, *rmdir*, *readdir*, *getattr*, and *lock*.

We may further optimize the prototypes. For example, considering alphabetical order in the metatada management of BAVFS, applications can sort data returned from *readdir*. Nevertheless, we implement the prototypes in a straightforward way while focusing on the strengths and overheads of context-based file systems.

3.3.1 The BAVFS File System

Overview

The data explosion problem has been observed in the past few years. The number and diversity of files are growing rapidly [45]. Recent studies [17][116] show that more than 50% of all files in a file system are small (*e.g.*, smaller than 4 KBytes); thus, it is critical to develop modern file systems that can efficiently process large number of small files [24][88][104].

Two common types of data access patterns in I/O-intensive applications are sequential and random reads. For example, all small files in a directory are sequentially scanned by antivirus softwares, whereas one of these files is simultaneously opened by an editor. Sequential data retrieval can be improved by prefetching techniques [68] [74] [90]; however, the unpredictability of future accesses is a natural barrier to adopting the prefetching techniques. Unnecessary prefetched data can significantly degrade system performance.

We implement the dual-mode I/O prefetching in a file system. A similar dual-mode prefetching mechanism can be found in the realm of instruction retrievals [72]. Aggressive prefetching is applied in the sequential context, thereby avoids unnecessary prefetches due to random data retrievals; on the other hand, conservative prefetching is employed in the random context. BAVFS is different from existing prefetching approaches in that BAVFS provides a context-based prefetching approach, where not only data in operating files but also in files to be accessed in future can be prefetched based on the contexts.

Implementation

To shorten relocation times in the sequential view, small files in a directory are aggregated into large files. As shown in Fig. 3.10, the physical data of files are merged into big *.data* files. The *.meta* files store the metadata in the views. The *.data* and *.meta* files appear in pair, and the order of metadata in *.meta* keeps the same with the ones in the

corresponding *.data* file. When a file is required, its metadata will be loaded in memory under the random view; whereas the entire *.meta* file will be loaded in the sequential view.

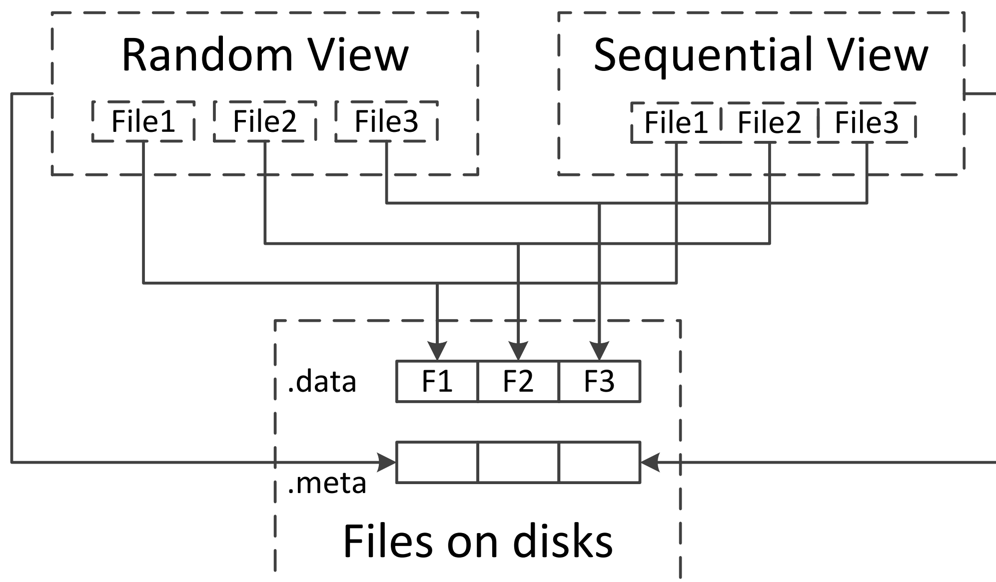


Figure 3.10: Structure of BAVFS.

An extendible hash method is used in file name mapping [39] (see Fig. 3.11). In both views, 128-bit hash values are generated from file names. The first 16 bits form the names of *.meta* files; the remaining 112 bits represent a unique index number in the *.meta* files. Each *.meta* file manages a set of entities, each of which is made up of an index number and file metadata information, including the location of physical data (i.e., *.log* file name, offset, and file size).

An update operation may be expensive. For instance, if the current block of a file is not sufficient to accommodate an update due to an increase in file size, the entire file must be moved to another location. Though such data movement incurs extra I/O costs, the design is reasonably effective for write-once-read-many access patterns [108]. Moreover, although two copies of metadata are managed in the two views, they actually share one copy of metadata on disk. The difference of metadata management in the views only exists in memory.

When a read operation is issued, 30 KBytes of data is prefetched and cached in memory in the sequential view. On the other hand, no data is prefetched in the random view. The

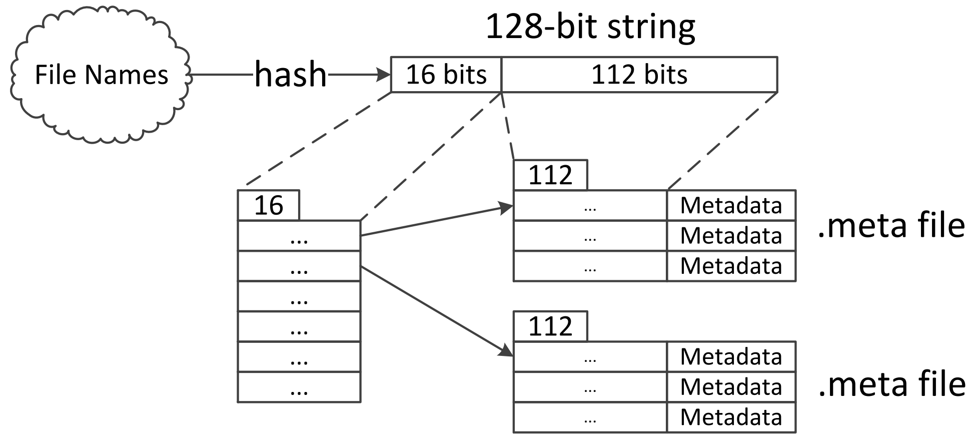


Figure 3.11: Name processing in two views.

reads are directly forwarded to the underlying file system. The random view is set as the default view of BAVFS.

3.3.2 The BHVFS File System

Overview

The efficiency of data retrieval and storage is critical in modern file systems. Two prevailing file systems, FFS [80] and LFS [99], embrace optimization techniques for reads or writes; however, few existing file systems are able to provide competitive performance on both reads like FFS does and writes like LFS does.

To maximize the strengths of LFS and FFS, hFS [138] incorporates them for both types of operations. Since either LFS or FFS technique is selected in hFS for a file according to the file size, the file can only benefit from one technique rather than two. In addition, quFile proposes a method that encapsulates physical representations of files for contexts at the file level. Enjoying the benefits from FFS and LFS cannot be addressed by quFile at the file system level. The aforementioned limitations of hFS and quFile motivate us to develop a BHVFS that can exhibits the strengths of both FFS and LFS. Their two completely different strategies are adopted in read-intensive and write-intensive views, each of which shows the strengths and weaknesses of either FFS or LFS. If the views are appropriately selected by

view-aware applications, the strengths of both views are maximally leveraged to achieve high performance for both reads and writes.

Implementation

Fig. 3.12 shows that both views individually manage metadata in the metadata space. File metadata in a directory is organized in form of a balanced search tree. In the physical data space, the FFS view applies the *update-in-place* strategy that overwrites data in the corresponding files; on the other hand, the LFS view uses the *update-out-of-place* strategy that appends updates to big *log* files. In addition, the LFS view maintains *meta* files to record the metadata information. We do not create a cleaner program to reorganize the *log* files as traditional log-structured file systems do because in most cases reads are issued in the FFS view, in which data are logically and continuously managed.

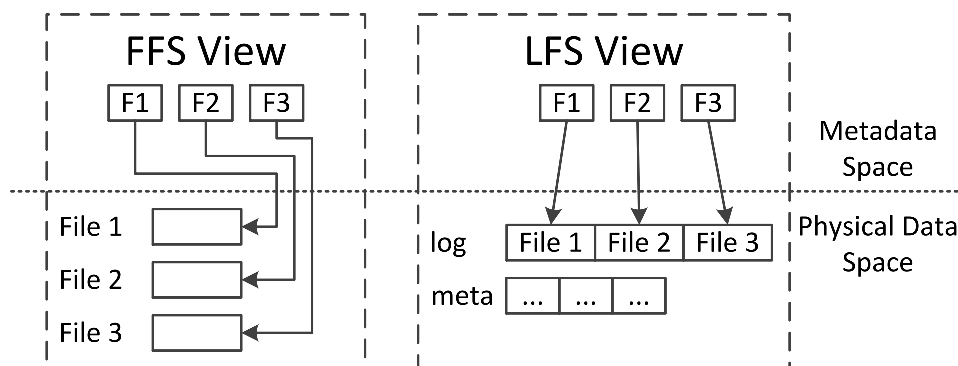


Figure 3.12: The structure of BHVFS.

3.4 Evaluations

In this subsection, we examine the performance of context-based file systems by evaluating BAVFS and BHVFS using our testbed. We mainly focus on 1) investigating strengths derived from context decomposition; and 2) quantifying overheads incurred by introducing multiple views. **BAVFS-CP** and **BAVFS-AP** denote conservative and aggressive prefetching views in BAVFS. **BHVFS-FFS** and **BHVFS-LFS** denote FFS and LFS views

in BHVFS. To fairly compare BAVFS and BHVFS with existing file systems, we have to consider the negative impacts imposed by the FUSE module and virtual file systems (VFS). Rather than minimizing the negative impacts of FUSE and VFS, we create the following four baseline file systems using FUSE. **FS-CP** and **FS-AP** are traditional file systems that apply identical metadata management, data layout and I/O operations with **BAVFS-CP** and **BAVFS-AP**, respectively. The only difference is that **FS-CP** and **FS-AP** do not contain views. **FS-FFS** and **FS-LFS** are fuse-based FFS and LFS.

We evaluate these file systems on a Ubuntu desktop computer with 2.2 GHz Intel Celeron CPU, 1 GBytes memory and two 160 GBytes Sata disks [11]. The Linux kernel version is 2.6.35. Ext 4 [78] is configured as the underlying file system, on which FUSE-based file systems are running. In the experiments, BAVFS is running on a single disk; whereas two disks are utilized for BHVFS, in which views maintain their data in separate disks.

We conduct the following three groups of experiments.

Metadata Operations. The first group is designed to measure the performance of file creation and deletion.

Random and Sequential Reads in BAVFS. We evaluate the impacts of aggressive and conservative prefetchings on sequential and random reads in BAVFS.

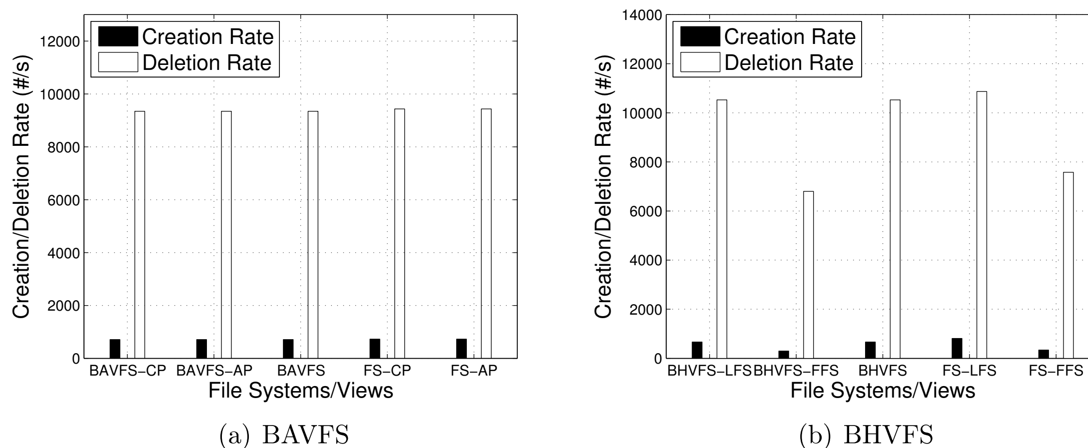
Random Reads/Writes in BHVFS. We quantify the strengths and weaknesses of LFS and FFS in the context of BHVFS by issuing random reads and writes to the file system.

During each experiment, we focus on testing a single view instead of the entire file system. This testing strategy allows us to demonstrate the performance of basic I/O operations in the light of multiple views. In the figures shown in the following subsections, BAVFS-CP, BAVFS-AP, BHVFS-LFS, and BHVFS-FFS indicate results of issuing operations on the views. In order to show the performance of BAVFS and BHVFS, we conduct the experiments by switching the default views that support non-view-aware applications with appropriate view selection. Note that the existing benchmarks and applications are non-view-aware. In

the two prototypes, write operations in the default view are pushed to disks by *fsync*, and update operations on the other view are processed by a thread at the back end.

3.4.1 File Creation/Deletion Rate

To test file creation/deletion rate, we use postmark [62] - a file system benchmark - with an initial set of 1,000,000 small files. We set file sizes anywhere between 1 to 2 KBytes. Fig. 3.13 reveals the file creation/deletion rates (*i.e.*, the number of files created/deleted per second) without other I/O transactions.



(a) BAVFS

(b) BHVFS

Figure 3.13: File creation/deletion.

Fig. 3.13(a) displays the comparisons among BAVFS, FS-CP, and FS-AP. We observe that the creation/deletion rates of BAVFS-CP (713/9345) and BAVFS-AP (712/9345) are slightly lower than those of FS-CP (727/9433) and FS-AP (726/9433). The differences in creation/deletion rates are less than 3%, because BAVFS does not actually maintain any replica on disks; even metadata are simply duplicated in memory. When a file is created or deleted, only one update is committed to disks. The creation/deletion process in BAVFS is similar to those in FS-CP and FS-AP.

Fig. 3.13(b) shows that the file creation/deletion rates of BHVFS-LFS are 664/10526, which are 18%/3% slower than that of FS-LFS (809/10869). Compared with FS-FFS

(339/7575), BHVFS-FFS (299/6802) suffers little (*i.e.*, 11%/10%) performance degradation in file creation/deletion. Thanks to committing updates to separate disks, the overhead is hidden by the back-end synchronization process. Moreover, creating/deleting files in the LFS manner is better than FFS, because LFS only appends updates at the end of *.meta* files rather than generating and removing files. BHVFS performs better than FS-FFS in both file creation and deletion, since BHVFS derives the performance from the LFS view.

The performance degradation of BAVFS and BHVFS can be attributed to metadata management. Apart from metadata, physical data duplications may contribute to the performance degradation. When a file is created or deleted, the file object should be constructed or destroyed in the two views; however, the execution time of creation and deletion is not doubled in all cases. The overheads can be avoided or reduced by a careful design of the file systems.

3.4.2 BAVFS

Small random and sequential reads are evaluated in BAVFS, FS-CP, and FS-AP. We create 2,000,000 files of size 1-2 KBytes by a modified postmark that does not execute transactions and delete files after file creation. We use ClamAV [64], an open source antivirus engine as a representative application with sequential access pattern, and Grep [3] as an example with random access pattern, searching a keyword "Frog" from 200,000 files in a random order. Fig. 3.14 displays the execution time of the two applications.

FS-CP (797/821) performs better than BAVFS-CP (857/842); FS-AP (1048/745) is better than BAVFS-AP (1096/755) for random and sequential access patterns, because view-maintenance costs adversely affect the performance of BAVFS. In addition, random reads are more efficient in BAVFS-CP and FS-CP. Conservative prefetching does not prefetch unnecessary data for random accesses. On the other hand, high hit rates of aggressive prefetching make BAVFS-AP and FS-AP perform better than BAVFS-CP and FS-CP in the case of sequential accesses.

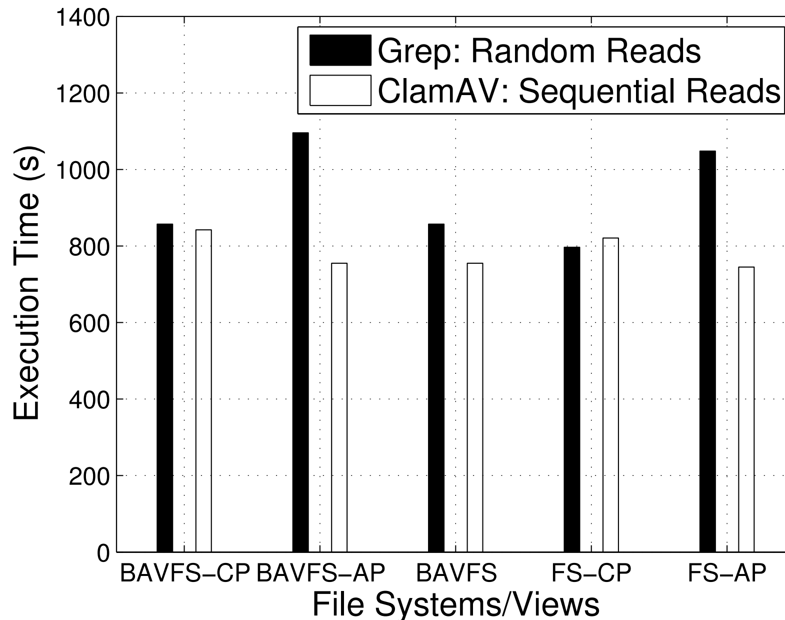


Figure 3.14: Random and sequential read evaluation in BAVFS.

If views are selected appropriately, BAVFS performs 9% better than FS-CP with respect to sequential reads, and 21% better than FS-AP when it comes to random reads. The performance degradation is lower than 8% for random reads with FS-CP, and 1% for sequential reads with FS-AP, because BAVFS experiences extra overhead induced by views. Our empirical study shows that the benefits of BAVFS outweighs its duplication overheads.

3.4.3 BHVFS

We compare the random read and write performance of BHVFS with those of FS-LFS and FS-FFS. To reduce the data consistency overhead, we deploy two disks in this group of experiments. Two views maintain their data on two separate disks. We use postmark to create 20,000 files, each of which is 100-200 KBytes in size. There are 2,000,000 writes and 20,000 reads issued to the tested file systems. The buffer size in each transaction is 4 KBytes. Since postmark creates files in a sequential way, we modify the file creation process so that data are randomly appended to initial files. In doing so, data among files are interleaved in

a big *log* file in the LFS view. We do not create a cleaner program in FS-LFS in order to fairly compare it with the BHVFS-LFS view.

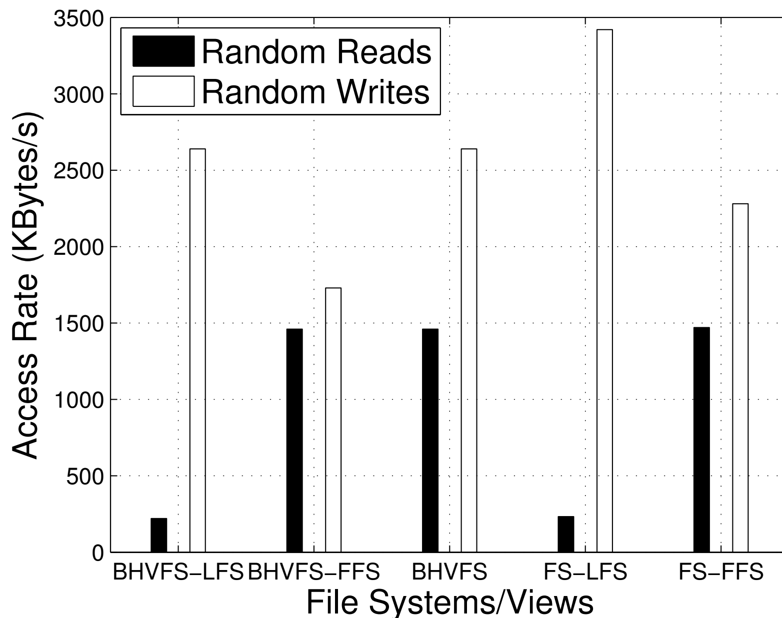


Figure 3.15: Random read and write evaluation in BHVFS.

Fig. 3.15 reveals the access rate (measured in KBytes/Second) of I/O operations. The write rates of BHVFS-LFS and FS-LFS are much higher than those of BHVFS-FFS and FS-FFS, because written data are appended to the big *log* files. No seek is needed for individual files on the disks. In BHVFS, updates in the FFS view are committed to another disk by the back-end process; the rate of BHVFS-LFS is not as low as the half of FS-LFS. The overheads are attributed to the synchronization process as well as resource contentions (*e.g.,* acquiring a lock on an update pending queue and memory consumption).

Due to a continuous organization, the read rates of BHVFS-FFS and FS-FFS are much higher than those of BHVFS-LFS and FS-LFS. When a read is issued to BHVFS-LFS or FS-LFS, the data is retrieved from a number of pieces triggering multiple seeks. Moreover, the difference in read rates between BHVFS-LFS and FS-LFS, as well as BHVFS-FFS and FS-FFS, is negligible. The way of retrieving data in BHVFS is similar to that in the traditional

file systems. The only difference is that BHVFS has to handle one more level (*i.e.*, views emulated by directories in our experiments) to reach files.

More importantly, the read rate of BHVFS (1460) is six times better than that of FS-LFS (223), and the write rate of BHVFS (2640) is 30% worse than that of FS-LFS (3420). The read rate of BHVFS is 1% worse than that of FS-FFS (1470); whereas the write rate is 16% better than that of FS-FFS (2280). After a comprehensive comparison, we conclude that BHVFS is better than both FS-LFS and FS-FFS; the benefits obtained by BHVFS outweigh its overhead.

3.5 Discussions

CBFS - the context-based file system - represents a broad research topic that cannot be fully covered in this document. We make an effort to raise a number of intriguing issues that are important for future studies. In this subsection, we first present two extensions that implement context-based file systems in kernel and distributed environments. Then, we compare several approaches to improving the performance of view-unaware applications. Last, we discuss context exposure methods.

3.5.1 Kernel Implementation

When it comes to kernel implementation, we must address two issues, namely, view allocation and block allocation. Currently, BAVFS and BHVFS are built in the user space, where the view allocation issue is not a concern. Directories are used to emulate views for a simple design. In the Frog framework, when a directory is created, its sub-views are automatically created. The observation indicates that a directory allocation is always followed by a fixed number of view allocations. Thus, bundling directory and view allocations is possibly more efficient than individually allocating them.

Fig 3.16 displays two allocation methods by presenting VED and views layout on disks. In the separated layout, VED and views are located in two separate spaces in the metadata

region. VED and views allocations are independent with each other. This method simplifies the system design because it does not modify the VED allocation method. A new view allocation method is integrated into the system. A group layout indicates a group allocation method in which a directory and views are allocated at the same time. The views are implemented in form of the sub-views of the directory. The group allocation method can avoid multiple allocations that may trigger multiple disk I/O operations.

Block allocation is another critical issue in a file system design. I/O operations can be optimized with a better data locality provided by block allocation methods. Block allocation might not be a big issue in a multiple-disk environment (e.g., BHVFS in our experiment), because each view has its own disk to run on. Existing block allocations can be applied in views. However, block allocation becomes complicated when two or more views run on a single disk. Applications may switch among views, which may incur disk I/Os at positions that are far away. In addition, committing an update among views may issue disk I/Os at multiple positions.

Now we discuss a feasible design that reduces I/O costs induced by multiple views. The fundamental idea is that Frog uses a block-mapping mechanism to map virtual blocks in views to actual blocks on disk. Fig. 3.17 depicts two block layouts in which the mapping is efficient. The mapping process can be completed by simple numeric calculation. The upper figure shows that views have their own block space on disk. Frog maps the virtual blocks in views to their corresponding block space on disk. In this mapping, blocks are continuously stored on the disk, which offers good sequential reads in either view. As we mentioned

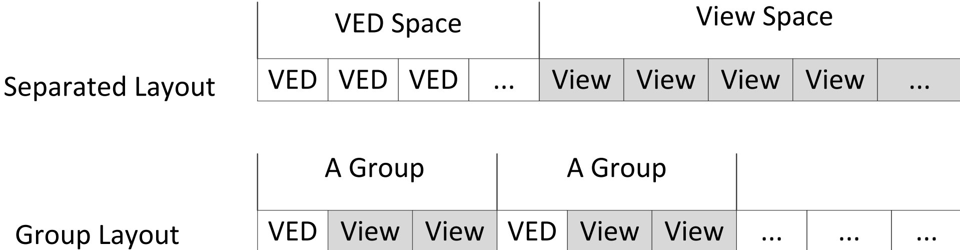


Figure 3.16: VED and views allocation.

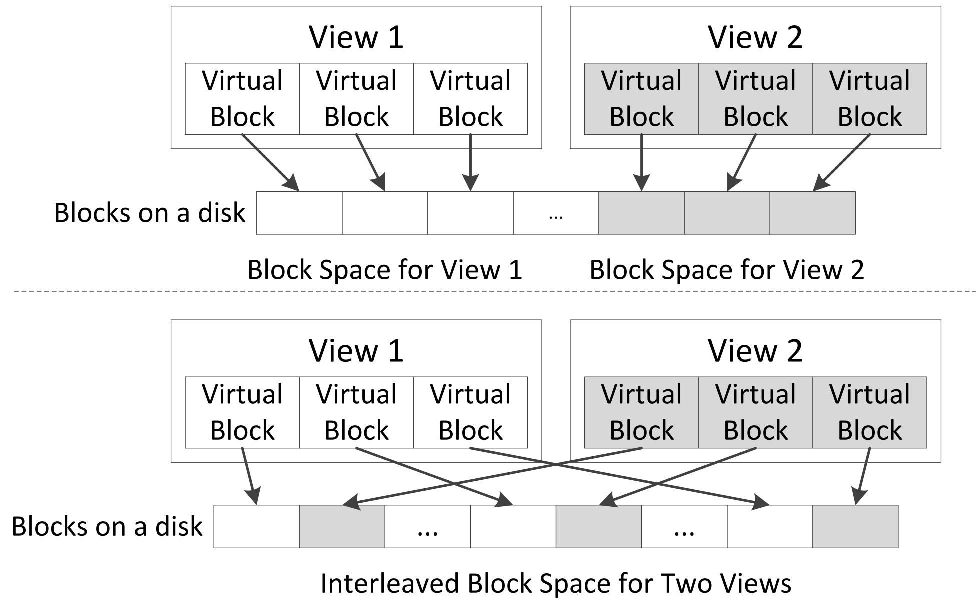


Figure 3.17: Two block mappings.

earlier, updates and frequent view switches are likely to issue I/Os among positions that are far away, thereby causing long disk seek times.

The lower figure shows an interleaved block mapping. This method is motivated by the observation that once an update is committed in a view, the update may be committed in other views in the near future. An interleaved layout increases the possibility that two updates may be completed on two continuous blocks on a disk, which avoids seek time in the second update. However, sequential reads in a view suffer from multiple seek overhead.

3.5.2 Work in a Distributed Environment

Distributed environments help in reducing the complexity of managing metadata/block allocations, where two or more views are running on a single disk. In a distributed environment where multiple computing resources and devices are coordinated, a context-based file system is likely to have each view located in a standalone disk. More importantly, data are usually duplicated in distributed environments for fault tolerance purpose (e.g., GFS [48] and HDFS [108]). All the issues related to replica maintenance have been investigated in the distributed systems. Adopting existing replica-management facilities can considerably

reduce the difficulty of the CBFS design. Thus, we conclude that integrating Frog into a replica-enable distributed file system is attractive.

It is worth noting that after the above integration, generating replicas in CBFS may be sightly different from that in distributed file systems. Rather than byte-to-byte duplications, each copy of replicas is independently managed by views. The content and format of a copy in a view may be different from the ones in other views. If the quality of the copies is not identical, the fault tolerance of entire system will be compromised. For example, a bi-context file system has two copies of a video file. A copy in the high resolution view is used for providing high quality services to VIPs, whereas another copy in the low resolution view is for regular customers. If the low quality copy is lost, the copy can be recovered from its high quality copy. However, the lost high-quality file cannot be recovered from the low quality copy. The information that does not exist in the low quality file will be lost permanently. To address this problem, we can create the low quality copy along with a complement file that stores the difference between high- and low- quality copies. In doing so, the high quality copy can be recovered from the low quality copy. The generation methods of complement files vary from application to application; these methods are out of the scope of our study.

3.5.3 Optimization for View-unaware Applications

Recall that CBFS provides compatibility to view-unaware applications by setting up a default view. All I/O requests issued by view-unaware applications are committed on the default view. Due to the unawareness of views, the view-unaware applications do not select any appropriate view. As a result, it is not guaranteed that the default view is the best view tailored for the applications, meaning that the view-unaware applications may not perform very well as view-aware applications in CBFS.

To improve the performance of view-unaware applications, we propose the following approaches to decide the most appropriate views on the behalf of the view-unaware applications. These approaches are categorized into the following two camps (i.e., static view selection and dynamic view selection).

Static View Selection. In this group of view-selection approaches, the decisions of view selections are statically made. The best view of a particular type of requests is explored during the development of CBFS; an appropriate view can be selected according to the request type. For example, file I/O requests are classified into file creation, file deletion, reads and writes. If a request is issued on BHVFS, the creation/deletion and write requests should be handled by BHVFS-LFS; read requests should be processed by BHVFS-FFS.

Dynamic View Selection. In the process of dynamic view selection, the views are determined at run-time. A dynamic-view-selection mechanism runs at the back-end of CBFS to collect the run-time information regarding types of I/O operations. When a request is issued, a view can be chosen by the dynamic-view-selection mechanism based on the run-time data analysis. Compared with the static approaches, the dynamic methods are automatic and flexible on view selections at the cost of extra overheads caused by data collection and analysis. Both the static and dynamic approaches can be incorporated into CBFS to choose the most appropriate views for view-unaware applications.

From the perspective of I/O access pattern recognition, these dynamic-view-selection methods can be envisioned as experienced-based or informed-based approaches.

Experienced-based View Selection. In an experienced-based view selection mechanism, historical data on access patterns are collected and used for making view selections. The effectiveness of these methods relies heavily on the access-pattern similarity between future requests and previously processed requests. Although a diversity of view selection policies can be incorporated in a dynamic view-selection mechanism, such an approach introduces extra overhead of historical data collection and maintenance.

Informed View Selection. Differing from the experienced-based methods, informed-view-selection approaches rely on view-selection hints provided by applications. In case the source code of existing view-unaware applications cannot be modified, the informed-view-selection approaches become impractical in CBFS. It is worth of attention that giving view names in a file path can be considered as one of informed-view-selection methods adopted for view-aware applications. Although the static methods are not flexible in nature, their efficiency is very high because of low overhead. If flexibility is more important than efficiency, a dynamic experience-based approach should be adopted.

3.5.4 Context Exposure

We discuss the methods of context exposure and perception in context-based systems and applications. In BAVFS and BHVFS, view set-ups in file systems are known as a priori during application development. The applications do not really perceive the contexts supported by the systems. However, context perception ability enables applications to be more adaptable in diverse computing environments. In what follows, let us discuss two approaches to providing applications with context information in Frog.

The first approach is **Context Detection**, in which view-aware applications are able to detect the existence of particular views by issuing operations (e.g., stat) on the views. The systems return information of the views if they exist; otherwise, an error code (e.g., ENOENT) is returned. The view detection operations are invisible to view-unaware applications because they can not issue any operation on views. Therefore, both types of applications can perform well in this approach. A limitation occurring in this method is that view names are required to be known in order to form a string path to views. In addition, detecting the existence of a number of views may cause multiple I/O operations that downgrade system performance.

The second approach is **View Listing**, where views are listed through directory listing operations. In this approach, a new type value (e.g., DT_VIEW in d_type field of struct

dirent) is added for views. After retrieving all objects under a directory, applications can find views by comparing their type values. As we mentioned in Section 3.2.2, the method exposes views to both types of applications, thereby possibly making view-unaware applications work incorrectly. For example, an application counts the number of files under a directory by calculating the number of objects that are not directories. If the views are returned along with files in the directory-listing operation, the application will retrieve incorrect information. Therefore, extra efforts on view-unaware application correctness should be paid in the case of view listing.

Chapter 4

ORCA: An Offloading Framework for I/O-Intensive Applications on Clusters

This chapter presents an offloading framework, ORCA, to map I/O-intensive code to a cluster that consists of computing and storage nodes. To reduce data transmission among computing and storage nodes, our offloading framework partitions and schedules CPU-bound and I/O-bound modules to computing nodes and active storage nodes, respectively. From developers' perspective, ORCA helps them to deal with execution path control, offloading executable code, and data sharing over a network. Powered by the offloading programming interfaces, developers without any I/O offloading or network programming experience are allowed to write new I/O-intensive code running efficiently on clusters.

We implement the ORCA framework on a cluster to quantitatively evaluate the performance improvements offered by our approaches. We run five real-world applications on both homogeneous and heterogeneous computing environments. Experimental results show that ORCA significantly speeds up the performance of all the five tested applications. Moreover, the results also confirm that ORCA considerably reduces network burden imposed by I/O-intensive applications.

This chapter is organized as follows. Section 4.1 illustrates the ORCA framework. Section 4.2 demonstrates four important design issues. Section 4.3 describes the implementation details of the framework. Sections 4.4 and 4.5 show our experimental results. Our experience on this study is discussed in section 4.6.

4.1 The ORCA I/O-Offloading Framework

We will begin this subsection by highlighting the main idea of our ORCA offloading framework for I/O-intensive applications. Then, we will discuss structures of applications designed to gain maximum benefit from the I/O offloading framework.

4.1.1 System Architecture

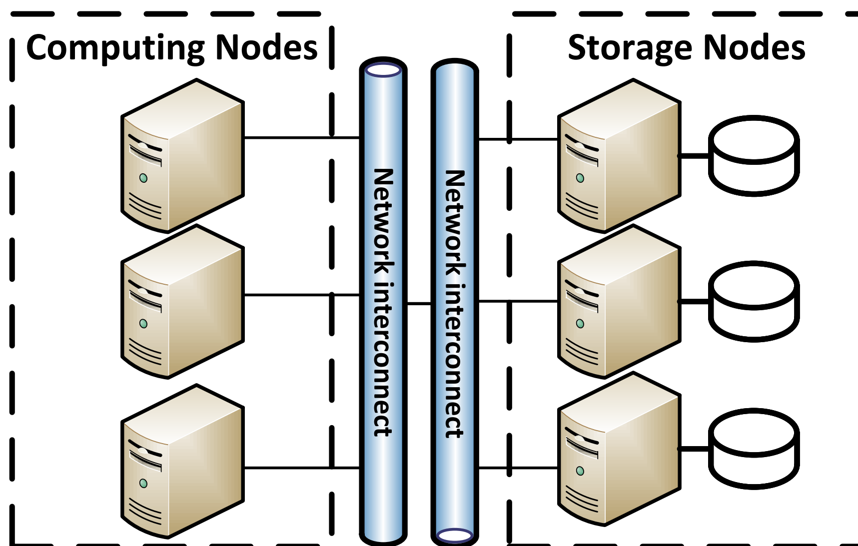


Figure 4.1: The architecture of commodity clusters, where a number of nodes are connected with each other through interconnects. We focus on clusters enhanced with active storage nodes that have computing capability.

Fig. 4.1 illustrates the architecture of commodity clusters, where a number of nodes are connected with each other through interconnects. In this work, we focus on clusters enhanced with active storage nodes that have computing capability. In our study, a cluster has two types of nodes: (1) computing nodes that deal with CPU-bound jobs and (2) storage nodes that are responsible for storing data and processing I/O-bound jobs.

In existing clusters, parallel file systems (see, for example, Lustre [105] and PVFS [84]) are employed to distributed data across multiple storage nodes. To support data-intensive applications, the parallel file systems need to transfer files back and forth between computing

and storage nodes. Although high peak aggregate I/O bandwidth can be achieved by accessing multiple storage nodes in parallel, moving data between computing and storage nodes will inevitably slow down the performance of I/O-intensive applications. Our preliminary evidence shows that reducing the amount of data transferred among nodes is a practical approach to boosting the overall performance of clusters.

4.1.2 Structure of Applications in ORCA

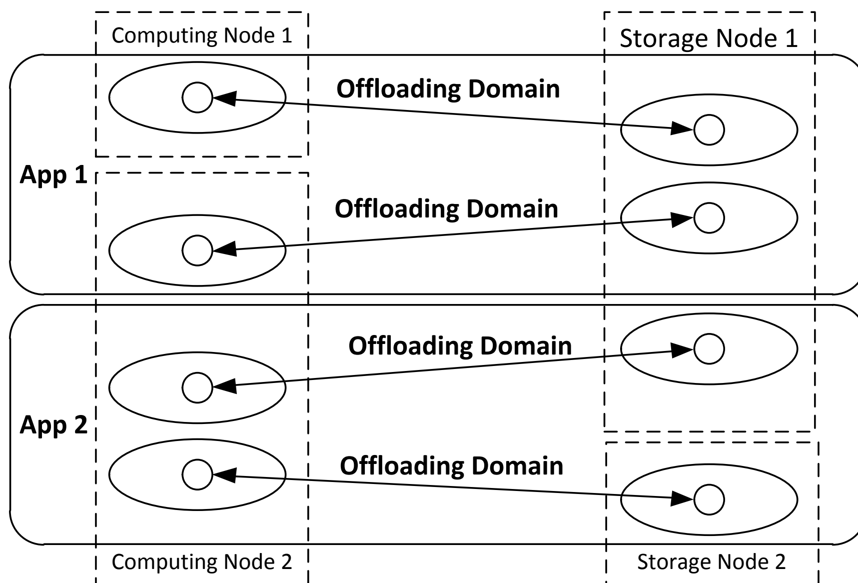


Figure 4.2: An offloading domain is a logic processing unit, in which a pair of computing and offloading modules are coordinated. I/O-bound modules are assigned to and executed on storage nodes; CPU-bound modules are handled by computing nodes. ORCA overlaps the executions of CPU-bound and I/O-bound modules.

Fig. 4.2 depicts our I/O offloading framework, in which I/O-bound modules are assigned to and executed on storage nodes. The goal of this framework is to reduce the amount of data transferred from storage nodes to computing nodes. Our offloading idea is inspired by the observation that I/O-intensive applications (see Section 4.4 for real-world examples) can be partitioned into CPU-bound and I/O-bound modules. CPU-bound modules are handled by computing nodes; whereas I/O-bound modules, running on storage nodes, are referred to as offloading parts. To achieve high performance, the framework makes an effort to overlap

the executions of CPU-bound and I/O-bound modules on computing and storage nodes in a cluster.

We now introduce the concept of offloading domains, which are used to group CPU-bound and I/O-bound modules. An offloading domain is a logic processing unit, in which a pair of computing and offloading modules are coordinated. An application may contain either only one or multiple offloading domains. The number of the offloading domains in an application heavily depends on the application's design and the number of offloading modules. Offloading domains are independent of, and isolated from each other in the sense that one offloading domain can not be interfered with by others.

Moreover, two simple applications that create two offloading domains are demonstrated in fig. 4.2. *App 1* is a multi-process program, in which the CPU-bound modules in both offloading domains are allocated on two computing nodes. The corresponding I/O-bound modules are executed on *storage node 1*. This is a typical n-to-1 model that multiple computing nodes and a storage node are used by *App 1*. On the other hand, *App 2* shows a typical 1-to-n model that a computing node and multiple storage nodes are utilized. The CPU-bound modules of *App 2* can be created as either threads or processes. The I/O-bound modules are offloaded to separated storage nodes. More complex applications can be derived from the two simple examples.

CPU-bound and I/O-bound modules in an offloading domain are, in some cases, serially and synchronously executed. Thus, while CPU-bound modules are running on computing nodes, their I/O-bound counterparts must be waiting and vice versa. In the case where CPU-bound and I/O-bound modules in an offloading domain are asynchronous, our framework can overlap the executions of the CPU-bound and I/O-bound modules on computing and storage nodes to achieve high performance in a cluster.

4.2 Design Issues

Before developing the proposed ORCA I/O-offloading framework, we need to address the following four design issues.

- How to identify I/O-bound modules in an application? (see Sec. 4.2.1)
- How to offload an I/O-bound module to an active storage node? (see Sec. 4.2.2)
- How to transfer an execution to a storage node? (see Sec. 4.2.3)
- How to share data between CPU-bound and I/O-bound modules within an offloading domain? (see Sec. 4.2.4)

4.2.1 Data-Intensive Module Identification

The first step in partitioning a data-intensive application is to identify the I/O-bound modules of the application. Intuitively, I/O-bound modules need to process huge amount of data, meaning that I/O time should dominate the performance of such modules. On the other hand, CPU-bound modules spend the majority of their time using CPUs to do calculations. A profiling and performance analysis tool can be employed to evaluate whether modules in a data-intensive application are CPU-bound or I/O-bound. With the performance analysis tool in place, programmers can evaluate whether applying the offloading technique improves overall application performance. Such an evaluation process should take into account various aspects such as computing workload, I/O workload, and network traffic.

4.2.2 Offloading a Program

The second design issue is that of an efficient way of offloading an executable file to an active storage node. Two practical approaches to offloading executable modules are dynamic offloading and static offloading. The main idea of dynamic offloading is to automatically transfer an executable file and its configurations to storage nodes in a cluster before loading

the file into the memory. In this method, the offloading platform must be aware of details of the run-time system implementation (e.g., programming languages and libraries) if the run-time system is platform dependent. If the run-time system is platform independent(e.g., implemented in scripts or java), the offloading platform does not have to consider run-time system details. Thus, the level of difficulty in implementing the offloading technique using dynamic distributions highly relies on the nature of the applications to be supported by the framework.

Dynamic offloading introduces another challenge - version management - for platform-dependent applications. In heterogeneous environments, all types of executable files, each of which is dedicated to a specific hardware platform, need to be precompiled. To invoke I/O-bound modules offloaded to storage nodes, applications must detect the type of hardware/software in the storage nodes and choose a proper version of the I/O-bound module to be offloaded on the fly. Moreover, this dynamic-distribution approach suffers from repeatedly transferring I/O-bound modules from computing to storage nodes. Although storage nodes are able to cache and reuse offloaded modules, it is time consuming for computing nodes to decide whether the cached ones on the storage end are valid and updated.

Unlike dynamic offloading, static offloading configures offloaded I/O modules a priori. Static offloading encompasses three distinct procedures if active storage systems are heterogeneous in nature. The first procedure is to manually compile I/O-bound modules for various hardware and run-time systems in heterogeneous storage systems. The second procedure is to write specific configuration files. The last procedure is to deploy the configuration files along with I/O-bound modules onto target storage nodes. Although these three procedures are seemingly complicated, they can be automatically completed by a simple yet efficient tool in our offloading framework. Moreover, the static offloading approach greatly simplifies the design of our offloading framework, because there is no need to address the platform-dependent issues. In this approach, when an application starts processing, its offloaded I/O-bound modules have already been compiled and installed on storage nodes.

4.2.3 Controlling an Execution Path

The third design issue is a mechanism for transferring executions back and forth between a pair of CPU-bound and I/O-bound modules in an offloading domain. To deal with this issue, we considered and compared two candidate mechanisms - CORBA and RPC.

CORBA [14] - a distributed programming model - is able to accommodate a number of components implemented by different languages. These components usually execute on different machines and communicate with each other through networks. However, CORBA's extreme complexity often prevents beginner programmers from learning and using it. It normally requires at least several months for programmers to become familiar with its fundamentals [54]. Another downside of adopting CORBA in our offloading solution is that storage nodes must be equipped with powerful processors in order to host a complex CORBA implementation. Using CORBA in our framework is likely to lead to very expensive storage nodes with high-performance CPUs. Otherwise, weak processors in storage nodes will constantly be busy running CORBA middleware.

A feasible option for our framework is Remote Procedure Call (RPC), which is a broadly accepted method of invoking a function to execute in a remote machine. Thanks to RPC's simplicity, it is easy for any programmer to learn and use. There are many RPC libraries implemented by various general-purpose programming languages. RPC was applied to implement Network File System (NFS) [103], MapReduce [33] and Hadoop [13]. Because of this, we choose RPC rather than CORBA to implement in our offloading framework.

4.2.4 Data Sharing among Storage and Computing Nodes

The last issue is data sharing between a pair of CPU-bound and I/O-bound modules in an offloading domain. Shared data include both global variables and code segments. A major challenge is that in an offloading domain, global variables can not be shared by CPU-bound and I/O-bound modules allocated to different computing and storage nodes.

An intuitive solution for the above challenge is to establish a synchronization mechanism to allow a pair of modules in an offloading domain to notify each other when any global variable is updated. For example, if a CPU-bound module modifies shared data on a computing node, a notification along with the updated data will be delivered to the corresponding I/O-bound module on a storage node.

A second solution is motivated by an observation that in some cases, I/O-bound modules are synchronized with their CPU-bound modules. In a synchronization process, offloaded I/O-bound modules are unable to access global data on storage nodes until control is regained from CPU-bound modules on computing nodes. Thus, CPU-bound modules can transfer updated shared variables to I/O-bound modules by appending the shared variables with offloading requests. In our approach, the framework updates global variables before processing offloading requests. In other words, the changes that occur at offloaded modules can be treated as results in response messages.

Code segments are considered to be a special type of global data. In applications implemented by compiled languages, function objects can not be shared directly. The reason for this is that addresses of an function in a pair of CPU-bound and I/O-bound modules may be different after being loaded into the main memory. On the other hand, in interpreted applications, functions are parsed by names rather than addresses. Hence, both the CPU-bound and I/O-bound modules in an offloading domain are able to obtain identical functions by their names.

In this subsection, we only highlighted the basic idea of data sharing supported in our offloading framework. Please refer to Section 4.3.4 for implementation details on the data-sharing mechanism .

4.3 Implementation Details

In this subsection, we describe the implementation details of our ORCA offloading framework and explain how to run offloading applications on clusters.

4.3.1 Configuration

Recall that we took the static offloading approach (see Section 4.2.2) by adopting the pre-configuration method to offload I/O-bound modules to storage nodes. The following five steps are required to run a data-intensive application in our ORCA I/O-offloading framework.

1. Design a data-intensive application and identify I/O-bound modules to be offloaded to storage nodes.
2. Convert the application into its offloading version by using the offloading programming interface (API) described in Section 4.3.3. Developers may need to write configuration files.
3. Create executable files for target storage nodes if the executables are implemented by compiled languages. If the application is developed by interpreted languages, then source files are executable.
4. Copy executable and configuration files to specified directories on computing and storage nodes.
5. Start I/O-bound modules on storage nodes followed by computing nodes. This order is important because offloaded modules must provide services to CPU-bound modules in an initial phase.

4.3.2 Workflow of an Application in ORCA

Normally, offloaded I/O-bound modules (see the right-hand side of Fig. 4.3) can be dispatched to multiple storage nodes. Storage nodes to which I/O-bound modules are offloaded depend on an allocation policy. For example, a typical policy is to allocate offloaded modules to the storage nodes where data is located [33] [13]. Another possible policy is to equally distribute offloaded modules across storage nodes. After an offloaded module completes,

it returns execution control back to the corresponding CPU-bound module of a computing node.

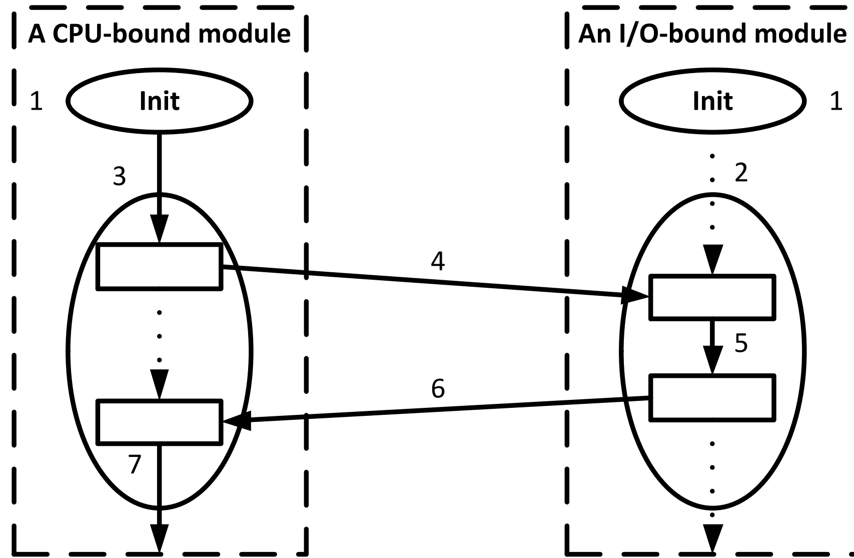


Figure 4.3: The execution flow of a data-intensive application running in the ORCA offloading framework.

Fig. 4.3 shows a workflow of an application running in our ORCA offloading framework. For simplicity, we only demonstrate the application with a single offloading invocation. The offloading framework manages to control the execution of the application in the following seven steps:

1. Both CPU-bound and I/O-bound modules are initialized.

2. Offloaded I/O-bound modules are suspended and wait for offloading requests issued from the computing node.

3. The CPU-bound module starts its execution on the computing node.

4. When an offloading invocation is required, the CPU-bound module sends a request to the I/O-bound module. The request includes the network address of a target storage node, the name of the offloading entry, and input parameters. Network addresses of storage nodes can be listed in a configure file so that the storage nodes can be

quickly accessed. Names of offloading entries can be hard-coded in the application, just like calling a function. All input parameters are transformed to a data stream to be transferred through the network.

5. After receiving an offloading request, the I/O-bound module is activated.

6. The I/O-bound module sends a response back to the CPU-bound module. The response contains the computation node's network address and results. The network address can also be obtained from the configuration file.

7. After receiving a response from the storage node, the computing node continues its processing.

4.3.3 Offloading APIs

The current version of the offloading framework provides an application programming interface (API) for C and C++ languages. Similar APIs can be implemented in other languages like java or python. Our ORCA offloading framework provides four API sets summarized in Table 4.1.

The `init` function in the first group initializes and sets up the offloading environments. Programs must execute `init` before issuing any offloading requests. First, `init` decides the role – a CPU-bound or I/O-bound module – that the program plays by identifying a dedicated command-line argument. After the role decision, `init` removes the dedicated argument which cannot longer be accessed. Then, a serial of `MARSHAL` and `UNMARSHAL` functions for primitive data types (*e.g.*, `char` and `unsigned short`) are registered for supporting primitive types serialization.

The second set of function in Table 4.1 is to register offloading entries. In C/C++ applications, offloading entries are addresses of functions in offloaded I/O-bound modules. After compilation, all functions are converted into addresses; an identical function may

Table 4.1 The ORCA Offloading Programming Interface

Interface & Description
`void init ()`   Initialize the system.

`void register_function (func_addr)`
 build a map from function addresses to their names.
 `func_name find_name_by_func_addr (func_addr)`
 Get a function name by a given address.
 `func_addr find_func_by_name (func_name)`
 Get a function address by a given name.

`void MARSHAL (void* obj, char**buf, int* len)`
 Serialize an object pointed by obj into a data stream. The address and size of the data stream are specified by buf and len.
 `void UNMARSHAL (void* obj, char*buf, int len)`
 Un-serialize an object pointed by obj from a data stream. The address and size of the data stream are specified by buf and len.

`void offload_call (addr, func_name, ins, outs)`   Invoke an offloading procedure named by func_name. addr indicates a network address (*e.g.*, an IP address) of the target node. The input parameters and results are specified by ins and outs.

have different addresses in CPU-bound and I/O-bound modules. In order to exchange offloading entries between a pair of CPU-bound and I/O-bound modules, we enable applications to call `register_function` to register functions and then exchange function names instead of addresses. Addresses are automatically converted to names in CPU-bound modules and reverse in offloaded I/O-bound modules by calling `find_name_by_func_addr` and `find_func_by_name` respectively.

The goal of the third API set in Table 4.1 is to send and receive parameters and results. Both `MARSHAL` and `UNMARSHAL` accept input parameters *object* in the type of *void ** in order to adapt all types of objects. The following two parameters specify the buffer of the data stream and its length. All data being exchanged between CPU-bound and I/O-bound modules must implement corresponding `MARSHAL` and `UNMARSHAL` functions that are automatically called by the offloading framework. If a function pointer need to be serialized or un-serialized, the

pointer has to be converted to the function name by a second set of interfaces and then processed as a regular string. These functions must be registered during initialization as well.

Definition

```
struct offloading_para
{
   void *          obj;
   MARSHAL         marshal;
   UNMARSHAL       unmarshal;
};
```

A simple example

```
void domain_entry(String files[])
{
   for (int index = 0; index < 2; index++)
   {
      unsigned int result;
      struct offloading_para ins =
            {files[index], marshal_string, unmarshal_string};
      struct offloading_para outs =
            {&result, marshal_uint, unmarshal_uint};
      offload_call("192.168.0.1", word_counter, ins, outs);
   }
}
```

Figure 4.4: A simple example of offload_call

`offload_call` is a real action for calling an offload. The parameter `addr` indicates the network address (*e.g.*, an IP address) of the node where an offloading part will take place. `func_name` specifies an offloading entrance. `ins` and `outs` are an input and output parameters defined as instances of the `offloading_para` structure.

Fig. 4.4 gives the definition for and an example of `offloading_para` recording an object and corresponding `MARSHAL` and `UNMARSHAL` functions. When `offload_call` is invoked, the offloading library will automatically serialize and un-serialize input and output parameters so that the computation and offloading parts are able to successfully communicate with each other.

4.3.4 Sharing Data

Recall that the complexity of offloading programs heavily depends on data sharing mechanisms (see Section 4.2.4). Because our goal is to keep offloading programs simple, our

framework offers a simple yet efficient way of passing data as input and output parameters. We consider two key issues regarding data sharing.

The first one is how to share global data between computing and storage nodes. All data accessed by both nodes should be overseen by input parameters and results (see Section 4.2.4), and is required to be deeply copied in `MARSHAL` and `UNMARSHAL` instead of merely copying object points. This is because objects created in address spaces are totally different in the two parts. A function pointer is the data that maintains the address of the function. The address has to be converted to the function name in `MARSHAL` and recovered in `UNMARSHAL`, since the function name keeps consistent in both CPU-bound and I/O-bound modules. The conversion can be completed by Dynamically Loaded (DL) libraries[1] if the function is defined as *extern*. The CPU-bound and I/O bound modules are responsible for handling global updates.

The second issue is how to share code segments. Function entries or executable objects are a special type of data in programs. The framework can not simply copy binary codes and transfer them to another node, because the code might be not executable. In our implementation, we link all object codes to each part, regardless of whether the codes are used or not; therefore, programmers do not need to identify which functions belong to either parts or both. To transfer a function entry, we build a map between function names and addresses, thereby placing function names in offloading requests and responses. Both computing and active storage nodes can resolve function names and addresses by using the offloading API.

4.4 Evaluations

4.4.1 Experimental Testbed for ORCA

We set up a homogeneous cluster and a heterogeneous cluster as two testbeds to evaluate real-world applications supported by our ORCA offloading framework. Both clusters are

[1] http://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html

comprised of 16 nodes, which form 8 independent offloading domains (see Fig. 4.2 for an example of offloading domains). All nodes are connected by the Gigabit Ethernet.

Table 6.3 summarizes the hardware and software configurations of the two types of nodes - Type I and Type II - used in our testbeds. Type I nodes have better CPU performance and larger main memory than Type II nodes. Interestingly, measurements collected by hdparm [4] indicate that Type II nodes have higher sequential I/O throughput (130.35 MBytes/Sec.) than Type I nodes (106.94 MBytes/Sec.).

Table 4.2 Hardware and Software Configurations

Name	Hardware	Software
Type I	1 × Intel Xeon 2.4 GHz processor 1 × 2 GBytes of RAM 1 × 1 GigaBit Ethernet network card 1 × Seagate 160 GBytes Sata disk (ST3160318AS [7])	Ubuntu 10.04 Linux kernel 2.6.23
Type II	1 × Intel Celeron 2.2 GHz processor 1 × 1 GBytes of RAM 1 × 1 GigaBit Ethernet network card 1 × WD 500 GBytes Sata disk (WD5000AAKS [12])	Ubuntu 10.04 Linux kernel 2.6.23

The homogeneous cluster is made up of 16 Type I nodes; the heterogeneous cluster contains 8 Type I nodes and 8 Type II nodes. In the second testbed, computing nodes are Type I and storage nodes are Type II. The configuration details of the testbeds are specified in Table 6.3.

Table 4.3 Configuration of the two Testbeds for ORCA

	Computing Nodes	Storage Nodes
Homogeneous testbed	8 × Type I	8 × Type I
Heterogeneous testbed	8 × Type I	8 × Type II

4.4.2 Benchmark Applications

Applications

We tested five benchmarks (see Table 4.4), which are well-known data-intensive applications. PostgreSQL, Word Count(WC), Sort, and Grep were downloaded from their official websites, whereas the Inverted Index application was implemented by our research group at Auburn. In our experiments, we ran the baseline applications on computing nodes and loaded data from the storage nodes through the Network File System (NFS) service [103].

NFS is an RPC-based solution commonly used in clusters. Due to the efficiency and ease of management of NSF, numerous commercial products rely on it to manage massive amount of data. For example, Oracle 11g - the latest DBMS product developed by Oracle corporation - is supported on Oracle Real Application Clusters (RAC) that provides shared storage through the NFS service [5]. Kassick *et al.* studied the impact of I/O coordination on an NFS-based environment [61]. So, we decided to test all the baseline applications using the NFS service in our testbeds.

We choose PostgreSQL as the first benchmark, because offloading I/O-bound modules of a database engine to storage nodes has been proposed by a number of researchers. This approach aims to accelerate I/O processing in database systems. For example, Pitman *et al.* studied a scheme to offload relational operations in DB2 to Active Storage Fabrics (ASF), thereby increasing parallel capability of data retrieval [41]. Choudhary *et al.* selected DBMS as a major benchmark application [29], in which five operations (i.e., scan, join, sort, group-by and aggregate) are offloaded to smart disks, to evaluate the performance impact of distributed smart disks on I/O-intensive workloads. Abouzeid *et al.* developed HadoopDB that combines MapReduce and DBMS technologies by considering a combination of efficiency, scalability, fault tolerance and flexibility of databases [15]. HadoopDB offloads parts of the database workload to storage servers as Mapper tasks in a hybrid system. The

major difference between our offloading framework and the above studies lies in the fact that ours offloads a database engine to storage nodes in shared nothing clusters.

We applied the ORCA offloading framework to the five benchmark applications, each of which has an I/O-bound module running on storage nodes. In particular, the "executor" is defined as an offloaded module in PostgreSQL. The ORCA framework offloads the I/O-bound modules of the other applications to storage nodes. Table 4.4 describes the implementation of these benchmarks.

Table 4.4 Real-World Benchmark Applications

Applications	Descriptions
PostgreSQL 9.0 [6]	It is a relational database management system. The offloading framework offloads the "executor" module to storage nodes. The I/O-bound module receives an execution plan and performs queries. The CPU-bound module manages connections to clients, converts SQL statements to execution plans and sends results back to clients.
Word Count in GNU coreutils 7.4 [2]	It counts the number of words in a set of files. Our framework partitions the Word Count application into an I/O-bound module that calculates word occurrences in one file, and a CPU-bound module that sums the occurrences up.
Sort in GNU coreutils 7.4 [2]	It sorts lines of a text file in alphabetical order. Our framework treats the entire Sort application as an offloaded I/O-bound module that receives a file name and stores sorted text in a file.
GNU Grep 2.7 [3]	It searches through a file for lines which contains a given keyword. The I/O-bound module in Grep finds desired lines in a file; the CPU-bound module in Grep transfers keywords and file names to the I/O-bound module.
Inverted Index (our benchmark)	It loads a set of files and builds a map between words to their occurrences. In the Inverted Index application, its I/O-bound modules constructs a map for each file; a CPU-bound module transfers file names to the I/O-bound module.

Data Preparation

To measure performance of PostgreSQL running in the ORCA offloading framework, we created four databases with sizes of 5 GBytes, 10 GBytes, 15 GBytes and 20 GBytes. No

index was generated in these databases; therefore, PostgreSQL had to directly access data in the tables rather than merely checking index structures during query processing. Each database is made up of 1,000 tables, each of which has 100 integer attributes. Tuples are equally distributed across these tables, so a larger database has more tuples in each table. We generated 1,000 queries, each of which scans only one table. Together, these queries cover all the tables in the database.

For the other four benchmark applications, we created five text files of relatively smaller sizes (i.e., 400 MBytes, 600 MBytes, 800 MBytes, 2 GBytes and 4 GBytes). Each text file contained a number of randomly generated words. Due to the limitation of the main memory, we tested the inverted index application using the first three text files on the homogeneous cluster. This was because frequent page faults made I/O noise in the experiments when the input file size was larger than the main memory. We also tested the other four applications on the heterogeneous cluster.

4.4.3 PostgreSQL: A case study

We briefly described how official and offloading PostgreSQL worked in our experiments. We chose PostgreSQL as an example, because it is a complicated application with a number of independent modules. Also, boundaries of I/O-intensive modules are highly distinguishable. We can easily partition PostgreSQL into computation and offloading parts.

Compare ORCA with ASF, SDs, and HadoopDB

Offloading I/O-bound modules of a database engine to storage nodes is an idea proposed by other researchers. This approach can be applied to further accelerate I/O processing in database systems. Pitman *et al.*, for example, studied a scheme that offloads relational operations in DB2 to *Active Storage Fabrics* (ASF) in order to increase performance of parallel data retrievals [41]. To evaluate the performance improvement of distributed *Smart Disks* (SDs) for I/O-intensive workloads, Choudhary *et al.* chose DBMS as a major benchmark

application, in which five operations (i.e., scan, join, sort, group-by and aggregate) are offloaded to SDs [29]. In a study conducted by Abouzeid *et al.*, parts of database workload are offloaded to the storage as Mapper tasks in a hybrid system called HadoopDB, which combines MapReduce and DBMS technologies for a combination of efficiency, scalability, fault tolerance and flexibility of databases [15]. The major difference between our ORCA and the aforementioned studies is that ORCA offloads a database engine on shared nothing clusters.

Official PostgreSQL

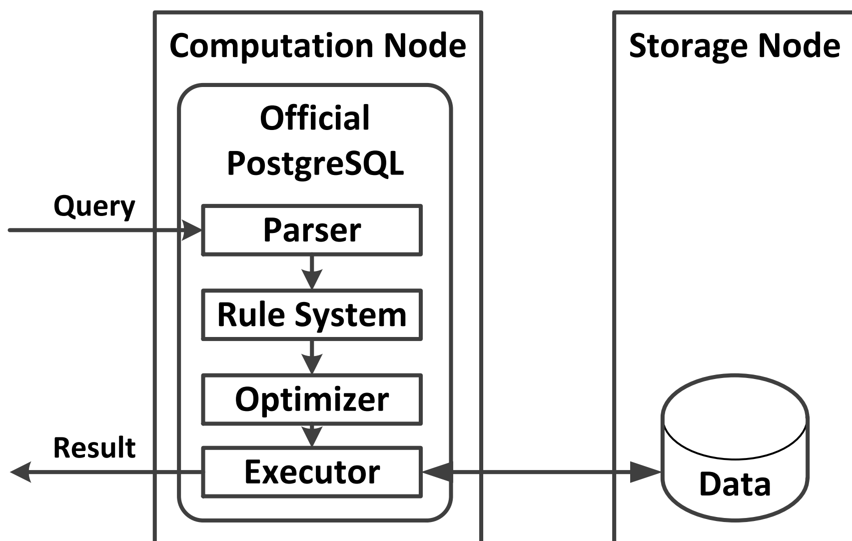


Figure 4.5: The execution flow of official PostgreSQL

PostgreSQL is an open source relational database management system. We chose the latest stable release, PostgreSQL 9.0, as a target application in these experiments.

Fig. 4.5 shows that there are four components in the PostgreSQL backend program, which mainly supports SQL queries in the background. Parser checks a query string for valid syntax and creates a parse tree after the validation process. The rule system applies a group of rules to rewrite the parse tree to an execution plan. The optimizer creates an optimal execution plan; the executor runs the query [49].

Offloading PostgreSQL in ORCA

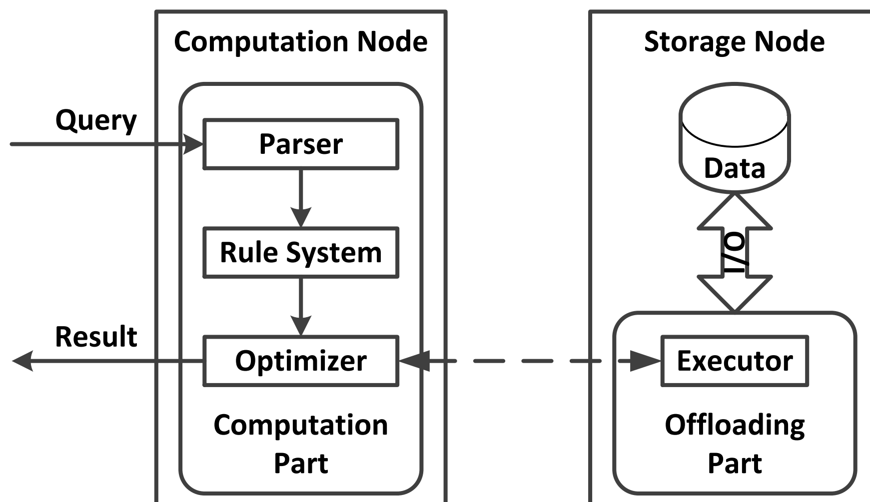


Figure 4.6: The execution flow of offloading PostgreSQL in ORCA. The computing node handles the parser, rule system, and optimizer; the executor is offloaded to the storage node.

In the query procedure, the executor is an I/O-intensive program that reads and/or writes a large amount of data from and/or to storage systems while processing expensive operations (e.g., scanning or joining tables). Fig. 4.6 illustrates the execution flow of offloading PostgreSQL in ORCA, which offloads the executor to storage nodes. ORCA does not modify other modules, such as access methods and disk space managers related to storage systems, because an offloading PostgreSQL can access the same data files. In the ORCA-based PostgreSQL, the executor receives an execution plan from the remote optimizer and transfers results back to the backend program.

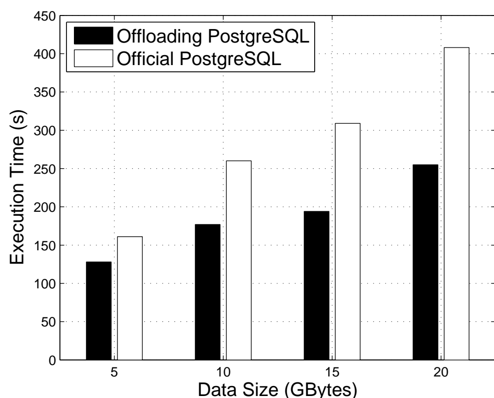
4.5 Experimental Results

4.5.1 Overall Performance Evaluation

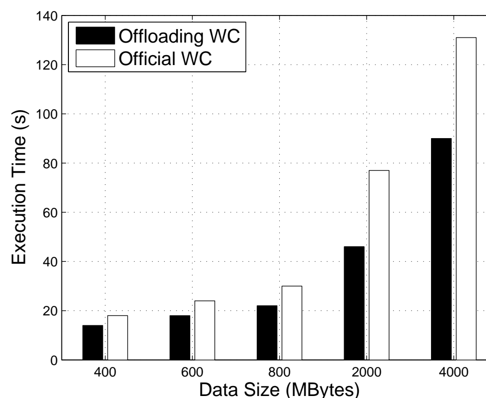
Homogeneous Clusters

Fig. 4.7 illustrates the execution times of the five applications (see Table 4.4) to compare the ORCA-enabled cluster against the same cluster without I/O offloading. The results plotted in Fig. 4.7 show that the ORCA offloading framework significantly reduces the execution

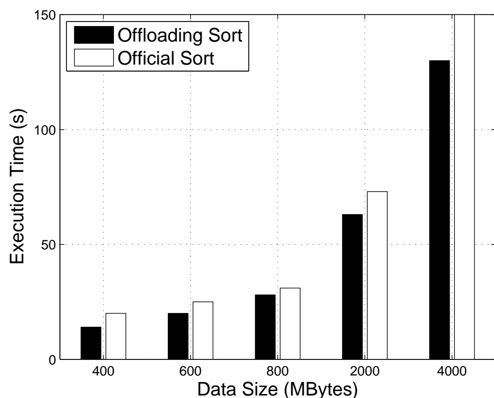
times of all five tested applications. For example, when data size is 4 GBytes, our scheme can reduce execution time of PostgreSQL, and Grep by 37.8% and 47.4%, respectively.



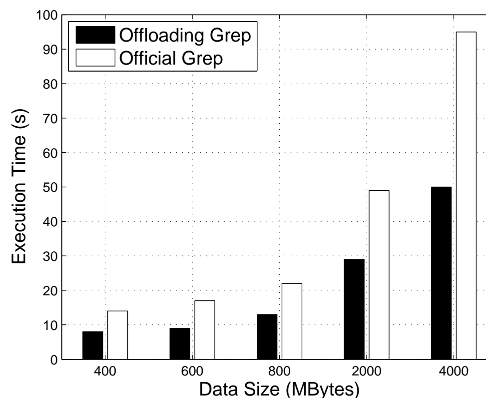
(a) PostgreSQL



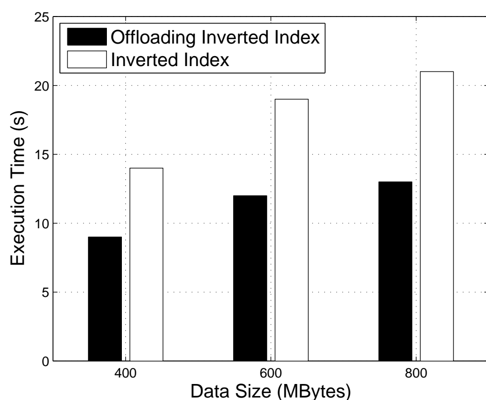
(b) Word Count



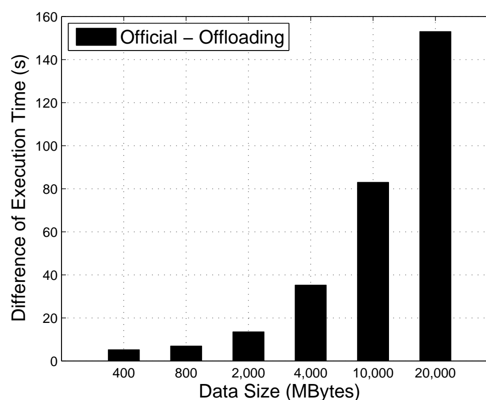
(c) Sort



(d) Grep



(e) Inverted Index



(f) Performance Improvement vs. Data Size

Figure 4.7: ORCA-based applications vs. Official applications. Execution times of the five real-world benchmark applications running on the homogeneous cluster (i.e., the first testbed).

The applications without the ORCA support are slowed down by remotely accessing huge amount of data, because the data must be transferred from storage nodes to computing nodes. Our framework solves this performance problem by offloading I/O-bound modules to storage nodes, thereby substantially reducing I/O time through local data accesses. Although the applications running in ORCA have to exchange input/output parameters among computing and storage nodes, the data size of input/output parameters is significantly smaller than the dataset size.

Fig. 4.7(f) shows the impact of data size on performance improvement gained by our ORCA offloading framework. We measured the performance of the four applications (i.e., WC, Sort, Grep and Inverted Index) on four datasets (400MB, 800MB, 2GB, and 4GB) and PostgreSQL on two large datasets (10GB and 20GB). We plotted performance improvement in terms of execution-time reduction in Fig. 4.7(f). This reveals that the performance improvements achieved by ORCA become more pronounced as the datasets grow in size. When data size is small, the non-offloading-enabled applications can take advantage of continuous I/O operations optimized by the NFS service. For example, NFS can cache entire datasets in the main memory so that the datasets can be repeatedly processed without further remote I/O accesses. Unfortunately, when the datasets grow in size, the non-ORCA-enabled applications benefit very little from caching due to limited caching ability in computing nodes.

Heterogeneous Clusters

Fig. 4.8 shows execution times of the five benchmark applications supported by the ORCA framework on a heterogeneous cluster, in which computing nodes and storage nodes have different performance. The results plotted in Fig. 4.8 are consistent with those shown in Fig. 4.7. In other words, we observed that ORCA reduces the execution times of the benchmarks. Moreover, the performance improvements are more distictive as the data size increases (see fig. 4.8(f)).

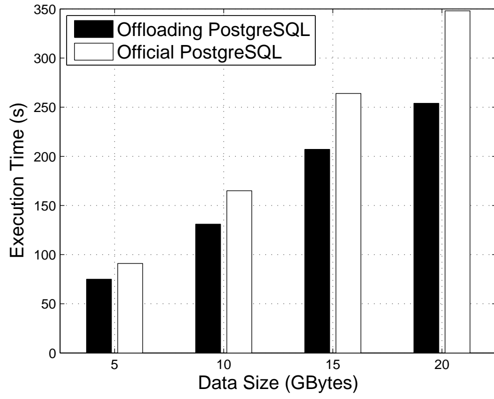
Comparing Figs. 4.7 and 4.8, we concluded that the heterogeneous cluster offers slightly better performance than the homogeneous one. This is because the storage nodes (i.e., Type II nodes) in the heterogeneous cluster have higher I/O bandwidth than the storage nodes (i.e., Type I nodes) in the homogeneous cluster. Although Type I nodes are superior to Type II nodes in terms of CPU speed and memory capacity, higher I/O throughput of Type II nodes cause the heterogeneous cluster to outperform its homogeneous counterpart.

4.5.2 Network Load Evaluation

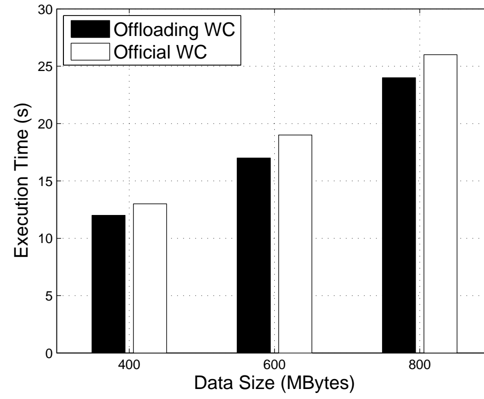
Homogeneous Clusters

Fig. 4.9 shows network load caused by both official and ORCA-based PostgreSQL when data size is set to 5GB, 10GB, 15GB, and 20GB, respectively. The results confirm that the ORCA offloading framework significantly reduces the network load of the homogeneous cluster running PostgreSQL. When the non-ORCA-based PostgreSQL is running, transferring data from the storage to computing nodes keeps the network resources very busy (e.g., from 30MB/s to 60MB/s). The performance bottleneck of non-ORCA-based applications is dominated by network resources. This problem becomes even worse as data size grows.

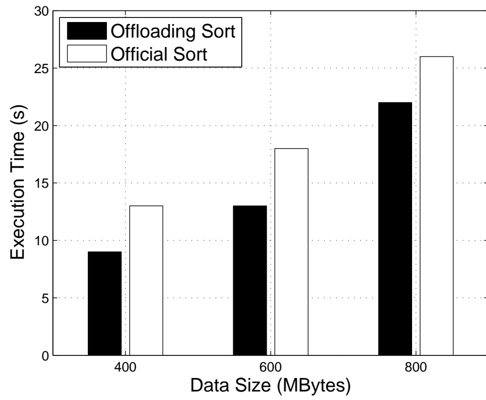
Fig. 4.10 shows network load imposed by the other four applications and their ORCA-based counterparts processing an 800MB dataset. WC, Grep, and Inverted Index share a similar network traffic pattern with PostgreSQL. Fig. 4.10(b) shows that the data transmission rate in Sort is constantly changing between 0 and 65MB/s. The reason is that after loading a certain amount of data, Sort becomes CPU-bound rather than I/O-bound to handle the sorting process. During the short period of sorting process, the network resource is sitting idle and waiting for the next I/O request.



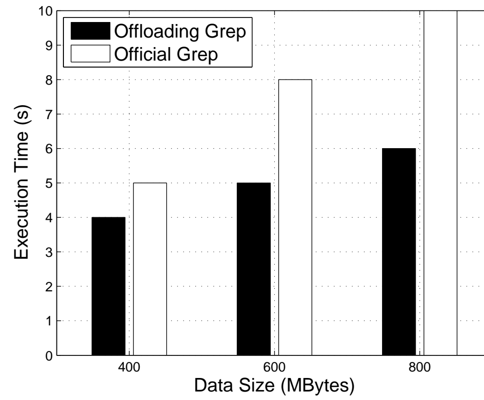
(a) PostgreSQL



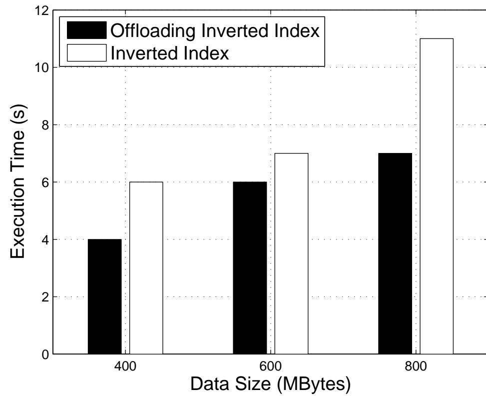
(b) Word Count



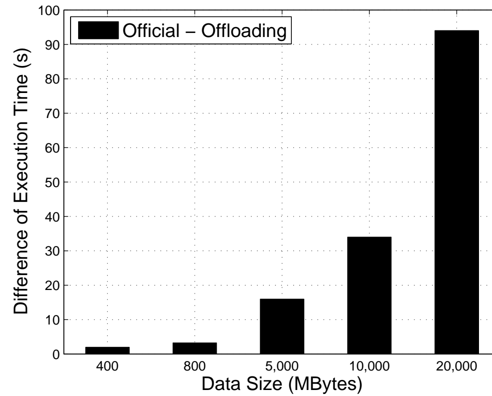
(c) Sort



(d) Grep

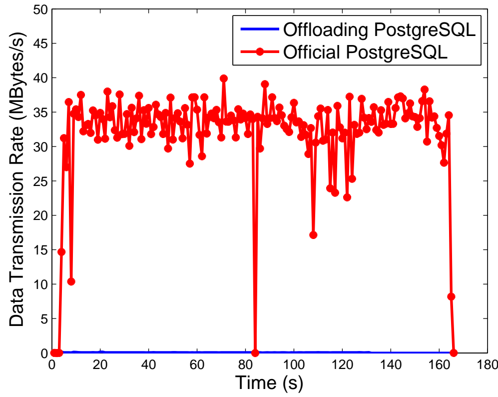


(e) Inverted Index

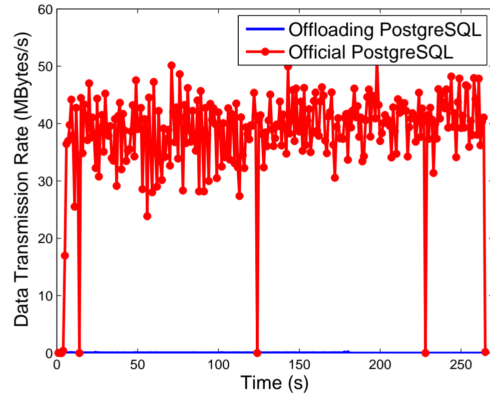


(f) Performance Improvement vs. Data Size

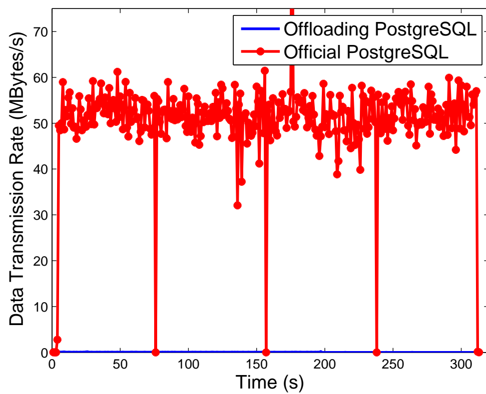
Figure 4.8: ORCA-based applications vs. Official applications. Execution times of the five real-world benchmark applications running on the heterogeneous cluster (i.e., the second testbed).



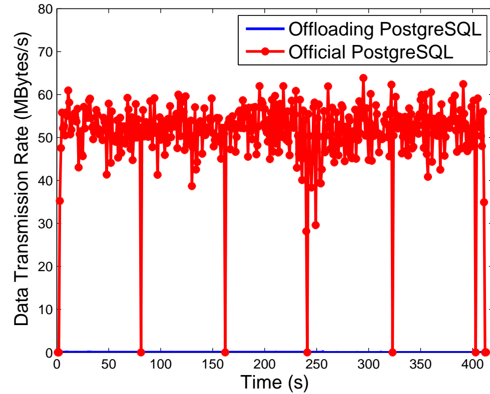
(a) 5 GBytes



(b) 10 GBytes



(c) 15 GBytes



(d) 20 GBytes

Figure 4.9: Network load imposed by both official and ORCA-based PostgreSQL accessing different databases on the homogeneous cluster (i.e., the first testbed).

Heterogeneous Clusters

Figs. 4.11 and 4.12 show the network traffic patterns of the five applications and their ORCA-based counterparts running on the heterogeneous cluster. The empirical results indicate that regardless of the type of a cluster, the non-ORCA-based applications keep network resources very busy in transferring data between computing and storage nodes.

We observe that the data transmission rate (ranging from 50MB/s to 70MB/s) of the non-ORCA-based PostgreSQL on the heterogeneous cluster is constantly higher than that of the homogeneous cluster. This observation implies that for non-ORCA-based applications,

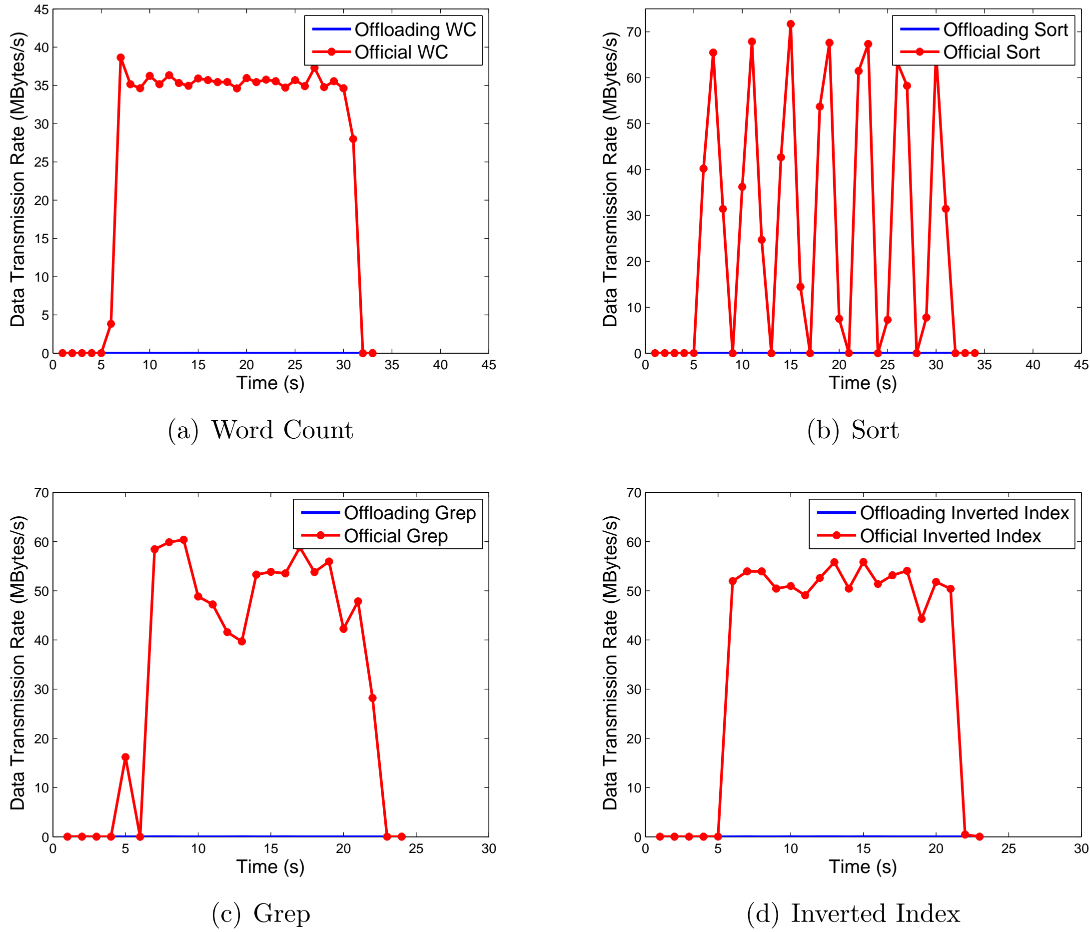
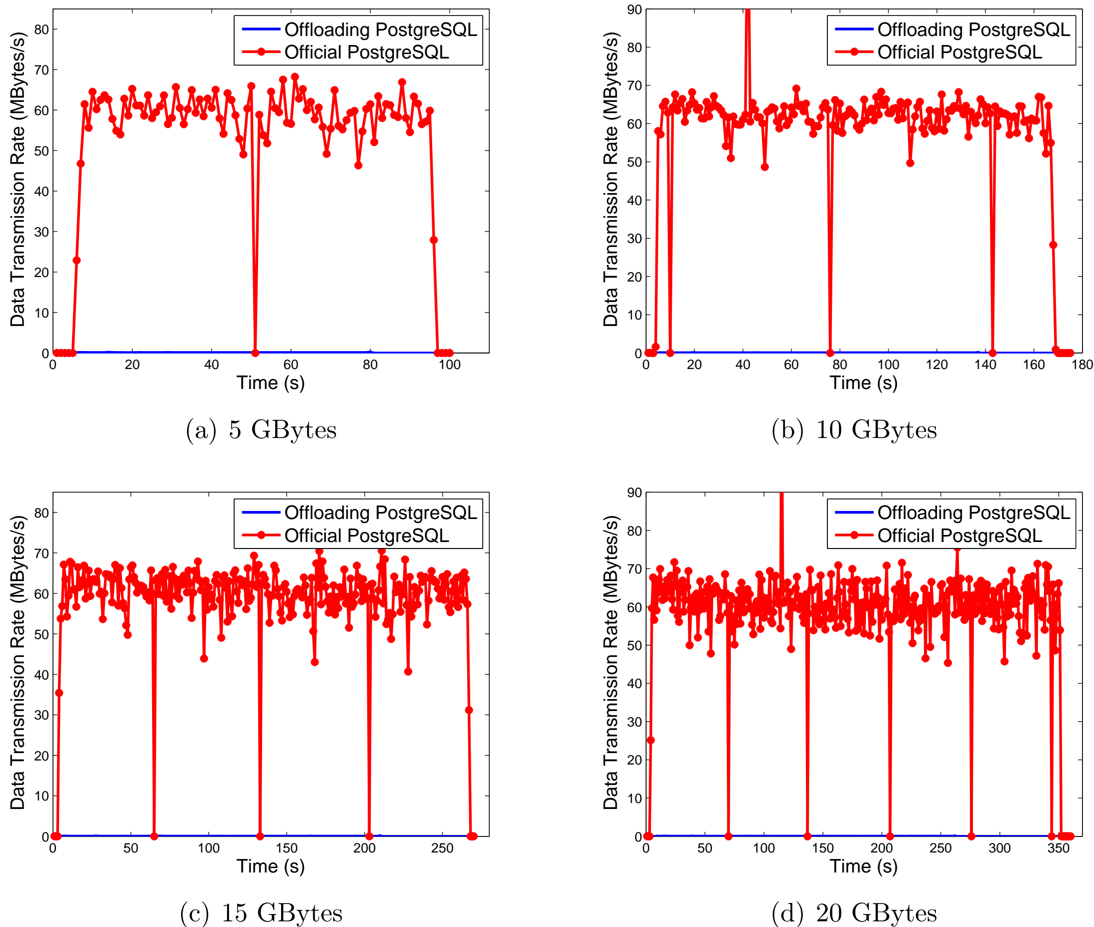


Figure 4.10: Network load imposed by the four real-world applications and their ORCA-based counterparts accessing 800 MB datasets on the homogeneous cluster (i.e., the first testbed).

improving I/O bandwidth of storage nodes can reduce the applications' execution times by increasing network utilization.

In addition, the network links of the homogeneous and heterogeneous clusters are not saturated, because the data transmission rates are below the maximum network bandwidth (i.e., 1 Gbps)in both cases. Data retrieved from the storage nodes can be immediately delivered to the computing nodes; therefore, accessing data in the heterogeneous cluster is faster than in its homogeneous counterpart.



(a) 5 GBytes (b) 10 GBytes

(c) 15 GBytes (d) 20 GBytes

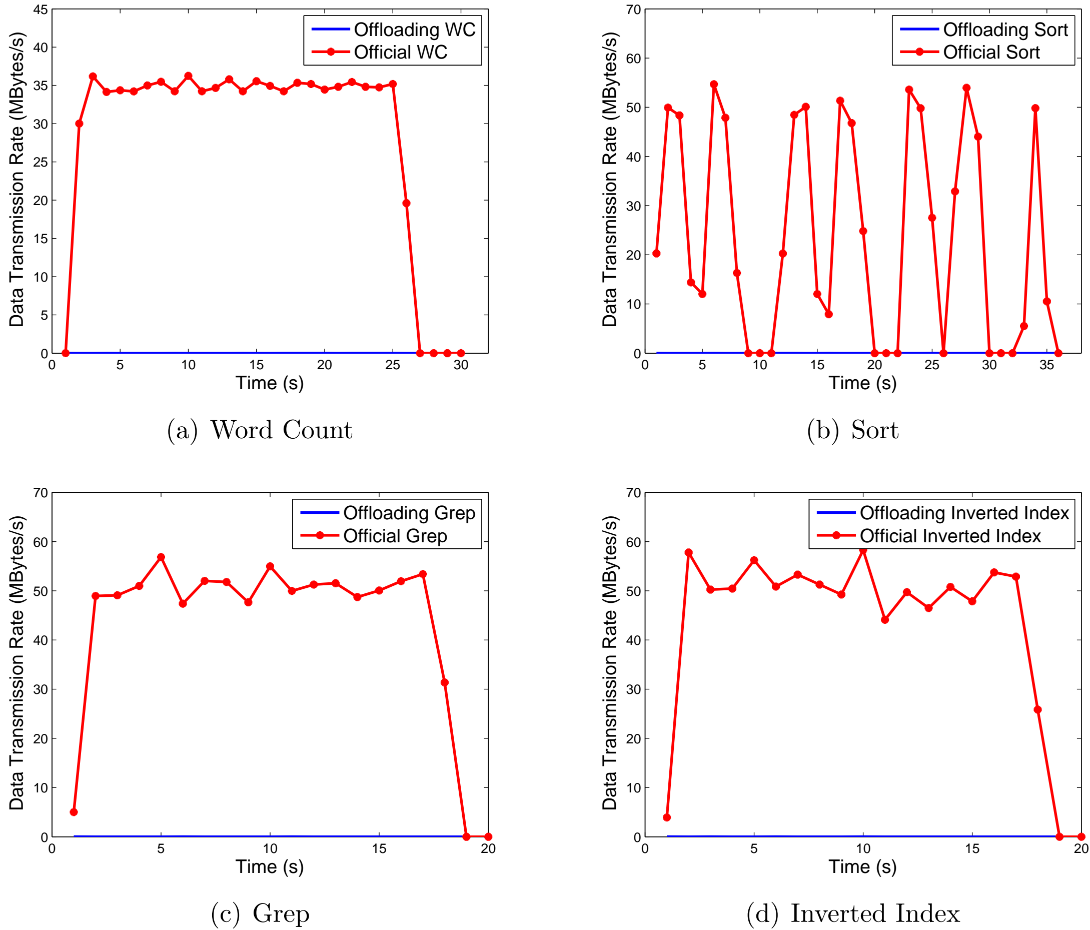
Figure 4.11: Network load imposed by both official and ORCA-based PostgreSQL accessing different databases on the heterogeneous cluster (i.e., the second testbed).

4.5.3 CPU Usage Evaluation

Homogeneous Clusters

The goal of this set of experiments was to assess the performance impact of offloaded I/O-bound modules on storage nodes in ORCA. This goal was achieved by evaluating CPU usage of storage nodes in the homogeneous cluster running the five data-intensive applications. Evaluating CPU usage of storage nodes is very important, because offloaded I/O-bound modules may have side effect on other I/O services running on the storage nodes.

Fig. 4.13 illustrates CPU utilization of PostgreSQL processing a 10 GB dataset and the other applications processing an 800 MB dataset. We observe that the CPU usage of ORCA,



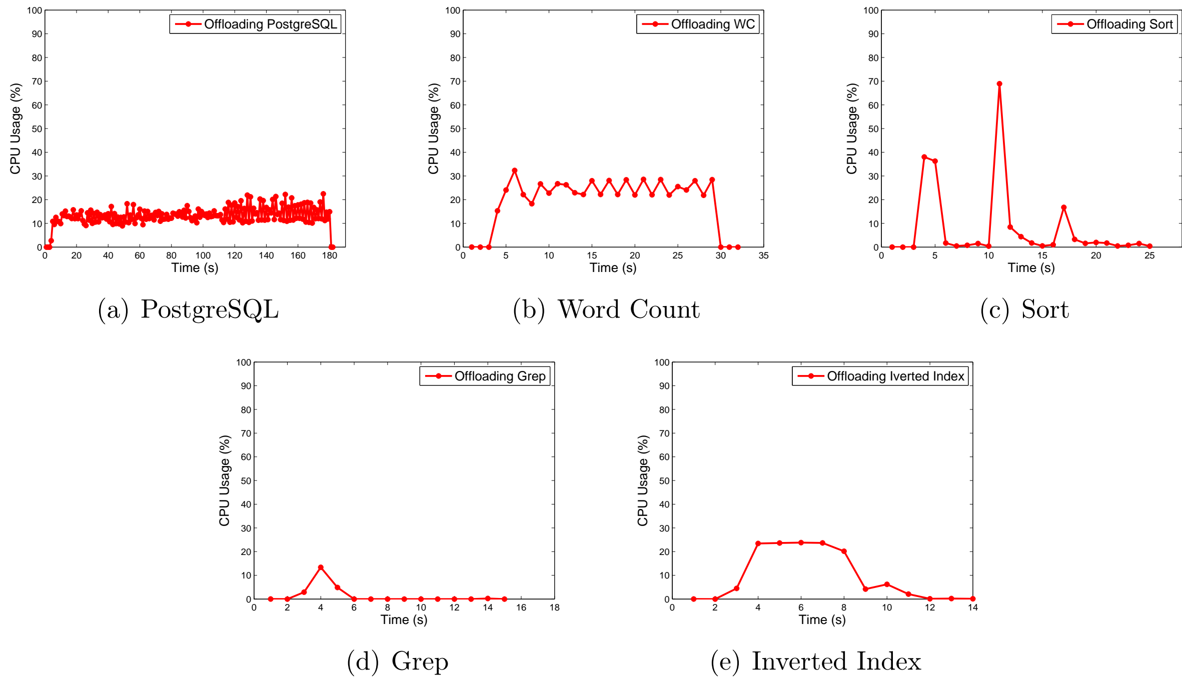
(a) Word Count (b) Sort

(c) Grep (d) Inverted Index

Figure 4.12: Network load imposed by the four real-world applications and their ORCA-based counterparts accessing the 800 MB datasets on the heterogeneous cluster (i.e., the second testbed).

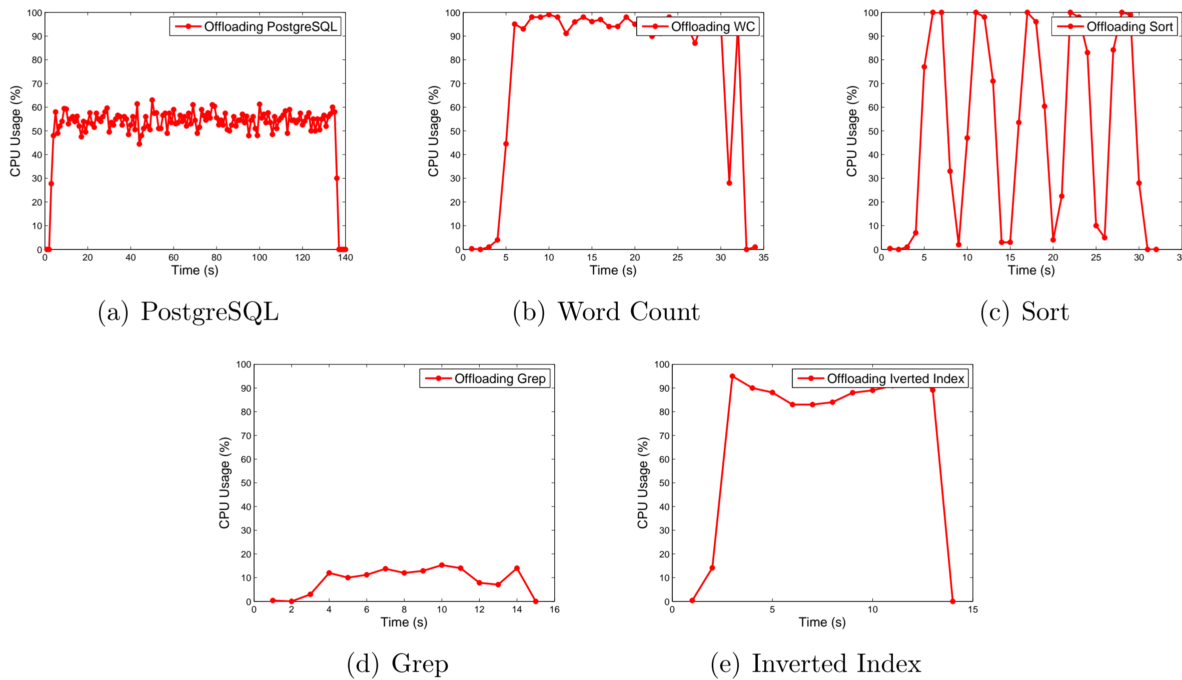
in most cases, is below 30% although there were two cases where the CPU utilization reaches 40% and 70% for a few seconds (see Fig. 4.13(c)). These two cases have little negative impact on storage nodes. Of all the five tested applications, Grep (see Fig. 4.13(d)) had the least overall impact on other services running on storage nodes. Overall, we concluded that our ORCA I/O-offloading framework has minimal negative impact on any services running on storage nodes in homogeneous clusters.

We confirm that improving performance of data-intensive applications in ORCA comes at the cost of increasing CPU usage in storage nodes. Fig. 4.13 indicates that different offloaded I/O-bound modules lead to different CPU-usage increases in storage nodes. An



(a) PostgreSQL (b) Word Count (c) Sort (d) Grep (e) Inverted Index

Figure 4.13: CPU load imposed by the five real-world ORCA-based applications in the storage nodes of the homogeneous cluster (i.e., the first testbed).



(a) PostgreSQL (b) Word Count (c) Sort (d) Grep (e) Inverted Index

Figure 4.14: CPU load imposed by the five real-world ORCA-based applications in the storage nodes of the heterogeneous cluster (i.e., the second testbed).

increase in CPU utilization of storage nodes heavily relies on the nature of the I/O-bound modules. For example, offloaded modules can vary from a simple problem solver (e.g., calculating the number of word occurrences in a file) to a complicated procedure (e.g., searching qualified tuples by scanning an entire table). Ideally, ORCA provides a good tradeoff between reducing execution time of a data-intensive application and increasing CPU usage on storage nodes.

Heterogeneous Clusters

Now we are in a position to evaluate ORCA's CPU usage of storage nodes in heterogeneous clusters. Fig. 4.14 shows CPU usages of storage nodes running the offloaded modules for the five benchmark applications. The results suggest that WC and Inverted Index give rise to a high CPU usage (i.e., $>$90%). The Sort application repeatedly pushes the CPU usage up to 100% and then drops down to nearly 0%. PostgreSQL and Grep keep CPU usage at a moderate level (i.e., 50%-60%) and a low level (i.e., $<$18%), respectively. If storage nodes in ORCA have low-performance CPUs or offloaded modules are CPU-intensive, then the offloaded modules can cause high CPU utilization in the storage nodes.

4.6 Experience

In this study, we successfully extend PostgreSQL, Word Count, Sort, Grep, and Inverted Index using the ORCA APIs (see Sec. 4.3.3) in our real-world shared-nothing clusters. We have learned a great deal about the design and implementation of the ORCA offloading framework. A summary of our experience and lessons accumulated during the development of ORCA and ORCA-based applications.

4.6.1 Offloading Module Identification

When we prepared to develop the first ORCA-based offloading application (see ORCA-based PostgreSQL in Sec. 4.4.3), we encountered a challenging issue: which modules should

be offloaded. In particular, developers have to figure out which modules process large volumes of data. After the initial analysis of PostgreSQL, we treated the entire DB engine as an offloading module.

The second challenge is to minimize the computational complexity of offloaded modules, because the offloaded tasks in ORCA are executed on storage nodes. After examining the DB engine, we discovered that only the executor accesses a large amount of data and; therefore, the executor was chosen as an offloaded module.

The third issue we faced is to decouple the offloaded module with other modules. The decoupling process should simplify both offloading interfaces and data dependency. Based on our further analysis of PostgreSQL, we determined that the execution tree is the most appropriate interface defined in the offloaded module. The execution tree is a good interface because of the twofold reason. First, this module contains a single data structure. Second, the execution tree specifies a module boundary between the optimizer and executor modules. Let us discuss the issue of data dependency in the following subsection.

4.6.2 Data Sharing

Recall that offloaded modules located in storage nodes and others modules in the host computing nodes can not share in-memory data (see Section 4.2.4 and Section 4.3.4). In the implementation of the ORCA-based PostgreSQL, no in-memory data set is shared between modules residing in computing nodes and offloaded modules in storage nodes. The data of the execution tree located in both computing and storage nodes must be synchronized by the ORCA offloading framework. However, the metadata of databases, originally stored in files and loaded into main memory at in the initialization phase, has to be shared by among computing and storage nodes. For example, the optimizer uses catelog data to make optimization decisions; the executor requires metadata (e.g., the locations of tables) to carry out the execution plan. Considering that the size of metadata is very small, we have these files shared by the NFS services in the ORCA-based PostgreSQL.

The offloaded code in storage nodes and other code in computing nodes are linked together as an executable program in ORCA. The ORCA-based program is dispatched to both computing and storage nodes. ORCA offers an additional command-line argument to distinguish whether or not a program is an offloaded program. This process is handled by the `init` function.

Chapter 5

MOLQ: Multi-Criteria Optimal Location Query with Overlapping Voronoi Diagrams

Chapters 3 and 4 are focused on I/O techniques and file systems. Now, we are positioned to investigate spatial query applications, which are data-intensive in nature. In this chapter, we present a novel optimal location selection problem, Multi-Criteria Optimal Location Query (MOLQ), which can be applied to a wide range of applications. After providing a formal definition of the novel query type, we explore two intuitive approaches that sequentially scan all possible object combinations and locations in the search space. Then, we propose an Overlapping Voronoi Diagram (OVD) model that defines OVDs and Minimum OVDs, and construct an algebraic structure under an OVD overlap operation. Based on the OVD model, we design an advanced approach to answer the query. Due to the high complexity of Voronoi diagram overlap computation, we improve the overlap operation by replacing the real boundaries of Voronoi diagrams with their Minimum Bounding Rectangles (MBR). We also propose a cost-bound iterative approach that efficiently processes a large number of Fermat-Weber problems. Our experimental results show that the proposed algorithms can efficiently evaluate the novel query type.

This chapter is organized as follows. Section 5.1 formally defines our novel multi-criteria optimal location query. Two intuitive solutions are described in section 5.2. The OVD model is illustrated in section 5.3. Our advanced solution is proposed in section 5.4. The experimental results are shown in section 5.5.

5.1 Preliminaries

5.1.1 Definitions

A spatial object is defined by the triple $<l, w^t, w^o>$, where l is its location in the search space, and w^t and w^o are the type weight and object weight associated with the object. $\mathbb{E} = \{P_1, ..., P_n\}$ denotes a universal set of objects, where $P_i = \{p_i^1, ..., p_i^m\}$ denotes a set of objects of a particular type. $G = \{p_1^u, ..., p_n^v\}$, where $p_1^u \in P_1, ..., p_n^v \in P_n$, denotes an object group, the objects of which are different types. ς^t and ς^o are weight functions applied to *type weight* and *object weight*. Notations used in this chapter are summarized in Table I.

Table 5.1 Symbolic Notations.

Symbol	Meaning
P_i	An object set
G	An object group
p_i^u	A spatial object in P_i
w^t	Type weight
w^o	Object weight
ς^t	A type weight function
ς^o	An object weight function
$\|S\|$	The number of elements in the set S
ϵ	An error bound
η	A distance bound
γ	A stopping rule used in iterative approaches [121, 127]
$d(.,.)$	Euclidean distance between two objects
$\mathbb{E}$	A set of object sets or groups
$\mathbb{V}$	A set of Voronoi diagrams
$\mathbb{R}$	The search space
$VD(P_i)$	Voronoi diagram of P_i
$Dom(p_j)$	Dominance region of p_j in a Voronoi diagram
OVD	An overlapped Voronoi diagram
OVR	An overlapped Voronoi region
$MOVD$	A minimum overlapped Voronoi diagram

Weighted Distance of Two Points

Given a point q, a spatial object p, a type weight function ς^t, and an object weight function ς^o, weighted distance considers both the distance between two points $d(.,.)$ and the weights of p. The formal definition is as follows:

$$WD(q, p, \varsigma^t, \varsigma^o) = \varsigma^t(\ \varsigma^o(\ d(q,\ p.l),\ p.w^o\ ),\ p.w^t) \tag{5.1}$$

Weighted Distance from a Query Point to an Object Group

Given a point q, an object group $G = \{p_1^u, ..., p_n^v\}$, a type weight function ς^t, and object weight functions $\sigma = \{\varsigma_1^o, ..., \varsigma_n^o\}$, we define the weighted distance from q to G as the sum of WD(q, p_i^s, ς^t, ς_i^o), where $p_i^s \in G$, $\varsigma_i^o \in \sigma$. The formal definition is

$$WGD(q, G, \varsigma^t, \sigma) = \sum_{p_i^s \in G, \varsigma_i^o \in \sigma} WD(q, p_i^s, \varsigma^t, \varsigma_i^o) \tag{5.2}$$

Minimum Weighted Distance from a Query Point to Object Groups

Given a point q, a set of object sets $\mathbb{E} = \{P_1, ..., P_n\}$, a type weight function ς^t, and object weight functions $\sigma = \{\varsigma_1^o, ..., \varsigma_n^o\}$, we define the minimum weighted distance from q to object combinations of $\mathbb{E}$ as:

$$MWGD(q, \mathbb{E}, \varsigma^t, \sigma) = \min(\{WGD(q, G, \varsigma^t, \sigma) | G \in P_1 \times ... \times P_n\}) \tag{5.3}$$

Multi-criteria Optimal Location Query (MOLQ)

Given a set of object sets $\mathbb{E} = \{P_1, ..., P_n\}$, a type weight function ς^t, and object weight functions $\sigma = \{\varsigma_1^o, ..., \varsigma_n^o\}$ where ς_i^o is applied to an object $p_i^u \in P_i$, the query is to find an optimal location l in the search space $\mathbb{R}$ that minimizes $MWGD(l, \mathbb{E}, \varsigma^t, \sigma)$.

$$MOLQ(\mathbb{E}, \varsigma^t, \sigma) = l, \quad \text{where} \\ MWGD(l, \mathbb{E}, \varsigma^t, \sigma) = \min(\{MWGD(l', \mathbb{E}, \varsigma^t, \sigma) \mid l' \in \mathbb{R}\}) \tag{5.4}$$

5.1.2 Voronoi Diagram

Ordinary Voronoi Diagram

Given a set of objects $P_i = \{p_i^1, ..., p_i^m\}$, the ordinary Voronoi diagram $VD^O(P_i)$ is defined as a number of dominance regions $\{Dom^O(p_i^u) | p_i^u \in P_i\}$, each of which is dominated by an object p_i^u. All locations in $Dom^O(p_i^u)$ are closer to p_i^u than other objects.

$$Dom^O(p_i^u) = \{\ l \mid d(l, p_i^u.l) \ \leq \ d(l, p_i^v.l), u \neq v, p_i^u, p_i^v \in P_i\} \tag{5.5}$$

Weighted Voronoi Diagram

In a weighted Voronoi diagram, generators have different weights reflecting their variable properties. Given a set of objects $P_i = \{p_i^1, ..., p_i^m\}$ and a weight function ς, the dominance regions are measured by weighted distance.

$$VD^W(P_i) = \{Dom^W(p_i^u) \mid p_i^u \in P_i\} \quad where \\ Dom^W(p_i^u) = \{\ l \mid \varsigma(\ d(\ l,\ p_i^u.l),\ p_i^u.w^o\ )\ \leq \varsigma(\ d(\ l,\ p_i^v.l),\ p_i^v.w^o\ ), u \neq v,\ p_i^u, p_i^v \in P_i\} \tag{5.6}$$

5.1.3 Fermat-Weber Point

Given a point group $G = \{p_1^u, ..., p_n^v\}$ in a d-dimensional space $\mathbb{R}^d$, the Fermat-Weber point is the point q which minimizes the cost function [26].

$$c(q, G) = \sum_{p_i^s \in G} p_i^s.w^t \times d(q, p_i^s.l) \tag{5.7}$$

The point exists for any point set and is unique except for the event that all the points lie on a single line [52]. In the noncollinear case, the cost function is strictly convex [121].

The solution to the three-point Fermat-Weber problem had been proposed in [58]. In the collinear case of any point set, an optimal point can be found in linear time [26]; however, to the best of our knowledge, if the number of points is greater than three, no exact solution has been reported for non-collinear cases. Instead, an iterative approach is used as an approximate solution proposed in [121, 127]. This approach converges monotonically to the unique optimal location during iterations.

The iterative approach starts with an arbitrary location q_0 ($q_0 \notin G$) in $\mathbb{R}^d$. In each iteration, a new location $q_i = f(q_{i-1}, G)$ is produced based on a location q_{i-1} found before the iteration. According to the monotonic convergence property, q_i is closer to the Fermat-Weber point than q_{i-1}; hence, theoretically, the Fermat-Weber point is located at $\lim_{n\to\infty} f^n(q_0, G)$. The function f is described below.

$$f(q, G) = \begin{cases} \sum_{p_i^s \in G} \{g_i^s(q) \times p_i^s.l\} & \text{if } q \notin G \\ q & \text{Otherwise} \end{cases} \quad (5.8)$$

where

$$g_i^s(q) = \frac{p_i^s.w^t}{d(q, p_i^s.l)} \times \{\sum_{p_{i'}^{s'} \in G} \frac{p_{i'}^{s'}.w^t}{d(q, p_{i'}^{s'}.l)}\}^{-1} \quad (5.9)$$

Three existing stopping rules for the iterative method are widely adopted. Üster and Love developed a generalized bounding method, by which the result is limited within a specified rectangular distance to the optimal location [120]. Verkhovsky and Polyakov adopted the difference of the costs between two successive iterations as the stopping rule in their experiments [123]. Setting an acceptable deviation from the cost of the optimal location as the stopping rule is widely used in applications [97]. For example, given an error bound ϵ, the location after the nth iterations l^n, and the optimal location l^∞, the iteration procedure

will stop when $\frac{c(l^n,G)-c(l^\infty,G)}{c(l^\infty,G)} \leq \epsilon$, where $c(l^\infty, G)$ is approximated by a lower bound of the cost at l^n:

$$lb(l^n) = \sum_{k=1}^{d}(\min_x(\sum_{p_i^s \in G} p_i^s.w^t \frac{|l^n.x_k - p_i^s.l.x_k||x - p_i^s.l.x_k|}{d(l^n, p_i^s.l)})) \tag{5.10}$$

5.2 Basic Algorithms

5.2.1 Sequential Scan Object Combinations

One basic algorithm to solve MOLQ sequentially checks optimal locations of all object combinations. In particular, given $\mathbb{E} = \{P_1, ..., P_n\}$, the optimal locations l's of all combinations $\{p_1^u, ..., p_n^v\}$, where $p_1^u \in P_1, ..., p_n^v \in P_n$, are calculated by the Fermat-Weber method, which considers both object locations and their type weights. The answer to the query is the best location among these l's. We call this algorithm the *Sequential Scan Combinations* (SSC) algorithm.

Algorithm 2 SSC($\mathbb{E}$, ς^t, σ)

1. $Ubound = \infty$
2. $l = < 0, 0 >$
3. **for** $< p_1^u, ..., p_n^v > \in P_1 \times ... \times P_n$ **do**
4. Calculate the optimal location l_1 of $< p_1^u, p_2^s >$
5. **if** WGD(l_1, $\{p_1^u, p_2^s\}$, ς^t, σ) $< Ubound$ **then**
6. Calculate the optimal location l_2 of $< p_1^u, ..., p_n^v >$
7. $Cost$ = WGD(l_2, $\{p_1^u, ..., p_n^v\}$, ς^t, σ)
8. **if** $Cost < Ubound$ **then**
9. $Ubound = Cost$
10. $l = l_2$
11. **end if**
12. **end if**
13. **end for**
14. **return** l

Since the computation of SSC is expensive, we can set an upper bound to reduce the complexity of the algorithm by filtering out a portion of combinations whose optimal locations cannot be the answer. For an example, two combinations (object groups), G_1 and G_2,

will be evaluated sequentially in SSC. We assume the optimal location of G_1 is at l_1. The weighted distance from l_1 to G_1 is denoted by d_1. Before processing $G_2 =< p_1^u, ..., p_n^v >$, we first set d_1 as an upper bound and calculate the optimal location l_2 of $< p_1^u, p_2^s >$, which costs much less than computing an optimal location of multiple points. If the weighted distance from l_2 to $< p_1^u, p_2^s >$ is greater than d_1, the weighted distance from any location to G_2 must be greater than d_1, thus calculating the optimal location of G_2 can be avoided. During SSC processing, the upper bound is initialized to infinity and will be reduced to the total weighted distance of the best solution found so far. The detailed steps of SSC are described in Algorithm 2.

5.2.2 Sequential Scan Locations

Another intuitive algorithm is to convert an infinite search space to a finite number of locations by dividing the search space into a grid. We assume that an answer with a distance-bound η is acceptable for the query. We calculate the weighted distance from all the line intersections of the grid to their nearest objects of different types. The best location is selected as the answer to the query. We call this algorithm the *Sequential Scan Locations* (SSL) algorithm.

Algorithm 3 SSL($\mathbb{E}$, η, ς^t, σ)

1. $Ubound = \infty$
2. $l = < 0, 0 >$
3. **for** $p_1^u \in P_1$ **do**
4.   **for** l' in the $range(p_1^u, Ubound)$ **do**
5.   /* l' stands for the location of an intersection */
6. $Cost$ = WGD(l', the nearest objects of l', ς^t, σ)
7.   **if** $Cost < Ubound$ **then**
8. $Ubound = Cost$
9. $l = l'$
10.   **end if**
11.   **end for**
12. **end for**
13. **return** l

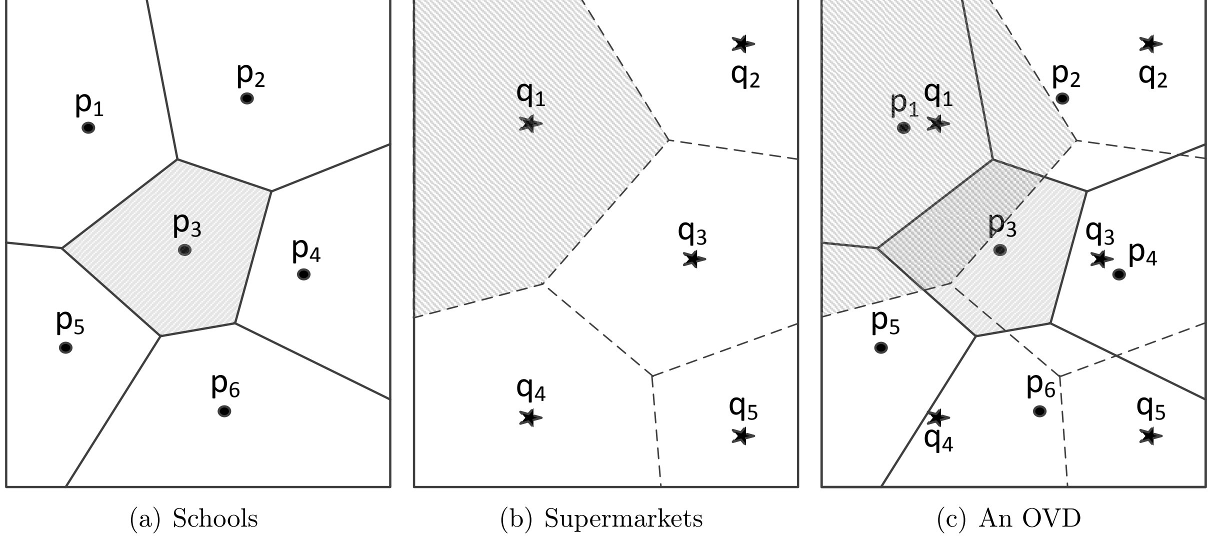
Since the number of intersections, depending on η, could be extremely large, we employ an upper bound to reduce the number of intersections checked in SSL. In particular, an upper bound is initialized to infinity and will be trimmed to the total weighted distance of the best solution found so far. Since Equation 5.2 is non-decreasing, given an object group $G = \{p_1^u, ..., p_n^v\}$, if the weighted distance between a location l and p_1^u is greater than the upper bound, the total weighted distance from l to G must be greater than the upper bound as well, and l cannot be the answer to the query. Based on this observation, SSL only scans the intersections within a range of p_1^u, which is limited by the upper bound. The steps of SSL are formalized in Algorithm 3. In the for loop in line 4, expanding l' from p_1^u to all directions is more efficient because the range may be reduced by finding a better location during the scan.

5.3 The OVD Model

Before describing our MOVD-based solution, we will first introduce the OVD model. In this subsection, we start with a simple OVD example which provides a basic understanding of the model. Then, we formally define OVD and Minimum OVD (MOVD) and systematically analyze their properties. We build an algebraic structure of MOVD on an overlap operation $+$.

5.3.1 An OVD Example

Fig. 5.1(a) and Fig. 5.1(b) display two ordinary Voronoi diagrams generated by schools and supermarkets, respectively. The shaded areas in the figures are dominance regions of generators p_3 and q_1. Fig. 5.1(c) shows an OVD that overlaps the two ordinary Voronoi diagrams. Apparently, the OVD is comprised of a number of overlapping regions, each of which is generated by overlapping two ordinary Voronoi polygons. For example, the doubly shaded area in Fig. 5.1(c) is the overlapping region in both shaded regions of two ordinary



(a) Schools (b) Supermarkets (c) An OVD

Figure 5.1: Ordinary Voronoi diagrams and OVDs.

Voronoi diagrams. According to the properties of Voronoi diagrams, p_3 and q_1 are the closest school and supermarket to any locations in the doubly shaded region.

5.3.2 Overlapped Voronoi Diagram Definition

Overlapped Voronoi Diagram (OVD)

Given a set of object sets $\mathbb{E} = \{P_1, ..., P_n\}$ and a set of Voronoi diagrams $\mathbb{V} = \{VD(P_i)|P_i \in \mathbb{E}\}$, where $VD(P_i)$ can be either an ordinary or a weighted Voronoi diagram generated by P_i in the search space $\mathbb{R}$. Overlapped Voronoi Diagram (OVD) is a set of Overlapped Voronoi Regions (OVR),

$$OVD(\mathbb{E}) = \{OVR_j \mid 1 \leq j \leq m\} \tag{5.11}$$

where OVR_j is

$$OVR(p_1^u, ..., p_n^v) = \{l|l \in Dom(p_1^u), ..., l \in Dom(p_n^v), p_1^u \in P_1, ..., p_n^v \in P_n\} \tag{5.12}$$

Property 1. *An OVD may have one or more empty set OVRs (e.g., $OVR_j = \emptyset$).*

Proof. By definition, an OVR is the intersection of dominance regions from different Voronoi diagrams. Potentially, these dominance regions may not overlap each other (see the dominance regions of p_1 in Fig. 5.1(a) and q_5 in Fig. 5.1(b)). If this is the case, no locations fall into both dominance regions, thus their overlapping region is an empty set. □

Minimum OVD (MOVD)

a Minimum Overlapped Voronoi Diagram (MOVD) is an OVD, in which all empty OVRs have been removed. An OVD is an MOVD if it does not have any empty OVR. The formal definition of MOVD is:

$$MOVD(\mathbb{E}) = OVD(\mathbb{E}) - \{\emptyset\} \tag{5.13}$$

In the extreme case that $\mathbb{E}$ is an empty set, no Voronoi diagrams are overlapped, and the search space is not decomposed into subregions. We define this case as:

$$MOVD(\emptyset) = OVD(\emptyset) = \{\mathbb{R}\} \tag{5.14}$$

OVD/MOVD Properties

A number of properties and proofs can be derived from OVD/MOVD definitions. These properties are the basis of the OVD/MOVD model utilized in our MOVD-based solution.

Property 2. $|MOVD(\mathbb{E})| \leq |OVD(\mathbb{E})| = \prod_{P_i \in \mathbb{E}} |P_i|$.

Proof. By Equation 5.12, OVRs are generated by a combination of selected Voronoi regions. So the number of OVRs in $OVD(\mathbb{E})$ is the product of the number of Voroni regions in these Voronoi diagrams. Because all the possible empty sets have been removed, the size of $MOVD(\mathbb{E})$ is less than or equal to $OVD(\mathbb{E})$. □

Property 3. *Any MOVD fully covers the entire search space* $\mathbb{R}$*.*

$$\bigcup^{OVR_j \in MOVD(\mathbb{E})} OVR_j = \mathbb{R} \tag{5.15}$$

Proof. According to the Voronoi diagram property that a Voronoi diagram fully covers the entire search space, $\forall l \in \mathbb{R}$, there must exist Voronoi regions $\{Dom(p_i^s) \in VD(P_i) | l \in Dom(p_i^s), p_i^s \in P_i, P_i \in \mathbb{E}\}$. By Equation 5.12, the location l is at the $OVR(p_1^u, ..., p_n^v)$, and an OVD fully covers the entire search space. Moreover, because $MOVD(\mathbb{E})$ only removes empty sets from $OVD(\mathbb{E})$, $MOVD(\mathbb{E})$ covers the entire search space as well. □

Property 4. *The overlapping area of two different OVRs is a subset of their common boundaries.*

Proof. By Equation 5.12, an OVR is the overlapping region of $\{Dom(p_1^u),..., Dom(p_n^v)\}$. If we have two OVRs from an OVD that $OVR = \bigcap_{p_i^s \in \{p_1^u,...,p_n^v\}} Dom(p_i^s)$, and $OVR' = \bigcap_{p_i^{s'} \in \{p_1^{u'},...,p_n^{v'}\}} Dom(p_i^{s'})$, then the overlapping area of OVR and OVR' is

$$\begin{aligned} & OVR \bigcap OVR' \\ & = (\bigcap_{p_i^s \in \{p_1^u,...,p_n^v\}} Dom(p_i^s)) \bigcap (\bigcap_{p_i^{s'} \in \{p_1^{u'},...,p_n^{v'}\}} Dom(p_i^{s'})) \\ & = \bigcap_{i=1}^{n} (Dom(p_i^s) \bigcap Dom(p_i^{s'})) \end{aligned} \tag{5.16}$$

According to the properties of the Voronoi diagram, $Dom(p_i^s) \cap Dom(p_i^{s'})$, where $p_i^s \neq p_i^{s'}, p_i^s, p_i^{s'} \in P_i$, is either their common boundaries or an empty set. Moreover, if OVR and OVR' are different, there must exist a p_i^s and $p_i^{s'}$ that are different. The boundaries of an OVR are comprised of the boundaries of corresponding Voronoi regions. Hence, the overlapping region of OVR and OVR' is a subset of their common boundaries. □

Property 5. *Given a type weight function* ς^t*, and object weight functions* $\sigma = \{\varsigma_1^o, ..., \varsigma_n^o\}$*, a point* q *in* $OVR(p_1^u, ..., p_n^v)$*, the total weighted distance from* q *to the corresponding object*

group $G = \{p_1^u, ..., p_n^v\}$ *is the minimum weighted distance from* q *to all object combinations* G'*, where* $G' \in P_1 \times ... \times P_n$.

$$WGD(q, G, \varsigma^t, \sigma) = \min(\{\ WGD(q, G', \varsigma^t, \sigma) \mid G' \in P_1 \times ... \times P_n\ \}\ ) \tag{5.17}$$

Proof. If $VD(P_1)$ is generated by P_1 and weight function $\varsigma_1^o \in \sigma$, a point q in $OVR(p_1^u, ..., p_n^v)$ must fall in $Dom(p_1^u)$ of $VD(P_1)$ so that p_1^u is the closest point in P_1 to q. $WD(q, p_1^u, \varsigma^t, \varsigma_1^o)$ is the minimum weighted distance from q to any points in P_1. We can get the same result in other sets $P_i \in \mathbb{E}$. After summing them up, we obtain Property 5 that $WGD(q, G, \varsigma^t, \sigma)$ has the minimum distance. □

Property 6. *|*$MOVD(\mathbb{E})$*| is bigger than or equal to |*$VD(P_i)$*|, where* $P_i \in \mathbb{E}$*.*

$$|MOVD(\mathbb{E})| \geq |VD(P_i)| \tag{5.18}$$

Proof. Overlapping two Voronoi diagrams is a process in which one Voronoi diagram is decomposed by another Voronoi diagram. Each Voronoi region is divided into a number of subregions, unless two Voronoi regions from different VDs are exactly the same, or one region contains the other. In these extreme cases, the Voronoi region remains unchanged. Thus after overlapping Voronoi diagrams, the number of overlapping regions in an MOVD is either greater than or equal to the basic Voronoi diagrams. □

Property 7. *When* $\mathbb{E}$ *is made up of only one object set* $\mathbb{E} = \{P\}$*, then*

$$MOVD(\mathbb{E}) = OVD(\mathbb{E}) = VD(P) \tag{5.19}$$

Proof. This property is straightforward. If $\mathbb{E}$ has only one object set P, there is no other Voronoi diagram overlapped on $VD(P)$. Obviously $VD(P)$ does not have any empty regions. So $OVD(\mathbb{E})$ and $MOVD(\mathbb{E})$ are identical to $VD(P)$. This property not only states an

extreme case of definitions, but also highlights basic units in the OVD/MOVD model. All OVDs are generated from these building blocks. □

5.3.3 Algebraic Structure of MOVD

After theoretically introducing the OVD/MOVD model, we will mainly focus on the overlap operation. We create an algebraic structure of MOVD by exploring MOVD space under the overlap operation and discussing its properties. The implementation details of the operation will be presented in Section 5.4.

MOVD space

MOVD space is a universal set of MOVDs that are fed into and produced by the overlap operation. Given a universal set of object sets $\mathbb{E} = \{P_1, ..., P_n\}$, the universal set of MOVD($\mathbb{E}$) is defined as

$$U(MOVD(\mathbb{E})) = \{MOVD(E_i) \mid E_i \subseteq \mathbb{E}\} \tag{5.20}$$

Property 8. *The number of MOVDs existing in the universal space is as follows:*

$$|U(MOVD(\mathbb{E}))| = \sum_{i=0}^{|\mathbb{E}|} \binom{|\mathbb{E}|}{i} \tag{5.21}$$

Proof. By definition, MOVD space consists of a number of MOVDs, each of which is generated by a subset of $\mathbb{E}$; thus the number of MOVDs in the space equals the number of subsets in $\mathbb{E}$, which is presented as Equation 5.21. The case that i equals 0 indicates a special subset, the empty set, defined in Equation 5.14. □

Overlap operation +

We define a binary operation + that overlaps two given MOVDs. The result of + is a new MOVD generated by the union of generator sets of input MOVDs. The formal definition is: given MOVD(E_i) and MOVD(E_j), where $E_i, E_j \subseteq \mathbb{E}$, then

$$MOVD(E_i) + MOVD(E_j) = MOVD(E_i \cup E_j) \tag{5.22}$$

A general implementation (RRB) of the operation will be discussed in Section 5.4.2.

+ Operation Properties

By properties of the union operation on sets, we can obtain the following three laws.

Property 9. *Idempotent Law*

$$MOVD(E_i) + MOVD(E_i) = MOVD(E_i) \tag{5.23}$$

Property 10. *Commutative Law*

$$MOVD(E_i) + MOVD(E_j) = MOVD(E_j) + MOVD(E_i) \tag{5.24}$$

Property 11. *Associate Law*

$$\begin{aligned}(MOVD(E_i)+&MOVD(E_j)) + MOVD(E_k) = \\ &MOVD(E_i) + (MOVD(E_j) + MOVD(E_k))\end{aligned} \tag{5.25}$$

Corollary 1. *MOVD(E_i), where $E_i \subseteq \mathbb{E}$, is unique.*

Proof. According to the commutative and associate laws of operation +, the order of overlapping Voronoi diagrams does not cause the result to change. Thus MOVD(E_i) is unique. □

Property 12. *$MOVD(\emptyset)$ is an identity element.*

Proof. $MOVD(\emptyset)$ equals $\{\mathbb{R}\}$ that leaves MOVDs unchanged under operation +. The following equation can be easily proved by the definition of +.

$$MOVD(E_i) + MOVD(\emptyset) = MOVD(E_i) \tag{5.26}$$

□

Property 13. *Closure: the universal MOVD space of* $\mathbb{E}$ *is closed under operation* +.

Proof. By definition, given any MOVD(E_i) and MOVD(E_j), where $E_i, E_j \subseteq \mathbb{E}$, the result of overlapping them is MOVD($E_i \cup E_j$). Obviously, $E_i \cup E_j$ is still a subset of $\mathbb{E}$, so the result is an element of $U(MOVD(\mathbb{E}))$. □

Definition Sequential Overlap Operations

$$\begin{aligned} \sum_{i=1}^{n} MOVD(E_i) &= MOVD(E_1) + ... + MOVD(E_n) \\ &= MOVD(\bigcup_{i=1}^{n} E_i) \end{aligned} \tag{5.27}$$

Definition Partial Order

If $MOVD(E_i) = MOVD(E_j) + MOVD(E_k)$ then,

$$\begin{aligned} MOVD(E_i) &\geq MOVD(E_j) \\ MOVD(E_i) &\geq MOVD(E_k) \end{aligned} \tag{5.28}$$

The partial order definition formalizes a comparison model for evaluating how much information MOVDs maintain. As Equation 5.28 shows, $MOVD(E_i)$ is generated by $MOVD(E_j)$ and $MOVD(E_k)$. $MOVD(E_i)$ has more information (*i.e.*, objects) than either $MOVD(E_j)$ or $MOVD(E_k)$. We use $\geq$ to denote the relationship.

Property 14. $MOVD(E_i) + MOVD(E_j) = MOVD(E_i)$ *if* $MOVD(E_i) \geq MOVD(E_j)$.

Proof. The following equation proves Property 14 by applying the partial order definition that decomposes $MOVD(E_i)$ into $MOVD(E_j)$ and $MOVD(E_k)$, and the commutative and idempotent laws of operation $+$.

$$\begin{aligned} & MOVD(E_i) + MOVD(E_j) \\ & = MOVD(E_j) + MOVD(E_k) + MOVD(E_j) \\ & = MOVD(E_j) + MOVD(E_j) + MOVD(E_k) \\ & = MOVD(E_j) + MOVD(E_k) \\ & = MOVD(E_i) \end{aligned} \tag{5.29}$$

□

5.4 Design

After introducing the OVD model, we now start to illustrate our MOVD-based solution for the query.

5.4.1 Framework of the MOVD-based Solution

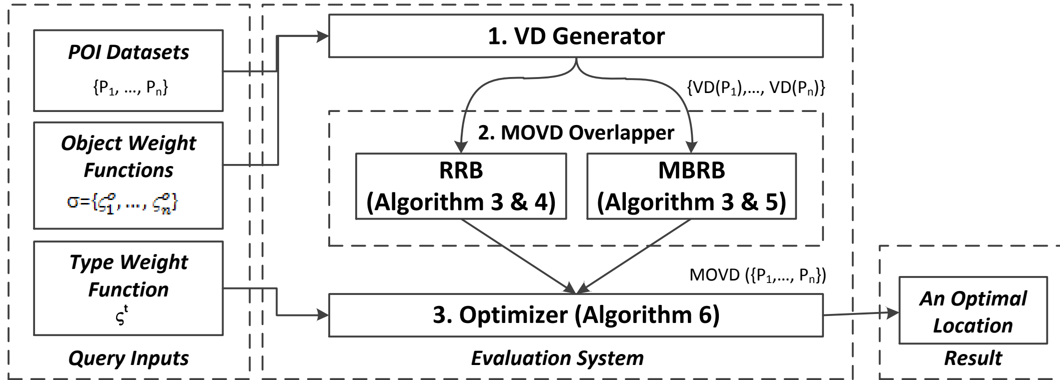


Figure 5.2: The Framework of the MOVD-based solution.

Fig. 5.2 illustrates the framework of our solution. The inputs are POI datasets ($P_i \in \mathbb{E}$), object weight functions $\sigma = \{\varsigma_1^o, ..., \varsigma_n^o\}$ and a type weight function ς^t. The result is an optimal location of the query.

In the evaluation system, the query is sequentially processed by three modules. In particular, based on POIs of particular types and the object weight functions, *VD Generator* generates Voronoi diagrams that are the basic MOVDs used in the next step (see Property 7). Then, a new MOVD is produced by overlapping the basic MOVDs with *MOVD Overlapper* (see Equation 5.27). Finally, *Optimizer* sequentially scans OVRs in the new MOVD, finding a locally optimal location in each OVR, and returns the best of these locations as the query result.

Essentially, two solutions are proposed in Fig. 5.2, illustrated by two paths from the *VD Generator* to the *Optimizer*. The solutions apply either Real Region as Boundary (RRB) or Minimum Bounding Rectangle as Boundary (MBRB) approaches in the *MOVD Overlapper*. RRB and MBRB are two MOVD overlapping approaches that will be described in the following two subsections. A cost-bound approach used in *Optimizer* will be presented in Section 5.4.4. The Voronoi diagram generation approaches used in the *VD Generator* can be found in [83].

5.4.2 The RRB Approach

In this subsection, we describe the RRB approach for MOVD overlapping operation. Since the basic MOVDs are identical to Voronoi diagrams (see Property 7), the generation methods of which has been extensively studied, we will mainly focus on the process of creating an MOVD from two MOVDs. For better explanation, overlapping two basic MOVD is illustrated by the simple example in Fig. 5.3.

A plane-sweep-based algorithm is designed in the RRB approach. As the typical plane sweep approach [32], the RRB approach maintains an event queue and two sweeping statuses. The event queue consists of a number of event points that are maximum and minimum values of projections of OVRs on the y axis. These maximum and minimum points are called start and end points, which indicate that when the sweeping line arrives at these points, the corresponding OVR starts or ends its intersection with the sweeping line. The event points

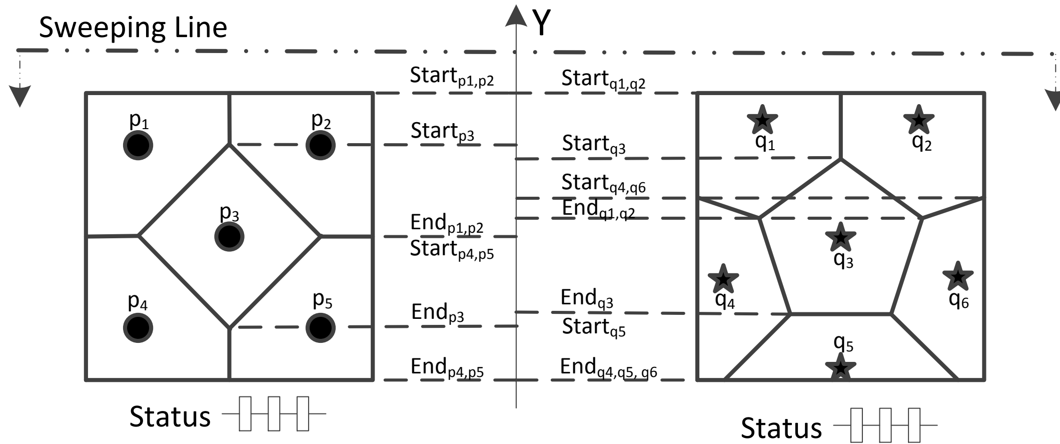


Figure 5.3: Overlapping two MOVDs.

of both MOVDs are sorted by their y-coordinates in descending order. The sweeping line vertically scans the plane from top to bottom, so that the start point of an OVR will be reached before its end point. The status structures are set up to record OVRs that intersect with the sweeping line. Two status structures are maintained individually and respectively for MOVDs. OVRs also have a range (minimum and maximum values) of projections on the x axis.

During the sweeping process, when an end point is arrived at, the corresponding OVR is removed from the status structure. When the sweeping line reaches a start point, the corresponding OVR is inserted into the status structure. Moreover, overlapping regions of the new OVR and OVRs in the other status structure are required to be detected. The detection process first identifies potential OVRs; the range of which overlaps with the new OVR on the x axis. Then, the overlapped region of the two OVRs is calculated. The details are described in Algorithms 4 and 5.

The essential idea of the algorithms is that minimum and maximum values on the x and y axes are an outer boundary of OVR. Two OVRs cannot overlap each other if their outer boundaries do not overlap. Overlapped outer boundary detection significantly reduces overlapping region calculations by avoiding the overlapping of two OVRs (*e.g.*, regions of p_1 and q_5 in Fig. 5.3), which are actually far away from each other.

As shown in Algorithm 4, the overlap operation receives two MOVDs as input parameters and produces a new MOVD. From lines 1-4, *Result*, *EventQueue*, *Status* and $Status'$ are initialized to be empty sets. *Status* keeps the status for $MOVD(E)$, and $Status'$ for $MOVD(E')$. Then, in lines 5-6, events are inserted into *EventQueue* and sorted. Finally, from lines 7-14, all events are iteratively handled by Algorithm 5.

Algorithm 4 Overlap($MOVD(E)$, $MOVD(E')$)

```
 1. Result = ∅
 2. EventQueue = ∅
 3. Status = ∅
 4. Status' = ∅
 5. Push events of MOVD(E) and MOVD(E') into EventQueue
 6. Sort(EventQueue)
 7. while ( EventQueue ≠ ∅ ) do
 8.    e = EventQueue.pop()
 9.    if ( e is from MOVD(E) ) then
10.       EventHandler(e, Status, Status', Result)
11.    else
12.       EventHandler(e, Status', Status, Result)
13.    end if
14. end while
15. return Result
```

Algorithm 5 describes the event handler that receives the following four parameters. e is an event object. *Current* is the status structure of MOVD, from which the event occurs. *Other* refers to the other status structure. *Result* is the MOVD produced by the overlap operation. As shown in Fig. 5.5, an MOVD manages a list of OVRs, each of which is represented as <*region*, *pois*>, where *region* maintains the shape of the OVR and *pois* is a list of objects associated with the OVR. If a start event occurs, the corresponding OVR is first inserted into *Current* status. Then, potentially overlapped OVRs in *Other* are detected by comparing their $Range_X$ with the current OVR. $Range_X$ denotes the range of possible x-coordinates of OVRs. If their $Range_X$ overlap, the overlapped region is calculated in line 5. If the new-generated overlapped region is not empty, a pair of the region and its associated

pois will be appended to $Result$. In the second branch, an end event takes place and the corresponding OVR is removed from $Current$.

Algorithm 5 EventHandler(e, $Current$, $Other$, $Result$)

1. **if** e is a start event **then**
2. Insert $e.ovr$ into $Current$
3. **for** $ovr \in Other$ **do**
4. **if** $Range_X(e.ovr) \cap Range_X(ovr) \neq \emptyset$ **then**
5. $region = e.ovr.region \cap ovr.region$
6. **if** $region \neq \emptyset$ **then**
7. $pois = e.ovr.pois \cup ovr.pois$
8. $Result$.append($< region, pois >$)
9. **end if**
10. **end if**
11. **end for**
12. **else**
13. Remove $e.ovr$ from $Current$
14. **end if**
15. **return**

A general overlapping approach is not presented; however, the RRB approach can be modified to be a general approach used for the OVD model if line 7 is removed and only *region* is appended to *Result* in line 8. *pois* contains the additional information for our specific query type. Algorithm 5 does not specify any methods for overlapping region calculation in line 5. The reason is that the shape of OVRs in a general model is difficult to predict. The case is worse after overlapping because the OVRs become more complex. Furthermore, overlap methods for regions vary greatly as well. The overlap methods for polygons are different from the ones for circles. The overlap methods applied in the model cannot be determined until the shapes of regions have been decided. We will discuss this issue in Section 5.4.3.

The RRB approach is an output-sensitive algorithm, the complexity of which depends on the size of the results, or more exactly the number of OVRs existing in the new MOVD. We denote the average size of input MOVDs by n. There are totally $4 \times n$ events, and sorting them in order takes $O(n \lg n)$ time. There are $2 \times n$ start and end events handled

by Algorithm 5. If status structures are organized as a balanced search tree that sorts OVRs in order by their start x-coordinates, inserting or deleting an OVR from status can be completed in $O(\lg n)$ time. The total cost of maintaining status is $O(n \lg n)$ as well. If status structures record the start and end x-coordinates of OVRs, a range specified by the points that are either immediately smaller than minimum or greater than maximum x-coordinate of the current OVR can be figured out in $O(\lg n)$ time. The OVRs, whose event points are located at the range are potentially required to overlap the current OVR. Moreover, we denote the number of OVRs in the result by I and costs of overlapping region computation by θ. The costs of calculating the overlap regions is $\theta \times I$. In the worst case, I becomes n^2, so that the total costs of operation $+$ is $O(\theta \times n^2)$.

5.4.3 The MBRB Approach

According to the variety of weight functions specified in the query inputs, various Voronoi diagrams are generated by the *VD Generator*. In addition to the ordinary Voronoi diagrams shown in Fig. 5.1, two typical weighted Voronoi diagrams are displayed in Fig. 5.4. The generation methods of additively and multiplicatively Voronoi diagrams have been presented in [22, 60, 18, 44, 19, 35, 81]. More practical Voronoi diagrams can be found in [83].

Although the generation methods of weighted Voronoi diagrams had been extensively studied, efficiently maintaining the shape of OVRs is extremely difficult since they are not in regular shapes. In general, their boundaries have to be modelled by a number of curves. More importantly, overheads of overlapping region calculations would be highly expensive due to the complexity of boundary representation.

To overcome this difficulty, we propose the MBRB approach that combines Algorithm 4 with an alternate event handler, MBRBHandler, for the overlap operation. The MBRB approach is motivated by an observation that the shapes of OVRs are not used in *Optimizer*. Instead, the POI locations and their weights are the criteria for optimal location selection; therefore, we set Minimum Bounding Rectangles (MBR) of OVRs as their shapes in this

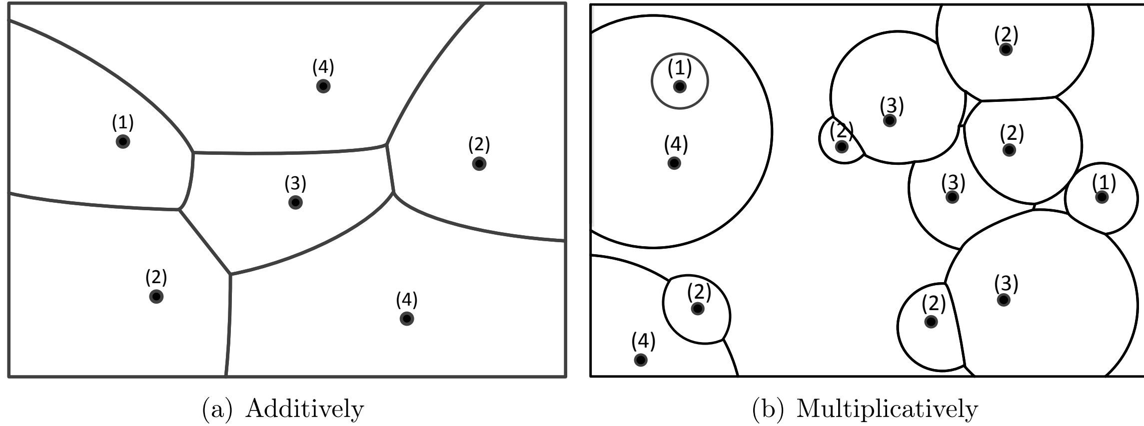


Figure 5.4: Weighted Voronoi diagrams (the numbers indicate weights).

approach. Two OVRs will be treated as overlapped if their MBRs are overlapped. This approach is able to significantly reduce the cost of the overlap operation by simplifying boundary maintenance and avoiding region overlapping calculations; however, the approach suffers from that unnecessary OVRs (false positives which are not really overlapped) would be appended to the new MOVD.

The data structure used in MBRBHandler is illustrated in Fig. 5.5. An OVR is indicated as $<MBR, pois>$, where MBR is comprised of minimum and maximum points on the x and y axes, and *pois* is a list of objects associated with the OVR.

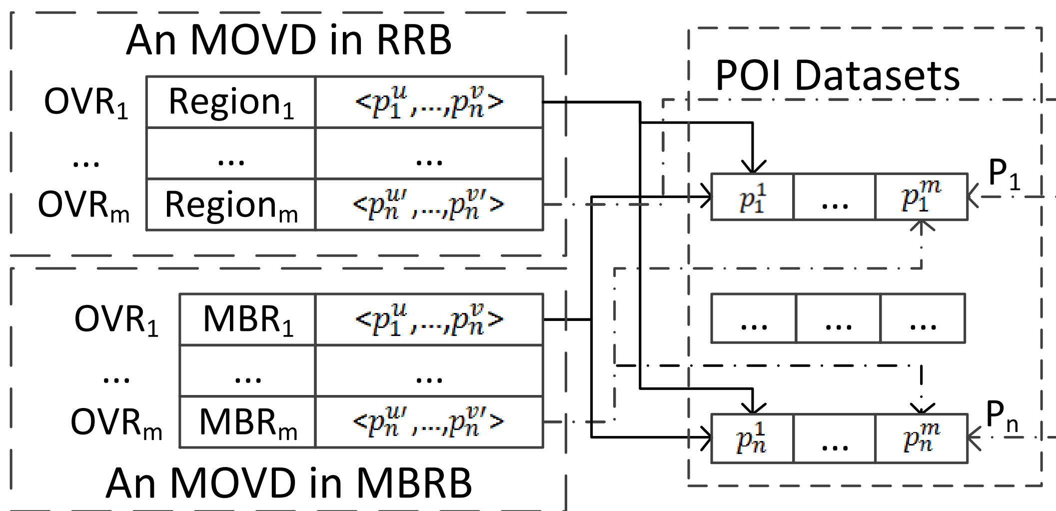


Figure 5.5: Data structures.

The MBRBHandler is described in Algorithm 6. In the branch that handles start event processing, the handler only detects whether two MBRs are overlapped. If this is the case, the MBRs are overlapped and the objects associated with the two OVRs are merged. The new OVR is appended to the result. The final branch remains unchanged.

Algorithm 6 MBRBHandler(e, $Current$, $Other$, $Results$)

```
1.  if e is a start event then
2.     Insert e.ovr into Current
3.     for ovr ∈ Other do
4.        if Range_X(e.ovr) ⋂ Range_X(ovr) ≠ ∅ then
5.           mbr = e.ovr.MBR ⋂ ovr.MBR
6.           pois = e.ovr.pois ⋃ ovr.pois
7.           Results.append(<mbr, pois>)
8.        end if
9.     end for
10. else
11.    Remove e.ovr from Current
12. end if
13. return
```

Compared to the RRB approach, the complexity of region overlapping θ decreases in constant time, but the size of output I increases, the impact on the performance of which is difficult to be evaluated. The upper bound of I is n^2; therefore, the complexity of the MBRB approach becomes $O(n^2)$ in the worst case.

It is worth noting that the basic principle of our solution is that the search space is decomposed into a number of OVRs, in which a locally optimal location is found by *Optimizer*; however, the shapes of OVRs are not calculated. How does our solution determine an optimal location in an OVR?

Our solution does not limit the locally optimal location in a particular OVR. Instead, we look for it in the entire search space. As shown in Fig. 5.6, if an optimal location L_k is found in OVR_k, L_k will undoubtedly be appended to the candidate list. If the optimal location L_i is outside of OVR_i, according to Property 3, L_i must be located in another OVR, say OVR_j, which must have an optimal location L_j. L_j must be identical or better than L_i. Appending them to the candidate list does not change the global optimum since only the

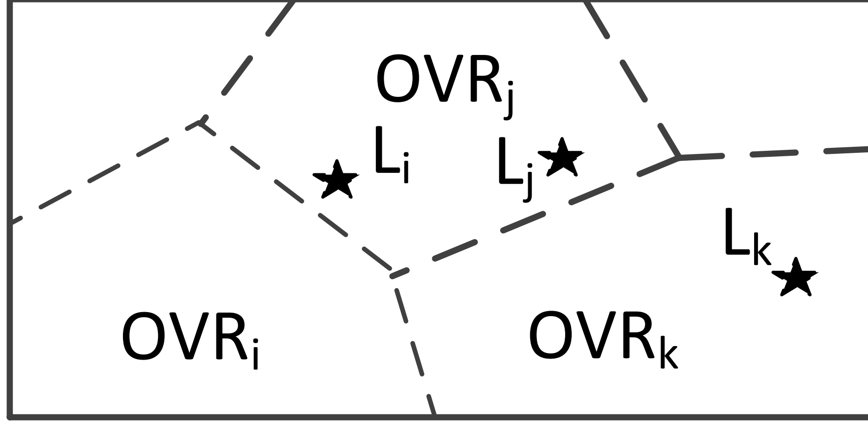


Figure 5.6: Optimal locations.

best one will be returned as the query result. Thus appending L_i to the candidate list does not change the global optimum.

5.4.4 A Cost-Bound approach in *Optimizer*

An optimal location q that minimizes $MWGD(q, \mathbb{E}, \varsigma^t, \sigma)$ is found in the third step of the proposed framework. The framework does not specify a weight function for type weight calculations; however, we mainly focus on a multiplicatively-based weight function, which is one of the practical methods used in real applications. Other weight functions can be applied in the framework as well.

If applying a multiplicatively-based weight function to type weights, the problem of finding an optimal location in each OVR is converted into a typical Fermat-Weber problem in two-dimensional space. The objects associated with OVRs are the points in the Fermat-Weber problems. The weights of the points are specified by the type weight function ς^t. The object weights are integrated into the distance from a location to points. As mentioned in Section 5.1.3, the problem had been solved theoretically. The optimal location in three-point cases and multiple-collinear-point cases can be found in constant and linear time, respectively. An approximate iterative approach has been proposed for other cases [127].

In the RRB and MBRB approaches, we observe that large number of OVRs will be created by *MOVD Overlapper* (see Property 2). In addition, the number of the problems increases rapidly when the number of objects grows. A basic approach is to sequentially calculate optimal locations of these Fermat-Weber problems, and select the best one as the query result; however, applying the iterative method to the Fermat-Weber problems is surprisingly expensive. Therefore, we propose a cost-bound approach, in which an optimal cost is set as a global lower bound. During the processing of a Fermat-Weber problem, a local lower bound of the cost in each iteration will be calculated. If the local lower bound is greater than the global lower bound, no matter how many iterations will be processed, its local optimal cost cannot be better than the global lower bound. Thus the following iterations can be avoided, even though the stopping condition has not been satisfied. The definition and the cost-bound approach of the problem are formally described as below.

Optimum Location of Multiple Fermat-Weber Problems

Given a set of object groups $\mathbb{E} = \{G_1, ..., G_n\}$, where G_i ($|G_i| \geq 3$) contains points of a Fermat-Weber problem, a type weight functions ς^t and object weight functions σ. Let l_j denote the optimal location of G_j under a stopping condition γ, the optimal location of $\mathbb{E}$ is a location $l \in \{l_j | 1 \leq j \leq n\}$ that minimizes WGD(l_j, G_j, ς^t, σ).

A Cost-Bound Approach

The cost-bound approach receives a set of object groups $\mathbb{E}$, a type weight function ς^t, object weight functions σ, and a stopping condition γ. The weights of the objects are indicated by ς^t. The distance from a location to points is calculated by their Euclidean distance and σ. The number of points in the Fermat-Weber problems ($|G_i|$) is unnecessarily fixed.

In Algorithm 7, the global lower bound, *Cbound*, is initialized to infinity and reduced to the minimum cost of the optimal location found so far. The algorithm sequentially checks

Algorithm 7 CostBoundApproach($\mathbb{E}$, ς^t, σ, γ)

1. $Cbound = \infty$
2. $l = < 0, 0 >$
3. **for** $G_i \in \mathbb{E}$ **do**
4. Initialize l_i to the center of G_i
5. **if** $|G_i| = 3$ or G_i is a collinear case **then**
6. Calculate the optimal location l_i of G_i
7. **else**
8. Let $G_i =< p_1^u, ..., p_n^v >$
9. Calculate the optimal location l' of $< p_1^u, p_2^s >$
10. **if** WGD(l', $\{p_1^u, p_2^s\}$, ς^t, σ) $>$ Cbound **then**
11. Continue
12. **end if**
13. **repeat**
14. l_i = f (l_i, G_i) /* Iterating, see Equation 5.8 */
15. $Lbound$ = lb(l_i) /* see Equation 5.10 */
16. **until** γ is satisfied or $Lbound \geq Cbound$
17. **end if**
18. $Cost$ = WGD(l_i, G_i, ς^t, σ)
19. **if** $Cbound > Cost$ **then**
20. $Cbound = Cost$
21. $l = l_i$
22. **end if**
23. **end for**
24. **return** l

the Fermat-Weber problems, in each of which a local optimal location is found in line 4-17. In the branch of the iterative method inside the loop, an optimal location of the first two points in G_i is first detected in line 8-12, as SSC solution does. If a better result of G_i potentially exists, a local lower bound is calculated in each iteration in line 15. If the local lower bound is greater than $Cbound$, the iteration will stop in line 16. The Cost-bound approach can be used in the SSC solution as well.

5.5 Experimental Validation

In this subsection, we evaluate the performance of the OVD model and proposed query solutions with real-world data sets. We implemented the proposed algorithms in C++. All data was loaded into the main memory during the execution of the simulations. All the

experiments were conducted on a Red Hat Enterprise Linux server equipped with four Intel Xeon X5550 2.67 GHz processors and 24GB of memory. All simulation results were recorded after the system model reached a steady state.

5.5.1 Data Sets

In our experiments, the data sets were downloaded from GeoNames[1]. We retrieved 230,762 streams (STM), 200,996 schools (SCH), 166,788 populated places (PPL), 225,553 churches (CH) and 110,289 buildings ($BLDG$) in the United States. By default, we set the type weight w^t and object weight w^o to 1. The multiplicatively-based weight functions are used as ς^t and σ. GPC library[2] is used for polygon overlapping calculations.

5.5.2 Cost-Bound Approach Evaluation

We evaluate the basic (Original) and cost-bound (CB) approaches by varying the number of Fermat-Weber problems and the error bound ϵ. The basic approach sequentially calculates the optimum locations of all Fermat-Weber problems, and selects the best location for the result. The number of points in each Fermat-Weber problem is fixed to 5. The coordinates and type weights of points are randomly generated from 0 to 10. The iterative method for a Fermat-Weber problem will stop when the deviation from the optimal cost is less than the error bound ϵ (see Section 5.1.3) [97].

Fig. 5.7 displays the execution time of the two approaches. As either the problem size increases or ϵ decreases, the execution time of both approaches rises. Obviously, the growth rate of the original approach is higher than the cost-bound approach because a significant number of unnecessary iterations can be avoided by setting a cost bound, which makes the cost-bound approach more efficient, even though it has to pay extra overhead on lower bound calculation in each iteration.

[1] http://www.geonames.org/

[2] http://www.cs.man.ac.uk/~toby/gpc/

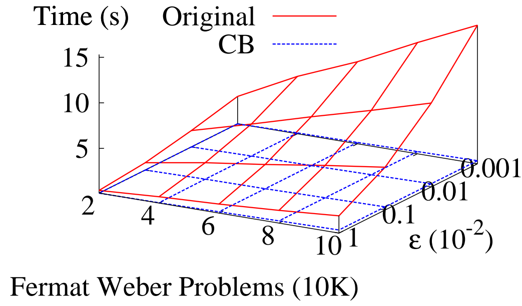


Figure 5.7: The CB approach evaluation.

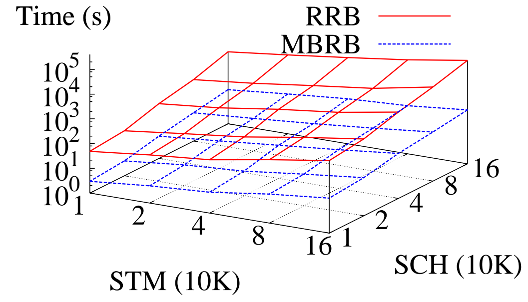


Figure 5.8: Execution time.

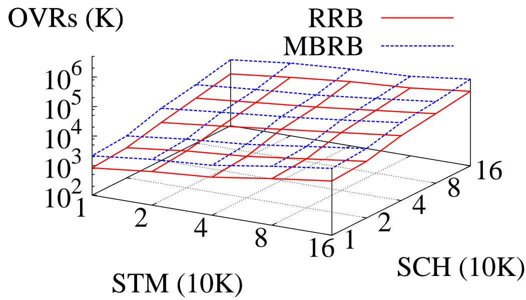


Figure 5.9: Number of OVRs.

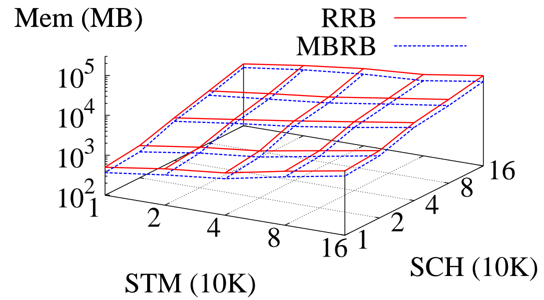


Figure 5.10: Memory consumption.

5.5.3 Overlapping Two Voronoi Diagrams

Two overlap approaches, RRB and MBRB, on two regular Voronoi diagrams are evaluated with various data set sizes. The Voronoi diagrams are generated by two object sets, which are randomly selected from STM and SCH. Their sizes are indicated by x and y axes in Fig. 5.8-5.10.

From Fig. 5.8, we observe that the MBRB performs better than the RRB in terms of execution time. In RRB, the OVRs in the new-generated MOVD are determined by region overlapping calculation (polygon overlapping calculation in this experiment), which is more expensive than the MBR detection (rectangle overlapping calculation) used by MBRB. Therefore, MBRB is able to complete the overlapping processing in a shorter time. Also, due to replacing real regions of OVRs with their MBRs, MBRB will generate more OVRs

than RRB, as shown in Fig. 5.9. Two OVRs that are not really overlapped with each other may be determined to be overlapped by the MBR detection. However, Fig. 5.10 shows that MBRB consumes less memory than RRB. The reason is that although MBRB has more OVRs, the regions (MBRs) of which can be represented by two points while all vertices of polygons have to be recorded in RRB. According to Fig. 5.10, the total number of points managed by MBRB approach is smaller than RRB.

5.5.4 Overlapping Multiple Voronoi Diagrams

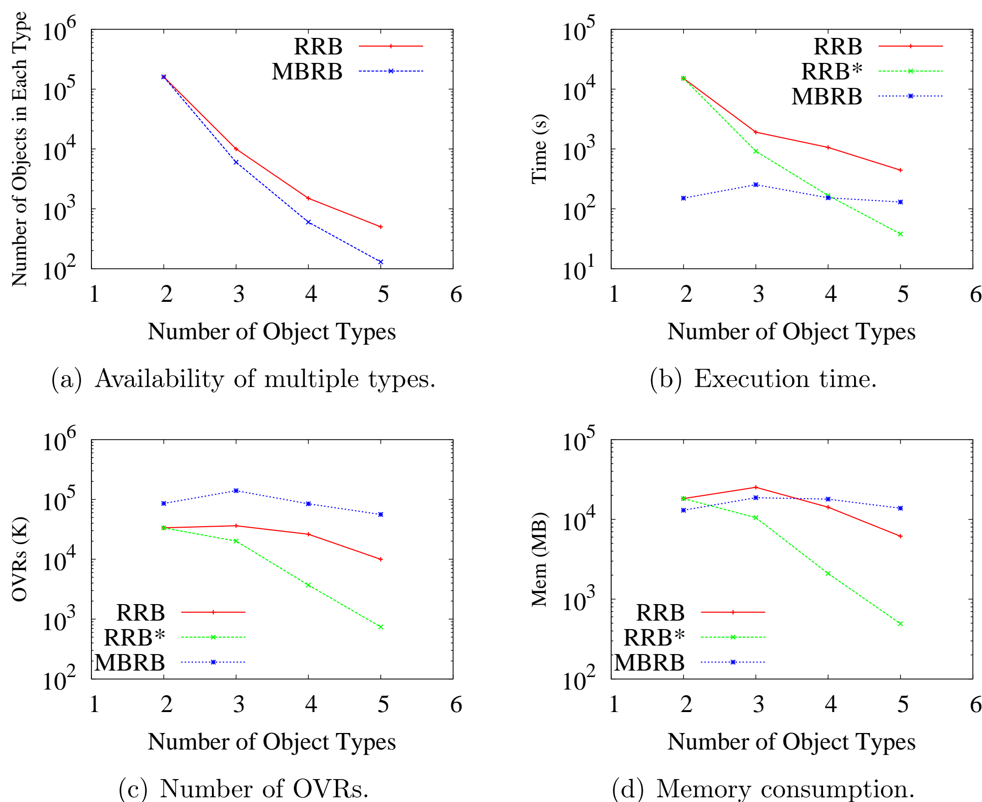


Figure 5.11: Varying number of object types.

In our experiment, we examine the overlap operation by varying the number of Voronoi diagrams. These Voronoi diagrams are generated by objects randomly selected from $\mathbb{E} = \{STM, SCH, PPL, CH, BLDG\}$. In addition to evaluating the performance, we explore the availability of the overlap operation, which is described by the maximum size of objects

that can be processed on the test bed. All data is assumed to be loaded into the memory (24GB).

Fig. 5.11(a) demonstrates the availability of the overlap operation by varying the number of object types. When the number of object types increases, the maximum numbers of objects in both the RRB and MBRB approaches drop rapidly. The more Voronoi diagrams overlap, the more OVRs are generated which requires more memory. Moreover, the dropping rate of the MBRB approach is higher than RRB because the MBRB approach consumes more memory when the number of object types is greater than three as shown in Fig. 5.11(d).

Fig. 5.11(b), 5.11(c) and 5.11(d) display corresponding execution time, the number of OVRs and memory consumption of both approaches with parameters that lie on the availability line. In order to fairly compare the two approaches, the performance of the RRB approach with parameters identical to the MBRB approach is highlighted by RRB*.

As we expect, the MBRB approach always generates more OVRs than RRB* as shown in Fig. 5.11(c). Moreover, in Fig. 5.11(b), when the number of object types is greater than 4, RRB* performs better than the MBRB approach because the computation complexity induced by surprisingly large number of OVRs dominates the entire process in the MBRB approach, which has greater impact than the benefits obtained from the region overlapping calculation. A turning point in terms of memory consumption is observed between 2 and 3 in Fig. 5.11(d). When overlapping three or more Voronoi diagrams, the MBRB approach consumes more memory due to the large number of OVRs, in which the total number of points is more than the vertices managed by RRB*.

5.5.5 MOLQ Evaluation

We evaluate the solutions for MOLQ queries with three and four object types that are popular applications in the real world. The type weights are randomly generated from 0 to 10. The objects are randomly selected from $\mathbb{E} = \{STM, SCH, PPL\}$ in three-type case and $\mathbb{E} = \{STM, SCH, PPL, CH\}$ in four-type case, respectively.

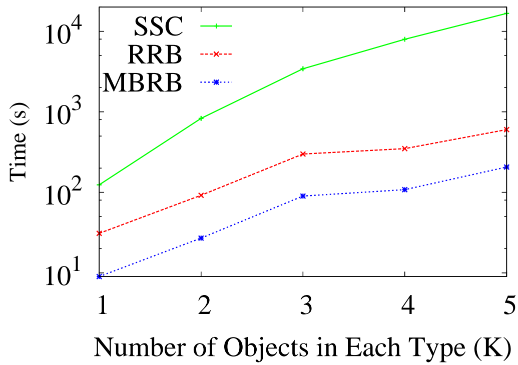


Figure 5.12: Three object types.

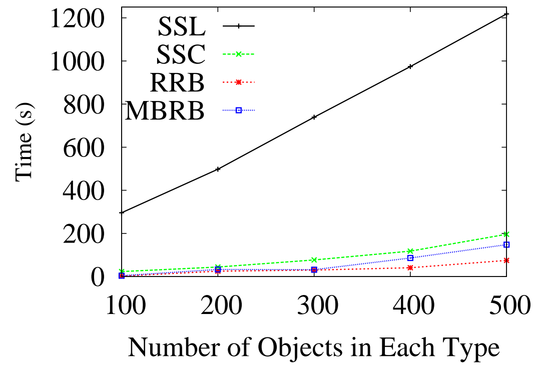


Figure 5.13: Four object types.

Fig. 5.12 displays the performance of SSC, proposed RRB and MBRB solutions. The cost-bound approach is used in all the three solutions. We do not compare with the SSL approach, since it only provides an approximate solution while the other three approaches produce exact solutions for the queries with three object types. As Fig. 5.12 shows, RRB and MBRB perform better than SSC because they avoid a significant number of object combinations. Overlapping Voronoi diagrams is a process of filtering out combinations that cannot be the closest objects of any location. Another observation is that MBRB completes the query processing in a shorter time than RRB. The evidence has been shown in Fig. 5.7 and 5.11; the benefit obtained by *MOVD Overlapper* in MBRB is larger than the overhead paid in *Optimizer*.

In the query with four object types, only approximate results can be provided by the four approaches. The distance-bound η in SSL approach is fixed at 1 km. The error bound ϵ is set to be 0.001. Fig. 5.13 shows the execution time of the four solutions, in which the RRB solution has the best performance. Although the execution time of overlapping processing in the MBRB approach is slightly shorter than RRB as shown in Fig. 5.11(b), more OVRs generated by MBRB increase the computation complexity of the Fermat-Weber calculation.

Chapter 6

Efficient Evaluation of Spatial Keyword Queries on Spatial Networks

In this chapter, we present efficient approaches to answer spatial keyword queries on spatial networks. In particular, we systematically introduce formal definitions of Spatial Keyword k Nearest Neighbor (SKkNN) and Spatial Keyword Range (SKR) queries. Then, we illustrate the framework of a spatial keyword query evaluation system, which consists of Keyword Constraint Filter (KCF), Keyword and Spatial Refinement (KSR), and the spatial keyword ranker. KCF employs an inverted index to calculate keyword relevancy of spatial objects, and KSR refines intermediate results by considering both spatial and keyword constraints with the spatial keyword ranker. In addition, we design novel algorithms for evaluating SKkNN and SKR queries. These algorithms employ the inverted index technique, shortest path search algorithm, and network Voronoi diagrams. Finally, we apply both real-world and synthetic data sets to evaluate the performance of the proposed solutions. Our extensive experimental results show that the proposed SKkNN and SKR algorithms can efficiently answer spatial keyword queries.

This chapter is organized as follows. Section 6.1 formally defines the Spatial Keyword queries. Section 6.2 illustrates our proposed solutions. Section 6.3 shows our experimental results.

6.1 Query Type Definition and Background

In order to explain definitions and algorithms in the following subsections, we prepare a sample data set of hotels in Table 6.1 and an example spatial network in Figure 6.1. All the hotels have three attributes which include their names, amenities, and distances from a specific location q. In Figure 6.1, road segments are assigned weights that stand for their

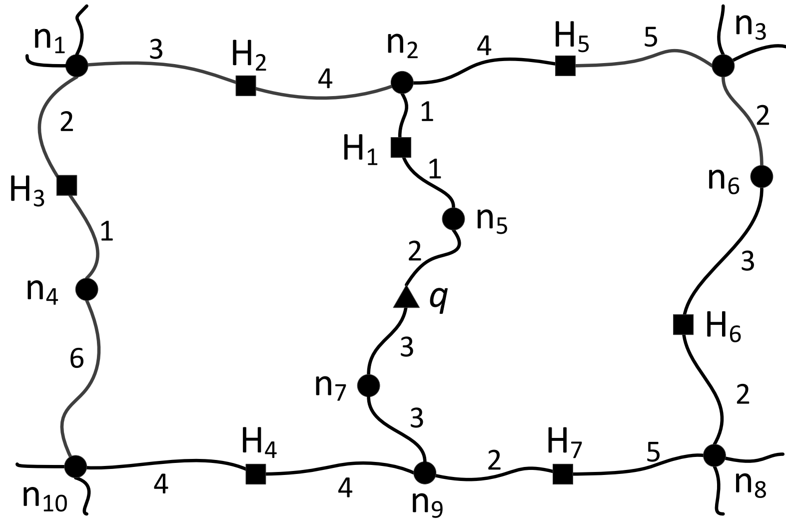


Figure 6.1: An example spatial network.

individual costs (e.g., distance or time). The location q and hotels are symbolized with a triangle and squares on road segments, respectively.

Table 6.1: A sample data set of hotels.

Name	$D_n(q, .)$	Amenities
H_1	3	Internet, Fitness Center, Pets Allowed, Parking
H_2	8	Pool, Parking, Room Service
H_3	13	Internet, Fitness Center, Pets Allowed, Parking
H_4	10	Parking, Airport Shuttle
H_5	8	Pets Allowed, Breakfast, Hot Tub, Restaurant Onsite
H_6	15	Internet, Pets Allowed, Restaurant Onsite
H_7	8	Fitness Center, Hot Tub, Parking

6.1.1 Foundation

In this subsection, we introduce the foundation of spatial keyword queries. In a SK query, a spatial object is defined as a pair $<l, t>$, where l is a location in the search space and t is a text description (e.g., amenities and features of a hotel) of the corresponding object. Table 6.2 summarizes notations used in this chapter.

Distance on Spatial Networks

Spatial networks are composed by undirected weighted graphs $G = (V, E)$, where V is a set of vertices and E is a set of edges. In general, the weight of each edge is determined by a metric measured in physical distance or time cost for traveling the road segment [71, 70]. The distance between two objects $D_n(.,.)$ on spatial networks is the summation of all segment weights on the shortest path connecting the two objects. For example, in Figure 6.1, $D_n(H_6, H_7) = D_n(H_6, n_8) + D_n(n_8, H_7) = 7$.

Matched Keywords

Matched-keywords is a set of keywords which are in both sets of $p.t$ and K, where p is a given spatial object, and K is a set of keywords specified by a user. For example, given a hotel H_2 with keywords {"Pool", "Parking", "Room Service"} and a set of keywords K {"Pool", "Parking", "Breakfast"}, $MK(H_2, K)$ is an intersection of $H_2.t$ and K, {"Pool", "Parking"}. The formal definition of the MK function is shown in Equation (6.1).

$$MK(p, K) = \{\, k_i \in K \mid k_i \in p.t \,\} \tag{6.1}$$

Fully Matched Keyword Search

With a given data set, the purpose of Fully Matched Keyword Search (FMKS) is to find objects whose descriptions completely match with a set of keywords K specified by a requester. As shown in Equation (6.2), the descriptions of search results of FMKS may be either identical to K or a superset of K. For example, given keywords "Internet" and "Pets Allowed" and the data set in Table 6.1, the result set of the FMKS is $\{H_1, H_3, H_6\}$.

$$FMKS(P, K) = \{\, p_i \in P \mid K \subseteq p_i.t \,\} \tag{6.2}$$

Table 6.2: Symbolic notations.

Symbol	Meaning
P	A set of spatial objects
Q	A spatial keyword query
K	A set of search keywords
q	The location of a requester
I	An inverted index
k	The requested number of objects in the result of a SKkNN query
r	The search range of a SKR query
s	The ranking score of an object
$\lvert\mathbb{S}\rvert$	The number of elements in set $\mathbb{S}$
$d(.,.)$	The Euclidean distance between two points
$D_n(.,.)$	The shortest network distance between two points
$\mathbb{R}$	The result set of a query
$\mathbb{E}$	The explored region of a VDkNN query
$\mathbb{C}$	The set of candidate spatial objects

Partially Matched Keyword Search

With a given data set, the purpose of Partially Matched Keyword Search (PMKS) is to retrieve objects which match at least one keyword in the user defined keyword set as shown in Equation (6.3). For example, given keywords “Internet" and “Pets Allowed" and the data set in Table 6.1, the results of the PMKS are $\{H_1, H_3, H_5, H_6\}$. The difference in search results from the previous FMKS is H_5, which matches only one keyword (“Pets Allowed") and is a valid answer of this PMKS.

$$PMKS(P, K) = \{\, p_i \in P \,|\, \exists k_j \in p_i.t \text{ } and \text{ } k_j \in K \,\} \tag{6.3}$$

Weighted Keyword Relevancy

We use a weight function TR to calculate keyword relevancy of a specific spatial object p [128]. We assume that each keyword k_i in a keyword set K is assigned with a weight $w(k_i)$, which indicates its importance in queries. Consequently, given an object p and a keyword set K, we have the following equation:

$$TR(p, K) = \sum_{k_i \in MK(p,K)} w(k_i) \tag{6.4}$$

For special cases where all keywords share identical weight, Equation (6.5) can be derived from Equation (6.4) where $w(k_i) = 1$ and $|MK(p, K)|$ is the number of keywords in $MK(p, K)$.

$$TR(p, K) = \sum_{k_i \in MK(p,K)} 1 = |MK(p, K)| \tag{6.5}$$

6.1.2 Spatial Keyword Ranker

A spatial keyword ranker is designed to determine the ranking of a given spatial object in a SKkNN query by employing both metrics, spatial network distance and keyword relevancy. We utilize a ranking function RK to compute how well an object matches a SKkNN query. Given a query Q $<l$, $K>$ and an object p $<l$, $t>$, the ranking function is defined as follows:

$$RK(Q, p) = \theta_1 \cdot TR(p.t, Q.K) - \theta_2 \cdot D_n(p.l, Q.l) \tag{6.6}$$

In Equation (6.6), θ_1 and θ_2 are parameters of each part of the function [53], and their values depend on user preferences. For example, if a user is more concerned about keyword match, θ_1 can be set to a larger value than θ_2 in order to make keyword relevancy dominant in the RK function. Moreover, intuitively, an object with either shorter distance or higher keyword relevancy would have a higher ranking in query results. Therefore, TR has a positive influence on the RK function while D_n has a negative one.

6.1.3 Spatial Keyword kNN Queries

Based on the spatial keyword ranker, the purpose of a spatial keyword kNN query is to retrieve k objects which have top k ranking values.

Definition Given a SKkNN query Q and an object set P, we define SKkNN(P, Q) as following:

$$\begin{aligned} &RK(p_i) \geq RK(p_j) \ where \ p_i \in SKkNN(P,Q) \ and \\ &p_j \in P \backslash \{SKkNN(P,Q)\} \ and \ |SKkNN(P,Q)| = k \end{aligned} \quad (6.7)$$

We utilize the data set in Table 6.1 and spatial network in Figure 6.1 to demonstrate a SKkNN query example. Assume a visitor wants to find the two nearest hotels that have the amenities, "Internet" and "Pets Allowed" from q. Partially matched results are acceptable when there are not enough fully matched objects in the vicinity. In addition, all keywords have identical weight and the values of θ_1 and θ_2 are 0.8 and 0.2, respectively. The result set for this query is $\{H_1, H_5\}$ where H_5 has only one matched keyword. Besides, if 4 hotels are requested instead of 2, the result set will be $\{H_1, H_5, H_3, H_6\}$.

6.1.4 Spatial Keyword Range Queries

The purpose of a SK Range query is to find all the objects that fully match the given keywords within a user specified distance.

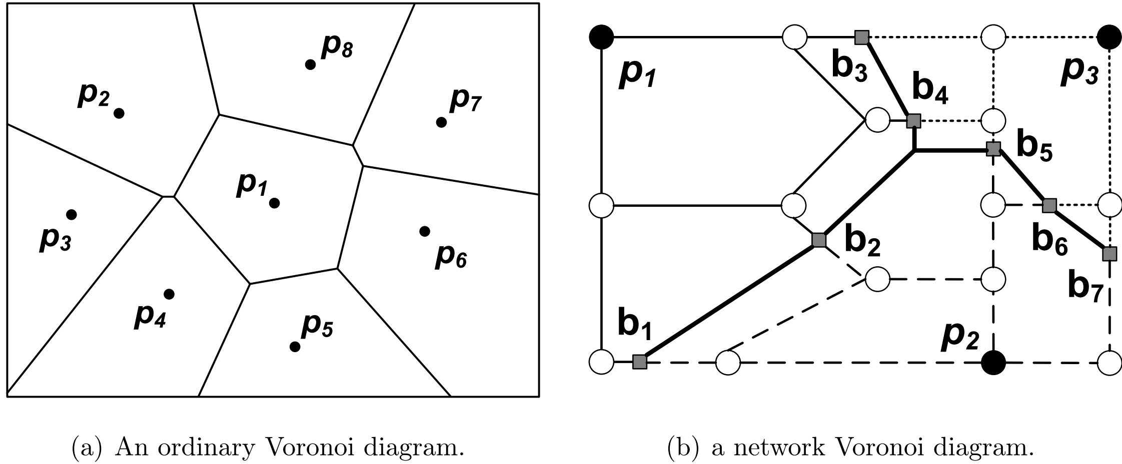
Definition Let P be a set of objects. Given a query location q, a search range r, and a set of keywords K, a SK range query is defined as follows:

$$SKR(q, r, K) = \{p_i \in P | K \subseteq p_i.t \ and \ D_n(p_i, q) \leq r\} \quad (6.8)$$

Assume a tourist wants to find all the hotels bearing the keywords "Internet" and "Pets Allowed" within 10 miles from q on the sample spatial network. The answer is $\{H_1\}$ based on the example data set (Table 1). Furthermore, if the range is extended to 20 miles, the result set will be $\{H_1, H_3, H_6\}$.

6.1.5 Network Voronoi Diagram

We employ network Voronoi diagrams in our approach for efficiently evaluating spatial keyword queries. A Voronoi diagram divides a metric space into disjoint polygons (Voronoi



(a) An ordinary Voronoi diagram.

(b) a network Voronoi diagram.

Figure 6.2: Voronoi diagram examples.

polygons) based on the distances to a specified set of points (generators) in the space. The nearest neighbor of any point inside a polygon is the generator of the polygon. The Voronoi Polygon (VP) and the Voronoi Diagram (VD) in the Euclidean plane can be formally defined as follows. Given a set of generators $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^2$, where $2 \leq n < \infty$ and $p_i \neq p_j$ for $i \neq j, i, j \in I_n = \{1, \ldots, n\}$. The region given by:

$$VP(p_i) = \{p \mid d(p, p_i) \leq d(p, p_j)\} \ for \ j \neq i, j \in I_n\} \tag{6.9}$$

is called the Voronoi polygon associated with p_i where $d(p, p_i)$ denotes the minimum Euclidean distance between p and p_i. In addition, the set given by:

$$VD(P) = \{VP(p_1), \ldots, VP(p_n)\} \tag{6.10}$$

is called the Voronoi diagram generated by P. Figure 6.2(a) demonstrates a Voronoi diagram in the Euclidean plane.

The Network Voronoi Diagram (NVD) is defined based on a planar geometric graph where the locations of objects are restricted to the links that connect the nodes of the graph. Distances between objects are defined as the length of the shortest path in the graph (network distance) [83]. In our problem, spatial networks can be modeled as a geometric graph where

the intersections are symbolized by nodes of the graph and edges are represented by the links connecting the nodes. Furthermore, the weights of links are the distances between corresponding nodes.

The network Voronoi diagram can be formally defined as follows. Consider a geometric graph $G(N, L)$ consisting of a set of nodes $N = \{p_1, \ldots, p_n, p_{n+1}, \ldots, p_l\}$, where the first n elements are the generators (i.e., $P = \{p_1, \ldots, p_n\}$), and a set of links $L = \{l_1, \ldots, l_k\}$ which form a connected network. We define the distance from a point p on a link in L to a node p_i in N, $D_n(p, p_i)$, by the length of the shortest path from p to p_i. For all $j \in I_n \backslash \{i\}$, let

$$Dom(p_i, p_j) = \{p | p \in \bigcup_{i=1}^{k} l_i, D_n(p, p_i) \leq D_n(p, p_j)\} \tag{6.11}$$

$$b(p_i, p_j) = \{p | p \in \bigcup_{i=1}^{k} l_i, D_n(p, p_i) = D_n(p, p_j)\} \tag{6.12}$$

We call the set $Dom(p_i, p_j)$ the dominance region of p_i over p_j on links in L and the set $b(p_i, p_j)$ the bisector (border) points between p_i and p_j on links in L. Accordingly, the Voronoi link set associated with p_i and the network Voronoi diagram are defined as follows, respectively:

$$V_{link}(p_i) = \bigcap_{j \in I_n \backslash \{i\}} Dom(p_i, p_j) \tag{6.13}$$

$$NVD(P) = \{V_{link}(p_1), \ldots, V_{link}(p_n)\} \tag{6.14}$$

where $V_{link}(p_i)$ specifies all the points in all the links in L that are closer to p_i than any other generator point in N. By properly connecting adjacent bisector points of a generator to each other without crossing any of the links, we can generate a bounding polygon, named Network Voronoi Polygon (NVP) [83, 65]. Figure 6.2(b) shows an NVD example where each line style corresponds to a Voronoi link set of a generator (NVPs are created by connecting adjacent bisector points).

6.2 System Design

In this subsection, we design a spatial keyword query evaluation system which is comprised of Keyword Constraint Filter (KCF), Keyword and Spatial Refinement (KSR), and the spatial keyword ranker. For the proposed spatial keyword query algorithms, if two or more objects have the same ranking score, our algorithms will sort the objects based on their distances to the query point (i.e., in an ascending order). In addition, in order to simplify the explanation, we assume all keywords have the same weight.

6.2.1 Framework of Query Evaluation

Before presenting the details of our spatial keyword query algorithms, we briefly introduce the framework of our system. As illustrated in Figure 6.3, the spatial keyword query evaluation system comprises three main components which are Keyword Constrain Filter (KCF), Keyword and Spatial Refinement (KSR) and the spatial keyword ranker. This system receives both spatial data sets and spatial keyword constraints as inputs and produces results after a two-step computation.

The system employs a filter-and-refine strategy to answer SK queries. The two key steps are KCF and KSR. KCF receives spatial data sets and keyword constraints and filters out objects that do not match any user specified keyword. Because spatial network distance computation is expensive, we do not take spatial constraints into account in this step. The main purpose of KCF is to reduce the number of candidate objects in order to decrease computation costs in the next step. In the second step, KSR receives inputs from KCF and refines the intermediate results based on both keyword and spatial constraints. Afterward, KSR returns the qualified objects sorted by their ranking scores provided by the ranker.

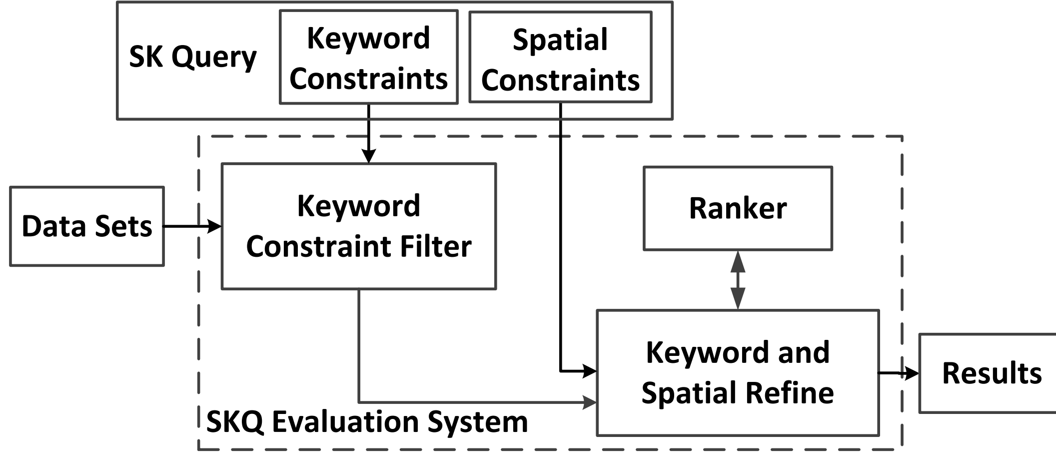


Figure 6.3: Framework of the spatial keyword query evaluation system.

6.2.2 Keyword Constraint Filter

Inverted Indexing Structure

Inverted indexes are primarily designed to support keyword searches from a set of text files. In our system, we utilize inverted indexes to search for objects related to specific keywords from spatial databases. As illustrated in Figure 6.4, an index of terms is maintained in our system where each term is a unique keyword, and each postings list contains a number of object identifiers. Each postings list is in sorted order (based on object identifiers) to facilitate the efficient search of objects related to a specific keyword. If an object has multiple keywords, its identifier will appear in each corresponding postings list. In addition, inverted indexes are independent of other dedicated index structures, such as R-trees or grids, in spatial databases.

Keyword Match Algorithm

Based on the proposed problem, we design a keyword match algorithm by employing the inverted index-based merge technique [77] to calculate keyword relevancy of spatial objects. With the keyword match algorithm, we measure keyword relevancy of a spatial object by counting the number of matched-keywords. The more matched-keywords the object has,

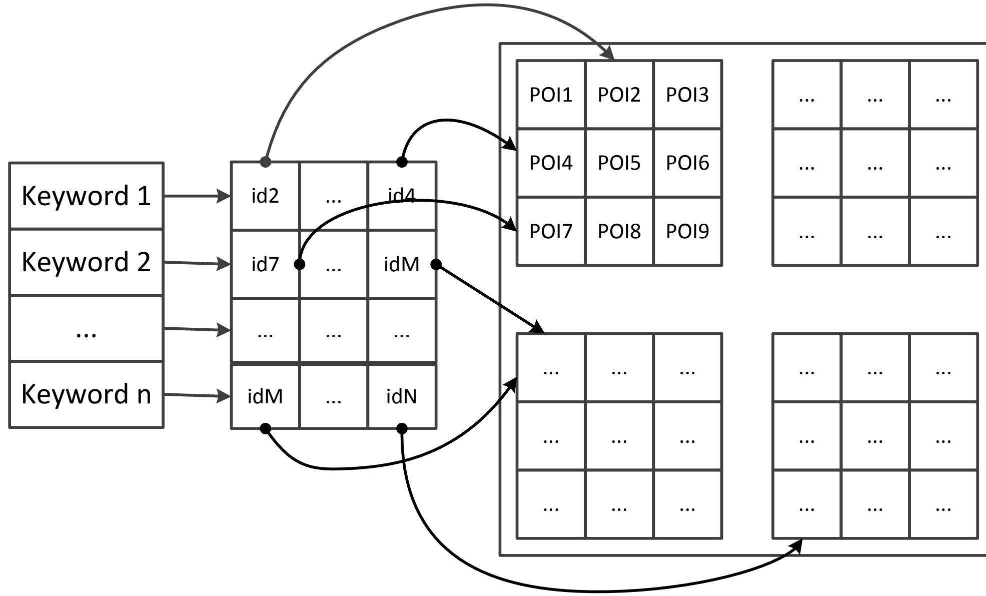


Figure 6.4: Inverted index structure.

the higher its keyword relevancy is. This algorithm receives an inverted index and a set of keywords as input parameters, and then returns the keyword relevancy of objects which match with at least one keyword.

In Algorithm 8, mergeList is a list, of which each element comprises a pair $<id, occurrence>$ where id is an object identifier and $occurrence$ is the corresponding keyword relevancy. With the for loop in line 1, the algorithm iteratively retrieves object lists of matched-keywords from the inverted index structure and merges these object lists into mergeList. This merge process, illustrated in lines 5 to 22, is an essential part which supports partially matched-keyword search. The worst-case time complexity of Algorithm 8 is $O(|K| * |P|)$, where $|P|$ is the number of spatial objects in the data set and $|K|$ stands for the number of search keywords.

Figure 6.5 illustrates how KeywordMatch works. We utilize the data set in Table 6.1 and spatial network in Figure 6.1 for explanation. Assume a query with keywords “Internet", “Pets Allowed", and “Parking" is evaluated. Algorithm 8 first finds object lists that are related to these keywords by searching the inverted index. As shown in Figure 6.5, three

Algorithm 8 KeywordMatch(I, K)

```
1.  for each term in I do
2.    if (term ∈ K) then
3.      iterator iterA = mergeList.begin
4.      iterator iterB = term.idList.begin
5.      while (iterA != mergeList.end and iterB != term.idList.end) do
6.        if (iterA.id > iterB.id) then
7.          newNode ← {iterB.id, 1}
8.          insert newNode at previous position of iterA
9.          iterB++
10.       else if (iterA.id == iterB.id) then
11.         iterA.occurrence += 1
12.         iterA++
13.         iterB++
14.       else
15.         iterA++
16.       end if
17.     end while
18.     while (iterB != term.idList.end) do
19.       newNode ← {iterB.id, 1}
20.       append newNode to the end of mergeList
21.       iterB++
22.     end while
23.   end if
24. end for
25. return mergeList
```

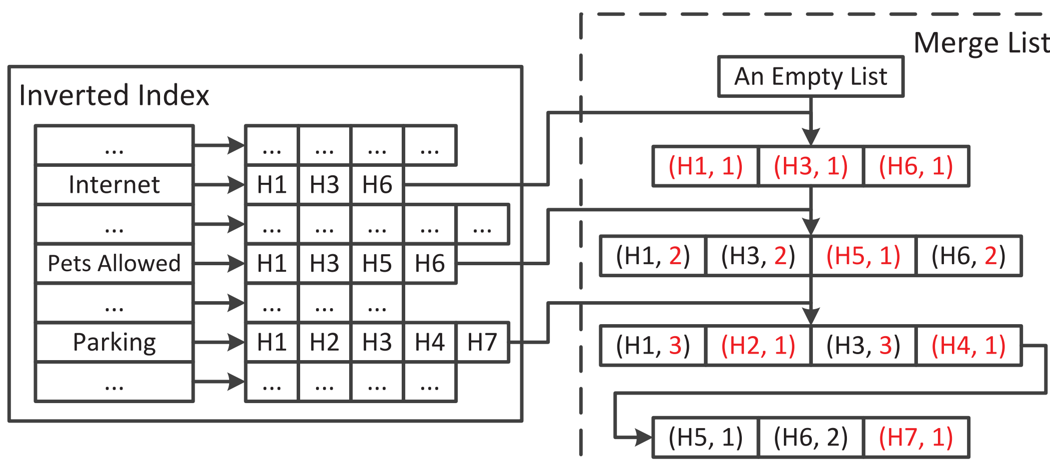


Figure 6.5: An example of KeywordMatch.

object lists, $\{H_1, H_3, H_6\}$, $\{H_1, H_3, H_5, H_6\}$ and $\{H_1, H_2, H_3, H_4, H_7\}$, are retrieved from the inverted index. Then it merges these lists into a mergeList.

In the first round, KeywordMatch simply copies objects in the "Internet" list to mergeList, and each object is marked by one occurrence. Then, in the second round, KeywordMatch compares objects in the "Pets Allowed" list with mergeList. If an object appears in the "Pets Allowed" list but does not exist in mergeList, it will be inserted into mergeList with occurrence marked by one. On the contrary, if an object already exists in mergeList, its counter will be increased by one. The third round of merging the "Parking" list is processed in the same way. After the iterations, mergeList contains the final result with seven objects and their keyword relevancy shown at the bottom of the dashed rectangle in Figure 6.5.

6.2.3 The Network Expansion-Based SKkNN Query Algorithm

In this subsection, we explain our algorithm for processing spatial keyword k nearest neighbor query based on network expansion techniques [87, 34]. As present in Section 6.1.3, the algorithm receives an inverted index I, a query point q, the value of k, and a set of keywords K as input parameters and returns the top k objects by considering both keyword and spatial constraints.

For searching the shortest path between objects on spatial networks, Dijkstra's algorithm-based approaches [34, 42, 43] have been widely utilized in various applications. Given a source point and a group of destinations, the algorithm recursively expands the unvisited paths and records distances of intermediate nodes. During the search, a distance record of a node will be updated if there is a shorter path than the present one. Such a process is continued until all the destinations have arrived, and the distances of all other possible paths are longer than their current distances. In addition, a solution named Incremental Network Expansion (INE) is presented in [87] by extending Dijkstra's algorithm to compute k nearest neighbors in a network space. Specifically, INE first locates the network segment e_i, which covers the query point q, and retrieves all objects on e_i. If any object p_i is found on e_i, p_i will be inserted into

the result set. Furthermore, the endpoint of e_i, which is closer to q, will be expanded while the second endpoint of e_i will be placed in a priority queue Q_p. INE repeats the process by iteratively expanding the first node in Q_p and inserting newly discovered nodes into Q_p until k objects are retrieved.

We develop a Network Expansion-based SKkNN (NEkNN) solution by leveraging INE. There are two main steps in the NEkNN algorithm. The first step is to filter out objects which do not match any user specified keywords by employing Algorithm 8. Then, we mark all the objects in mergeList in the spatial network as candidates (e.g., set a bit of these points of interest). The next step is to expand the network from q with INE and the ranking function (Section 6.1.2). When an object p_i is discovered, NEkNN checks whether p_i is a candidate object or not. If p_i is a candidate object, NEkNN calculates its ranking score s by executing the ranking function (otherwise the algorithm ignores p_i). Meanwhile, NEkNN keeps a result set $\mathbb{R}$ which is sorted in descending order based on the ranking score. If $\mathbb{R}$ has fewer than k objects and p_i is a candidate object, p_i is inserted into $\mathbb{R}$. Otherwise, NEkNN compares the ranking score of p_i with the last object p_j in $\mathbb{R}$. p_j will be replaced by p_i if $p_i.s > p_j.s$. In addition, when $|\mathbb{R}| \geq k$, NEkNN calculates ranking scores for network nodes as well by assuming that they match all the search keywords to restrict the search space. In other words, any spatial object p_i, which is further away from q than a network node n_i, must have a lower ranking score than n_i even if p_i matches all the search keywords. Consequently, NEkNN iterates the search process until $\mathbb{R}$ contains k objects and the next network node to be expanded in Q_p has an equal or lower ranking score than the last object in $\mathbb{R}$.

By employing the data set in Table 6.1 and spatial network in Figure 6.1, we demonstrate an example to retrieve the two nearest hotels that have the amenities, “Internet" and “Pets Allowed" from q with NEkNN. We assume that all keywords have identical weight and the values of θ_1 and θ_2 are 0.5 and 0.5, respectively. First, NEkNN executes Algorithm 8 and marks candidate hotels H_1, H_3, H_5, and H_6 on the network. Then, NEkNN locates the segment n_5n_7 that covers q. Since no hotel is covered by n_5n_7, the node (n_5) closer

Algorithm 9 NEkNN(I, q, k, K)

1. mergeList = KeywordMatch(I, K)
2. **if** mergeList $== \emptyset$ **then**
3.   **return** $\emptyset$
4. **end if**
5. mark each object in mergeList in the spatial network
6. $n_i n_j$ = the segment covers q
7. **if** $n_i n_j$ covers candidate objects **then**
8.   calculate their ranking scores and insert them into $\mathbb{R}$
9. **end if**
10. $\{p_1, \ldots, p_k\}$ are the top k objects in $\mathbb{R}$ sorted in descending order of their ranking scores
11. $s_{min} = p_k.s$ // if $p_k = \emptyset$, $s_{min} = -\infty$
12. $Q_p = <(n_i, D_n(q, n_i)), (n_j, D_n(q, n_j))>$ // sorted in ascending order of D_n
13. **repeat**
14.   de-queue the first node n_f in Q_p
15.   **if** $|\mathbb{R}| \geq k$ **then**
16.   calculate $n_f.s$ by assuming that n_f fully matches K
17.   **else**
18. $n_f.s = 0$
19.   **end if**
20.   **for** each non-visited adjacent node n_x of n_f **do**
21.   search $n_x n_f$
22.   **if** $n_x n_f$ covers candidate objects **then**
23.   **for** each candidate object p_i **do**
24.   **if** $|\mathbb{R}| < k$ **then**
25. $\mathbb{R} \cup p_i$
26.   **else**
27.   **if** $p_i.s > p_k.s$ **then**
28.   replace p_k by p_i
29.   **end if**
30.   **end if**
31.   **end for**
32.   **end if**
33.   en-queue $(n_x, D_n(q, n_x))$
34.   **end for**
35. $s_{min} = p_k.s$ // if $p_k = \emptyset$, $s_{min} = -\infty$
36. **until** $s_{min} < n_f.s$ **or** $|\mathbb{R}| < k$
37. **return** $\mathbb{R}$

to q is expanded and the other endpoint n_7 is placed in Q_p. On $n_2 n_5$, H_1 is discovered and inserted into $\mathbb{R}$ with $s = -0.5$. Meanwhile, n_2 is inserted to $Q_p = <(n_7, 3), (n_2, 4)>$. The expansion of n_7 reaches n_9 and $Q_p = <(n_2, 4), (n_9, 6)>$. Next, the expansion of n_2

reaches n_1 and n_3, after which $Q_p = <(n_9, 6), (n_1, 11), (n_3, 13)>$ and H_5 is found on n_2n_3. Afterward, H_5 is inserted into $\mathbb{R}$ with $s = -3.5$. Subsequently, n_9 is expanded and $Q_p = <(n_1, 11), (n_3, 13), (n_8, 13), (n_{10}, 14)>$. The ranking score of the next node in Q_p (n_1) is -4.5; the algorithm terminates because $\mathbb{R}$ contains two hotels and $H_5.s > n_1.s$. The complete algorithm of NEkNN is formalized in Algorithm 9.

6.2.4 The Voronoi Diagram-Based SKkNN Query Algorithm

Although NEkNN is able to restrict the search space and retrieve top k objects based on their ranking scores, the main limitation of NEkNN is that it has to explore a large portion of the network when candidate objects are not densely distributed in the network. Therefore, we propose a Voronoi diagram-based SKkNN (VDkNN) solution by leveraging the network Voronoi diagram (NVD) [65] to improve performance. In order to be independent of the density and distribution of candidate objects, we first partition the spatial network into small regions by generating a network Voronoi diagram over all the spatial objects (points of interest). Each cell of the NVD is centered by one spatial object and contains the nodes that are closest to that object in network distance. Afterward for each NVD cell, we pre-compute the distances between all the edges of the cell to its center as well as the distances only across the border points of the adjacent cells. Consequently, for a new cell, we can quickly extend the searched region to the border points without expanding all the internal network segments.

With the NVD of the search space, for a SKkNN query, VDkNN first filters out unqualified objects with Algorithm 8 and marks all the objects in mergeList in the NVD as candidates. Then, VDkNN finds the network Voronoi polygon NVP(p_i) that contains q where p_i is the generator of the polygon. This step can be accomplished by employing a spatial index (e.g., the R-tree), which is generated based on the NVD cells. Next, we verify that p_i is a candidate object. If p_i is a candidate object, VDkNN calculates its ranking score by running the ranking function. In addition, VDkNN maintains a result set $\mathbb{R}$ which

is sorted in descending order according to the ranking score. When $\mathbb{R}$ contains fewer than k objects, newly discovered candidate objects are inserted into $\mathbb{R}$. However, if $\mathbb{R}$ already includes k objects, VDkNN replaces the kth object p_k of $\mathbb{R}$ when a newly retrieved candidate object has a higher s than p_k. Also, VDkNN keeps a queue Q_n which stores the neighbors (adjacent cells) of p_i and a set $\mathbb{E}$ which consists of all the searched cells (i.e., $\mathbb{E}$ covers the current explored region).

Subsequently, VDkNN searches the adjacent cells of $\mathbb{E}$ (i.e., NVP(p_i)) stored in Q_n for the next candidate object. Every time after a cell NVP(p_j) been explored, the neighboring generators of p_j are unioned with Q_n, NVP(p_j) is unioned with $\mathbb{E}$, and $\mathbb{R}$ is updated according to the aforementioned rules if p_j is a candidate object. Moreover, when $|\mathbb{R}| \geq k$, VDkNN calculates the ranking score of all the border points of the current explored region by assuming that they match all the search keywords to restrict the search space. VDkNN iterates the search process until $\mathbb{R}$ contains k objects and the ranking scores of all the border points of $\mathbb{E}$ are equal or worse than the ranking score of the kth object in $\mathbb{R}$ (i.e., there will not be any changes in $\mathbb{R}$ even if we search further).

Figure 6.6 illustrates an example of retrieving two nearest points of interest (POI) which match keywords in K from q with VDkNN. First, VDkNN executes Algorithm 8 and marks candidate POIs on the NVD. Then, VDkNN locates the network Voronoi polygon, NVP(p_1), which contains q. Next, VDkNN verifies that p_1 is a candidate POI and inserts the neighboring generators, p_2, p_5, p_6, p_7, p_8, p_9, and p_{10} into Q_n. Also, $\mathbb{E}$ covers NVP(p_1). Afterward VDkNN searches the objects in Q_n for the next candidate POI. Assume that NVP(p_2) is the second explored NVP and both p_1 and p_2 are candidate POIs. Then, $\mathbb{R}$ contains p_1 and p_2, $\mathbb{E}$ covers NVP(p_1) and NVP(p_2) (the shaded region in Figure 6.6), and Q_n comprises nine generators (p_3 to p_{11}). Since $\mathbb{R}$ covers two POIs, VDkNN computes the ranking score of all the border points of $\mathbb{E}$ (b_1 to b_{12}) by assuming that they match all the search keywords in K. Here we suppose that $s_{min} > b_{max}$ and the algorithm terminates. The complete algorithm of VDkNN is formalized in Algorithm 10.

Algorithm 10 VDkNN(I, q, k, K)

1. Generate a NVD based on P
2. mergeList = KeywordMatch(I, K)
3. **if** mergeList $== \emptyset$ **then**
4.   **return** $\emptyset$
5. **end if**
6. mark each object in mergeList in the NVD
7. NVP(p_i) = the NVP covers q
8. en-queue the neighboring generators of p_i into Q_n
9. $\mathbb{E} \cup$ NVP(p_i)
10. **if** p_i is a candidate object **then**
11.   calculate s of p_i and insert p_i into $\mathbb{R}$
12. **end if**
13. $\{p_1, \ldots, p_k\}$ are the top k objects in $\mathbb{R}$ sorted in descending order of their ranking scores
14. $s_{min} = p_k.s$ // if $p_k = \emptyset$, $s_{min} = -\infty$
15. **if** $|\mathbb{R}| \geq k$ **then**
16.   calculate s of all the border nodes of $\mathbb{E}$ by assuming that they fully match K
17. b_{max} = the largest s of all the border nodes of $\mathbb{E}$
18. **else**
19. $b_{max} = 0$
20. **end if**
21. **while** $s_{min} < b_{max}$ **or** $|\mathbb{R}| < k$ **do**
22. p_j = de-queue Q_n
23.   **if** p_j is a candidate object **then**
24.   **if** $|\mathbb{R}| < k$ **then**
25. $\mathbb{R} \cup p_j$
26.   **else**
27.   **if** $p_j.s > p_k.s$ **then**
28.   replace p_k by p_j
29.   **end if**
30.   **end if**
31.   **end if**
32. $Q_n \cup$ all the neighboring generators of p_j
33. $\mathbb{E} \cup$ NVP(p_j)
34. $s_{min} = p_k.s$ // if $p_k = \emptyset$, $s_{min} = -\infty$
35.   **if** $|\mathbb{R}| \geq k$ **then**
36.   calculate s of all the border nodes of $\mathbb{E}$ by assuming that they fully match K
37. b_{max} = the largest s of all the border nodes of $\mathbb{E}$
38.   **else**
39. $b_{max} = 0$
40.   **end if**
41. **end while**
42. **return** $\mathbb{R}$

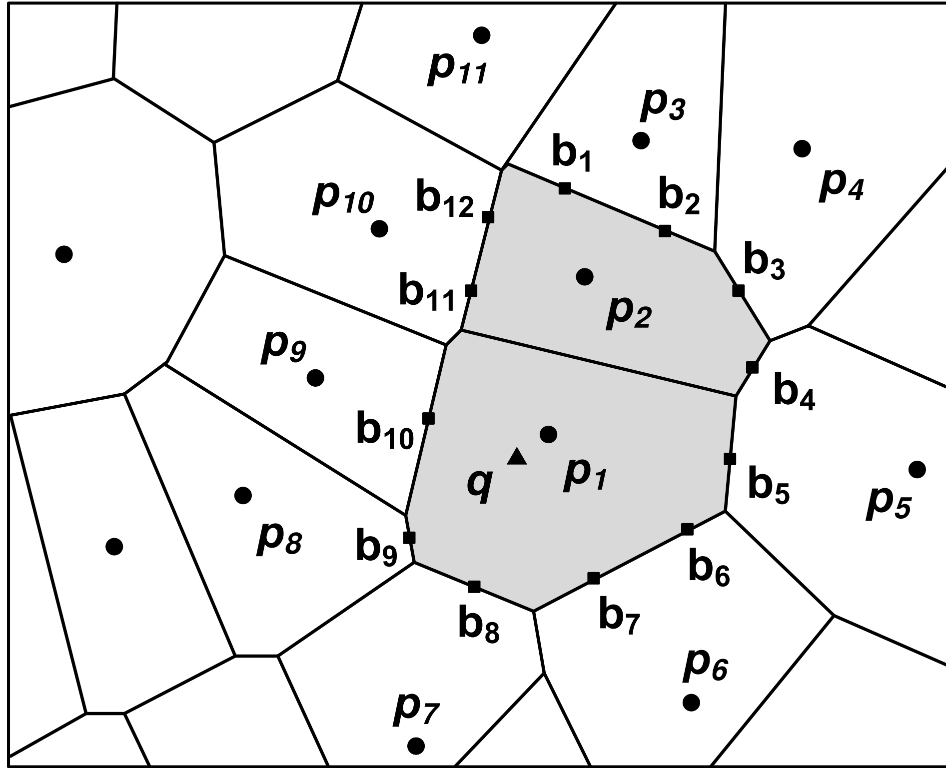


Figure 6.6: A VDkNN query example.

6.2.5 The Spatial Keyword Range Query Algorithm

As defined in Section 6.1.4, given a query point q, a search range r and a set of keywords K, SKR query is to retrieve all the objects which fully match all the keywords within r. SKR query first calculates the keyword relevancy of objects by utilizing KeywordMatch (Algorithm 8). Then, it retrieves objects which fully match all the given keywords and stores the qualified objects in $\mathbb{C}$. Afterward, it calls Dijkstra's algorithm for calculating distances from q to all the candidate objects. Finally, SKR query removes objects which are out of the search range from $\mathbb{R}$. The complete algorithm of SKR query is illustrated in Algorithm 11.

The worst-case running time of Algorithms 2, 3, and 4 on a spatial network with a set of nodes N is $O(|K| * |P| + |N|^2)$ by considering both the keyword match and spatial network search subroutines.

Algorithm 11 SKR(I, q, r, K)

```
1.  mergeList = KeywordMatch(I, K)
2.  for each object o in mergeList do
3.      if o.occurrence == |K| then
4.          ℂ.append(o)
5.      end if
6.  end for
7.  if (|ℂ| != 0 ) then
8.      ℝ = ShortestPath(q, ℂ)
9.  end if
10. filter out objects beyond r in ℝ
11. return ℝ
```

6.3 Experimental Validation

In this subsection, we evaluate the performance of our spatial keyword query solutions with both real-world and synthetic data sets. We implemented the proposed algorithms and related experimental components in C++. The inverted index structure was loaded into the main memory during the execution of simulations. All the experiments were conducted on an Ubuntu Linux server equipped with an Intel Xeon 2.4GHz processor and 2GB memory. More details of the simulation environment are shown in Table 6.3. All simulation results were recorded after the system model reached a steady state.

Table 6.3: Simulator configurations.

	Configurations
Hardware	Intel Xeon X3430 2.4GHz processor 2GB RAM 160GB SATA disk
Software	Ubuntu 10.04 Linux kernel 2.6.23 g++ 4.4.3

6.3.1 Data Sets

In our experiments, a real-world data set was downloaded from edigitalz[1], which provides a wide range of general data sets for free. We retrieved 9,483 restaurants in the state of California and utilized their menus (e.g., pizza, hamburger, steak, etc.) and cuisine (e.g., American, Chinese, French, etc.) as keywords for searches. The data sets of road networks in both the state of California (containing 21,692 roads and 21,047 intersections) and the continental United States (containing 179,178 roads and 175,812 intersections) were downloaded from the US Census Bureau (TIGER/Line Shapefiles)[2].

For the synthetic data set, we generated around 160,000 restaurants, of which the density follows the real-world data set in order to investigate the scalability of our algorithms. In addition, each restaurant has a similar number of keywords to the real-world data set. The network of the continental United States is used with the synthetic data set. Table 6.4 summarizes the numbers of the two data sets.

Table 6.4: Data sets employed in experiments.

Data Sets	Total Number of Restaurants	Total Number of Keywords	Total Number of Roads	Total Number of Intersections
Real	9,483	34,091	21,692	21,047
Synt.	160,000	575,200	179,178	175,812

Table 6.5 displays the default values of parameters in our experiments. We varied an essential parameter in each experiment set in order to evaluate its impact on the performance of the proposed algorithms. Other parameters were kept constant during all the experiments in the same set. The default values of parameters are used in experiments if we do not explicitly specify other values. The selection of θ_1 and θ_2 values depends on preference for keyword relevancy and distance of users. We fixed the ratio of θ_1 to θ_2 (1/20) in all the experiments.

[1] http://www.edigitalz.com/

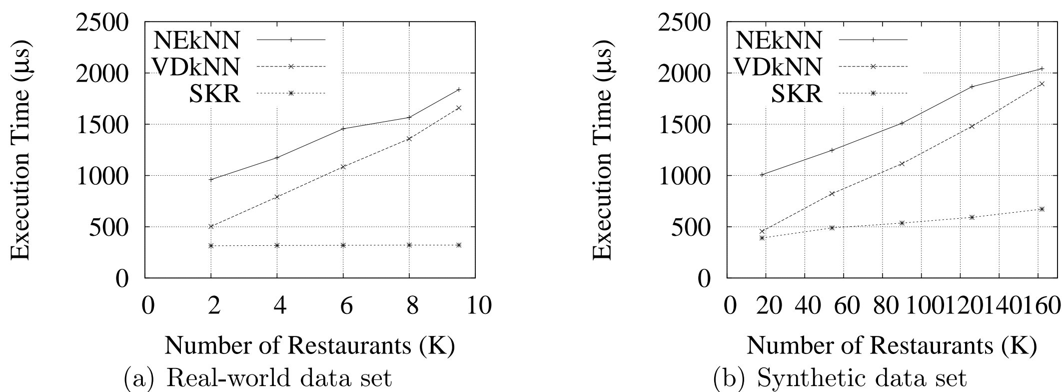
[2] http://www.census.gov/geo/www/tiger/

Table 6.5: Default values of parameters used in experiments.

Parameters	Default Values
θ_1	0.04762
θ_2	0.95238
k	5
$\|K\|$	2
r	20 km
q	randomly selected

6.3.2 Data Set Size Experiment

In this experiment, we evaluate NEkNN, VDkNN, and SKR queries with various data set sizes. The main purpose of this experiment is to analyze the influence of different data set sizes on query execution time. For both real-world and synthetic data sets, we generate five subsets of restaurants with an increasing number of data objects. The number of restaurants in consecutive subsets is increased by 2,000 for real-world data sets and 35,000 for synthetic data sets. Objects in smaller subsets are included in bigger ones.



(a) Real-world data set (b) Synthetic data set

Figure 6.7: Execution times of NEkNN and VDkNN queries as a function of data set size.

The results of real-world and synthetic data sets are demonstrated in Figure 6.7(a) and Figure 6.7(b), respectively. VDkNN always outperforms NEkNN with the default parameters in all the experiments. From Figure 6.7, we observe that the execution time of most queries increases linearly with the increment of data set size. The reason is that more POIs and

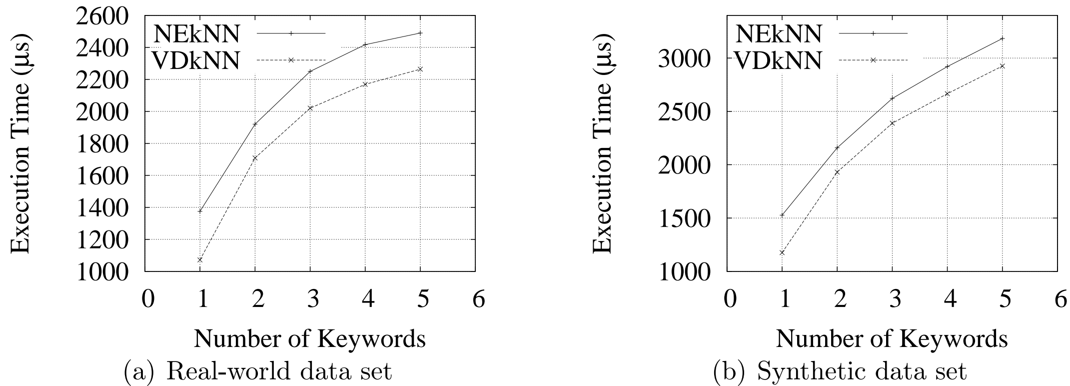
keywords have to be processed in these queries. In addition, the difference of execution time between NEkNN and VDkNN queries decreases gradually as data set size grows. In other words, the time costs of these two solutions become close with a larger data set. The reason that faster performance degradation is caused in VDkNN is that it has an extra overhead of searching on Voronoi diagrams in addition to the cost of processing POIs, which is suffered by both solutions. The higher density of POIs on spatial networks, the more border nodes are generated in Voronoi polygons. Hence, VDkNN has to spend more time on border node related calculation when it tests the stopping condition.

Another observation is that queries run faster on a bigger synthetic data set (e.g., 20,000 data objects) than a smaller real-world data set (e.g., 9,483 data objects). The reason is that the density of POIs is a dominant factor in these queries. Although there are more POIs involved in the keyword-match process in the synthetic data set, there are fewer candidates which are discovered in the search area due to lower POI density.

6.3.3 Number of Keywords Experiment

The number of keywords is an essential parameter of both NEkNN and VDkNN queries. In order to investigate the impact of the number of keywords on query performance, we vary the number of specified keywords on both data sets. We conduct experiments from queries with one keyword to ones with five keywords by adding a new keyword after each experiment. Figure 6.8 shows that the execution time of queries increases when more keywords are specified by users. In order to retrieve partially matched query results, POIs that match any of the given keywords have to be taken into account. Consequently, more keywords will increase the number of POIs to be processed in the keyword match and query evaluation processes.

The difference in execution time between NEkNN and VDkNN remains nearly constant in all queries. As the number of keywords becomes larger, more POIs are considered in the keyword match process in both solutions. Moreover, varying the number of keywords does



(a) Real-world data set

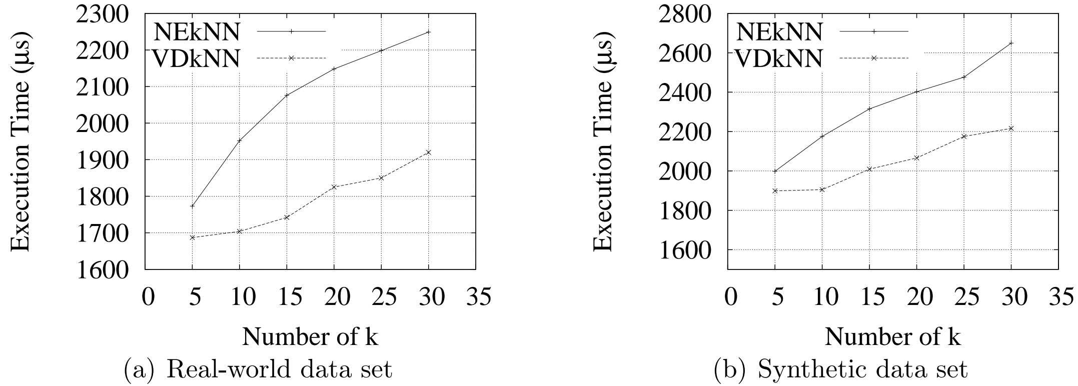
(b) Synthetic data set

Figure 6.8: Execution times of NEkNN and VDkNN queries as a function of number of keywords.

not directly enlarge or shrink the search area of both methods (i.e., the score of a POI is determined by the ranking function). Therefore, no apparent change of the difference in query performance between the two methods is observed. However, VDkNN always exceeds NEkNN in execution time in this experiment.

6.3.4 Number of k Experiment

Next, we evaluate the impact of k on the performance of NEkNN and VDkNN queries with the two data sets. We vary the value of k from 5 to 30 with an increment of five. Figure 6.9 illustrates that the execution time of queries increases as the number of k becomes larger. With both NEkNN and VDkNN, a larger search area has to be processed in order to retrieve more qualified results when we increase the k value. The performance difference between NEkNN and VDkNN becomes clear when k increases. Such a difference is proportional to the k value if POIs and networks are equally distributed. Apparently, given specific keywords, the cost of the keyword match process of NEkNN and VDkNN is identical. Therefore, the performance gain of VDkNN queries is from searches on the NVD where VDkNN can retrieve the top k candidates faster than NEkNN. When k increases, the search area is enlarged correspondingly and VDkNN is able to achieve more performance gains in the expanded search region.

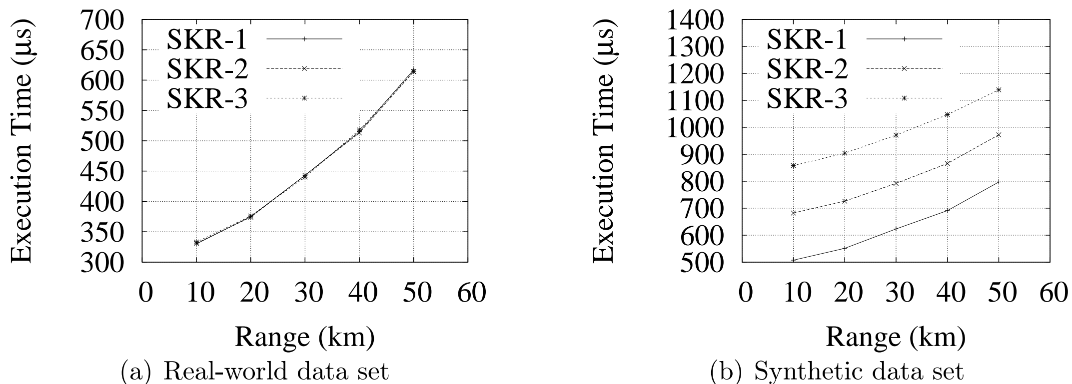


(a) Real-world data set (b) Synthetic data set

Figure 6.9: Execution times of NEkNN and VDkNN queries as a function of number of k.

6.3.5 Query Range Experiment

We examine the effect that varying the query range would have on the performance of SKR queries. In the experiments, SKR queries with various query ranges are evaluated in three different cases, which are queries with one (SKR-1), two (SKR-2), and three (SKR-3) keywords. Both Figures 6.10(a) and 6.10(b) illustrate that the execution time of queries grows exponentially with increasing query range. This is because the search area expands equally in all directions.



(a) Real-world data set (b) Synthetic data set

Figure 6.10: Execution time of SKR queries as a function of query range.

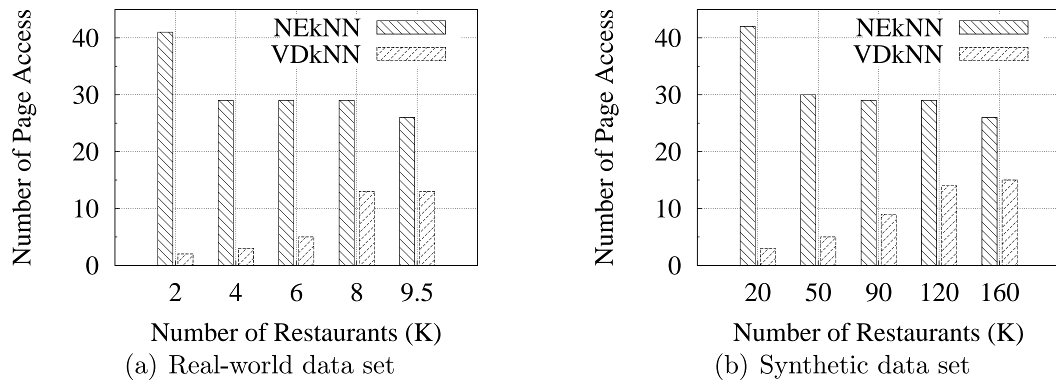
Interestingly, the execution time of the queries on real data sets are very close. Two factors mainly affect SKR. The first one is the number of POIs involved in the keyword match step. More POIs will be processed if more keywords are given. Furthermore, POIs that are

fully keyword-matched are qualified candidates in SKR and a large number of partially keyword-matched POIs are filtered out by KCF. Consequently, fewer candidate POIs need to be processed in the range search phase. The two factors offset each other in range queries with relatively small search distances and data sets. However, when large amounts of POIs are searched with SKR, the overhead of the keyword match process becomes dominant in execution time. As shown in Figure 6.10(b), SKR-1 becomes the best and SKR-3 is the worst.

6.3.6 Page Access Experiment

Finally, we evaluate the number of page accesses by our proposed solutions. In these tests, we mainly focus on the comparison of NEkNN and VDkNN queries. Given a specific query, both solutions have the keyword match process. An identical number of keywords are retrieved from data sets. In addition, the POIs detected by NEkNN are required to be processed in VDkNN as well, and vice versa. The only difference is that NEkNN searches on spatial networks, whereas VDkNN explores on NVDs. Therefore, we evaluate the page access regarding network retrieval in these experiments. The page size is set to be 4 KB. The size of intersections or border nodes is 20 Bytes, containing their identifiers and coordinates. The road segments have a size of 20 Bytes as well, encompassing their identifiers, identifiers of two endpoints, and the length of the road segment. A single page can accommodate either 200 nodes or road segments. The nodes and segments are stored continuously in pages. During a query process, each page is loaded only once.

Figure 6.11(a) and Figure 6.11(b) display the number of page accesses of NEkNN and VDkNN queries in real-world and synthetic data sets, respectively. The trend shared by the two figures is that as the data set size grows, the number of page accesses decreases in NEkNN, whereas it increases in VDkNN. The main reason is that NEkNN searches in a smaller area for qualified results in a larger dataset. Fewer intersections and road segments are retrieved by NEkNN. On the other hand, NVD becomes more complex when more



(a) Real-world data set

(b) Synthetic data set

Figure 6.11: Page Access evaluation with different data set sizes.

border nodes and connections between borders are generated. Therefore, more page access is required in VDkNN with a larger POI data set.

Chapter 7

Conclusion and Future Work

In this dissertation, we have demonstrated our novel design of frameworks for context-based file systems, offloading application development, and advanced solutions to Multi-Criteria Optimal Location Query and Spatial Keyword Query. This chapter concludes the dissertation study by summarizing the contributions and future work.

7.1 Framework for Context-Based File Systems

We present a general informed-based framework, Frog, for context-based file systems, where contexts are encapsulated in views. Frog integrates context-specific solutions that may conflict with each other in terms of metadata management, physical data organization, and I/O operations. We show the generality, transparency, diversity, and flexibility of Frog by implementing two Bi-Context File Systems, namely, BAVFS and BHVFS. In the two case studies, we first demonstrate that BAVFS optimizes performance of sequential and random reads on small files by the virtue of dual-mode prefetching. Then, we illustrate that BHVFS speed up random reads and writes by incorporating the *update-in-place* and *update-out-of-place* strategies. Our experimental results show that the benefits gained from context-based file systems far outweigh the overhead induced by creating and maintaining duplications for multiple views.

A few open issues in Frog and context-based file systems will be addressed in our future work. Duplicating and managing metadata (potentially physical data) is the fundamental idea behind the design of Frog. Creating data replicas can substantially improve data reliability; thus, it is intriguing to quantitatively study the reliability impacts on context-based file systems. Moreover, managing data in multiple views is a challenge in disk scheduling.

Disk scheduling policies applied in one view may either positively or negatively affect disk scheduling in another view. We will investigate disk scheduling optimization in a few particular contexts. Last, but not least, we plan to design a view-allocation mechanism that will make a tradeoff between performance and space effectiveness in a native context-based file system.

7.2 Offloading Framework

The emergence of active storage coupled with computation capability inspires us to offload I/O-bound modules of data-intensive applications to active storage nodes in a cluster computing system. In this study, we proposed a programming framework - ORCA - to automatically offload I/O-bound modules of applications to storage nodes in a cluster. The ORCA offloading framework handles configurations, execution-path control, offloading executable code, and data sharing. An ORCA application programming interface (API) and a run-time system in the framework allow programmers without any I/O offloading experience to easily write new I/O-bound modules or partition existing code to run efficiently on clusters.

The proposed ORCA framework can achieve the following two objectives for data-intensive applications running in both homogeneous and heterogeneous clusters. First, our ORCA framework accelerates data-intensive applications by allocating I/O-bound modules to active storage nodes in clusters. Second, ORCA can significantly reduce network burden imposed by transferring massive amounts of data from storage nodes to computing nodes.

In this work, we pay attention to offloading domains, each of which contains a pair of a CPU-bound module and an I/O-bound module. We introduce the offloading domain as an important concept, because this pair structure is a simple yet power model representing a wide range of data-intensive applications. For future research directions, this offloading-domain model will be extended to a multi-offloading-domain model in which multiple offloading domains can be properly coordinated. In light of this new model, we will

upgrade the offloading management in the ORCA framework. In addition, we plan to implement a dispatch management module to allocate I/O-bound processes to appropriate storage nodes. The offloading and dispatch management modules will address the challenge of how to control multiple collaborative offloading domains.

Another intriguing research issue is taking energy consumption into account when applying ORCA to offload I/O-bound modules on a data-redundancy cluster. Modern data centres often use data redundancy techniques, generating and keeping several copies of data to provide good services in terms of data throughput and availability. A number of energy-saving approaches are proposed to balance energy consumption and I/O performance of cluster storage systems. In a future study, we will investigate the trade-off issue among energy consumption, performance, and availability when we deploy the ORCA framework in a cluster storage system employing data-redundancy techniques.

7.3 MOLQ Evaluation

We formulated a novel optimal location selection problem. Except for designing two straightforward approaches that sequentially scan all object combinations and possible locations, we propose an MOVD-based approach (RRB) that efficiently answers the query. Moreover, in order to minimize the costs induced by region overlapping, we propose the MBRB approach, in which MBRs are used as the boundaries of OVRs, since overlapping two rectangles is much cheaper than overlapping two arbitrary regions. In addition, a cost-bound iterative approach is proposed to efficiently process large number of Fermat-Weber problems. We demonstrate the excellent performance of the proposed approaches through extensive simulations.

For integrity consideration, we plan to create and evaluate an inverse operation "$-$" that removes an MOVD from another MOVD. Moreover, we will evaluate "+" and "$-$" operations by varying the object weights. Overlapping weighted Voronoi diagrams is more

expensive than overlapping two regular diagrams because of difficulty in representing Voronoi regions.

7.4 Spatial Keyword Query

Geographic information systems are becoming increasingly sophisticated, and spatial keyword search represents an important class of queries. Most existing solutions for evaluating spatial keyword queries are based on Euclidean distance and cannot provide partially matched results. In this research, we introduce efficient techniques to answer spatial keyword k nearest neighbor and spatial keyword range queries on spatial networks. We demonstrate the excellent performance of the proposed algorithms through extensive simulations.

For future work, we plan to extend our spatial keyword query evaluation framework to support other common spatial query types such as spatial join, reverse nearest neighbor, spatial skyline, etc.

Bibliography

[1] BRTFS: The Linux B-tree Filesystem. http://domino.watson.ibm.com/library/CyberDig.nsf/papers/6E1C5B6A1B6EDD9885257A38006B6130/$File/rj10501.pdf.

[2] Gnu core utilities. http://www.gnu.org/software/coreutils/.

[3] Gnu Grep. http://www.gnu.org/software/grep/.

[4] hdparm. http://en.wikipedia.org/wiki/Hdparm.

[5] Oracle 11g release 1 rac on linux using nfs. http://www.oracle-base.com/articles/11g/OracleDB11gR1RACInstallationOnLinuxUsingNFS.php.

[6] Postgresql. http://www.postgresql.org/.

[7] Seagate product manual of barracuda 7200.12 serial ata. http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.12/100529369b.pdf.

[8] The IBM JFS project. http://www.ibm.com/developerworks/wikis/display/WikiPtype/JFS.

[9] The ReiserFS project. http://http://marc.info/?l=reiserfs-devel.

[10] The SGI XFS project. http://oss.sgi.com/projects/xfs/.

[11] WD1600AAJS specification. http://wdc.custhelp.com/app/answers/detail/search/1/a_id/1400#.

[12] Wd5000aaks specification. http://www.wdc.com/en/products/products.aspx?id=110.

[13] Apache hadoop. http://lucene.apache.org/hadoop/, 2006.

[14] Corba. http://www.corba.org/, 2010.

[15] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2:922–933, August 2009.

[16] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, Oct. 1998.

[17] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3(3), Oct. 2007.

[18] F. Anton, D. Mioc, and C. M. Gold. Dynamic additively weighted voronoi diagrams made easy. In *CCCG*, 1998.

[19] P. F. Ash and E. D. Bolker. Generalized dirichlet tessellations. *Geometriae Dedicata*, 20:209–243, 1986.

[20] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition.* Pearson Education Ltd., Harlow, England, 2011.

[21] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[22] J.-D. Boissonnat and C. Delage. Convex Hull and Voronoi Diagram of Additively Weighted Points. In *ESA*, pages 367–378, 2005.

[23] J. Bonwick. ZFS: The last word in file systems. `http://www.opensolaris.org/os/community/zfs/docs/zfs_last.pdf`.

[24] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, may 2009.

[25] A. Cary, O. Wolfson, and N. Rishe. Efficient and Scalable Method for Processing Top-k Spatial Boolean Queries. In *SSDBM*, pages 87–95, 2010.

[26] R. Chandrasekaran and A. Tamir. Algebraic Optimization: The Fermat-Weber Location Problem. *Math. Program.*, 46:219–224, 1990.

[27] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li. Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. *VLDB J.*, 21(1):69–95, 2012.

[28] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD Conference*, pages 277–288, 2006.

[29] S. Chiu, W.-k. Liao, and A. Choudhary. Design and evaluation of distributed smart disk architecture for i/o-intensive workloads. ICCS'03, pages 230–241, Berlin, Heidelberg, 2003. Springer-Verlag.

[30] W. Chu, W. Li, T. Mo, and Z. Wu. A Context-Source Abstraction Layer for Context-aware Middleware. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 1064 –1065, april 2011.

[31] G. Cong, C. S. Jensen, and D. Wu. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *PVLDB*, 2(1):337–348, 2009.

[32] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[33] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[34] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[35] P. Dong. Generating and updating multiplicatively weighted Voronoi diagrams for point, line and polygon features in GIS. *Computers & Geosciences*, 34(4):411–421, 2008.

[36] D. H. C. Du. Intelligent storage for information retrieval. NWESP '05, pages 214–, Washington, DC, USA, 2005. IEEE Computer Society.

[37] Y. Du, D. Zhang, and T. Xia. The Optimal-Location Query. In *SSTD*, pages 163–180, 2005.

[38] F. E.J., F. K., R. K., and N. J. Active Storage Processing in a Parallel File System. *In Proc. of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, 2006.

[39] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing-a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, Sept. 1979.

[40] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword Search on Spatial Databases. In *ICDE*, pages 656–665, 2008.

[41] B. G. Fitch, A. Rayshubskiy, M. C. Pitman, T. J. C. Ward, and R. S. Germain. Using the active storage fabrics model to address petascale storage challenges. PDSW '09, pages 47–54, New York, NY, USA, 2009. ACM.

[42] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[43] L. Fu, D. Sun, and L. Rilett. Heuristic shortest path algorithms for transportation applications: State of the art. In *Computers and Operations Research*, volume 33, pages 3324–3343, 2006.

[44] M. Gahegan and I. Lee. Data structures and algorithms to support interactive spatial analysis using dynamic Voronoi diagrams. *Computers, Environment and Urban Systems*, 24(6):509–537, 2000.

[45] J. F. Gantz. The Diverse and Exploding Digital Universe. *IDC white paper*, 2:1–16, 2008.

[46] Y. Gao, B. Zheng, G. Chen, and Q. Li. Optimal-Location-Selection Query Processing in Spatial Databases. *IEEE Trans. Knowl. Data Eng.*, 21(8):1162–1177, 2009.

[47] N. H. Gehani, H. V. Jagadish, and W. D. Roome. OdeFS: A File System Interface to an Object-Oriented Database. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 249–260, San Francisco, CA, USA, 1994.

[48] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.

[49] T. P. G. D. Group. Postgresql developer's guide. `http://www.postgresql.org/docs/9.0/interactive/index.html`.

[50] T. Gu, H. K. Pung, and D. Q. Zhang. A middleware for building context-aware mobile services. In *Proceedings of IEEE Vehicular Technology Conference (VTC-Spring 2004)*, volume 5, pages 2656–2660, Milan, Italy, 17-19 May 2004.

[51] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[52] J. B. S. Haldane. Note on the Median of a Multivariate Distribution. *Biometrika*, 35:414–415, 1948.

[53] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing Spatial-Keyword (SK) Queries in Geographic Information Retrieval (GIR) Systems. In *SSDBM*, page 16, 2007.

[54] M. Henning. The rise and fall of corba. *Queue*, 4(5), 2006.

[55] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[56] H. Hu, D. L. Lee, and V. C. S. Lee. Distance Indexing on Road Networks. In *VLDB*, pages 894–905, 2006.

[57] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-level isolation and recovery with solitude. *SIGOPS Oper. Syst. Rev.*, 42(4):95–107, Apr. 2008.

[58] G. Jalal and J. Krarup. Geometrical Solution to the Fermat Problem with Arbitrary Weights. *Annals OR*, 123(1-4):67–104, 2003.

[59] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest Neighbor Queries in Road Networks. In *GIS*, pages 1–8, 2003.

[60] M. I. Karavelas and M. Yvinec. Dynamic Additively Weighted Voronoi Diagrams in 2D. In *ESA*, pages 586–598, 2002.

[61] R. Kassick, F. Boito, and P. Navaux. Impact of i/o coordination on a nfs-based parallel file system with dynamic reconfiguration. pages 199–206, oct. 2010.

[62] J. Katcher. PostMark: a New FileSystem Benchmark. *Technical Report TR3022*, pages 1–8, 1997.

[63] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, Sept. 1998.

[64] T. Kojm. ClamAV. `http://www.clamav.net`, 2004.

[65] M. R. Kolahdouzan and C. Shahabi. Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases. In *VLDB*, pages 840–851, 2004.

[66] F. Korn and S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In *SIGMOD Conference*, pages 201–212, 2000.

[67] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse Nearest Neighbor Aggregates Over Data Streams. In *VLDB*, pages 814–825, 2002.

[68] D. Kotz and C. S. Ellis. Practical prefetching techniques for parallel file systems. In *Proceedings of the first international conference on Parallel and distributed information systems*, PDIS '91, pages 182–189, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[69] A. Krause, A. Smailagic, and D. P. Siewiorek. Context-aware mobile computing: learning context- dependent personal preferences from a wearable sensor array. *Mobile Computing, IEEE Transactions on*, 5(2):113 – 127, feb. 2006.

[70] W.-S. Ku, R. Zimmermann, H. Wang, and T. Nguyen. Annatto: Adaptive nearest neighbor queries in travel time networks. In *MDM*, page 50, 2006.

[71] W.-S. Ku, R. Zimmermann, H. Wang, and C.-N. Wan. Adaptive nearest neighbor queries in travel time networks. In *GIS*, pages 210–219, 2005.

[72] M. Lee, S. Min, C. Park, Y. Bae, H. Shin, and C. Kim. A dual-mode instruction prefetch scheme for improved worst case and average case program execution times. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 98 –105, dec 1993.

[73] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1):61–65, Mar. 2010.

[74] C. Li, K. Shen, and A. E. Papathanasiou. Competitive prefetching for concurrent sequential I/O. *SIGOPS Oper. Syst. Rev.*, 41(3):189–202, Mar. 2007.

[75] H. Lim, V. Kapoor, C. Wighe, and D. H.-C. Du. Active disk file system: A distributed, scalable file system. MSS '01, pages 101–, Washington, DC, USA, 2001. IEEE Computer Society.

[76] X. Ma and A. L. N. Reddy. MVSS: Multi-View Storage System. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 31–38, apr 2001.

[77] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval.* Cambridge University Press, 2008.

[78] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the 2007 Linux Symposium*, pages 21–33, June 2007.

[79] M. K. McKusick and G. R. Ganger. Soft updates: a technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, pages 24–24, Berkeley, CA, USA, 1999. USENIX Association.

[80] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2:181–197, August 1984.

[81] L. Mu. Polygon Characterization With the Multiplicatively Weighted Voronoi Diagram. *The Professional Geographer*, 56(2):223–239, 2004.

[82] K. Muller and J. Pasquale. A high performance multi-structured file system design. *Proceedings of the 13th ACM symposium on Operating systems principles*, 25(5):56–67, Sept. 1991.

[83] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* Probability and Statistics. Wiley, NYC, 2nd edition, 2000.

[84] R. B. R. P. H. Carns, W. B. Ligon III and R. Thakur. Pvfs: a parallel file system for linux clusters. *Proceedings of the 4th annual Linux Showcase and Conference*, pages 28–28, 2000.

[85] F. Pagliara, J. Preston, and D. Simmonds. Residential Location Choice: Models and Applications. 2010.

[86] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, Feb. 2000.

[87] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query Processing in Spatial Network Databases. In *VLDB*, pages 802–813, 2003.

[88] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. GIGA+: scalable directories for shared file systems. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07*, PDSW '07, pages 26–29, New York, NY, USA, 2007. ACM.

[89] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 17(3):109–116, June 1988.

[90] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *SIGOPS Oper. Syst. Rev.*, 29:79–95, Dec. 1995.

[91] J. Piernas, T. Cortes, and J. M. García. DualFS: a new journaling file system without meta-data duplication. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 137–146, New York, NY, USA, 2002. ACM.

[92] J. Piernas and J. Nieplocha. Efficient management of complex striped files in active storage. Euro-Par '08, pages 676–685, Berlin, Heidelberg, 2008. Springer-Verlag.

[93] J. Piernas, J. Nieplocha, and E. J. Felix. Evaluation of active storage strategies for the lustre parallel file system. SC '07, pages 28:1–28:10, New York, NY, USA, 2007. ACM.

[94] J. Qi, R. Zhang, L. Kulik, D. Lin, and Y. Xue. The Min-dist Location Selection Query. In *ICDE*, 2012.

[95] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Science, 2002.

[96] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[97] J. G. M. Robert F. Love and G. O. Wesolowsky. Facilities location, models and methods. 1988.

[98] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1(4):74–83, Oct. 2002.

[99] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10:26–52, February 1992.

[100] D. S. H. Rosenthal. Evolving the Vnode Interface. *USENIX Conference Proceedings*, pages 107–118, 1990.

[101] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *SIGMOD Conference*, pages 71–79, 1995.

[102] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD Conference*, pages 43–54, 2008.

[103] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation or the sun network filesystem. *In Proc. of USENIX Summer Technical Conf.*, 1985.

[104] J. Schindler, S. Shete, and K. A. Smith. Improving throughput for small disk requests with proximal I/O. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[105] P. Schwan. Lustre: Building a file system for 1000-node clusters. *In Proceedings of the 2003 Linux Symposium*, 2003.

[106] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: asynchronous meta-data protection in file systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[107] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice Hall, 2003.

[108] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 0:1–10, 2010.

[109] D. Siewiorek, A. Smailagic, J. Furukawa, A. Krause, N. Moraveji, K. Reiger, J. Shaffer, and F. L. Wong. SenSay: a context-aware mobile phone. In *Wearable Computers, 2003. Proceedings. Seventh IEEE International Symposium on*, pages 248 – 249, oct. 2003.

[110] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W.-K. Liao, and A. Choudhary. Enabling active storage on parallel i/o software stacks. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –12, may 2010.

[111] G. Soundararajan, M. Mihailescu, and C. Amza. Context-aware prefetching at the storage server. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 377–390, Berkeley, CA, USA, 2008. USENIX Association.

[112] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[113] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of Influence Sets in Frequently Updated Databases. In *VLDB*, pages 99–108, 2001.

[114] S. Sundararaman, L. Visampalli, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Refuse to crash with Re-FUSE. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 77–90, New York, NY, USA, 2011. ACM.

[115] M. Szeredi. File system in user space(FUSE). `http://fuse.sourgeforge.net`.

[116] A. S. Tanenbaum, J. N. Herder, and H. Bos. File size distribution on UNIX systems: then and now. *ACM SIGOPS Operating Systems Review*, 40:100–104, January 2006.

[117] Y. Tao, D. Papadias, and X. Lian. Reverse kNN Search in Arbitrary Dimensionality. In *VLDB*, pages 744–755, 2004.

[118] Y. Tao, D. Papadias, X. Lian, and X. Xiao. Multidimensional reverse k NN search. *VLDB J.*, 16(3):293–316, 2007.

[119] Y. Tao, M. L. Yiu, and N. Mamoulis. Reverse Nearest Neighbor Search in Metric Spaces. *IEEE Trans. Knowl. Data Eng.*, 18(9):1239–1252, 2006.

[120] H. Üster and R. Love. A generalization of the rectangular bounding method for continuous location models. *Computers & Mathematics with Applications*, 44(1-2):181–191, 2002.

[121] Y. Vardi and C.-H. Zhang. A modified Weiszfeld algorithm for the Fermat-Weber location problem. *Mathematical Programming*, 90:559–566, 2001.

[122] K. Veeraraghavan, J. Flinn, E. B. Nightingale, and B. Noble. quFiles: The right file at the right time. *ACM Transactions on Storage (TOS)*, 6:12:1–12:28, September 2010.

[123] B. S. Verkhovsky and Y. S. Polyakov. Feedback algorithm for the single-facility minisum problem. *Annals of the European Academy of Sciences*, 1:127–136, 2003.

[124] W. Wang, Y. Zhao, and R. Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 145–158, Berkeley, CA, USA, 2004. USENIX Association.

[125] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[126] G. Weisbrod, M. Ben-Akiva, and S. Lerman. Tradeoffs in Residential Location Decisions: Transportation versus Other Factors. *Transportation Policy and Decision-Making*, 1(1), 1980.

[127] E. Weiszfeld and F. Plastria. On the point for which the sum of the distances to n given points is minimum. *Annals OR*, 167(1):7–41, 2009.

[128] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient Continuously Moving Top-K Spatial Keyword Query Processing. In *ICDE*, 2011.

[129] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On Computing Top-t Most Influential Spatial Sites. In *VLDB*, pages 946–957, 2005.

[130] X. Xiao, B. Yao, and F. Li. Optimal location queries in road network databases. In *ICDE*, pages 804–815, 2011.

[131] C. Yang and K.-I. Lin. An Index Structure for Efficient Reverse Nearest Neighbor Queries. In *ICDE*, pages 485–492, 2001.

[132] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse Nearest Neighbors in Large Graphs. *IEEE Trans. Knowl. Data Eng.*, 18(4):540–553, 2006.

[133] E. Zadok. UnionFS: A Stackable Unification File System. `http://www.fsl.cs.sunysb.edu/project-unionfs.html`.

[134] E. Zadok, I. Badulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, pages 5–5, Berkeley, CA, USA, 1999. USENIX Association.

[135] E. Zadok and J. Nieh. FIST: a language for stackable file systems. *SIGOPS Oper. Syst. Rev.*, 34(2):38–, Apr. 2000.

[136] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword Search in Spatial Databases: Towards Searching by Document. In *ICDE*, pages 688–699, 2009.

[137] D. Zhang, Y. Du, T. Xia, and Y. Tao. Progressive Computation of the Min-Dist Optimal-Location Query. In *VLDB*, pages 643–654, 2006.

[138] Z. Zhang and K. Ghose. hFS: a hybrid file system prototype for improving small file and metadata performance. *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 41:175–187, March 2007.

[139] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid Index Structures for Location-based Web Search. In *CIKM*, pages 155–162, 2005.

[140] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[141] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted Files Versus Signature Files for Text Indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.