AUSPICE: AUTOMATIC SERVICE PLANNING IN CLOUD/GRID ENVIRONMENTS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of the Ohio State University

By

David Chiu, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2010

Dissertation Committee:

Gagan Agrawal, Advisor

Hakan Ferhatosmanoglu

Christopher Stewart

ABSTRACT

Scientific advancements have ushered in staggering amounts of available data and processes which are now scattered across various locations in the Web, Grid, and more recently, the Cloud. These processes and data sets are often semantically loosely-coupled and must be composed together piecemeal to generate *scientific workflows*. Understanding how to design, manage, and execute such data-intensive workflows has become increasingly esoteric, confined to a few scientific experts in the field. Despite the development of scientific workflow management systems, which have simplified workflow planning to some extent, a means to reduce the complexity of user interaction without forfeiting some robustness has been elusive. This violates the essence of scientific progress, where information should be accessible to anyone. A high-level querying interface tantamount to common search engines that can not only return a relevant set of scientific workflows, but also facilitate their execution, may be highly beneficial to users.

The development of such a system that can abstract the complex task of scientific workflow planning and execution from the user is reported herein. Our system, *Auspice: AUtomatic Service Planning In Cloud/Grid Environments*, consists of the following key contributions. Initially, a two-level metadata management framework is introduced. In the top-level, Auspice captures semantic dependencies among available, shared processes and data sets with an ontology. Our system furthermore indexes these shared resources for facilitating fast planning times. This metadata

framework enables an automatic workflow composition algorithm, which exhaustively enumerates relevant scientific workflow plans given a few key parameters - a marked departure from requiring users to design and manage workflow plans.

By applying models on processes, time-critical and accuracy-aware constraints can be realized in this planning algorithm. During the planning phase, Auspice projects these costs and prunes workflow plans in an *a priori* fashion if they cannot meet the specified constraints. Conversely, when feasible, Auspice can adapt to certain time constraints by trading accuracy for time. To simplify user interaction, both natural language and keyword search interfaces have been developed to invoke the said workflow planning algorithm. Intermediate data caching strategies have also been implemented to accelerate workflow execution over emerging Cloud environments. A focus on cache elasticity is reported, and to this end, we have developed methods to scale and relax resource provisioning for cooperating data caches. Finally, costs of supporting such data caches over various Cloud storage and compute resources have been evaluated.

To my family, friends, and mentors.

ACKNOWLEDGMENTS

This work is far from complete without proper acknowledgment of those who supported me over the years.

My family. When my parents made the decision to move to Ohio from Taiwan over twenty years ago, it was not the ideal situation for either of them. My mother spent tireless hours teaching me English and we struggled through heaps of schoolwork together. My father kept the family afloat through his ongoing work overseas. He also brought me up with great values in life, including humility and patience. Without their sacrifice and guidance, my education would have been impossible. I also thank my sister, Jenn, who has always been my partner in crime in childhood and now.

My wife, Michelle, for her infinite patience and unwavering love and understanding during even the most trying times. I am thankful for the many drop-offs and pick-ups to/from Dreese Labs she has had to perform on early mornings and late evenings. I respect her deeply, and she is a great inspiration to me.

My mentors. I am grateful to have worked with my dissertation advisor, Gagan Agrawal, who supported me throughout the program at Ohio State, offering not only guidance and leadership, but also personal advice. Working with him marked an auspicious beginning to my doctoral upbringing. I also thank Hakan Ferhatosmanoglu and Christopher Stewart for serving on my committee. I also have great respect for Tim Long, Raghu Machiraju, and Laurie Maynell, who always opened their doors to me when I needed advice, or just a chat. Help from my friend, Jason Sawin, and Joan

Krone, my PFF mentor, allowed me to ultimately succeed during my job search, and I thank them tremendously during this very difficult time. I also received impeccable guidance from my MS advisor, Paul Wang, and Yuri Breitbart at Kent State.

My friends and extended family, who always understood, and avoided the temptation of making me feel guilty, when I lacked time to see them. I am also happy to have made very close friends in my department, who put up with me on a daily basis: Qian, Tekin, Jiedan, Vignesh.1, Vignesh.2, Wenjing, Fan, Smita, Sai, Bin, Raff, Jim, and Leo.

My students. I would like to thank all my students, without whom I never would have chosen this path. Thanks for your patience during my early days of teaching, and for your feedback that continues to allow me to evolve and improve my teaching.

Coffee. Ever since my grandmother first introduced me to Mr. Brown's canned coffee at a very young age, I have been addicted. I have counted on coffee for years to get me through my program, so I would thus like to acknowledge coffee farmers around the world, and cafés in Columbus, including *Apropos* and *Stauf's*.

VITA

2002	B.S. Computer Science, Kent State University
2004	M.S. Computer Science, Kent State University
2010	Ph.D. Computer Science and Engineering, The Ohio State University
2005-Present	Graduate Teaching Associate, Department of Computer Science and Engineering, The Ohio State University
2007-2009	Graduate Research Associate, Department of Computer Science and Engineering, The Ohio State University

PUBLICATIONS

Vignesh Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations. In *Proceedings of the 24th ACM/SIGARCH International Conference on Supercomputing (ICS'10)*, ACM, 2010.

David Chiu, Sagar Deshpande, Gagan Agrawal, and Rongxing Li. A Dynamic Approach toward QoS-Aware Service Workflow Composition. In *Proceedings of the 7th IEEE International Conference on Web Services (ICWS'09)*, IEEE, 2009.

David Chiu and Gagan Agrawal. Enabling Ad Hoc Queries over Low-Level Scientific Data Sets. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM'09)*. 2009.

David Chiu and Gagan Agrawal. Hierarchical caches for grid workflows. In *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE, 2009.

David Chiu, Sagar Deshpande, Gagan Agrawal, and Rongxing Li. Composing geoinformatics workflows with user preferences. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'08)*, New York, NY, USA, 2008.

David Chiu, Sagar Deshpande, Gagan Agrawal, and Rongxing Li. Cost and Accuracy Sensitive Dynamic Workflow Composition over Grid Environments. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid'08)*, 2008.

Fatih Altiparmak, David Chiu, and Hakan Ferhatosmanoglu. Incremental Quantization for Aging Data Streams. In *Proceedings of the 2007 International Conference on Data Mining Workshop on Data Stream Mining and Management (DSMM'07)*, 2007.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Continuous progress in various scientific fields including, but not limited to, geoinformatics [80, 79, 64], astronomy [145], bioinformatics [75, 76, 77], and high-energy physics [56, 13], is generating a cornucopia of massive data sets. The Large Hadron Collider (LHC) project, for instance, is projected to produce 15 petabytes of data annually. Despite ongoing success in database technologies, scientists have resisted storing data within database systems in favor of traditional low-level files for a variety of reasons [161]. Moreover, today's database systems lack many functionalities on which many scientific applications depend. As such, most scientific data are persisted on decentralized systems, known as Mass Storage Systems (MSS), at various institutions. Such MSS systems are currently deployed over networks including clusters, scientific Data Grid. More recently, the Cloud-based persistent storages such as Amazon S3 [14] and SQLAzure [16] have also emerged as highly practical and cost-effective solutions to store mounds of data.

At the same time, advent in Web technologies has made it possible to easily share data sets and processing algorithms over service-oriented architectures. As a result, interoperable Web services have become abundantly deployed onto distributed cluster, Grid, and Cloud environments, offering the capacity to share scientific data manipulation processes with others. Indeed, such service-oriented science [68] paradigms

have proved useful and have been the answer to managing many large-scale projects. Figure 1.1 illustrates a current model of scientific data storage and processing.

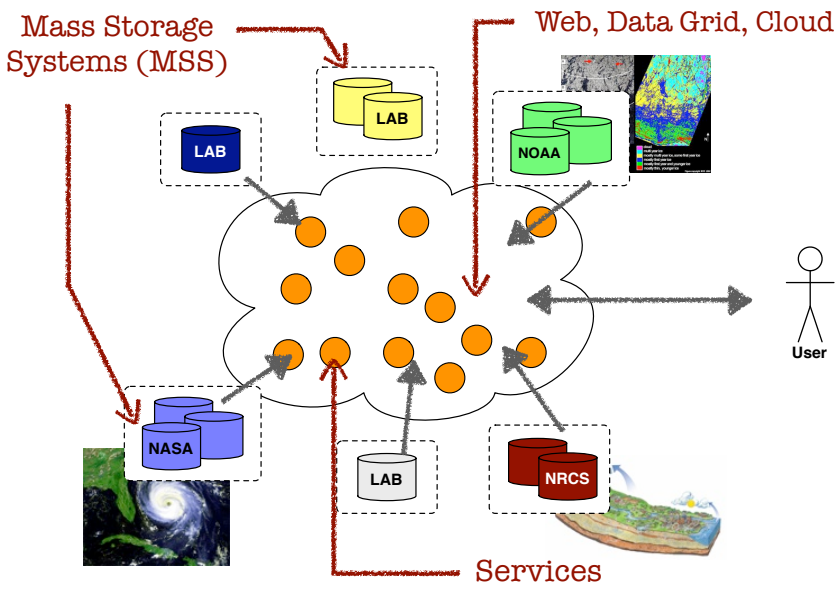


Figure 1.1: Scientific Data Repositories and Services

The resulting processing paradigm is one where services and data sets are constructed together piecemeal over geographically distributed environments to arrive at some desiderata. These so-called service chains, known formally as *scientific workflows*, facilitate users in the planning and execution of large-scale scientific processes. However, understanding how to manage scientific workflows is restricted to only a small group of experts, and rendering this process intuitive for the common user requires significant consideration toward novel designs. For instance, a keyword-search driven interface similar to the popular Google search engine, may be highly desirable.

In this work, we focus on such enabling designs and implementations in our workflow system, Auspice: Automatic Service Planning in Cloud/Grid Environments.

In general, Auspice processes high-level user queries through ad hoc, automatic composition and execution of workflow plans involving a correct set of services and data sets. Our system also enables Quality of Service (QoS) through adaptive service planning, allowing it to dynamically scale execution times and data derivation accuracies to user specifications. Auspice also exploits the emergence of the emergent pay-as-you-go computing infrastructure provided via the Cloud, by offering a flexible intermediate data cache.

1.1 Our Vision and the Auspice System

With scientific data sets, services, and metadata descriptions being made increasingly available, enabling high-level, intuitive access to these loosely coupled resources over existing scientific data infrastructures becomes possible. Empowered by a nuanced semantics framework, our system constructs and executes workflow plans automatically for the user. This approach is motivated by the following observations:

- *The aforementioned trend towards service-oriented solutions in scientific computing.* The Data Grid community, for instance, has benefitted greatly from borrowing Web service standards for interoperability through the Open Grid Service Architecture (OGSA) initiative [66].

- *The incentives for Cloud-based scientific applications implementations.* On-demand computing is causing many researchers to considering utilizing the Cloud paradigm to possibly save costs and to access elastic, infinite compute and storage resources.

- *The requirement of metadata standards in various scientific domains.* Metadata standards allow for clarity in both user and machine interpretability of otherwise cryptic data sets. However, in many sciences, metadata formats are often heterogeneous, and a unified method to index similar information is lacking, specifically for the purposes of workflow composition.

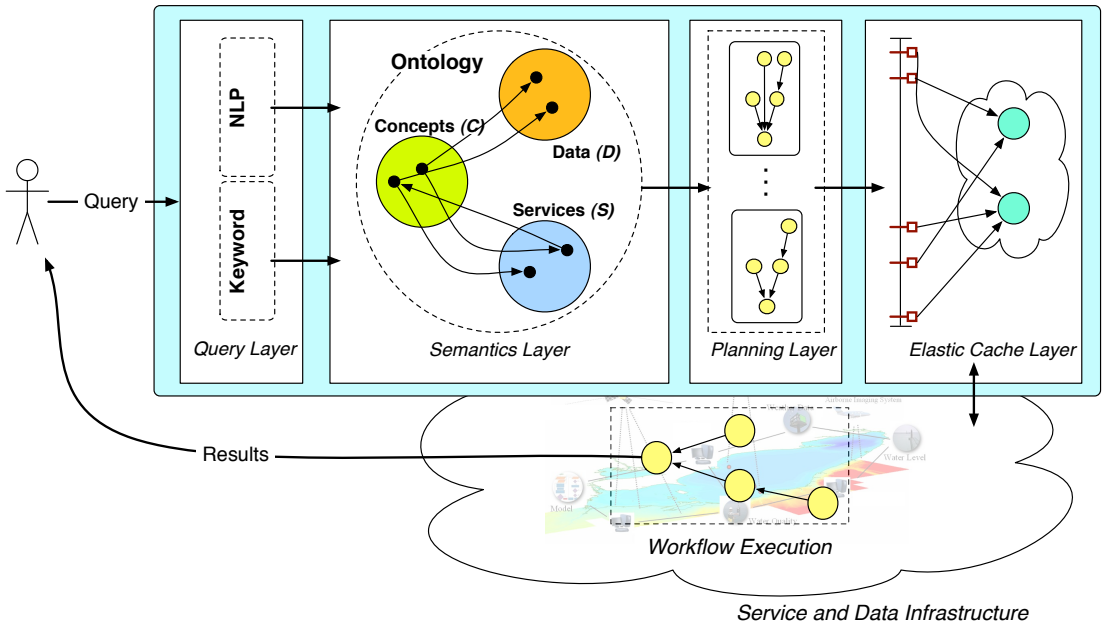


Figure 1.2: Auspice System Architecture

Auspice, shown in Figure 1.2, comprises multiple independent layers, each encapsulated with certain functionalities. In the ensuing chapters, detailed descriptions will be provided for each layer. As a whole, Auspice transparently overlays some underlying network, that is, the system is enabled with as little invasive specifications as possible required for existing infrastructures.

An overview of the system is presented here while detailed descriptions can be found in ensuing chapters. Auspice functions as a *service composition broker*. As users submit high-level queries to the system, the broker automatically plans and executes the workflows involved in deriving the desired result (derived data) while hiding such complexities as service composition, domain knowledge, and QoS optimization from the user.

The *Query Layer* offers two distinct ways for users to interact with Auspice: natural language and keyword search. In the former, the StanfordNLP [102] parser and WordNet [61] libraries are employed to decompose a natural language query first into a set of keywords. With the benefit of the language grammar, certain query semantics can be established. For instance, the query's target is typically the *direct object*. Similarly, *adjectives* can be associated with their respective *objects* – in the ontology (in the Semantics Layer), this would correspond with merging multiple concepts to form a hyper-concept node.

On the other end, keyword queries do not have the benefit of grammars for semantic guidance. Instead, we first map all keywords with corresponding ontological concepts when feasible. Then based on this set of concepts, we invoke our workflow planner repeated, on various ontological concepts, to enumerate possibly relevant workflow plans. Our planner will ranks workflow plans as a function of number of query concepts matched.

The task of the *Semantics Layer* is two-fold. First, this layer maintains an active list of available compute nodes, services, data sets, and their accompanying metadata. Services are described in WSDL [41] and data sets in their respective metadata standards. Since the work described herein deals with geospatial data sets, the Content

Standard for Digital Geospatial Metadata (CSDGM) [63], is used for data annotation. The second task of this layer is providing a domain-specific ontology, which is discussed further in Chapter 3.

Next, the *Planning Layer* assumes that the ontology and metadata are in place and defined. The planning algorithm, discussed in detail in the following section, relies heavily on the Semantics Layer. The planner enumerates workflows to answer a particular query through traversals of the domain ontology. The existence of needed services and data sets is identified by the ontological index. A set of workflows capable of answering the user query can be pruned or adjusted according to some user specified QoS parameters. A filtered set of workflow candidates can then be sent for execution, and the resulting derived data is finally returned back to the user.

Upon workflow execution, the intermediate cache within the *Elastic Cache Layer* is checked for precomputed data whenever possible to acclerate this process. This cache layer maintains a data cache while autonomously provisioning and relaxing Cloud resources for storage. The workflow execution itself is carried out in the underlying network environment, and its results are sent back to the user (and the cache) upon completion.

1.2 Contributions

While we discuss in depth the contributions of Auspice in the next few chapters, a summary work is given here.

1.2.1 Data-Driven Service Composition

The scientific domain contains many loosely coupled data sets and services, being made available to various users, ranging from novice to expert. Scientific processes are often data-intensive and dependent on results of previous computations. For

users, identifying the class of data, service operations, and linking their dependencies together is nontrivial, regardless of the user's skill set. Our vision is a system that can automatically index data sets and services and then automatically compose service workflow plans to answer a high-level user query.

In Auspice, data sets and services are *registered* into an index containing metadata information, e.g., time, creator, data quality, etc. for easy identification. A simple domain ontology is superimposed across the available data sets and services for enabling workflow planning. Although traditional database and data integration methods (such as the use of federated databases [149] or mediator-based systems [74, 150]) can be applied, our approach does not require a standardized format for storing data sets or the implementation of a complex mediator-based querying framework. Auspice thus combines workflow composition with machine-interpretable metadata, a domain ontology, and a natural language interface to offer simple and intuitive ways for querying a variety of data sets stored in their original low-level formats.

In the meantime, many of today's search engines no doubt owe much of their success to the ease of usability in their keyword search interfaces. This incentive, consequently, has invoked a tremendous amount of effort toward supporting keyword queries in modern information systems. At the same time, the staggering growth of scientific data and Web services has ushered in the development of scientific workflow managers. In these systems, users are required to chain together a number of dependent processes, e.g., Web services, with disparate data sets, in order to generate some desiderata. Scientific workflows are typically planned and invoked by experts within certain fields, which restricts the access of scientific information to a small group of users. To this end, we believe that using keyword search queries to invoke automatic scientific workflow planning would be highly desirable for the overall

scientific community. With the widespread proliferation of scientific metadata, coupled with advancements in ontological reasoning, such a system can be enabled, but not without challenges. We describe a way to index scientific data and services on their ontological attributes. A keyword-to-ontology mapper is then used in conjunction with an automatic planning algorithm to enable on-the-fly retrieval of relevant workflow plans. A relevance metric is further discussed for ranking workflow results.

1.2.2 Workflow Planning with Adaptive QoS Awareness

A myriad of recent activities can be seen towards dynamic workflow composition for processing complex and data intensive problems. Meanwhile, the simultaneous emergence of the Grid has marked a compelling movement towards making datasets and services available for ubiquitous access. This development provides new challenges for workflow systems, including heterogeneous data repositories and high processing and access times. But beside these problems lie opportunities for exploration: The Grid's magnitude offers many paths towards deriving essentially the same information albeit involving varying execution times and errors.

We discuss a framework for incorporating QoS in a dynamic workflow composition system in a geospatial context. Specific contributions include a novel workflow composition algorithm which employs QoS-aware apriori pruning and an accuracy adjustment scheme to flexibly adapt workflows to given time restrictions. A performance evaluation of our system suggests that our pruning mechanism provides significant efficiency towards workflow composition and that our accuracy adjustment scheme adapts gracefully to time and network limitations.

1.2.3 Methods for Caching Intermediate Results

From personal software to advanced systems, caching mechanisms have steadfastly been a ubiquitous means for reducing workloads. It is no surprise, then, that under the Grid and cluster paradigms, middlewares and other large-scale applications often seek caching solutions. Among these distributed applications, scientific workflow management systems have gained ground towards mitigating the often painstaking process of composing sequences of scientific data sets and services to derive virtual data. In the past, workflow managers have relied on low-level system cache for reuse support. But in distributed query intensive environments, where high volumes of intermediate virtual data can potentially be stored anywhere on the Grid, a novel cache structure is needed to efficiently facilitate workflow planning. We describe an approach to combat the challenges of maintaining large, fast virtual data caches for workflow composition. A hierarchical structure is described for indexing scientific data with spatiotemporal annotations across Grid nodes. Our experimental results show that our hierarchical index is scalable and outperforms a centralized indexing scheme by an exponential factor in query intensive environments.

1.2.4 Elastic Cloud Caches for Accelerating Workflows

Computing as a utility, that is, on-demand access to computing and storage infrastructure, has emerged in the form of the Cloud. In this model of computing, elastic resource allocation, i.e., the ability to scale resource allocation for specific applications, should be optimized to manage cost versus performance. Meanwhile, the wake of the information sharing/mining age is invoking a pervasive sharing of Web services and data sets in the Cloud, and at the same time, many data-intensive scientific applications are being expressed as these services. In this paper, we explore an approach to accelerate service processing in a Cloud setting. We have developed a cooperative

scheme for caching data output from services for reuse. We introduce algorithms for scaling our cache system up during peak querying times, and back down to save costs. Using the Amazon EC2 public Cloud, a detailed evaluation of our system has been performed, considering speed up and elastic scalability in terms resource allocation and relaxation.

With the availability of on-demand compute and storage infrastructures, many users are deploying data-intensive scientific applications onto Clouds. To accelerate these applications, the prospect of caching intermediate data using the Cloud's elastic compute and storage framework has been promising. To this end, we believe that an in-depth study of cache placement decisions over various Cloud storage options would be highly beneficial to a large class of users. While tangential analyses have recently been explored, ours in contrast focuses on cost and performance tradeoffs of maintaining a data cache with varying parameters of any Cloud application. We have compared several Amazon Web Service (AWS) Cloud resources as possible cache placements: within machine instances (in-memory, on-disk, on large mountable disk volumes) and in inexpensive persistent Cloud storage. We found that application-dependent attributes like unit-data size, total cache size, persistence requirements, etc., have far-reaching implications on the cost to sustain their cache. Also, while instance-based caches expectedly yield higher cost, the performance that they afford may outweigh lower cost options.

The remainder of this thesis is organized as follows. In Chapter 2, a detailed discussion of related works is given. We present descriptions on past efforts in scientific data management, workflow systems, quality of service in workflow management, caching strategies, and finally, resource allocation. Chapter 3 describes a metadata registration framework, keyword search, and automatic workflow planning algorithms. A Quality of Service-aware variant of the workflow planning algorithm is discussed

in Chapter 4. In Chapter 5, strategies for caching and reusing a workflow's intermediate and final results are presented for accelerating execution. We extend this cache framework onto the Cloud context, and discuss various cost-performance tradeoffs in Chapter 6. Finally, we will discuss future works and conclude in Chapter 7.

CHAPTER 2

RELATED WORKS

Auspice is a workflow system which supports several functionalities, outlined below. In literature, there exists a large amount of related works on each of these issues. This chapter discusses related efforts from each of the following topics.

- Enabling a nonintrusive framework for sharing and querying scientific data sets and Web services through metadata indexing.
- Composing known services and data sets in disparate ways to derive user queries.
- Adaptively scaling workflow execution vis à vis tradeoffs between execution time and application error to meet QoS constraints.
- Caching intermediate results for trivializing future service invocations.
- Employing Clouds for scientific processing.

2.1 Scientific Data Management

The deluge of scientific data has prompted a major challenge in data management issues. For instance, the Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) is projected to produce around 15PB of data annually [112]. One of its experiments, ATLAS (A Toroidal LHC ApparatuS) [110], which will

measure the particles created in the collisions, is single-handedly expected to produce data on the rate of 5PB per year. Elsewhere, the LSST telescope [117] is reporting that their archive will consist of 60PB of captured image data. These are just two, among many, scientific (let alone other domains, such as business) projects that are currently generating large volumes of data sets. Additionally, due to the nature of scientific processes, there is a push for persistent store policies, where the data derived from past computations must also be kept for historical/lineage, repeatability, and reuse purposes.

In order to store these data sets, it is no longer possible to use centralized storage systems. Moreover, the cost of storing these data sets is prohibitive for disk-based solutions – the even cheaper (but far slower) tapes must generally be used. Hence, large data centers normally consist of these slow, archival-based Mass Storage Systems (MSS). For instance, Fermilab uses such an storage infrastructure to store its 18+PB of data in its MSS, Enstore, with an intermediate buffer, known as dCache [17]. In another effort, the Storage Resource Broker (SRB) [19] can be used to support a logical distributed file system, that is, the user sees a single file hierarchy for data that are actually distributed across multiple storage systems. SRB also optimizes data movement through its transparent parallel I/O capabilities.

With current cyberinfrastructures, such as the Open Science Grid and Europe's EGEE, already in place, federated storage approaches have also been undertaken to harness the existing infrastructure. As a specific and recent example, to provide access to a large number of scientists, the Worldwide LHC Computing Grid (WLCG), has been built [111]. The architecture of WLCG is tier-based. At Tier-0 lies the LHC itself. Raw data produced by the LHC is first saved on a mass storage system (MSS) at CERN. Another copy of this data is split into chunks and distributed onto the Tier-1 sites. Tier-1 sites are themselves massive data centers across the world and

are connected to CERN via dedicated high speed links (10GB/s). The job of Tier-1 sites is essentially to provide data to the compute farms located in Tier-2. This tier provides most of the compute resources needed for scientists to run their processes. From our perspective at the Ohio State University, the Ohio Supercomputing Center, for instance, could be a Tier-2 site. Finally, Tier-2 sites are accessed by workstations and servers within research labs, which deal directly with the user (Tier-3).

Another aspect of scientific data management is access. The raw data captured directly from experiments and simulations are often difficult to interpret by humans. Thus, frequently, raw data sets are given a pass by algorithms close to the sources to translate them into something more interpretable and interesting. For instance, images captured in the spatial-frequency domain must first be converted into the spatial domain for human interpretation. During the reconstruction, data sets are given meaning by using metadata specific to its scientific domain. Over a decade ago, the invocation of metadata became reified through the timely emergence of XML [179], a declarative language which allows anyone to describe data using structured markup elements. Having an accessible means for anyone to invent and provide data descriptions, metadata were eventually substantiated and standardized. As a result, metadata has become essential in many of today's applications.

The Dublin Core Metadata Initiative, for instance, has instituted a set of cross-domain metadata elements (e.g., author, title, date, etc.) [46]. Attuned the importance of metadata, the scientific community also began undertaking tremendous efforts toward standardizing metadata formats. These efforts produced such format standards as the Content Standard for Digital Geospatial Metadata (CSGDM) [63] and the Biological Data Profile (BDP) [125]. With the growing number of reliable scientific metadata, relevant data sets can be identified and accessed more efficiently. In fact, the spread of metadata has become so marked that high-end systems research

has already begun addressing issues in *very large metadata management systems* [154, 91, 168]. With the growing number of reliable scientific metadata, relevant data sets can be identified and accessed both accurately and efficiently.

2.1.1 Scientific Data Processing

Another issue with scientific data is facilitating ways to process them. Scientists became aware that a centralized computing infrastructure is no longer viable. In order to facilitate the management, processing, and dissemination of this amount of data, a distributed framework must be employed. Research in this direction has brought forth the Data Grid [66], which builds on the notion of managing large amounts of data and supporting compute services from geographically dispersed nodes. Data Grids have become the *de facto* cyber-infrastructure of choice for hosting large-scale sciences, including the TeraGrid [141], and its peers, the Open Science Grid [130], Worldwide LHC Computing Grid (WLCG) [111], and others.

On the topic of actual data processing, the sheer size of scientific data must be processed in chunks. After this data distribution, the data chunks (typically independent) can then be processed in parallel, and their individual results merged at the end of the computation. These parallel approaches have been around for quite awhile, leading to the adoption of OpenMP, MPI, and GPGPU programming models. However, it is also generally known to the community that programming becomes more difficult for science researchers. To reach this community, new compiler techniques have facilitated automatic source-to-source code transformations (e.g., [28, 118]) to optimize parallel computations over multi-core (and GPGPU) architectures. This movement has been timely, as multi-core clusters, Data Grids, and Cloud-based infrastructures emerge.

One class of scientific data processing involves real-time data streams. For instance, "the LSST telescope rapidly scan the sky, charting objects that change or move: from exploding supernovae to potentially hazardous near-Earth asteroids" [117]. Similarly, other sensing applications must monitor real-time change in the environment. An exhaustive array of data stream processors, including Aurora [36], and many others, have been developed for this reason. Their ways of handling real-time streams (over given hardware resources) differ from load shedding to summarization, to accuracy tradeoffs. GATES [37], for instance, splits data stream processes into *stages*, where each stage (composed of multiple processes) performs some computation and summarization, before outputting results to the next stage. Based on the resources' ability to process at each stage, the application's accuracy may be tuned to involve more or less processing. Its counterpart, HASTE [186], handles time critical events by satisfying time constraints while seeking to maximize an application specific benefit function. In other works utilize discretization approaches, such as predictive quantization [8] to represent the original streaming data in a sliding window timeframe as accurately as possible. In a related effort, we proposed a method to "age" and shed accuracy for older, quantized elements within the sliding window by removing bits dynamically for their representation [7].

Many analysis and mining processes also fall under the umbrella of a "general reduction" structure (explained above, where data sets can be split into smaller processes, summarized locally, then ultimately merged). Systems that facilitate these reductions automatically (and reduces programming complexities) have been built, e.g., FREERIDE [84] and MapReduce systems [47]. These so-called Map-Reduce class applications have lent well to general scientific computing.

The derived scientific data results from such processes, are often input into another process, and thus, analysis frequently involve dependent processes to be run in

forms of workflows. Many scientific workflow management systems, such as Pegasus [52] and Kepler [5] have been employed to manage large-scale dependent computations. These systems typically allow scientists to design workflows from a high-level perspective. The execution of the individual workflow processes is abstracted by workflow managers, which hide such nuances as scheduling, resource allocation, etc. However, the planning of such workflow structures is still performed by the user. This approach may sometimes be sufficient, but in a time when users are becoming increasingly goal-oriented, solutions which leverage effective use of domain information would require significantly less effort. Certainly, one ultimate goal for enabling these workflows is to automate their composition while simultaneously hiding such esoteric details as service and data discovery, integration, and scheduling from the common user.

2.1.2 Metadata and Querying

The call for a semantically conscious Web has been addressed by such efforts as the World Wide Web Consortium's (W3C) Metadata initiative, developing such specifications as the Resource Description Framework (RDF) [120], its complement, the Web Ontology Language (OWL) [49], and SPARQL [159], the SQL-like querying language for RDF. The Ontology Web Language for Services (OWL-S), provides developers a way to describe capabilities of services [59]. These standards supply common machine-readable resources with a means for enbling machine-understandability and interpretability. In Auspice, we utilize RDF+OWL to formalize a general ontology which describes the relationships between certain domain-specific concepts and resources (data sets and services) available over the Web. This resource description is imperative to our system, as it semantically drives our workflow/service composition algorithm. Auspice differs from other systems in the sense that its semantic

descriptions are generated with little cryptic input from users through its Metadata Registration framework.

Mainstream search engines have ushered the keyword paradigm into ubiquity, as it requires no a priori knowledge about a system's underlying structure to construct queries. Typically, keyword search queries derive a list of results ranked against their relevance to the user's needs. While loosely related to our proposal, it is worthwhile to note the efforts made toward keyword search over relational databases. In DataSpot [44] and BANKS [25], the database is represented as a directed graph where nodes denote tuples and directed edges represent "links" from foreign keys to primary keys across tables. To answer a query in DataSpot, it returns the connected subgraphs where the nodes contain all keyword terms. Because it the information represented by these subgraphs are sometimes ambiguous, BANKS differs in that the results to a query are those directed trees (Steiner trees) whose nodes correspond to the given keywords. The trees are rooted on a node that connects to all other keyword nodes, and therefore it conveys the strongest information. The nodes in BANKS's structure employ a PageRank-style [32] weighting, which affects the ranking of results. One notable problem with these system is that their directed graphs must be resident in memory, which is impractical for large databases. In Auspice, only the ontology is required to be memory-resident. Data and service identification for workflow planning is performed on the fly.

DISCOVER [170] and DBXplorer [3] generate relevant SQL on-the-fly and returns the tuple trees (from potentially joined tables) that contain all keywords. These efforts ranked the results based on the number of table joins needed to generate each result, i.e., the more joins needed, the fuzzier (and thus, less relevant) the results are. This is similar to the approach taken by our system. Hristidis et al. [90] and Liu et al. [116] later proposed IR-style ranking strategies which, among other

things, leverages the underlying database's native relevance methods for text search within attributes. Saddyadian et al.'s Kite [144] extends these efforts to distributed, heterogeneous databases. In some systems [136, 162] the entire database is crawled and indexed offline. Particularly, Qu and Widom's EKSO [162] crawls the database offline and generates *virtual documents* from joining relational tuples. Each virtual document associates with some database object that should be understandable to the user when returned. These virtual documents are then indexed using traditional keyword/IR indexing techniques.

Another related area of effort involves keyword queries over XML documents, which have become pervasive in storing structured data and metadata. In [65], Florescu et al. integrated keyword search into XML-QL [53], a formerly proposed XML querying language. Their approach, however, was intended for simplifying the specification of XML-QL queries and lacked any relevance metrics. In contrast, several other works were proposed generate ranked results. Perhaps most notably, XRANK [62] enabled two granularities of IR-style ranked retrieval: within and across XML documents. These efforts, while prevalent, are orthogonal to Auspice's goals. To the best of our knowledge, our proposed work is the first to enable keyword queries for retrieving automatically planned workflows toward deriving scientific information.

Ways to handle the heterogeneity of metadata have prompted many works on metadata cataloguing and management. Particularly, in the volatile Grid computing environment, data sources are abundant and metadata sources are ever-changing. Metadata Catalog Service (MCS) [154] and Artemis [168] are collaborative components used to access and query repositories based on metadata attributes. MCS is a self-sufficient catalog which stores information on data sets. Its counterpart Artemis, on the other hand, can be used to integrate many versions of MCS for answering interactive queries.

Their interface takes users through a list of questions guided by a domain ontology to formulate a query. The planned query is then sent to the Artemis mediator to search for relevant items in the MCS instances. While the MCS and Artemis is somewhat tantamount to our metadata registration and automatic query formulation processes, our systems differ in the following ways. (i) Ours not only facilitates accurate data identification based on metadata querying, but also combining these data items with similarly registered services to compose workflows. (ii) Although both systems allow higher level querying frameworks, our approach is enabled through natural language and keyword mapping of domain ontology concepts.

Research in high performance scientific data management has produced such systems as the Scientific Data Manager (SDM), which employs the Meta-data Management System (MDMS) [126]. SDM provides a programming model and abstracts low-level parallel I/O operations for complex scientific processing. While MDMS uses a database for metadata storage, the metadata itself is specific to the scientific process at hand, containing information on execution (e.g., access patterns, problem size, data types, file offsets, etc). This metadata is used by SDM to optimize the runtime of these parallel processes. Another system, San Diego Supercomputing Center's Storage Resource Broker (SRB) [19], seeks to store massive volumes of data sets split across clusters or nodes within heterogenous environments. SRB allows parallel and transparent data access by offering a simplified API to users which hides complexities such as merging data sets, allowing restricted access, etc. In [147], Shankar et al. explored the potentials of integrating database systems with workflow managers. They envision using SQL to query and invoke programs within the workflow, thereby harnessing the database's inherent capabilities to manage large-scale scientific processes. Compared to Auspice, there is a fundamental difference in functionality. Ours provides a way to store heterogeneous metadata specific to scientific domains inside

a database, and that the metadata are invoked not for process optimization, but for data identification purposes for automatic workflow planning.

2.2 Scientific Workflows and Service Composition

Among the first systems to utilize workflows to manage scientific processes is ZOO [93], which employs an object-oriented language to model the invocation of processes and the relationships between them. Another notable system, Condor [115], was originally proposed to harvest the potential of idle CPU cycles. Soon after, dependent processes (in the form of directed acyclic graphs), were being scheduled on Condor systems using DAGMan [43]. Recently, with the onset of the Data Grid, Condor has been integrated with the Globus Tookit [69] into Condor-G [71]. Pegasus [52] creates workflow plans in the form of Condor DAGMan files, which then uses the DAGMan scheduler for execution.

The general acceptance of Web service technologies has encouraged the deployment of heterogeneous, but interoperable, processes on the Web. Considered the modern day approach to Remote Procedure Calls (RPC), a Web service consists of a set of related operations that be invoked remotely. Each service's application interface is described using the Web Service Description Language (WSDL) [41]. Web services elude the low-level complexities by simplifying the communications protocol to ride over the Simple Object Access Protocol (SOAP) [157], which encapsulates services messages and sends them over HTTP. Envisioning far into the breadth of services, the Web community also prescribed Universal Description Discovery and Integration (UDDI) [169], essentially a registry for service discovery and search. This vision of a "web" of available and accessible services preceded the idea of ultimately being able to link together disparate services in order to compose complex, distributed, and

interoperable software. Enabling this process has become a popular research topic, known simply as service composition.

Service composition [58, 139, 128] is not a new concept. It is in fact deeply rooted in legacy workflow management systems, where a streamlined execution of well-defined "processes" are used to define complex tasks in business and scientific operations. By itself, service composition has become important enough to warrant such standards as the WSBPEL (Web Service Business Process Execution Language) [177] to describe the orchestration of service execution. Implementations of WSBPEL engines have already sprawled into realms of proprietary and opensource communities – an auspicious indication of the high optimism for the movement towards composite service solutions.

Static service composition systems, e.g., Microsoft BizTalk Server [26], are effective with the absence of changes in the computing environment such as the introduction or replacement of services. Such systems typically exist under proprietary domains, where the set of workflow processes and their components are rigorously defined and maintained. These static systems, however, are inadequate in the face of a dynamic computing environment where new services are made available and reimplemented on a daily basis. Many systems have been proposed to alleviate the painstaking task of maintaining consistency and correctness of the composite services. Several efforts [122, 57], for instance, describe a hybrid support for static workflows under dynamic environments. In Casati et al.'s eFlow [35], a workflow's structure (known as a process schema) is first defined by some authorized users, but the instantiation of services within the process is dynamically allocated by the eFlow engine. It is also worth noting that eFlow also supports high level modification of schemas when necessary. Sirin et al. proposed a user interactive composer that provides semi-automatic composition [155]. In their system, after each time that a

particular service is selected for use in the composition, the user is presented a filtered list of possible choices for the next step. This filtering process is made possible by associating semantic data with each service. To complement the growing need for interoperability and accessibility, many prominent workflow managers, including Taverna [129], Kepler [5] (the service-enabled successor to the actor and component-based Ptolemy II [33]), and Triana [119] have evolved into service-oriented systems. These systems typically allow domain experts to define static workflows through a user-friendly interface, and map the component processes to known Web services.

The Data Grid and the Cloud opened up various opportunities for the scientific community to share resources including, among others, large-scaled datasets and services. This prompted the emergence of scientific workflows for modeling and managing complex applications and processes for data and information derivation. Pioneering works towards this front include Chimera [70], which supports detailed recording of data provenance and enables the reuse of well-designed workflows to generate or recreate derived data. This allows domain experts to define workflows through intuitive interfaces, and the workflow components are then automatically mapped to the Grid for execution. The class of service/workflow composition systems for supporting composite business and scientific processes has been studied extensively in a number of works [81, 27, 163]. These systems, which borrow techniques from AI Planning, typically enable end-users to compose workflows from a high level perspective and automate workflow scheduling and execution, with the benefit of some domain specific ontology.

Some service composition systems [113, 148, 21, 124] require low-level programming details, which not suitable for all users. SELF-SERV [148, 21], for instance, is user-guided but at a more abstract level where the actual instantiation of services

is dynamically chosen. Traverso et al. discussed the importance of exploiting semantic and ontological information for automating service composition [167]. Their approach generates automata-based plans, which can then be translated into WS-BPEL processes. The goals and requirements for these plans, however, must be expressed in a formal language, which may be cryptic for the average user. Other automatic planning systems also require similar complexity in expressing workflows. For instance, SHOP2 [178] uses an AI planning technique, namely, Hierarchical Task Network (HTN), where its concept of task decomposition into primitives and compound service operations is not unlike the concept derivation planning employed by Auspice. However, SHOP2 must be used in conjunction with OWL-S [59] (formerly DAML-S), a set of markups for describing Web service capabilities and properties. In a related effort, SWORD [133] utilizes a service's pre/post-conditions to guarantee correctness. However, a service request must specify the initial and final states for the service composition. Fujii and Suda [72] proposed a tiered architecture for semantics-based dynamic service composition. The system uses natural language processor to parse queries into "components," and performs semantic matching to assure that a composed service satisfies the semantics of the query. Medjahed et al. also focused on semantics-enabled automatic service composition [121]. Auspice is tantamount in the way that services are composed with help of an ontology, for guidance in semantic correctness. Auspice, however, also seeks to reduce composition time by indexing metadata belonging to services and scientific files. This effectively enables a framework for data/service sharing.

Specifically in geospatial applications, the impetus and merits behind geospatial service composition have previously been highlighted in [4]. Di et al. described their system for ontology-based automatic service composition [54]. Here, workflows are

composed via a rule-based system, and correctness is kept through backwards reasoning. However, base rules that are needed to generate a specific geospatial concept must explicitly defined in the system. Studies on the use of geospatial ontologies for automated workflow composition have been carried out. The work of Lemmens et al. [108] describes a framework for semi-automatic workflow composition. Yue et al. demonstrated that automatic construction of geospatial workflows can be realized using their ontological structure [183, 55]. Hobona et al. [87] combines a well-established geospatial ontology, SWEET [140], with an adopted notion of semantic similarity of the constructed workflows and the user's query.

2.3 QoS Management in Workflow Systems

In the past, workflow systems with QoS support have been developed. For instance, within the Askalon Grid workflow environment [134], Brandic et al. introduced system-oriented QoS support, such as throughput and transfer rates [30, 31]. Some efforts in this area focus on process or service scheduling optimization techniques in order to minimize total execution times. Eder et al. suggests heuristics for computing process deadlines and meeting global time constraints [60]. Other works, including Zeng et al.'s workflow middleware [184], Amadeus [29], and stochastic modeling approaches [2, 176] exploit such Grid service technologies, where data sets are inherently assumed heterogeneous and intelligent workflow scheduling on resource availability becomes a greater issue in meeting time constraints. The scheduling optimization is also the approach chosen by the well-established Taverna [129], Pegasus [52, 82], and Kepler [5] to reduce overall execution times.

In other works, Glatard's service scheduler exploits parallelism within service and data components [83]. Lera et al. have developed a performance ontology for dynamic QoS assessment [109]. Kumar et al. [104] have developed a framework for

application-level QoS support. Their system, which integrates well-known Grid utilities (the Condor scheduler [71], Pegasus [52], and DataCutter [24], a tool which enables pipelined execution of data streams) considers *quality-preserving* (e.g., chunk size manipulation, which does not adversely affect accuracy of workflow derivations) and *quality-trading* QoS parameters (e.g., resolution, which could affect one QoS in order to optimize another). In quality-preserving support, the framework allows for parameters, such as chunksize, to be issued. These types of parameters have the ability to modify a workflow's parallelism and granularity, which potentially reduces execution times without performance tradeoffs. For quality-trading QoS support, an extension to the Condor scheduler implements the tradeoff between derived data accuracy for improved execution time.

While Auspice adopts strongly established features from the above efforts, it has the following distinguishing characteristics. We envision an *on-demand* domain level querying framework that is applicable to users from naïve to expert. Data derivation is made available immediately through a high-level keyword interface and abstraction of user-level workflow creation via automatic service composition. The aforementioned workflow managers require some user intervention involving definition of an abstract workflow template or recondite rules and goals for the planning algorithms in [133, 178, 148, 185]. Another feature of our work involves the adaptability to QoS constraints. Some workflow managers, such as Askalon [134], attempt to minimize the execution time of workflows by employing performance predictors which factor into scheduling decisions. Our system differs in that the overall goal is not specifically the minimization of execution time. Instead, we focus on an accuracy-oriented task where workflow execution times may be manipulated to fit within the required QoS through error-level adjustment. Auspice assumes that multiple workflow candidates can be composed for any given user query. This set of candidates is pruned on the

apriori principle from the given user preferences, making the workflow enumeration efficient. Furthermore, we focus on an accuracy-oriented task by allowing the user to specify application/domain specific time and error propagation models. Our system offers the online ability to adjust workflow accuracies in such a way that the modified workflow optimizes the QoS constraints.

2.4 Intermediate Result Caching for Service Composition

In the direction of data reuse, distributed file caching and replication architectures have long been established as an effective means for reducing processing times for compute and data intensive processes [160, 23, 164]. Meanwhile, recording provenance, or detailed information representing the procedure from which intermediate data are derived, became very popular for helping users accurately reproduce certain results. Chimera [70] was an early endeavor on building technologies for cataloging data provenance. Recognizing its benefits, provenance technologies also emerged in many scientific workflow systems, including Keper [6], Pegasus [101], and Karma 2 [152].

Shankar and Dewitt integrated distributed data caches with previous jobs' metadata (essentially a form of provenance) in their Condor-based scientific workflow manager, DAG-Condor [71]. Their goals of exposing distributed, preprocessed intermediate data for the purposes of memoization are consistent with our objectives. In DAG-Condor, workflow plans are declarative: Users submit a job file that specifies input/output files, variables, dependencies, etc. DAG-Condor thus assumes intermediate data used for input/output can be indexed on checksums of job histories. This user-guided nature marks a fundamental contrast between our systems.

Because our system automates workflow composition, our indexing scheme must be robust enough to correctly and efficiently identify intermediate data during the

planning process. To automate this process, domain level semantics must be injected to our cache identification mechanism. Thus, our index is not only physically structured across the Grid to quickly identify relevant cache locations, but the individual indices themselves are strongly tied to the scientific domain with knowledge and spatiotemporal connotations. In the geospatial domain, many indexing techniques exist. For example, our system relies on B^x-Trees [96, 95], an index originally provided by Jensen et al. for moving objects. B^x-Trees are fast because its underlying structure is the ubiquitous B+Tree [20], a *de facto* standard in most database systems. It linearizes two-dimensional space into the one-dimensional B+Tree key through using space-filling curves [123, 106]. Other well-established spatial indexing methods are also heavily utilized in practice: R-Trees and its variants [86], QuadTrees [143], and K-*d* Trees [22]. To the best of our knowledge, Auspice is the first to manage a hierarchical cache of intermediate results in a workflow composition framework.

Auspice's caching framework is much-inspired by efforts done in the general area of Web caching [135]. To alleviate server load, intelligently placed proxies have historically been employed to replicate and cache popular Web pages. The Internet Cache Protocol (ICP) is employed by many Web proxy systems to exchange query messages [175], although our system does not currently implement this standard.

Several methods can be used to evenly distribute the load among cooperating caches. Gadde, Chase, and Rabinovich's CRISP proxy [73] utilizes a centralized directory service to track the exact locations of cached data. But this simplicity comes at the cost of scalability, i.e., adding new nodes to the system causes all data to be rehashed. Efforts, such as Karger et al.'s consistent hashing [99, 100] have been used to reduce this problem down to only rehashing a subset of the entire data set. Also a form of consistent hashing, Thaler and Ravishankar's approach maps an object

name consistently to the same machine [165]. Karger et al.'s technique is currently employed in our cooperative cache.

Research efforts in storage management have proposed a cache layer for alleviating long access times to persistent storage. For instance, Cardenas et al.'s uniform, collaborative cache service [34] and Tierney et al.'s Distributed-Parallel Storage System (DPSS) [166] offer a buffer between clients and access to mass storage systems including SDSC's Storage Resource Broker [19]. Other efforts, including works done by Otoo et al. [131], Bethel et al. [23], and Vazhkudai et al. [171], consider these intermediate caching issues in various storage environments for scientific computing. Work has also been produced in the direction of optimal replacement policies for disk caching in data grids [97]. Other utilities have sought for the storage of more detailed information on the scientific, such as virtual data traces, known as provenance. Chimera [70] is a system that stores information on virtual data sets which affords scientists a way to understand how certain results can be derived, as well as a way to reproduce data derivations.

Tangential to our work is *multi-query optimization* (MQO) in the area of databases, where related queries may share, and can benefit from reusing, common data. Methods for processing like-queries together, rather than independently, could thus greatly improve performance [146]. Toward this goal, *semantic caching* [45, 142], i.e., based on semantic locality, has been considered in past efforts. In this approach, cached data associates a certain semantic region, represented by a formula, e.g., $Q_1 = (A \leq 30 \wedge B > 1000)$, where A and B are attributes. Now consider that a future query asks for all records where $Q_2 = A < 20$. A *probe query* would first retrieve the intersected subset from the local semantic cache, i.e., all records where $Q_1 \wedge Q_2$. Subsequently, a *remainder query* is submitted to the backend to retrieve the rest of the data, $Q_2 \wedge \neg Q_1 = (A < 20 \wedge (A > 30 \vee B \leq 1000))$.

Andrade, *et al.* described an active semantic caching middleware to frame MQO applications in a grid environment [10]. This middleware combines a proxy service with application servers for processing, which dynamically interacts with cache servers. The proxy acts as a coordinator, composing suitable schedules for processing the query over the supported environment. Our system differs, like all the aforementioned effort, in that it considers cost-based pressures of the Cloud. Specifically, our cache is sensitive to *user interest* over time, and it allocates compute resources to improve performance during query intensive periods, only to relax the resources later to save costs.

The recently proposed *Circulate* architecture employs Cloud proxies for speeding up workfow applications [174, 18]. In their work, proxies close to the computation are used to store intermediate data. This data is then directly routed to the nodes involved in the next stage of the computation. While the their overarching goal of reducing composite service time is tantamount to ours, we clarify the distinctions. Their system focuses on eluding unnecessary data transfers to and from some orchestrating node. Ours deliberately caches service results to accelerate processing times under a *query-intensive* scenario. Our work also focuses on strategies for caching, managing, and altering underlying Cloud structures to optimize the availability of cached results under these circumstances. Virtual services [94], like our approach, stores service results in intermediate caches architectures, have also been briefly proposed.

2.5 Scientific Computing in Cloud Environments

Since its emergence, there has been growing amounts of interest in testing the feasibility of performing data- and compute-intensive analysis in the Cloud. For instance, a plethora of efforts have been focused around the popular MapReduce paradigm [47],

which is being supported in various Cloud frameworks, including Google AppEngine [85], AWS Elastic MapReduce [15], among others. One early experience with scientific workflows in the Cloud is discussed in [89]. They showed that the running their scientific workflow over the Cloud was comparable to performance in a cluster, albeit that certain configuration overheads do exist in the Cloud. This specific scientific application is among several others that have been mapped onto the Cloud [172, 151, 114].

While Cloud-based data-intensive applications continues to grow, the cost of computing is of chief importance [12]. Several efforts have been made to assess costs for various large-scaled projects. Kondo, *et al.* compared cost-effectiveness of AWS against volunteer grids with the [103]. Deelman, *et al.* reported the cost of utilizing Cloud resources to support a representative workflow application, Montage [51]. They reported that CPU allocation costs will typically account for most of the cost while storage costs are amortized. Because of this, they found that it would be extremely cost effective to cache intermediate results in Cloud storage. Palankar *et al.* conducted an in-depth study on using S3 for supporting large-scale computing [132]. Our results on S3 echoed their findings, and we agree that S3 can be considered when average data size is large, and persistence is desirable.

In terms of utilizing the Cloud for caching/reusing data, Yuan, *et al.* proposed a strategy [182] for caching intermediate data of large-scale scientific workflows. Their system decides whether to store or evict intermediate data, produced at various stages of a workflow, based cost tradeoffs between storage and recomputation, and reuse likelihood. Their analysis ignores network transfer costs, which we showed to be a dominant factor. A tangential study by Adams *et al.*, which discussed the potentials of trading storage for computation [1].

Resource allocation is another related issue. Traditionally, a user requests a provision for some fixed set number of compute resources and reserve a span of time for exclusive usage. A common way to request for resources is batch scheduling, which is a ubiquitous mechanism for job submissions at most supercomputing or Grid sites [156]. Condor [115, 71] manages a distributed compute pool of otherwise idle machines, and puts them to use. Condor allocates resources for jobs based on a "matchmaking" approach [138]. Machines in the pool advertise their resource specifications as well as conditions under which it would be willing to take on a job. A submitted job must also advertise its needs, and Condor allocates the necessary resources by matching machines based on these specifications.

In [98], Juve and Deelman discussed the consequences of applying current resource provisioning approaches on the Grid/Cloud and argued that traditional queue-based reservation models can suffer massive delays due to the heterogeneity of Grid sites (different rules, priority, etc.). Raicu, *et al.*'s Falkon framework [137] describes an ad hoc resource pools which are preemptively allocated from disparate Grid sites by submitting a provisioning job. User applications submit jobs directly to the provisioner, rather than to the external site with the usual methods. Since the provisioner has already preemptively allocated the necessary resources, overheads of job submissions and dispatch are avoided. These advance reservation schemes, however, abstracts the actual provisioning of resources. As per Sotomayor, *et al.*'s observation, "... resource provisioning typically happen[s] as a side-effect of job submission" [158]. In their paper, they describe a lease-based approach toward resource provisioning, in an effort which seemingly precurses today's Cloud-usage methods, by leveraging virtual machine management. Singh, *et al.*'s provisioning model selects a set of resources to be provisioned that optimizes the application while minimizing the resource costs [153]. Providers advertise slots to the users, and each slot denotes the availability of

resources, (e.g., number of processors), which can be reserved for a certain timeframe, for a price. This allows application schedulers to optimize resource provisioning for their application based on cost. In a related effort, Huang, *et al.*'s scheme helps users by automatically selecting which resources to provision on a given workflow (DAG-based) application [92]. Our system exploits on-demand elastic resource management provided by the Amazon EC2 Cloud. In this paper, we have proposed algorithms to scale and relax compute resources to handle varying workloads driven by user interest.

Since the Cloud offers on-demand resource provisioning, the above problems appears inherently managed through the dynamic allocation of VMs. But this introduces a new class of problems, namely, the VM instantiation time is slow – on the order of minutes for Amazon EC2. This overhead is contributed by the fact that an entire OS image has to be loaded, and transference of memory (for statefulness) can take significant time [42, 105]. Of course, preallocated VM pools and Falkon-like preemptive allocation methods can be applied here, but the cost of preallocating machines versus the allocation overhead becomes difficult to justify in a Cloud setting. Work in this area involves ways of minimizing VM allocation times. In Vrable *et al.*'s Potemkin honeypot system, light-weight VMs are "*flash cloned*" from a static copy in the same machine using the memory optimization technique, copy-on-write. Observing that most VM functionalities are read, not written, new instances contain pointers from the static VM copy, until write operation is called [173]. Also on the forefront of this research area, SnowFlock [105] implements a VM Fork, which carries the same semantics as a regular process fork (stateful), but on the scale of a virtual machine. SnowFlock relies on a VM Descriptor, which is a condensed image with necessary metadata to spawn a new VM quickly. Its "lazy state replication" and "avoidance heuristics" reduces the amount of memory propagation to the child VM initially. The child's state is not initially transferred, but only read from the parent

when accessed. Both Potemkin and SnowFlock reported VM instance creation of under 1 second.

CHAPTER 3

DATA-DRIVEN SERVICE COMPOSITION

In Auspice, whose goal is to automatically, and more importantly, correctly synthesize service-based workflow plans, the need for domain knowledge is present on multiple levels. To exemplify, consider the case of having to answer the following user query:

‘‘`return an aerial image of (x,y) at time t`’’

Assuming that one correct way to plan for this workflow is to simply invoke a known service, getImage(x, y, t). Already, several major issues (among others) can be observed: (i) Of all the available services, how does the system know that getImage is the correct one to invoke? (ii) How does the system accurately assign the user values to their respective parameters? (iii) Has the user input enough information to invoke the service? We approach the former two problems here and address the third problem later in this chapter.

Available web services and data sets can be associated with certain *concepts* within a scientific domain. Abstractly, then, one can envision a concept-derivation scheme as a means to automatically compose the necessary service operations and data sets to answer queries. In such a scheme, a domain concept, c, is derived by a service s, whose parameter inputs, $(x_1, \ldots, x_p)$ are again substantiated by concepts $c(x_1), \ldots, c(x_p)$ respectively. Each of these concepts may be further derived by a service or data set. This chaining process continues until a terminal element (either a service without input or a file) has been reached on all concept paths.

We describe an ontology to capture these concept derivation relationships. Let ontology $O = (V_O, E_O)$ be a directed acyclic graph where its set of vertices, V_O, comprises a disjoint set of classes: concepts C, services S, and data types, D, i.e., $V_O = (C \cup S \cup D)$. Each directed edge $(u, v) \in E_O$ must denote one of the following relations:

- $(u\ \delta^{c \to s}\ v)$: concept-service derivation. Service $u \in S$ used to derive $v \in C$.
- $(u\ \delta^{c \to d}\ v)$: concept-data type derivation. Data type $u \in D$ used to derive $v \in C$.
- $(u\ \delta^{s \to c}\ v)$: service-concept derivation. Concept $u \in C$ used to derive service $v \in S$.

The ontological definition, depicted in Figure 3.1, simplifies the effort to indicate which services and data types are responsible for deriving specific domain concepts. A subset of the ontology we could expect for our aerial image example shown earlier is illustrated in Figure 3.2.

Focusing on the relevant portion of the ontology, the concepts, *image* and aerial should first be merged together to represent aerial-image. In this concept merge method, all outgoing edges are eliminated except those which share a common destination, in this case, the getImage service. We also see that domain knowledge is supplied to the getImage parameters through the *inputsFrom* edges connecting *latitude*, *longitude*, and *time*. These concepts are derived from $Q[\ldots]$, which denotes the user-supplied query values. Of course, other ways to derive aerial images might also exist, which allows our system to compose multiple plans for derivation. This *workflow enumeration* algorithm is discussed later.

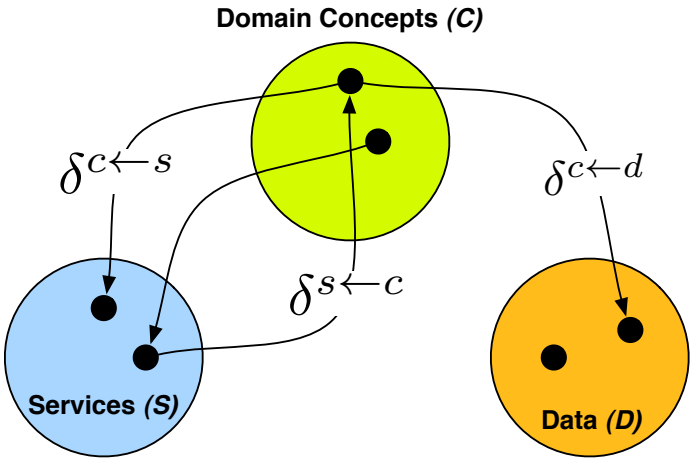


Figure 3.1: Ontology for Domain Description

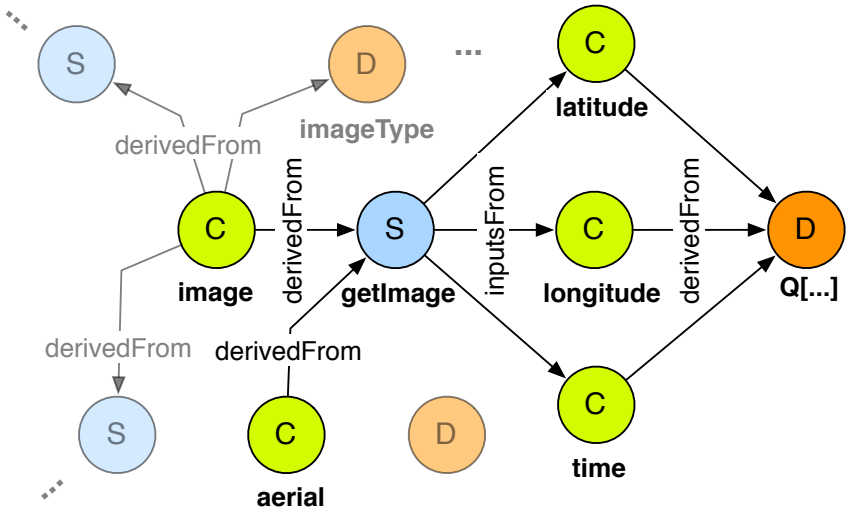


Figure 3.2: Subset Ontology for the Aerial Image Example

3.1 Metadata Registration

Because workflow planning is a necessary overhead, the existence of data sets (and services) must be identified quickly. Our goal, then, is to provide fast data identification. On one hand, we have the challenge of supplying useful domain knowledge to the workflow planner, and on the other, we have a plethora of pre-existing database/metadata management technologies that can be leveraged. The result is to utilize an underlying database to store and index domain-specific elements and, with the advantage of fast indices, the overhead of data identification for workflow planning can be optimized. For each data set, its indexed domain concepts can be drawn from an accompanying metadata file. However, metadata formats for describing scientific data sets can vary. There exists, for instance, multiple annotation formats from just within the geospatial community. But while their structures differ, the descriptors are similar, storing essential information (data quality, dates, spatial coverage, and so on) pertaining to specific data sets.

Domain experts initialize the system with the following[1]: (i) $\Upsilon = \{\upsilon_1, \ldots, \upsilon_n\}$, a set of XML Schema or Data Type Definitions (DTD) which defines the supported metadata formats used for validation. (ii) C_{idx}, a set of domain concepts that the system should index, and (iii) $xpath(\upsilon, c) : (\upsilon \in \Upsilon \wedge c \in C_{idx})$, For each indexed concept and schema, an XPath query [180] that is used to access the indexed value for concept c from a given the metadata document corresponding to schema υ. Once in place, domain users should be able to upload and not only share new data sets, but to also make it available for answering high-level queries.

To *register* new data sets with the system, users can invoke the Data Registration algorithm. This procedure, shown in Algorithm 1, takes three inputs: a data file

[1]Our implementation assumes that metadata is defined in XML.

Algorithm 1 registerData(d, $meta_d$[, K, υ_d])

1: ▷ identify and validate metadata
2: **if** $\upsilon_d \in \Upsilon \wedge \upsilon_d$.validate($meta_d$) = true **then**
3: $\quad\delta \leftarrow \upsilon_d$ ▷ input schema checks out
4: **else**
5: $\quad$**for all** $\upsilon \in \Upsilon$ **do**
6: $\quad\quad$**if** υ.validate($meta_d$) = true **then**
7: $\quad\quad\quad\delta \leftarrow \upsilon$ ▷ δ holds the corresponding schema
8: $\quad\quad$**end if**
9: $\quad$**end for**
10: **end if**
11: $c_K \leftarrow ConceptMapper$.map($K$) ▷ solve for concept derived by d
12: $d_K \leftarrow c_K \cdot$ "*type*"
13: **if** $\nexists\ d_K \in Ontology.D$ **then**
14: $\quad Ontology.D \leftarrow Ontology.D \cup \{d_K\}$
15: $\quad Ontology.Edges \leftarrow Ontology.Edges \cup \{(c_K, derivedFrom, d_K)\}$
16: **end if**
17: ▷ build database record
18: $R \leftarrow (datatype = d_K)$
19: **for all** $c \in C_{idx}$ **do**
20: $\quad v \leftarrow meta_d$.extract($xpath(\delta, c)$)
21: $\quad R \leftarrow record \cup (c = v)$ ▷ concatenate record
22: **end for**
23: DB.insert(R, d)

d, its metadata file $meta_d$, and an optional keyword array, $K[\ldots]$ that describes d, and an optional schema for validating $meta_d$, υ_d. The domain concept to which this data set derives can be computed. Optionally, but not shown, the user could select concepts directly from the ontology to describe d's type instead of providing $K[\ldots]$. To avoid confusion of schema validity and versioning, we emphasize here that the set of valid schemas, Υ, should only be managed by domain experts or administrators. That is, although users may potentially discover new metadata schemas, our system cannot allow them to update Υ directly.

Algorithm 1 starts by identifying the type of metadata, δ, prescribed by the user via validating $meta_d$ against the set of schemas, or directly against the user provided schema, υ_d (Lines 2-10). Next, the domain concept that is represented by $K[\ldots]$ is solved for on Line 11. On Line 12, d_K is assigned the name representing the

type of data in the ontology, where c_K is the matched concept and $\cdot$ denotes string concatenation. If necessary, d_K is added into the ontology's data type class, D, and an edge from c_K to d_K is also established (Lines 13-16). Finally, on Lines 18-23, a record is constructed for eventual insertion into the underlying database. The constructed record, R, is inserted into the database with a pointer to the data set, d. This process is illustrated in Figure 3.3.

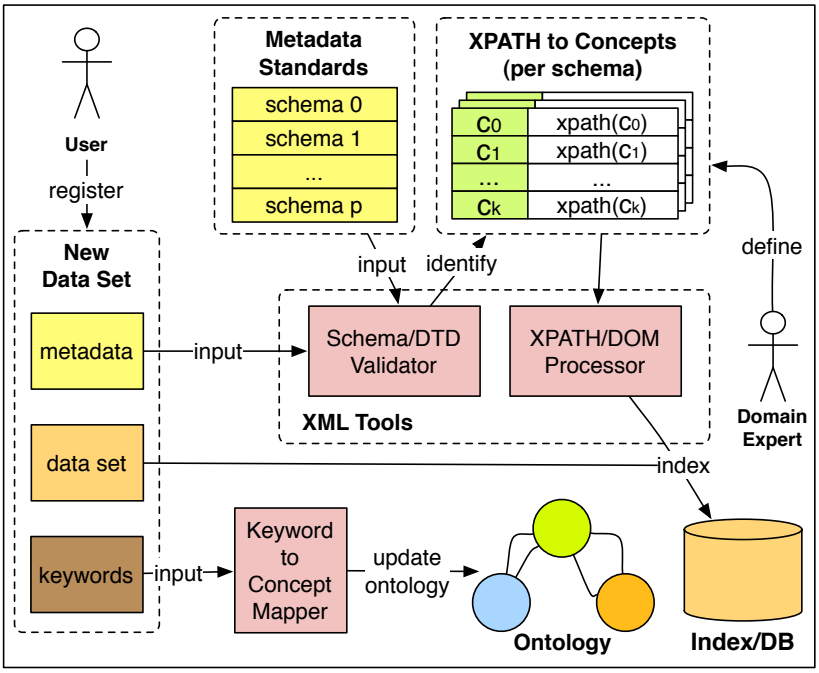


Figure 3.3: Data Registration

We clarify this process with an example. Consider an ontology that contains a *satellite image* concept, *sat*, derivable by files within two known data types, *LandSAT* and *IKONOS*. Let us now assume that a new, and more precise instrument, *quickbird*, becomes available, which generates a new type of data type becomes available. On

registering a quickbird image, the user would create the new data type, $QBIRD$, and because it still derives a satellite image, the ontology's edge record is updated as $\{(sat\ \delta^{c \to d}\ QBIRD)$, $(sat\ \delta^{c \to d}\ LandSAT)$, $(sat\ \delta^{c \to d}\ IKONOS)\}$. The file's metadata is also prepared for extraction. For each concept-XPath, which might point to the image's location, date and time the image was captured, among other concepts that can help systems identify it.

A nuance not captured by Algorithm 1 is index transformation, which modifies the extracted metadata values to fit the suitable underlying index. To illustrate, consider the following XPath queries that extract spatial and temporal information from CSDGM metadata:

```
date     //metadata/idinfo/timeperd/timeinfo/sngdate/caldate
northbc  //metadata/idinfo/spdom/bounding/northbc
southbc  //metadata/idinfo/spdom/bounding/southbc
westbc   //metadata/idinfo/spdom/bounding/westbc
eastbc   //metadata/idinfo/spdom/bounding/eastbc
```

```
<timeinfo>
   <caldate>2003-07-08</caldate>
   <time>14:18</time>
</timeinfo>
.
.
.
<bounding>
   <westbc>345708.61</westbc>
   <eastbc>349397.59</eastbc>
   <northbc>3058100.14</northbc>
   <southbc>3062563.34</southbc>
</bounding>
```

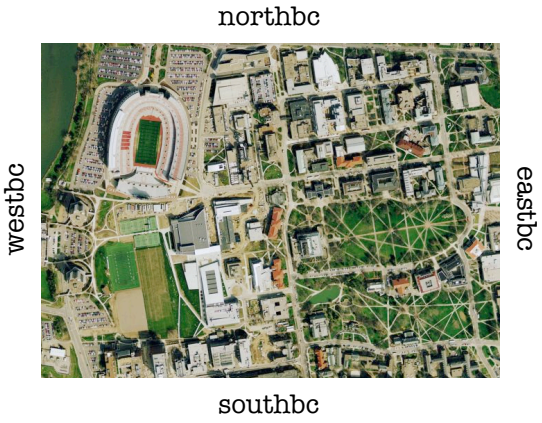


Figure 3.4: A Case for Index Transformation

The spatial boundary, denoted by northbc, southbc, westbc, and eastbc, is displayed in Figure 3.4. The problem observed with indexing these values independently is obvious: most database systems are equipped with spatial indices, and not employing their use in this case would be imprudent. Thus, we apply the following rule to transform extracted spatial values into the spatial index as supported by Open Geospatial Consortium's (OGC) OpenGIS extensions in the MySQL database [127]:

```
location   GeomFromText('POLYGON((westbc northbc, eastbc
                   northbc, eastbc southbc, westbc southbc,
                   westbc northbc))')
```

Here, the spatial *location* field in the database replaces the four independently extracted values, thus reducing the dimensionality and utilizes native indexing support.

Service registration is also enabled in Auspice. However, its usefulness will not be clear until we discuss cost modeling in the next chapter.

3.2 Service Composition and Workflow Enumeration

Often in practice, scientific tasks are composed of disparate processes chained together to produce some desired values [4]. Although workflows are rooted in business processes, their structures lend well to the realization of complex scientific computing [52, 129, 5, 119]. Also referred to as composite services in some literature, workflows are frequently expressed as directed acyclic graphs where the vertices denote services and data elements and directed edges represent flows of execution. Workflows, in our context, can also be recursively defined as follows. Given some set of data, D and a set of services S, a workflow, w is defined

$$w = \begin{cases} \epsilon \\ d \\ (op, (p_1, \ldots, p_k)) \end{cases}$$

such that terminals ϵ and $d \in D$ denote a null workflow and a data instance respectively. Nonterminal $(op, (p_1, \ldots, p_k)) \in S$ is a tuple where op denotes a service operation with a corresponding parameter list $(p_1, \ldots, p_k)$ and each p_i is itself a workflow. To put simply, a workflow is a tuple which either contains a single data instance or a service operation whose parameters are, recursively, (sub)workflows.

3.2.1 Workflow Enumeration Algorithm

Given some query q, the goal of workflow planning algorithm is to enumerate a list of workflows $W_q = (w_1, \ldots, w_n)$ capable of answering q from the available services and data sets. The execution of each $w_i \in W_q$ is carried out, if needed, by an order determined by cost or QoS parameters. Thus, upon workflow execution failure, the system can persistently attempt alternative, albeit potentially less optimal *vis-à-vis* QoS parameters (ensuing chapter), workflows.

Domain concept derivation is the goal behind constructing each workflow. Thus, our algorithm, WFEnum, relies heavily on the metadata and semantics provided in the *Semantics Layer*. Recall that the Query Decomposition component outputs the query's target concept, t, and a hashed set of query parameters, $Q[\ldots]$ (such that $Q[concept] \rightarrow \{val_1, val_2, \ldots\}$). The WFEnum algorithm takes both t and $Q[\ldots]$ as input, and outputs a list W of distinct workflows that are capable of returning the desiderata for the target concept.

WFEnum, shown in Algorithm 2, begins by retrieving all $d \in D$ (types of data registered in the ontology) from which the target concept, t, can be derived. On Line 2, a statically accessible array, $W'[\ldots]$, is used for storing overlapping workflows to save redundant recursive calls in the later half of the algorithm. The workflows are memoized on a hash value of their target concept and parameter list. On Line 5, a set of indexed concepts, C_{idx}, is identified for each data type, and checked against the

Algorithm 2 WFEnum(t, $Q[\dots]$)

```
1: W ← ()
2: global W'[...] ▷ static table for memoization
3: Λ_data ← Ontology.derivedFrom(D, t)
4: for all d ∈ Λ_data do
5:     C_idx ← d.getIndexConcepts()
6:     ▷ user-given values enough to substantiate indexed concepts
7:     if (Q.concepts() − C_idx) = {} then
8:         cond ← (datatype = d)
9:         for all c ∈ C_idx do
10:            cond ← cond ∧ (c = Q[c]) ▷ concatenate new condition
11:        end for
12:        F ← σ_<cond>(datasets) ▷ select files satisfying cond
13:        for all f ∈ F do
14:            W ← (W, (f))
15:        end for
16:    end if
17: end for
18:
19: Λ_srvc ← Ontology.derivedFrom(S, t)
20: for all op ∈ Λ_srvc do
21:    Π_op ← op.getPreconditions();
22:    (p_1, ..., p_k) ← op.getParameters()
23:    W_op ← ()
24:    for all p ∈ (p_1, ..., p_k) do
25:        ▷ forward query parameters s.t. preconditions are not violated
26:        Q_p[...] ← Q[...]
27:        for all (concept, value) ∈ Q_p[...] do
28:            if (concept, value).violates(Π_op) then
29:                Q_p[...] ← Q_p[...] − (concept, value)
30:            end if
31:        end for
32:        if ∃ W'[h(p.target, Q_p[...])] then
33:            W_p ← W'[h(p.target, Q_p[...])] ▷ recursive call is redundant
34:        else
35:            W_p ← WFEnum(p.target,Q_p[...]) ▷ recursively invoke for p
36:        end if
37:        W_op ← W_op × W_p ▷ cartesian product
38:    end for
39:    ▷ couple parameter list with service operation and concatenate to W
40:    for all pm ∈ W_op do
41:        W ← (W, (op, pm))
42:    end for
43: end for
44: W'[h(t, Q_p[...])] ← W ▷ memoize
45: return W
```

parsed user specified values in the query. To perform this check, if the set difference between the registered concepts, C_{idx}, and the query parameters, $Q[\ldots]$, is nonempty, then the user clearly did not provide enough information to plan the workflow unambiguously. On Lines 7-11, if all index registered concepts are substantiated by elements within $Q[\ldots]$, a database query is designed to retrieve the relevant data sets. For each indexed concept c, its *(concept=value)* pair, $(c = Q[c])$ is concatenated (AND'd) to the query's conditional clause. On Lines 12-15, the constructed query is executed and each returned file record, f, is an independent file-based workflow deriving t.

The latter half of the algorithm deals with concept derivation via service calls. From the ontology, a set of relevant service operations, Λ_{srvc} is retrieved for deriving t. For each operation, *op*, there may exist multiple ways to plan for its execution because each of its parameters, p , is a subproblem. Therefore, workflows pertaining to each parameter p must first be solved with its own target concept, $p.target$ and own subset of relevant query parameters $Q_p[\ldots]$. While $p.target$ is easy to identify from following the *inputsFrom* links belonging to *op* in the ontology, the forwarding of $Q_p[\ldots]$ requires a bit more effort. Looking past Lines 25-31 for now, this query parameter forwarding process is discussed in detail in Section 3.2.2.

Once the $Q_p[\ldots]$ is forwarded appropriately, the recursive call can be made for each parameter, or, if the call is superfluous, the set of workflows can be retrieved directly (Line 32-36). In either case the results are stored in W_p, and the combination of these parameter workflows in W_p is established through a cartesian product of its derived parameters (Line 37). For instance, consider a service workflow with two parameters of concepts a and b: $(op, (a, b))$. Assume that target concepts a is derived using workflows $W_a = (w_1^a, w_2^a)$ and b can only be derived with a single workflow $W_b = (w_1^b)$. The distinct parameter list plans are thus obtained as $W_{op} = W_a \times W_b =$

$((w_1^a, w_1^b), (w_2^a, w_1^b))$. Each element from W_{op} is a unique parameter list. These lists are coupled with the service operation, *op*, memoized in W' for avoiding redundant recursive calls in the future, and returned in W (Lines 39-45). In our example, the final list of workflows is obtained as $W = ((op, (w_1^a, w_1^b)), (op, (w_2^a, w_1^b)))$.

The returned list, W, contain planned workflows capable of answering an original query. Ideally, W should be a queue with the "best" workflows given priority. Mechanisms identifying the "best" workflows to execute, however, depends on the user's preferences. Our previous effort have led to QoS-based cost scoring techniques leveraging on bi-criteria optimization: workflow execution time and result accuracy. The gist of this effort is to train execution time models and also allow domain experts to input error propagation models per service operation. Our planner, when constructing workflows, invoke the prediction models based on user criteria. Workflows not meeting either constraint are pruned on the a priori principle during the enumeration phase. In the special case of when W is empty, however, a re-examination of pruned workflows is conducted to dynamically adapt to meet these constraints through data reduction techniques. This QoS adaptation scheme is detailed in the next chapter.

3.2.2 Forwarding Query Parameters

It was previously noted that planning a service operation is dependent on the initially planning of the operation's parameters. This means that WFEnum must be recursively invoked to plan (sub)workflows for each parameter. Whereas the (sub)target concept is clear to the system from *inputsFrom* relations specified in the ontology, the original query parameters must be forwarded correctly. For instance, consider some service-based workflow, $(op, (L_1, L_2))$ that expects as input two time-sensitive data files: L_1 and L_2. Let's then consider that *op* makes the following two assumptions: (i) L_1 is obtained at an earlier time/date than L_2 and (ii) L_1 and L_2 both represent the

same spatial region. Now assume that the user query provides two dates, 10/2/2007 and 12/3/2004 and a location (x, y), that is,

$$Q[\ldots] = \begin{cases} location \rightarrow \{(x, y)\} \\ date \rightarrow \{10/2/2007, 12/3/2004\} \end{cases}$$

To facilitate this distribution, the system allows a set of preconditions, Π_{op}, to be specified per service operation. All conditions from within Π_{op} must be met before allowing the planning/execution of *op* to be valid, or the plan being constructed is otherwise abandoned. In our case, the following preconditions are necessary to capture the above constraints:

$$\Pi_{op} = \begin{cases} L_1.date \preceq L_2.date \\ L_1.location = L_2.location \end{cases}$$

In Lines 25-31, our algorithm forwards the values accordingly down their respective parameter paths guided by the preconditions, and thus implicitly satisfying them. The query parameter sets thus should be distributed differently for the recursively planning of L_1 and L_2 as follows:

$$Q_{L_1}[\ldots] = \begin{cases} location \rightarrow \{(x, y)\} \\ date \rightarrow \{12/3/2004\} \end{cases} \qquad Q_{L_2}[\ldots] = \begin{cases} location \rightarrow \{(x, y)\} \\ date \rightarrow \{10/2/2007\} \end{cases}$$

The recursive planning for each (sub)workflow is respectively supplied with the reduced set of query parameters to identify only those files adhering to preconditions.

3.2.3 Analysis of WFEnum

In terms of time complexity, though it is hard to generalize its input, the enumeration algorithm is conservatively reducible to Depth-First Search (DFS). We can observe that, from the ontology, by initiating with the target concept node, it is necessary

to traverse all intermediate nodes until we reach the sinks (data), leaving us with a number of distinct paths and giving our algorithm the same time complexity as DFS in its worst case: $O(|E| + |V|)$. For clarity, we decompose its set of vertices into three familiar subsets: concepts nodes C, services nodes S, and data nodes D, i.e., $V = (C \cup D \cup S)$. Since the maximum number of edges in a DAG is $|E| = \frac{|V| * (|V| - 1)}{2}$, we yield a $O(|C \cup D \cup S|^2)$ upper bound. Although theoretically sound, it is excessively conservative. A recount of our ontological structure justifies this claim.

- $\nexists\ (u, v) : (u \in K \wedge v \in K) \mid K \in \{C, S, D\}$ — No edges exist within its own subgraph.
- $\nexists\ (u, v) : u \in S \wedge v \in D$ — Edges from service to data nodes do not exist.
- $\nexists\ (u, v) : u \in D$ — Data nodes are sinks, and thus contain no outgoing edges.

A more accurate measurement of the maximum number of edges in our ontology should be computed with the above constraints. We obtain $|E| = |C| \times (|S| + \frac{|D|}{2})$ and thus a significantly tighter upper bound.

3.3 Evaluating Workflow Enumeration

The experiments that we conducted are geared towards exposing two particular aspects of our system: (i) we run a case study from the geospatial domain to display its functionality, including metadata registration, query decomposition, and workflow planning. (ii) We show scalability and performance results of our query enumeration algorithm, particularly focusing on data identification.

To present our system from a functional standpoint, we employ an oft-utilized workflow example from the geospatial domain: shoreline extraction. This application

requires a Coastal Terrain Model (CTM) file and water level information at the targeted area and time. CTMs are essentially matrices (from a topographic perspective) where each point represents a discretized land elevation or bathymetry (underwater depth) value in the captured coastal region. To derive the shoreline, and intersection between the effective CTM and a respective water level is computed. Since both CTM and water level data sets are spatiotemporal, our system must not only identify the data sets efficiently, but plan service calls and their dependencies accurately and automatically.

For this example, the system's data index is configured to include the *date* and *location* concepts. In practice however, it would be useful to index additional elements such as resolution/quality, creator, map projection, and others. Next, we provided the system with two metadata schemas, the U.S.-based CSDGM [63] and the Australia and New Zealand standard, ANZMETA [11], which are both publicly available. Finally, XPaths formed from the schemas to index concepts *date* and *location* for both schemas are defined.

Next, CTM files, each coupled with corresponding metadata and keywords $K = \{$*"CTM", "coastal terrain model", "coastal model"*$\}$, are inserted into the system's registry using the data registration procedure provided in Algorithm 1. In the indexing phase, since we are only interested in the spatiotemporal aspects of the data sets, a single modified B^x-Tree [96] is employed as the underlying database index for capturing both *date* and *location*.[2] For the ontology phase, since a *CTM* concept is not yet captured in the domain ontology, the keyword-to-concept mapper will ask the user to either (a) display a list of concepts, or (b) create a new domain concept mapped from keywords K. If option (a) is taken, then the user chooses the relevant

[2]Jensen et al.'s B^x-Tree [96], originally designed for moving objects, is a B+Tree whose keys are the approximate linearizations of time and space of the object via space-filling curves.

concept and the incoming data set is registered into the ontology, and K is included the mapper's dictionary for future matches. Subsequent CTM file registrations, when given keywords from K, will register automatically under the concept *CTM*.

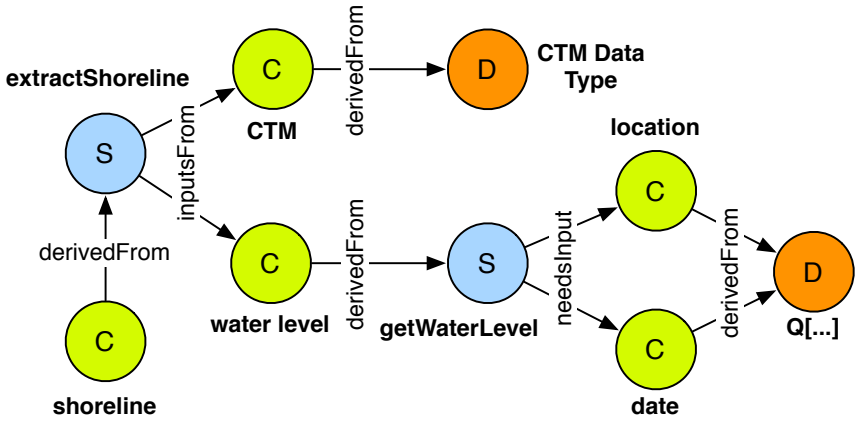


Figure 3.5: Example Ontology after Registration

On the service side, two operations are required for registration, shown below as $(op, (c_{p1}, c_{p2}, \ldots, c_{pk}))$, where op denotes the service operation name and c_{pi} denotes the domain concept of parameter i:

1. *(getWaterLevel, (date, location))*: retrieves the average water level reading on the given date from a coastal gauging station closest to the given location.

2. *(extractShoreline, (CTM, water level))*: intersects the given CTM with the water level and computes the shoreline.

For sake of simplicity, neither operation requires preconditions and cost prediction models. After metadata registration, the resulting ontology is shown in Figure 3.5,

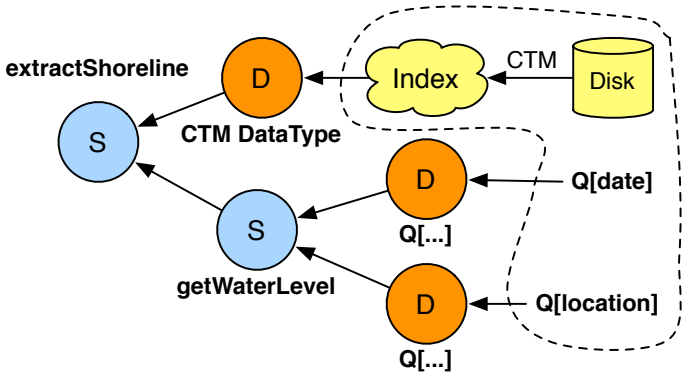


Figure 3.6: Shoreline Workflow Structure

unrolled for clarity. Albeit that there are a multitude of more nodes in a practical system, it is easy to see how the WFEnum algorithm would plan for shoreline workflows. By traversing from the targeted concept, *shoreline*, and visiting all reachable nodes, the workflow structure is a reduction of *shoreline*'s reachability subgraph with a reversal of the edges and a removal of intermediate concept nodes. The abstract workflow shown in Figure 3.6 is the general structure of all plannable workflows. In this particular example, WFEnum will enumerate more than one workflow candidate only if multiple CTM files (perhaps of disparate resolutions) are registered in the index at the queried location and time.

Auspice is distributed by nature, and therefore, our testbed is structured as follows. The workflow planner, including metadata indices and the query parser, is deployed onto a Linux machine running a Pentium 4 3.00Ghz Dual Core with 1GB of RAM. The geospatial processes are deployed as Web services on a separate server located across the Ohio State University campus at the Department of Civil and Environmental Engineering and Geodetic Science. CTM data sets, while indexed on the

workflow planner node, are actually housed on a file server across state, at the Kent State University campus.

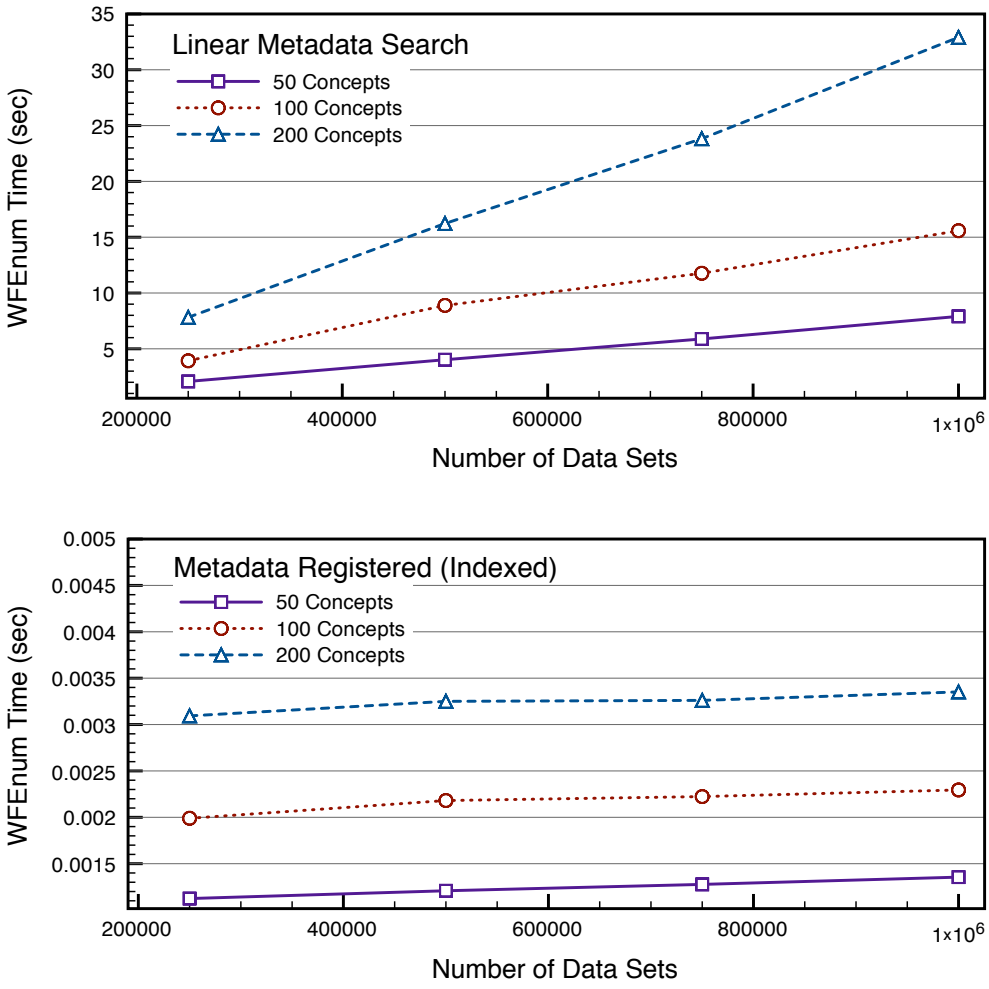


Figure 3.7: Planning Times with Increasing Data Sets and Concept Indices

In the first experiment, we are interested in the runtime of WFEnum with and without the benefit of metadata registration when scaled to increasing amounts of data files and concepts needing indexed (thus resulting in both larger index structures

and a larger number of indices). Shown in Figure 3.7 (top), the linear search version consumes significant amounts of time, whereas its counterpart (bottom) consumes mere milliseconds for composing the same workflow plan. Also, because dealing with multiple concept indices is a linear function, its integration into linear search produces drastic slowdowns. And although the slowdown can also be observed for the indexed runtime, they are of negligible amounts.

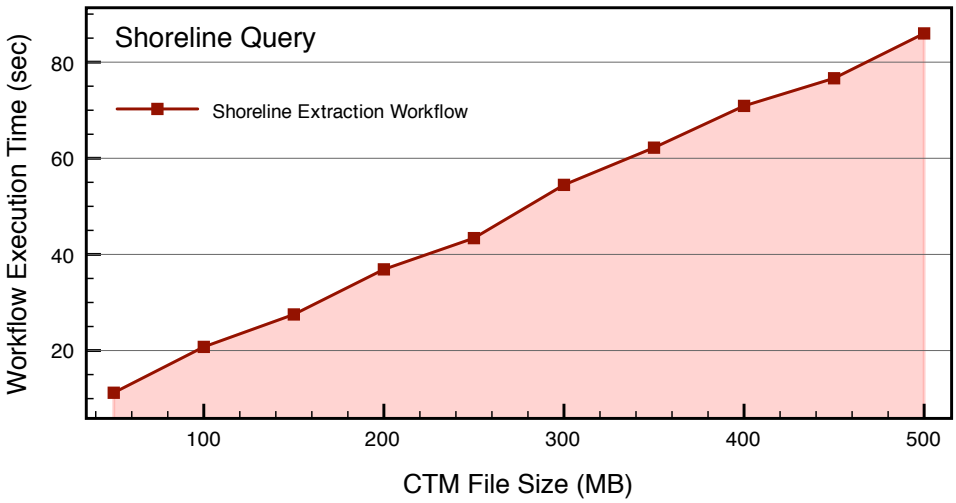


Figure 3.8: Shoreline Workflow Execution Times

Once the shoreline extraction workflow has finished planning, its execution is then carried out by our system. If we juxtaposed Figure 3.7 with Figure 3.8, the importance of minimizing planning time becomes clear. Especially for smaller CTM files, the cases when planning times dominate execution times should be avoided, and metadata indexing decreases the likelihood for this potential.

As seen in Figure 3.8, the workflow's execution time is heavily dependent on the CTM file size. Due to its data-intensive nature, we would expect that much larger CTM sizes will render the execution time prohibitive in time-critical scenarios. In the following chapter, we discuss ways to adjust accuracy on the fly as a means to meet time constraints.

3.4 Auspice Querying Interfaces

With the description of our ontology in place, which is a core component in Auspice, we are now able lead into the discussion of the querying interfaces.

3.4.1 Natural Language Support

We begin with another working example:

```
‘‘return water level from station=32125 on 10/31/2008’’
```

One functionality we wish to enable is the ability to process user queries in the form of high-level keyword or natural language. The job of the Query Decomposition Layer is to extract relevant elements from the user query. These elements, including the user's desiderata and other query attributes, are mapped to domain concepts specified in the Semantics Layer's ontology. Thus, these two layers in the system architecture are tightly linked. Shown in Figure 3.9, the decomposition process is two-phased.

In the Mapping Phase, StanfordNLP [102] is initially employed to output a list of terms and a parse tree from the query. The list of extracted query terms is then *stemmed* and *stopped*. This filtered set is further reduced using a synonym matcher provided through WordNet libraries [61]. The resulting term set is finally mapped to individual domain concepts from the ontology. Some terms, however, can only be

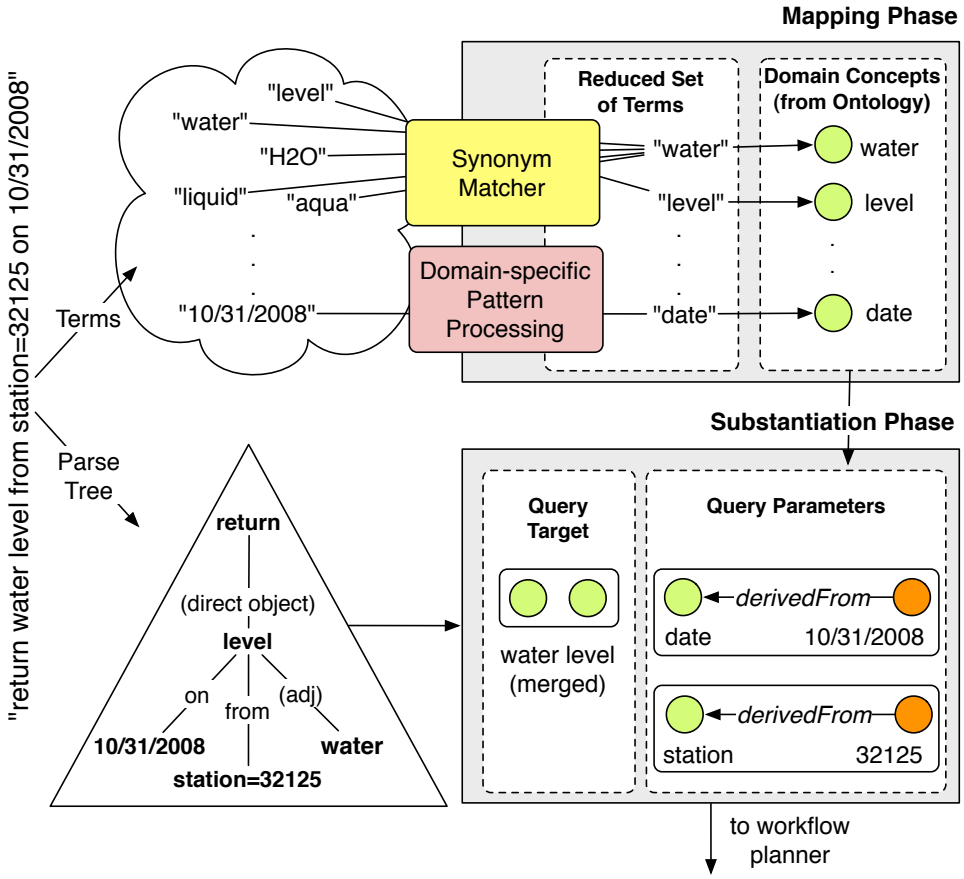


Figure 3.9: Query Decomposition Process

matched by their patterns. For example, "13:00" should be mapped to the concept, *time*. Others require further processing. A coordinate, (x, y), is first parsed and assigned concepts independently, (i.e., $x \leftarrow$ longitude and $y \leftarrow$ latitude). Because Auspice is currently implemented over the geospatial domain, only a limited number of patterns are expected. Finally, the last pattern involves value assignment. In our keyword system, values can be given directly to concepts using a *keyword=value* string. That is, the keyword query, "water level (x, y)" is equivalent to "water level latitude=y longitude=x". Finally, each query term is matched against this set of terms to identify their corresponding concepts. Indeed, a keyword may correspond with more than one concept.

Upon receiving the set of relevant concepts from the previous phase, the Substantiation Phase involves identifying the user's desired concept as well as assigning the given values to concepts. First, from the given parse tree, concepts are merged with their descriptors. In our example, since "water" describes the term "level", their respective domain concepts are merged. The pattern matcher from the previous phase can be reused to substantiate given values to concepts, resulting in the relations ($date$ $\delta^{c \to d}$ 10/31/2008) and ($station$ $\delta^{c \to d}$ 32125). These query parameter substantiations is stored as a hash set, $Q[\ldots] = Q[date] \rightarrow \{10/31/2008\}$ and $Q[station] \rightarrow \{32125\}$. This set of query parameters is essential for identifying accurate data sets in the workflow planning phase. Query parameters and the target concept, are sent as input to the workflow planning algorithm in the Planning Layer of the system.

We take prudence in discussing our query parser as not to overstate its functionalities. Our parser undoubtedly lacks a wealth of established natural language query processing features, for it was implemented ad hoc for interfacing with our specific domain ontology. We argue that, while related research in this area can certainly be

leveraged, the parser itself is ancillary to meeting the system's overall goals of automatic workflow planning and beyond the current scope of this work. Nonetheless, incorrectly parsed queries should be dealt with. Currently, with the benefit of the ontology, the system can deduce the immediate data that users must provide as long as the target concept is determined. The user can then enter the required data into a form for querying.

3.4.2 Keyword Search Support

Modern search engines have become indispensable for locating relevant information about almost anything. At the same time, users have probably also become aware of a common search engine's limitations. That is, sites, such as Yahoo! and Google, only unilaterally search Web pages' contents. However, with the continuous production of data from various domains, especially from within the sciences, information hidden deep within these domains cannot be reached with current search engines. To exemplify, let us consider that an earth science student needs to find out how much an area inside a nearby park has eroded since 1940. Certainly, if this exact information had previously been published onto a Web page, a common search engine could probably locate it without problems. But due to the query's *specificity*, the likelihood such a Web page exists is slim, and the chances are, our student would either have to be content with an approximate or anecdotal answer, or worse, give up. Various avenues for obtaining this information, however, probably do exist but are unknown to the student. In this section, we describe an approach and a system addressing this need.

Emerging technology and data sources have permitted the development of large-scale scientific data management projects. In one recent example, the Large Hadron

Collider (LHC Project) at CERN is projected to produce around 15PB of data annually [112]. In fact, just one of its experiments, ATLAS [110], is single-handedly expected to generate data on the rate of 5PB per year. This influx of data invokes a need for a similar dissemination of programs used for data analysis and processing. Fortunately, timely developments within the Web have enabled these programs to be shared and accessed remotely by anyone via Web services [41]. For many, including our earth science student, the explosion of data sets and Web services has been bittersweet. Within these resources lies the potential for making great discoveries, but deriving interesting results from using these resources has proved challenging for a number of reasons [161]. Among those, understanding where to find and how to compose existing Web services together with specific low-level data sets has been confined to a small class of experts. We believe that this situation betrays the spirit of the Web, where information is intended to be intuitively accessible by anyone, anywhere.

What elude users like our earth science student are possibilities hidden within the Web for answering complicated queries. Specifically, many queries cannot be answered by a single source, e.g., a Web page. But rather, these queries may involve an invocation of multiple resources, whose results are often combined together to form new information. The erosion information that our student seeks is perhaps one that can be derived from executing a series of computations, for instance, by composing geological Web services together, as one would with procedure calls, with relevant data found in various Web pages and data repositories. And even if users understood the steps toward deriving the desired information, the service composition process itself can be painstaking and error-prone.

In our approach, we maintain that a necessary ingredient to drive automatic workflow planning is domain knowledge. That is, as had been pressed in the previous sections, the system must understand the semantic relationships between the

available data sets and Web services. Returning to our example, this requires knowing the following: Which services can generate erosion results? What types of data (e.g., topographical, climate, water shed) do these erosion-producing Web services require as input? Which attributes from data sets are relevant to the query? In our student's case, only those data sets representing the park's location and the time of interest (1940 to today) should be considered for input into the erosion-producing Web services.

In the ensuing subsections, we discuss an approach for supporting keyword search in an automatic scientific workflow system. Upon receiving some keywords, our system returns a ranked list of relevant scientific workflow plans. Particularly, the result set is ranked according to the number of concepts (mapped by the keyword terms) that each workflow plan can derive.

Upon receiving the set of relevant concepts from the previous phase, the Substantiation Phase involves identifying the user's desired concept as well as assigning the given values to concepts. First, from the given parse tree, concepts are merged with their descriptors. In our example, since "water" describes the term "level", their respective domain concepts are merged. The pattern matcher from the previous phase can be reused to substantiate given values to concepts, resulting in the relations ($date\ \delta^{c \to d}$ 10/31/2008) and ($station\ \delta^{c \to d}$ 32125). These query parameter substantiations is stored as a hash set, $Q[\ldots] = Q[date] \to \{10/31/2008\}$ and $Q[station] \to \{32125\}$. This set of query parameters is essential for identifying accurate data sets in the workflow planning phase. Query parameters and the target concept, are sent as input to the workflow planning algorithm in the Planning Layer of the system.

As users submit keyword queries to the system, Auspice first applies some traditional stopping and stemming [181] filters to the terms. The terms are then mapped

to some respective concepts within the domain ontology. Recall that the ontology describes a directed acyclic graph which represents the relationships among available data sets, Web services, and scientific concepts. Once the set of ontological concepts has been identified, it is sent to the workflow planner. Guided by the set of ontological concepts, the planner composes Web services together with data files automatically and returns a ranked list of workflow candidates to the user. The user can then select and choose which workflow plan to execute. Next, we describe the ontology followed by the system's support for building such an ontology.

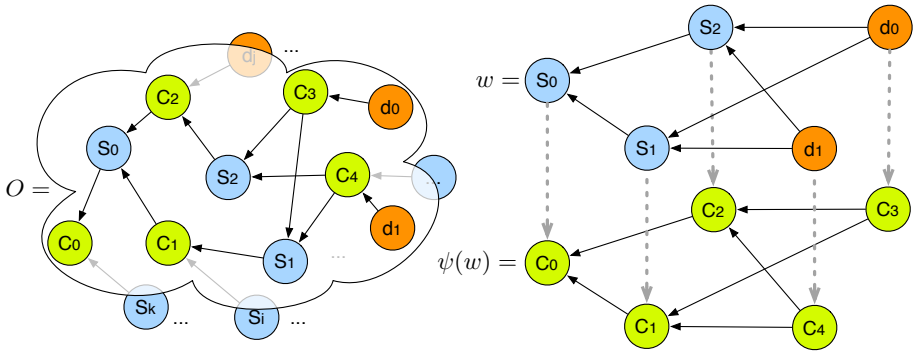


Figure 3.10: Examples of Ontology, Workflow, and ψ-Graph of w

Consider the ontology subset illustrated in the left side of Figure 3.10 as an example. If c_0 is the targeted concept in the query, one could traverse its edges in reverse order to reach all services and data types that are used to derive it. After executing this process, the respective workflow, w, is produced, shown on the upper-right side of the figure. We will revisit this figure in further detail later this section as we discuss the query planning algorithm, but first, we describe the process in which Auspice aids

Identifier	Description
O	The ontology, a directed acyclic graph, $O = (V_O, E_O)$
V_O	Set of instances (vertices) in O, $V_O = (C \cup S \cup D)$
E_O	Set of derivation edges in O
C, S, D	V_O's subset class of concepts, services and data types respectively
$(u\ \delta^{c \to s}\ v) \in E_O$	Concept-service derivation edge. Concept-service derivation edge. Service $u \in S$ used to derive $v \in C$
$(u\ \delta^{c \to d}\ v) \in E_O$	Concept-data type derivation edge. Data type $u \in D$ used to derive $v \in C$
$(u\ \delta^{s \to c}\ v) \in E_O$	Service-concept derivation edge. Concept $u \in C$ used to derive service $v \in S$
w	A workflow, which may be expressed as ϵ, d, or s, where ϵ is null, $d \in D$ is a data type, and $s \in S$ is a service. s is a non-terminal, i.e., the parameters for invoking s are themselves workflows.
ψ	The concept derivation graph. A reduced form of the ontology, but contains only concept vertices and edges. $\psi = (V_\psi, E_\psi)$
$\psi(c)$, $\psi(w)$	ψ-graph pertaining to $c \in C$ and to workflow w, respectively.

Table 3.1: Definitions of Identifiers Used throughout

users in constructing this ontology. For reading comprehension, we have summarized a list of identifiers in Table 3.1.

3.4.3 Keyword-Maximization Query Planning

To support keyword queries, we enumerate all workflows relevant to the most number of keywords in the user query, K. We currently support only AND-style keyword queries, and in this section, we discuss the process of the algorithms for automatically planning workflows given some set of keywords. Auspice's querying algorithm is to return all workflow plans, w, whose concept-derivation graph, $\psi(w)$ (to be discussed later), contains the most concepts from K, while under the constraints of the user's query parameters, Q. To exemplify the algorithms, we prescribe the ontology subset shown in Figure 3.11 to our discussion. Furthermore, we interweave the description of the algorithms with the keyword query example:

‘‘`wind coast line CTM image (41.48335,-82.687778) 8/20/2009`’’

Here, we note that the given coordinates point to Sandusky, Ohio, a location where we have abundant data sets.

3.4.4 Concept Mapping

The data and service metadata registration procedure, discussed previously, allows the user to supply some keywords that describe their data set or the output of the service. These supplied keywords are used to identify the concepts in which the new resource derives, and if such a concept does not exist, the user is given an option to create one in the ontology. As such, each concept, c, has an associated set of keywords, K_c. For instance, the concept of *elevation* might associate $K_{elevation} = \{$ "height", "elevation", "DEM"$\}$. The WordNet database [61] was also employed to expand K_c for the inclusion of each term's synonyms.

Before we describe the workflow enumeration algorithm, WFEnum_Key (shown as Algorithm 3), we introduce the notion of concept derivation graphs (or ψ-graphs) which is instrumental in WFEnum_Key for pruning. ψ-graphs are obtained as concept-derivation relationships, $\psi(c) = (V_\psi, E_\psi)$, where c is a concept, from the ontology. All vertices within $\psi(c)$ denote only concepts, and its edges represent derivation paths. As an aside, ψ can also be applied on workflows, i.e., $\psi(w)$ extracts the concept-derivation paths from the services and data sets involved in w. The graphic on the right of Figure 3.10 exemplifies $\psi(w)$. We revisit the left side of the figure, which illustrates an ontology, O, and vertices c_i, d_j, and s_k denote instances from the classes C, D, and S respectively. In the top-right side of the graphic, we show one derivation of c_0: $w = (s_0, ((s_1, (d_0, d_1)), (s_2, (d_0, d_1))))$. In the bottom-right of the figure, $\psi(w)$ is extracted from w. Although not shown, $\psi(c_0)$ would extract a significantly larger DAG; specifically, $\psi(c_0)$ would also include all concept paths leading in from services

s_i and s_k, but these have been hidden/ignored in this example. Indeed, for a concept c and a workflow w that derives c, $\psi(w) \subseteq \psi(c)$.

3.4.5 Planning with Keywords

WFEnum_Key's inputs include c_t, which denotes the targeted concept. That is, all generated workflows, w, must have a ψ-graph rooted in concept c_t. Specifically, only workflows, w, whose $\psi(w) \subseteq \psi(c_t)$ will be considered for the result set. The next input, Φ, is a set of required concepts, and every concept in Φ must be included in the derivation graph of c_t. A set of query parameters, Q, is also given to this algorithm. These would include the coordinates and the date given by the user in our example query. Q is used to identify the correct files and also as input into services that require these particular concept values. Finally, the ontology, O, supplies the algorithm with the derivation graph.

On Lines 2-8, the planning algorithm first considers all data-type derivation possibilities within the ontology for c_t, e.g., $(c_t\ \delta^{c \to d}\ d_t)$. All data files are retrieved with respect to data type d_t and the parameters given in Q. Each returned file record, f, is an independent file-based workflow deriving t. Next, the algorithm handles service-based derivations. From the ontology, O, all $(c_t\ \delta^{c \to s}\ s_t)$ relations are retrieved. Then for each service, s_t, that derives c_t, its parameters must first be recursively planned. Line 15 thus retrieves all concept derivation edges $(s_t\ \delta^{s \to c}\ c_{s_t})$ for each of its parameters. Opportunities for pruning are abundant here.

For instance, if the required set of concepts, Φ, is not included in the ψ-graphs of all s_t's parameters combined, then s_t can be pruned because it does not meet the query's requirements. For example, on the bottom left corner of Figure 3.11, we can imply that another service, *img2*, also derives the *image* concept. Assuming that $\Phi = \{shore\}$, because the ψ-graphs pertaining to all of *img2*'s parameters

Algorithm 3 WFEnum_Key(c_t, Φ, Q, O)

static W
for all concept-data derivation edges w.r.t. c_t, $(c_t\ \delta^{c\to d}\ d_t) \in E_O$ **do**
  ▷ data type d_t derives c_t; build on d_t
  $F \leftarrow \sigma_{<Q>}(d_t)$ //select files w.r.t. Q
  for all $f \in F$ **do**
    $W \leftarrow W \cup \{f\}$
  end for
end for
▷ any workflow enumerated must be reachable within Φ
for all concept-service derivation edges w.r.t. c_t, $(c_t\ \delta^{c\to s}\ s_t) \in E_O$ **do**
  ▷ service s_t derives c_t; build on s_t
  $W_{s_t} \leftarrow ()$
  ▷ remove target, c_t, from requirement set
  $\Phi \leftarrow \{\Phi \setminus c_t\}$
  for all service-concept derivation edges w.r.t. s_t, $(s_t\ \delta^{s\to c}\ c_{s_t}) \in E_O$ **do**
    ▷ prune if elements in Φ do not exist in c_{s_t}'s derivation path, that is, the union of all its parents' ψ graphs
    if $(\Phi \subseteq \bigcup \psi(c_{s_t}))$ **then**
      $W' \leftarrow$ WFEnum_Key(c_{s_t}, $\Phi \cap \psi(c_{s_t})$, Q, W, O)
      if $W' \neq ()$ **then**
        $W_{s_t} \leftarrow W_{s_t} \times W'$
        $W \leftarrow W \cup W'$
      end if
    end if
  end for
  ▷ construct service invocation plan for each $p \in W_{s_t}$, and append to W
  for all $p \in W_{s_t}$ **do**
    $W \leftarrow W \cup \{(s_t, p)\}$
  end for
end for
return W

does not account for the elements in Φ, *img2* can be immediately pruned here (Line 17). Otherwise, service s_t is deemed promising, and its parameters' concepts are used as targets to generate workflow (sub)plans toward the total realization of s_t. Recalling the workflow's recursive definition, this step is tantamount to deriving the nonterminal case where $(s_t, (w_1, \ldots, w_p)) \in S$. Finally whereas the complete plan for s_t is included in the result set (Line 27), W, each (sub)plan is also included because they include some subset of Φ, the required keyword concepts and therefore could be somewhat relevant to the user's query (Line 21).

With the planning algorithm in place, the natural extension now is to determine its input from a given list of keywords.

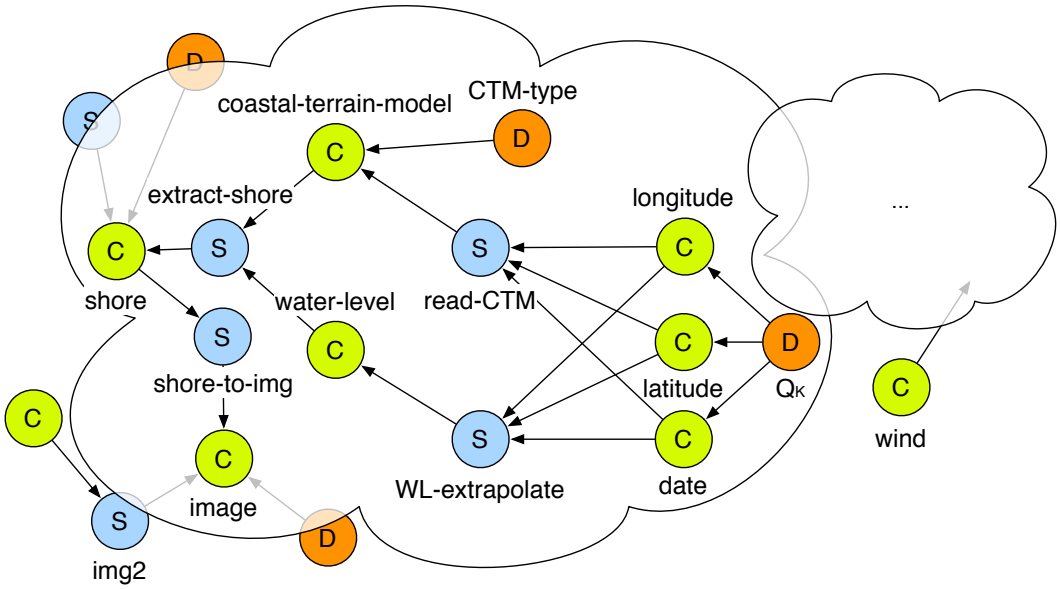


Figure 3.11: An Exemplifying Ontology

Algorithm 4 KMQuery(K, O)

$R \leftarrow ()$ $\quad$ R will hold the list of derived workflow results
$Q_K \leftarrow O$.mapParams(K)
$C_K \leftarrow O$.mapConcepts($K \setminus Q_K$)
▷ compute the power set, $\mathcal{P}(C_K)$, of C_K
for all $\rho \in \mathcal{P}(C_K)$, in descending order of $|\rho|$ **do**
    ▷ $\rho = \{c_1, \ldots, c_n\}, \{c_1, \ldots, c_{n-1}\}, \ldots, \{c_1\}$
    ▷ check for reachability within ρ, and find successor if true
    reachable $\leftarrow$ false
    for all $c_i \in \rho \ \wedge \ \neg$*reachable* **do**
        if $(\rho \setminus \{c_i\}) \subseteq \psi(c_i)$ **then**
            $c_{root} \leftarrow c_i$
            reachable $\leftarrow$ true
        end if
    end for
    if *reachable* **then**
        ▷ from ontology, enumerate all plans with c_{root} as target
        $R \leftarrow R \cup$ WFEnum_Key(c_{root}, $(\rho \setminus \{c_{root}\})$, Q_K, O)
        ▷ prune all subsumed elements from $\mathcal{P}(C_K)$
        for all $\rho' \in \mathcal{P}(C_K)$ **do**
            if $\rho' \subseteq \rho$ **then**
                $\mathcal{P}(C_K) \leftarrow \mathcal{P}(C_K) \setminus \{\rho'\}$
            end if
        end for
    end if
end for
return R

The query planning algorithm, shown in Algorithm 4, simply takes a set of keywords, K, and the ontology, O, as input, and the resulting list of workflow plans, R, is returned. First, the set of query parameters, Q_K, is identified using the concept pattern mapper on each of the key terms. Because user-issued parameter values are essentially data, they define a $\delta^{c \to d}$-type derivation on the concepts to which they are mapped. Here, (*longitude* $\delta^{c \to d}$ x), (*latitude* $\delta^{c \to d}$ y), (*date* $\delta^{c \to d}$ 8/20/2009), can be identified as a result (Line 2). The remaining concepts from K are also determined, $C_K = \{$*wind*, *shore*, *image*, *coastal-terrain-model*$\}$ (note that "coast" had been deduced to the concept, *shore*, and that "line" had been dropped since it did not match any concepts in O).

Next (Lines 5-14), the algorithm attempts to plan workflows incorporating all possible combinations of concepts within C_K. The power set, $\mathcal{P}(C_K)$ is computed for C_K, to contain the set of all subsets of C_K. Then, for each subset-element $\rho \in \mathcal{P}(C_K)$, the algorithm attempts to find the root concept in the derivation graph produced by ρ. For example, when $\rho = \{$*shore*, *image*, *coastal-terrain-model*$\}$, the root concept is *image* in Figure 3.11. However, when $\rho = \{$*shore*, *coastal-terrain-model*$\}$, then $c_{root} =$ *shore*. But since any workflows produced by the former subsumes any produced by the latter ρ set of concepts, the latter can be pruned (thus why we loop from descending order of $|\rho|$ on Line 5). In order to perform the root-concept test, for each concept element, $c_i \in \rho$, its ψ-graph, $\psi(c_i)$ is first computed, and if it consumes all other concepts in ρ, then c_i is determined to be the root (recall that $\psi(c_i)$ generates a concept-derivation DAG rooted in c_i).

Back to our example, although *wind* is a valid concept in O, it does not contribute to the derivation of any of the relevant elements. Therefore, when $\rho = \{$*wind*, *image*, *shore*, *coastal-terrain-model*$\}$, no plans will be produced because *wind* is never reachable regardless of which concepts is considered root. The next ρ, however,

produces {*image*, *shore*, *coastal-terrain-model*}. Here, $\psi(image)$ incorporates both *shore* and *coastal-terrain-model*, and thus, *image* is determined to be c_{root}. The inner loop on Line 9 can stop here, because the DAG properties of O does not permit $\psi(shore)$ or ψ(coastal-terrain-model) to include *shore*, and therefore neither can be root for this particular ρ.

When a reachable ρ subset has been determined, the planning method, WFEnum_Key can be invoked (Lines 15-24). Using c_{root} as the targeted with $\rho \setminus \{c_{root}\}$ being the concepts required in the derivation paths toward c_{root}, WFEnum_Key is employed to return all workflow plans. But as we saw in Algorithm 3, WFEnum_Key also returns any workflow (sub)plans that were used to derive the target. That is, although *image* is the target here, the *shore* concept would have to be first derived to substantiate it, and it would thus be included in R as a separate plan. Due to this redundancy, after WFEnum_Key has been invoked, Lines 18-23 prunes the redundant ρ's from the power set. In our example, every subset element will be pruned except when $\rho = \{wind\}$. Therefore, *wind* would become rooted its workflows will likewise be planned separately.

3.4.6 Relevance Ranking

The resulting workflow plans should be ordered by their relevance. Relevance, however, is a somewhat loose term under our context. We simply define relevance as a function of the number of keyword-concepts that appear in each workflow plan. We, for instance, would expect that any workflow rooted in *wind* be less relevant to the user than the plans which include significantly more keyword-concepts: *shore*, *image*, etc. Given a workflow plan, w, and query, K, we measure w's relevance score, as

follows:

$$r(w, K) = \frac{|V_\psi(w) \cap C(K)|}{|C(K)| + \log(|V_\psi(w) \setminus C(K)| + 1)}$$

Recall that $V_\psi(w)$ denotes the set of concept vertices in w's concept derivation graph, $\psi(w)$. Here, $C(K)$ represents the set of concept nodes mapped from K. This equation corresponds to the ratio of the amount of concepts from $C(K)$ that w captures. The log term in the denominator signifies a slight *fuzziness* penalty for each concept in w's derivation graph that was not specified in K. The motivation for this penalty is to reward "tighter" workflow plans are that more neatly represented (and thus, more easily understandable and interpreted by the user). This metric is inspired by traditional approaches for answering keyword queries over relational databases [170, 3].

3.4.7 A Case Study

We present a case study of our keyword search functionality in this section. Our system is run on an Ubuntu Linux machine with a Pentium 4 3.00Ghz Dual Core with 1GB of RAM. This work has been a cooperative effort with the Department of Civil and Environmental Engineering and Geodetic Sciences here at the Ohio State University. Our collaborators supplied us with various services that they had developed to process certain types of geospatial data. A set of geospatial data was also given to us. In all, the ontology used in this experiment consists of 29 concepts, 25 services, 5 data types. The 25 services and 2248 data files were registered to the ontology based on their accompanying metadata, solely for the purposes of this experiment. We note that, although the resource size is small, the given is sufficient for evaluating the *functionality* of keyword search support. A set of queries, shown in Table 3.2, are used to evaluate our system.

Query ID	Description
1	"coast line CTM 7/8/2003 (41.48335,-82.687778)"
2	"bluff line DEM 7/8/2003 (41.48335,-82.687778)"
3	"(41.48335,-82.687778) 7/8/2003 wind CTM"
4	"waterlevel=174.7cm water surface 7/8/2003 (41.48335,-82.687778)"
5	"waterlevel (41.48335,-82.687778) 13:00:00 3/3/2009"
6	"land surface change (41.48335,-82.687778) 7/8/2003 7/7/2004"

Table 3.2: Experimental Queries

First, we present the search time of the six queries issued to the system. In this experiment, we executed the search using two versions of our algorithm. Here, the search time is the sum of the runtimes for KMQuery and WFEnum_Key algorithms. The first version consists of the a-priori pruning logic, and the second version does not prune until the very end. The results of this experiment are shown in Figure 3.12, and as we can see, a typical search executes on the order of several milliseconds, albeit that the ontology size is quite small.

We can also see that the pruning version results in slightly faster search times in almost all queries, with the exception of QueryID=3. It was later verified that this query does not benefit from pruning with the given services and data sets. In other words, the pruning logic is an overhead for this case. Along the right y-axis, the result set size is shown. Because the test data set is given by our collaborators, in addition to the fact that our search algorithm is exhaustive, we can claim (and it was later verified) that the recall is 100%. Recall by itself, however, is not sufficient to measuring the effectiveness of the search.

To measure the precision of the result set, we again required the help of our collaborators. For each workflow plan, w in the result set, the domain experts assigned a score, $r'(w, K)$ from 0 to 1. The precision for each plan is then measured relative to the difference of this score to the relevance score, $r(w, K)$, assigned by our search

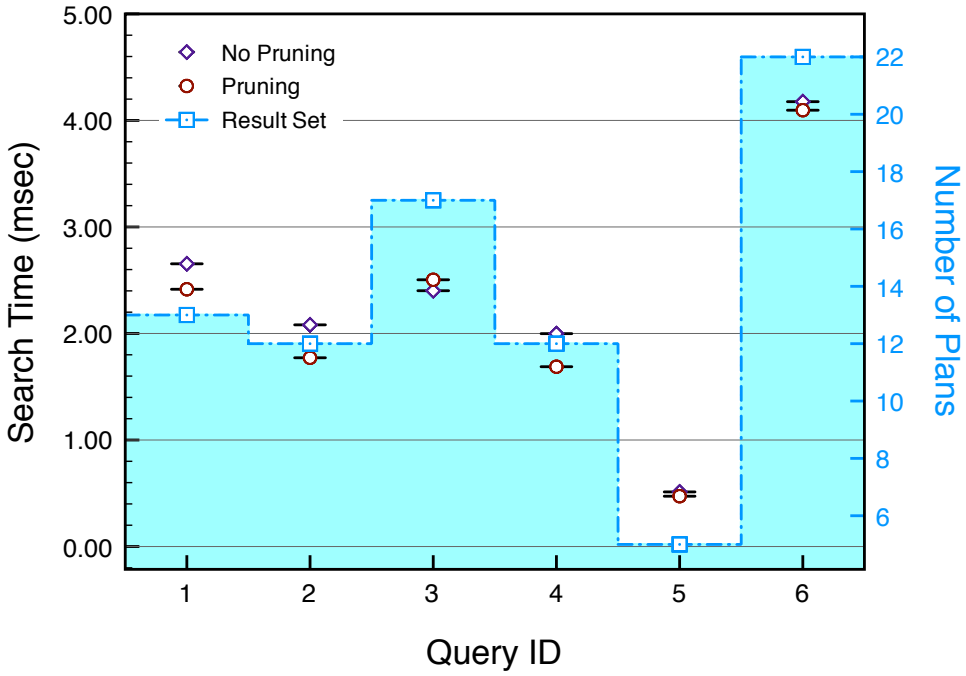


Figure 3.12: Search Time

engine. For a result set R, its precision is thus computed,

$$prec(R, K) = \frac{1}{|R|} \sum_{w \in R} 1 - (|r(w, K) - r'(w, K)|)$$

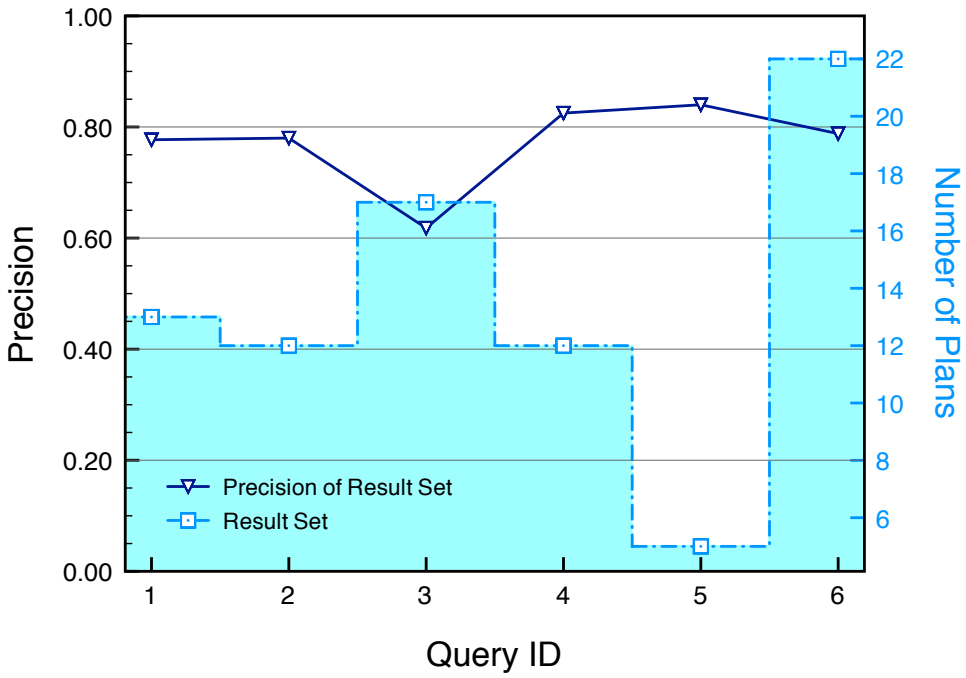


Figure 3.13: Precision of Search Results

The precision for our queries is plotted in Figure 3.13. Most of the variance are introduced due to the fact that our system underestimated the relevance of some plans. Because Query 3 appeared to have performed the worst, we show its results in Table 3.3.

The third query contains five concepts after keyword-concept mapping: *wind*, *date*, *longitude*, *latitude*, and *coastal-terrain-model*. The first five plans enumerated captures all five concepts plus "water surface", which is superfluous to the keyword

Workflow Plan	r	r'
GetWindVal(GetWaterSurface(getCTMLowRes(CTM42.dat)))	0.943	0.8
GetWindVal(GetWaterSurface(getCTMMedRes(CTM42.dat)))	0.943	0.8
GetWindVal(GetWaterSurface(getCTMHighRes(CTM42.dat)))	0.943	0.8
GetWindVal(GetWaterSurface(CreateFromUrlLowRes(CTM42.dat)))	0.943	0.8
GetWindVal(GetWaterSurface(CreateFromUrlHighRes(CTM42.dat)))	0.943	0.8
getCTMLowRes(CTM42.dat)	0.8	0.3
getCTMMedRes(CTM42.dat)	0.8	0.3
getCTMHighRes(CTM42.dat)	0.8	0.3
CreateFromUrlLowRes(CTM42.dat)	0.8	0.3
CreateFromUrlHighRes(CTM42.dat)	0.8	0.3
CTM42.dat	0.8	0.3
GetWaterSurface(getCTMLowRes(CTM42.dat))	0.755	0.3
GetWaterSurface(getCTMMedRes(CTM42.dat))	0.755	0.3
GetWaterSurface(getCTMHighRes(CTM42.dat))	0.755	0.3
GetWaterSurface(CreateFromUrlLowRes(CTM42.dat))	0.755	0.3
GetWaterSurface(CreateFromUrlHighRes(CTM42.dat))	0.755	0.3

Table 3.3: QueryID 3 Results Set and Precision

query. Therefore, any plans generating a water surface will be slightly penalized. Note that, while the variance is relatively high when compared with the user's expectations, the scores do not affect the user's expected overall ordering of the results. Although, it certainly can be posited that other properties, such as cost/quality of the workflow, can be factored into the relevance calculation.

CHAPTER 4

PLANNING WORKFLOWS WITH QOS AWARENESS

Advancements in Web and Grid computing have propelled a movement towards making data sets and computing services widely available. The result is a high number of distributed data repositories storing large volumes of data sets over links with potentially high latency and access costs. In these scenarios a workflow's overall execution time can be impacted by the high access costs of moving Grid-based data sets.

Often in these distributed computing models, there are multiple ways of answering a given query, using different combinations of data sources and services. For instance, in the simple case of obtaining water level readings for a coast line region in Lake Erie, one way would be to retrieve it directly from available data sets. A different method involves extracting this information from water surface models.

Some methods are likely to result in higher cost, but better accuracy, whereas others might lead to quicker results and lower accuracy. This could be due to that some data collection methods involve higher resolution or because some data sets are available at servers with lower access latencies. Meanwhile, different classes of users can have different querying requirements. For instance, some users may want the fastest answers while others require the most accurate response. Users may also prefer the faster of the methods which can meet certain accuracy constraints. While most efforts in scientific workflow systems focus directly on minimizing execution

times [134, 184, 2] through scheduling heuristics, it would be highly desirable to enable user preferences for both accuracy and time.

Meanwhile, different users interacting with the query framework can have different requirements. Some users may want the answers the fastest, some may demand the most accurate answers, and others might prefer the faster of the methods which can meet certain accuracy constraints. It will be highly desirable if a workflow composition system can incorporate user constraints and preferences. In other words, we want to alleviate the users from the need of understanding the cost and accuracy tradeoffs associated with different data sets and services that could be used to answer a query.

Auspice seeks to remove from users the need of having to understand the cost and accuracy tradeoffs associated with different data sets and services that could be used to answer a query. To automate the time-accuracy tradeoff in service composition, we allow developers to expose an accuracy parameter, e.g., sampling rate. Our system takes unrestricted models as input for predicting process completion time and error/accuracy propagation of the applications. Our service composition algorithm employs an efficient procedure to automatically regulate the accuracy parameter based on the defined cost models to meet the user-specified QoS constraints. We conducted experiments to evaluate two aspects of our algorithm. First, we show that, although the cost models can be invoked quite frequently during workflow planning, they contribute little overhead to the overall planning time. Secondly, we present the effect that the accuracy parameter adjustment scheme has on planned workflows.

We present a framework for workflow composition that supports user preferences on time and accuracy. Our framework includes models for assessing cost associated with retrieving each data set or executing any services on a given input. Similarly, we take as input models that predict accuracy associated with the results of a particular

service, as a function of the accuracy of the data input to the service. User-specified constraints on accuracy and cost are also taken as input. The system automatically composes workflows while pruning candidates that cannot meet the constraints.

The uses for our framework is three-fold. First, the integration of cost to our workflow composition algorithm allows the system to support user constraints. Secondly, these constraints increase the efficiency of our workflow composition algorithm by enabling the pruning of workflows at an early stage if QoS constraints are not met on the apriori principle. Lastly, we enable opportunities for service developers to expose parameters such that, when tuned, can affect the execution time and accuracy of the composed workflows. Given time restrictions or such physical limitations as low network bandwidth, our system is capable of dynamically suggesting recommended values for the exposed accuracy parameters in order to maximize the user's expectations. To the best of our knowledge, ours is the first automatic service composition system with these capabilities. This framework has been incorporated in the Planning Layer of Auspice.

Recall from the previous chapter that Auspice provides support for keyword queries. The QoS constraints, if available, are input by the user with the keyword pairs: *QoS:Time=seconds* and *(QoS:ErrConcept=concept, QoS:Err=value)*. The latter pair for applying error restriction is important to note, since workflows may involve multiple error models. For instance, consider a service such that, when given an aerial image of a city and corner coordinates, crops and returns a new image with the specified boundaries. Here, errors can include the resolution of the cropped image or measurement discrepancies involving the actual cropping, e.g., the cropping service is accurate up to $\pm$5cm. Multiple error constraints may be coupled together in the same query. Conversely, the only constraint the user has is time. In this case, a user might request:

```
‘‘aerial northbound=(x,y) southbound=(x,y), ... (QoS:Time=120s)’’
```

It is worth noting that if only one constraint is given, then system attempts to abide the restriction while optimizing the undefined constraint, and that if neither is provided, the system will execute the workflow containing the lowest error. The user may also request that all workflows meeting the constraints be returned. In this case the user is given time and error predictions of each workflow, and he/she selects which to execute. Given this well-structured query, appropriate services and data sets must be selected for use and their composition is reified dynamically through consultation with the domain ontology. Through this process, the workflow construction engine enumerates a *set* of valid workflow candidates such that when each is executed, returns a suitable response to the query.

From the set of workflow candidates, the service composition engine must then examine the cost of each in order to determine a subset that meet user constraints. Additionally, this component can dynamically adjust accuracy parameters in order to meet expected time constraints set by the user. Although shown as a separate entity for clarity, the pruning mechanism is actually pushed deep within the workflow construction engine. The remaining candidate set is sorted top-down according to either the time or accuracy constraint (depending on preference) to form a queue. The execution of workflows is carried out and the presence of faults within a certain execution, caused by such factors as network downtime or data/process unavailability, triggers the next queued workflow to be executed to provide the most optimal possible response.

4.1 Modeling Service Workflow Cost

Two cost functions are introduced for aggregating workflow execution time and error propagation respectively. Recall the definition of a workflow from the previous chapter reduces to a service, data, or null. A workflow w's time cost can be estimated by:

$$T(w) = \begin{cases} 0, & \text{if } \ w = \epsilon \\ t_{net}(d), & \text{if } \ w \in D \\ t_x(op, P_{op}) + t_{net}(op, P_{op}) + \max\limits_{p_i \in P_{op}} T(p_i), & \text{if } \ w \in S \end{cases}$$

If workflow w is a base data element, then $w = d$, and the cost is trivially the data transmission time, t_{net}. When w is a service, then $w = (op, P_{op})$, and its time can be summarized as the sum of the service's execution time t_x, network transmission time of its product, and, recursively, the maximum time taken by its parameters (assuming their execution can be carried out concurrently).

The error aggregation function, $E(w)$, which represents the error estimation of a given workflow, is also in a recursive sum form:

$$E(w) = \begin{cases} 0, & \text{if } \ w = \epsilon \\ \sigma(d, \gamma), & \text{if } \ w \in D \\ \sigma(op, P_{op}, \gamma) + f(E(p_i)), & \text{if } \ w \in S \end{cases}$$
$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad p_i \in P_{op}$$

Due to the heterogeneity of data sets and processes, it is expected that disparate workflows will yield results with fluctuating measures of accuracy. Again, at the base case lies the expected error of a particular data set, $\sigma(d, \gamma)$. Here, γ denotes an accuracy parameter with respect to the data set, e.g., resolution, sampling rate, etc. An error value can also be attributed to a service execution, $\sigma(op, P_{op}, \gamma)$. For instance, errors will be introduced if a sampling service is called to reduce data size or some interpolation/extrapolation service is used estimate some value. In the third

case, function f depends on the operation op, i.e., f is max when op is independent. However, f could denote multiplication when op is a join operation.

The obvious goal is providing prudent and reliable measures since cost is the determining factor for pruning workflow candidates. Furthermore, the online computation of cost should only require diminutive overhead. For each service, we are interested four separate models: The $T(w)$ term itself involves the implementation of three distinct models for service execution time (t_x), network transmission time (t_{net}), and, implicitly, an estimation of output size ($size_d$). For t_x, we sampled service runtime by controlling various sized inputs and generating multi-regression models. $size_d$ was computed on a similar basis (note that $size_d$ is known for files). The network transmission time t_{net} was approximated as the ratio of $size_d$ over bandwidth between nodes that host each service or data. Regression, however, cannot be used to reliably capture the capricious nature of an error model. It depends heavily on the application's mechanisms and is largely domain specific. Thus, our model must capture arbitrarily complex equations given by domain experts.

4.2 Workflow Enumeration and Pruning

The goal of a workflow planning algorithm is to enumerate a sequence of workflows $W_q = (w_1, \ldots, w_n)$ capable of answering some query q by employing the available services and data sets. The execution of each $w_i \in W_q$ is carried out, if needed, by an order determined by cost or QoS parameters. Thus, upon workflow execution failure, the system can persistently attempt alternative, albeit potentially less optimal, workflows.

Our QoS-aware service composition algorithm, WFEnumQOS (Algorithm 5), is a slight modification of the Algorithm 2. We summarize the original algorithm here, and discuss the additional modification. The WFEnumQOS algorithm takes as input

the query's targeted domain concept, $target$, the user's time constraint, QoS_{time}, and error constraint, QoS_{error}. WFEnumQoS runs a modification of Depth-First Search on the domain ontology starting from $target$. It is defined by the ontology that every concept can be realized by various data types or services. WFEnumQOS starts (Line 2) by retrieving a set, Λ_{data} of all data types that can be used to derive the input concept, $target$. Each element in Λ_{data} is a potential data workflow candidate, i.e., $target$ can be derived by the contents within some file. Correctly and quickly identifying the necessary files based on the user's query parameters (Line 4) is a challenge and out of the scope of this work. On Line 7, each file is used to call an auxiliary procedure, QoSMerge, to verify that its inclusion as a workflow candidate will not violate QoS parameters.

Algorithm 5 WFEnumQOS($target, QoS_{time}, QoS_{error}$)

```
 1: W ← ()
 2: Λ_data ← Ontology.derivedFrom(D, target)
 3: for all dataType ∈ Λ_data do
 4:     F ← dataType.getFiles()
 5:     for all f ∈ F do
 6:         w ← (f)
 7:         W ← (W, QoSMerge(w, ∞, ∞, QoS_time, QoS_error))
 8:     end for
 9: end for
10: Λ_srvc ← Ontology.derivedFrom(S, target)
11: for all op ∈ Λ_srvc do
12:     P_op ← op.getParams()
13:     W_op ← ()
14:     for all p ∈ P_op do
15:         W_p ← WFEnumQoS(p.target, QoS)
16:         W_op ← W_op × W_p
17:     end for
18:     for all pm ∈ W_op do
19:         w ← (op, pm)
20:         W ← (W, QoSMerge(w, ∞, ∞, QoS_time, QoS_error))
21:     end for
22: end for
23: return W
```

The latter half of the WFEnumQoS algorithm handles service-based workflow planning. From the ontology, a set of relevant service operations, Λ_{srvc} is retrieved for deriving $target$. For each service operation, op, there may exist multiple ways to plan for its execution because each of its parameters, p, by definition, is a (sub)problem. Therefore, workflows pertaining to each parameter p must first be computed via a recursive call (Line 15) to solve each parameter's (sub)problem, whose results are stored in W_p. The combination of these parameter (sub)workflows in W_p is then established through a cartesian product of its derived parameters (Line 16). For instance, consider a service workflow with two parameters of concepts a and b: $(op, (a, b))$. Assume that target concepts a is derived using workflows $W_a = (w_1^a, w_2^a)$ and b can only be derived with a single workflow $W_b = (w_1^b)$. The distinct parameter list plans are thus obtained as $W_{op} = W_a \times W_b = ((w_1^a, w_1^b), (w_2^a, w_1^b))$. Each tuple from W_{op} is a unique parameter list, pm. Each service operation, when coupled with a distinct parameter list (Line 19) produces an equally distinct service-based workflow which again invokes QoSMerge for possible inclusion into the final workflow candidate list (Line 20). In our example, the final list of workflows is obtained as $W = ((op, (w_1^a, w_1^b)), (op, (w_2^a, w_1^b)))$.

When a workflow becomes a candidate for inclusion, QoSMerge (Algorithm 6) is called to make a final decision: prune, include as-is, or modify workflow accuracy then include. For simplicity, we consider a single error model, and hence, just one adjustment parameter in our algorithm. QoSMerge inputs the following arguments: (i) w, the workflow under consideration, (ii) t' and (iii) e' are the predicted time and error values of the workflow from the previous iteration (for detecting convergence), and (iv) QoS_{time} and QoS_{error} are the QoS objects from the query.

Initially, QoSMerge assigns convergence thresholds C_E and C_T for error and time constraints respectively. These values are assigned to ∞ if a corresponding QoS is not given. Otherwise, these thresholds assume some insignificant value. If the current

Algorithm 6 QoSMerge($w, t', e', QoS_{time}, QoS_{error}$)

```
1: ▷ no time constraint
2: if QoS.Time = ∞ then
3:     C_T ← ∞
4: end if
5: ▷ no accuracy constraint
6: if QoS.Err = ∞ then
7:     C_E ← ∞
8: end if
9: ▷ constraints are met
10: if T(w) ≤ QoS_time ∧ E(w) ≤ QoS_error then
11:     return w    ▷ return w in current state
12: end if
13: ▷ convergence of model estimations
14: if |T(w) − t'| ≤ C_T ∧ |E(w) − e'| ≤ C_E then
15:     return ∅    ▷ prune w
16: else
17:     α ← w.getNextAdjustableParam()
18:     γ ← suggestParamValue(α, w, QoS_error, C_E)
19:     w_adj ← w.setParam(α, γ)
20:     return QoSMerge(w_adj, T(w), E(w), QoS_time, QoS_error)
21: end if
```

workflow's error and time estimations, $E(w)$ and $T(w)$, meet user preferences, the workflow is included into the result set. But if the algorithm detects that either of these constraints is not met, the system is asked to provide a suitable value for α, the adjustment parameter of w, given the QoS values.

Taken with the suggested parameter, the QoSMerge procedure is called recursively on the adjusted workflow, w_{adj}. After each iteration, the accuracy parameter for w is adjusted, and if both constraints are met, w is returned to WFEnumQoS for inclusion in the candidate list, W. However, when the algorithm determines that the modifications to w provide insignificant contributions to its effects on $T(w)$ and $E(w)$, i.e., the adjustment parameter converges without being able to meet both QoS constraints, then w is left out of the returned list. As an aside, the values of t' and e' of the initial QoSMerge call on (Lines 7 and 20) of Algorithm 2 are set to ∞ for dispelling the possibility of premature convergence.

Algorithm 7 suggestParamValue($\alpha, w, QoS_{error}, C_E$)

```
1: ▷ trivially invoke model if one exists for suggesting α
2: if ∃ model(α, w.op) then
3:     M ← getModel(w.op, α)
4:     return M.invoke(QoS.Err)
5: else
6:     min ← α.min
7:     max ← α.max
8:     repeat
9:         m′ ← (min + max)/2
10:        w_adj ← w.setParam(α, m′)
11:        if QoS.Err < E(w_adj) then
12:            min ← m′
13:        else
14:            max ← m′
15:        end if
16:    until (max < min ∨ |E(w_adj) − QoS.Err| < C_E)
17:    return m′
18: end if
```

QoSMerge employs the suggestParamValue procedure (Algorithm 7) to tune the workflow's adjustment parameters.[‡] This algorithm has two cases: The trivial case is that a model is supplied for arriving at an appropriate value for α, the adjustment parameter. Sometimes this model is simply inverse of either the time or error models that exist for solving $T(w)$ and $E(w)$.

However, finding a model for suggesting α can be nontrivial. In these cases, α can be found by employing the existing models, $T(w)$ and $E(w)$ in a forward fashion. For example, consider that α is the rate at which to sample some specific file as a means for reducing workload. The goal, then, is to maximize the sampling rate while being sure that the workflow it composes remains below QoS_{time} and QoS_{error}. Since sampling rates are restrained to a specific range, i.e., $\alpha \in [0, 1]$, binary search can be utilized to find the optimal value (Lines 5-16 in Algorithm 7). For this to work

[‡]Although Algorithm 7 only shows error QoS awareness, time QoS is handled in much the same way.

properly $T(w)$ and $E(w)$ are assumed to be monotonically increasing functions, which is the case in most cases.

If either QoS constraint is not given by the user, its respective models is actually never invoked. In this case, QoSMerge becomes the trivial procedure of immediately return workflow candidate w. In Algorithm 6 this is equivalent to assigning the $QoS.*$ constraints to ∞.

4.3 Service Registration Framework

In the previous chapter, we discussed the data registration, a framework where users can share files. In this process, our system automatically extracts and indexes metadata associated with these files. Data registration is a necessity, as fast file identification is imperative to automatic, on-the-fly planning. Indexing services is not nearly as crucial since the number of available files far outnumber services. However, it is a convenient means to store useful information about the service, such as its associated cost models.

Depicted in Figure 4.1, is not unlike data registration. To register a service, a domain expert initially inputs the service description (WSDL [41]) file. Our system validates the WSDL and peruses through each supported operation. For each operation, (op, P_{op}), the system asks the expert for multiple inputs: (i) K_p — keywords describing each of the operation's parameters $p \in P_{op}$, (ii) K_{out} — keywords describing the operation's output, and (iii) a set of models/equations defining the operation's error propagation and execution time, i.e., those models discussed previously in Section 4.1.

Upon the given inputs, Auspice registers each prescribed service operation in the following way. First, the system's ontology must be updated to reflect the new service operation. A new service instance for op is added into the ontology's service class, S.

Next, relational edges for the new service are computed. To do this, WordNet [61] libraries are used to match like-terms in each of the provided K_p sets. The reduced set of terms is then matched to concepts within the system's domain ontology (keyword-to-concept mapping is assumed to be already provided). For each parameter p, an *inputsFrom* edge to the computed domain concept is added into the ontology. The same process is taken for prescribing the *derivedFrom* edge for K_{out}. With ontological updates in place, the new operation is now available to the WFEnumQoS algorithm for workflow planning.

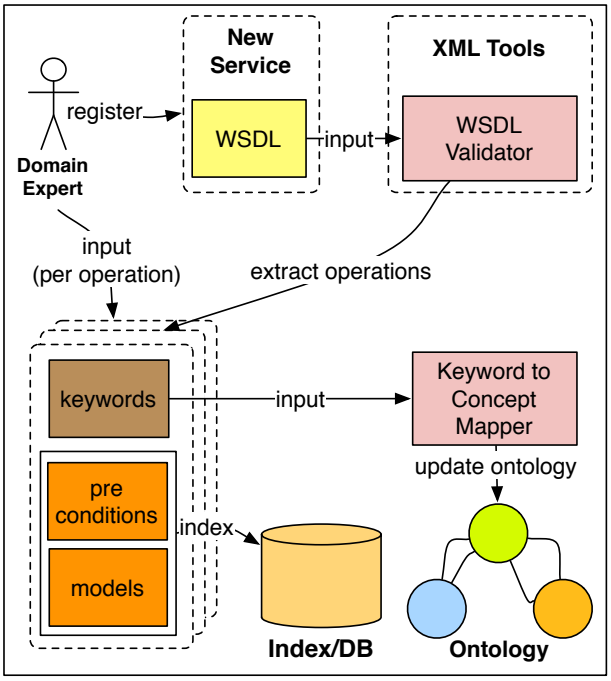


Figure 4.1: Service Registration

Next, the registration process handles the input cost models per operation. Equipped

with an equation parser, our framework allows general infix equations to be specified representing each model. Alternatively, algorithms can also be defined (in Java) for realizing more complex models. These models are appended onto the system's service configuration file, srvcModels.xml. In Figure 4.2, models are defined for an operation, DEMDiff, which inputs two files, DEM_1 and DEM_2. A Digital Elevation Model (DEM) is a file consisting of a matrix whose points represent surface elevation. For this particular operation, notice that the error model, σ, is defined as a method call. Upon the need to invoke σ, our system dynamically loads the specified class, *geodomain.GeoError*, and calls the *DEMComputeError* method on some given sampling rate.

```
<operation name="DEMDiff">
 <!-- sized -->
 <model type="outputModel"
   equation="max(DEM1.SIZE, DEM2.SIZE)" />
 <!-- tx -->
 <model type="execTimeModel"
   equation="8.11E-7 * max(DEM1.SIZE, DEM2.SIZE) + .."  />
 <!-- σ -->
 <model type="errorModel"
   class="geodomain.GeoError"
   method="DEMComputeError(RATE)" />
 <!-- ... -->
</operation>
```

Figure 4.2: SrvcModels.xml Snippet

4.4 An Example Query

‘‘water level (482593, 4628522) 01/30/2008 00:06’’

This query may be solved in two directions, i.e., the target concept of water level contains two distinct service nodes for derivation. One approach employs services to retrieve Deep Web data from some K nearest water gauge stations to the queried location and interpolates their readings for a more accurate result. Another method consults a water surface simulation model, whose access is made available over the Grid. Error models for both workflows were developed by our collaborators in the Department of Civil and Environmental Engineering and Geodetic Science [40].

The following is the actual output from our system given the above query. The values within [...] are the time and error predictions made by our system models. The actual execution times of both workflows are 3.251 sec for w_1 and 1.674 sec for w_2. Note that QoS constraints were not set as to show the comprehensive workflow candidate set with their estimated costs without the effect of pruning. With pruning, if QoS.Time was assigned a value of 2.0, then w_1 would have been discarded at the time of composition of its initial workflow, SRVC.GetGSListGreatLakes().

The service plan, w_1:

```
w_1 =
[t_total=3.501, err=0.004 --> t_x=1 t_net=0 d_size=0 ]
SRVC.getWL(
    X=482593,
    Y=4628522,
    StnID=
        [t_total=2.501, err=0.004 --> t_x=0.5, t_net=0, d_size=0]
        SRVC.getKNearestStations(
            Long=482593,
            Lat=4628522,
            ListOfStations=
                [t_total=2.01, err=0 --> t_x=2.01 t_net=0 d_size=47889]
                SRVC.GetGSListGreatLakes()
            RadiusKM=100,
            K=3
        )
```

```
    time=00:06,
    date=01/30/2008
)
```

The service plan, w_2:

```
w_2 =
[t_total=2.00, err=2.4997 --> t_x=2 t_net=0 d_size=0]
SRVC.getWLfromModel(
    X=482593,
    Y=4628522,
    time=00:06,
    date=01/30/2008
)
```

4.5 Experimental Evaluation

Two main goals are addressed in our experiments: First, to assess the overhead of workflow enumeration and the impact of pruning. The second set of experiments focused on evaluating our system's ability to consistently meet QoS constraints.

For our experiments, we employ three nodes from a real Grid environment. The local node runs our workflow system, which is responsible for composition and execution. Another node containing all needed services is located within the Ohio State University campus on a 3MBps line. Finally, a node containing all data sets is located in another campus, Kent State University, about 150 miles away. Here the available bandwidth is also 3.0MBps. The error models for all services involved in these experiments were developed by our collaborators in the Department of Civil and Environmental Engineering and Geodetic Science [39].

4.5.1 Overheads of Workflow Enumeration

The performance evaluation focuses on two goals: (i) To evaluate the overhead of workflow enumeration algorithm and the impact of pruning. (ii) To evaluate the efficiency and effectiveness of our adaptive QoS parameter scheme.

The initial goal is to present the efficiency of Algorithm 2. This core algorithm, called upon every given query, encompasses both auxiliary algorithms: QoSMerge — the decision to include a candidate and SuggestParamValue — the invocation of error and/or time models to obtain an adjustment value appropriate for meeting user preferences. Thus, an evaluation of this algorithm offers a holistic view of our system's efficiency. A synthetic ontology, capable of allowing the system to enumerate thousands of workflows, consisting of five activities each, for a user query, was generated for purposes of facilitating this scalability experiment. The results, depicted in Figure 4.3, was repeated for an increasing number of workflow candidates (i.e., $|W| = 1000, 2000, \ldots$) enumerated by WFEnumQoS on four configurations (solid lines). These four settings correspond to user queries with (i) no QoS constraints, (ii) only error constraints, (iii) only time constraints, and (iv) both constraints.

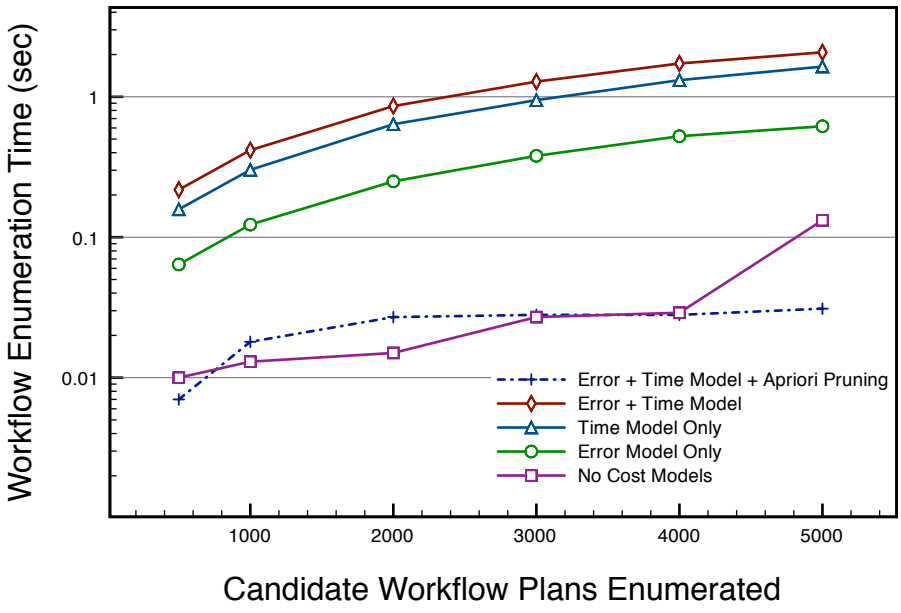


Figure 4.3: Cost Model Overhead and Pruning

Expectedly, the enumeration algorithm runs in proportional time to the numbers of models supported. To evaluate our algorithm's efficiency, we altered our previous experimental setting to contain exactly one workflow within each candidate set that meets both time and error constraints. That is, for each setting of $|W| + 1$, the algorithm now prunes $|W|$ workflows (dashed line). The results show that cost-based pruning algorithm is as efficient as *no-cost* model since the amount of workflows considered is effectively minimized due to their cost being unable to fulfill QoS requirements.

4.5.2 Meeting QoS Constraints

$Query_{DEM}$	"return surface change at (482593, 4628522) from 07/08/2000 to 07/08/2005"
$Query_{SL}$	"return shoreline extraction at (482593, 4628522) on 07/08/2004 at 06:18"

Table 4.1: Experimental Queries

The experimental queries (Table 4.1) are designed to demonstrate QoS management. Specifically, $Query_{DEM}$ must take two digital elevation models (DEM) from the given time periods and location and output a new DEM containing the difference in land elevation. The shoreline extraction in $Query_{SL}$ involves manipulating the water level and a DEM for the targeted area and time. Although less computationally intense than $Query_{DEM}$, execution times for both are dominated by data movement and computation.

This becomes problematic for low QoS time constraints, but can be mitigated through data reduction, which we implement via sampling along each of the DEM's dimensions. In both queries the sampling rate is the exposed accuracy adjustment

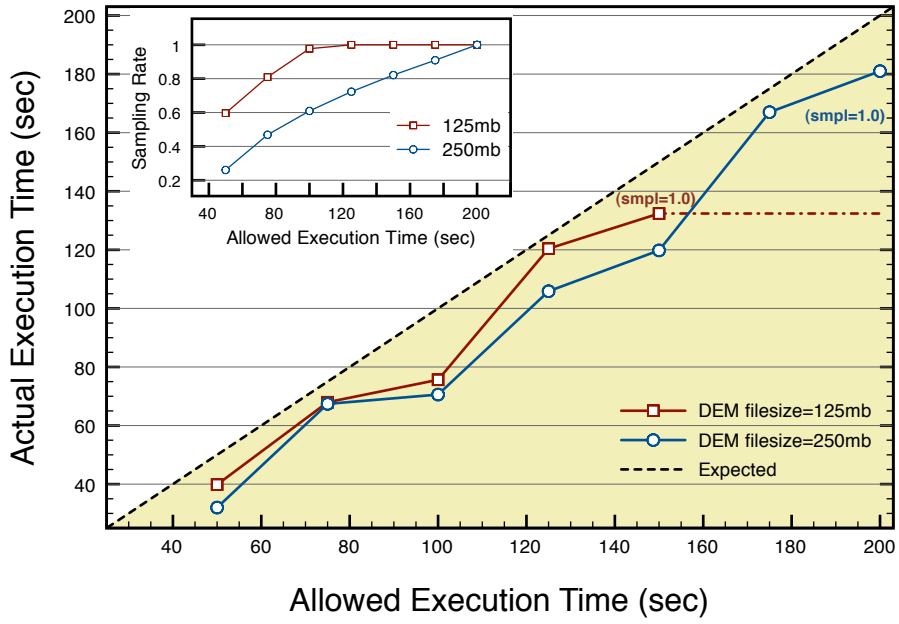


Figure 4.4: Meeting Time Expectations: $Query_{DEM}$

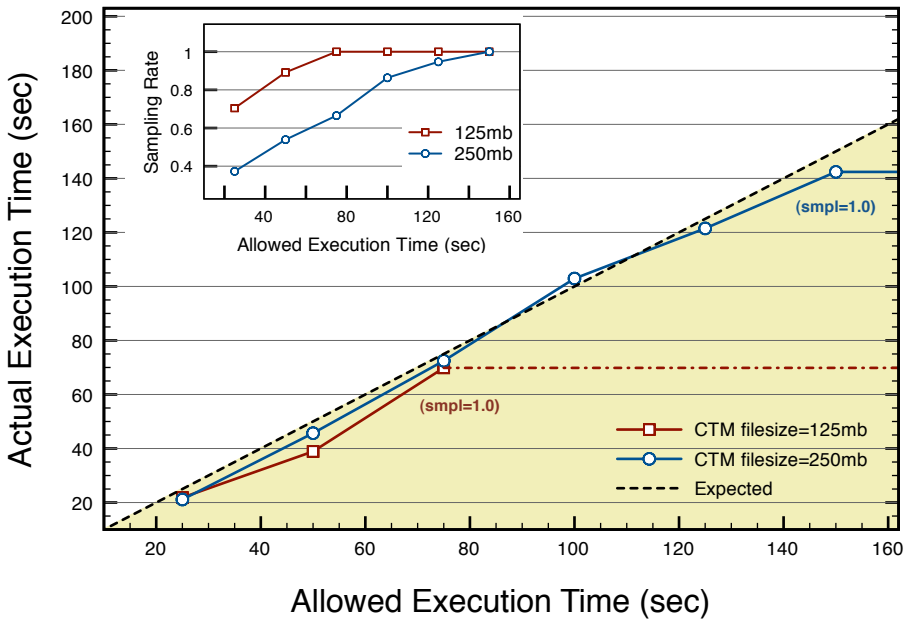


Figure 4.5: Meeting Time Expectations: $Query_{SL}$

parameter, and the goal of our system is to suggest the most appropriate sampling rates such that the actual execution time is nearest to the user allowance. All services involved in these queries have been trained to obtain prediction models for cost estimation.

Figures 4.4 and 4.5 show the actual execution times of each query against user-allowed execution times. The dashed line which represents the end-user's expectations is equivalent to the time constraint. The DEM sampling rate, which is embedded in the figures, is inversely proportional to the error of our workflow's payload. A juxtaposition of the outer and embedded figures explains why, in both results, the actual execution time of the workflow pertaining to smaller DEMs flattens out towards the tail-end: at the expected time constraint, it has already determined that the constraint can be met without data reduction.

The gap observed when $AllowedExecutionTime = 100$ in Figure 4.4 is exposing the fact that the system was somewhat conservative in suggesting the sampling rate for that particular point, and a more accurate workflow could probably have been reached. Situations like these exist due to imprecisions in the time model (we used multi-linear regression). The implementation of the models, Between the two DEM size configurations, $Query_{DEM}$ strays on an average of 15.65 sec (= 14.3%) from the expected line and $Query_{SL}$ by an average of 3.71 sec (= 5.2%). Overall, this experiment shows that our cost model and workflow composition scheme is effective. We obtained consistent results pertaining to error QoS, but these results are not shown due to space constraints.

The next experiment shows actual execution times against varying bandwidths of our network links. Ideal expectations in this experiment are much different than the linear trend observed in the previous experiment. When bandwidth is low, sampling is needed to fit the execution within the given time constraint (we configured this

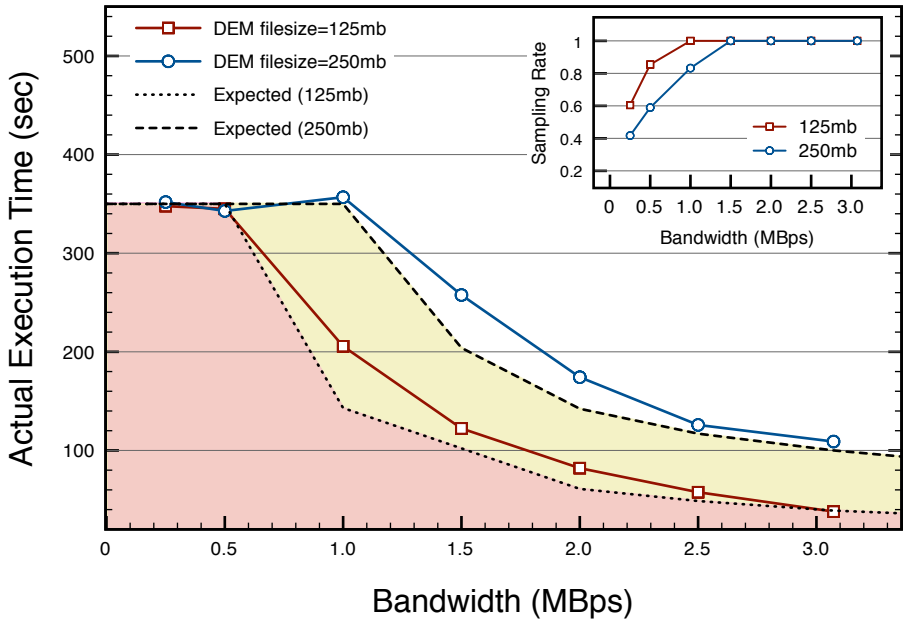


Figure 4.6: Against Varying Bandwidths: $Query_{DEM}$

at 350 sec in both experiments). Next, when the bandwidth is increased beyond the point where sampling is necessary, we should observe a steady drop in actual execution time. Finally, this declining trend should theoretically converge to the pure execution time of the services with ideal (zero) communications delay and network overhead.

As seen in Figures 4.6 and 4.7, the actual execution times lie consistent with the expected trends. Between the two data sizes, $Query_{DEM}$ strays on average 16.05 sec (= 12.4%) from the ideal line and $Query_{SL}$ 13.79 sec (= 6.7%) on average. It is also within our expectations that the actual execution times generally lie above the ideal lines due to communication overheads and actual network fluctuations.

We believe that our experimental results suggest that the system provides and

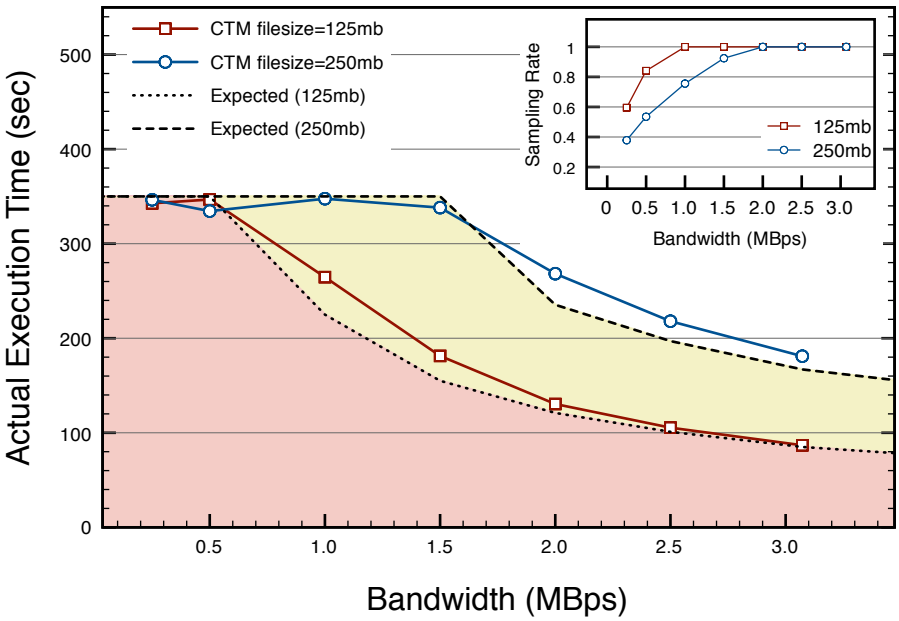


Figure 4.7: Against Varying Bandwidths: $Query_{SL}$

maintains robustness against user defined cost as well as computing costs within dynamic Grid environments. Specifically, the accuracy parameter suggestion algorithm was shown to gracefully adapt workflows to restrictions on time and networks.

Next, we demonstrate the system's efforts for supporting user preferences. We begin by presenting an evaluation of the adaptive workflow parameter suggestion procedure. For this experiment, the sampling rate is the exposed workflow accuracy adjustment parameter. Table 4.2 shows the ideal and actual, i.e., system provided, error targets. On the left half of the table, the ideal accuracy % is the user provided accuracy constraint and the ideal error is the error value (from the model) expected given this corresponding accuracy preference. The right half of the table shows the actual accuracy % and errors that the system provided through the manipulation on sampling rate. As seen in the table, although the error model appears to be

extremely sensitive to diminutive amounts of correction, our system's suggestion of sampling rates does not allow a deviation of more than 1.246% on average. It is also observable that the % of deviation causes minute, if not negligible, differences (in meters) as compared to the ideal accuracies.

Ideal		**System Suggested**	
Acc %	Error (meters)	Acc %	Error (meters)
10	8.052	11.81	8.052001
20	7.946	21.15	7.945999
30	7.911	28.61	7.911001
40	7.893	34.96	7.892999
50	7.868	50.52	7.867996
60	7.859	60.16	7.858989
70	7.852	70.65	7.851992
80	7.847	80.71	7.847001
90	7.8437	89.07	7.843682
100	7.8402	99.90	7.840197

Table 4.2: Suggested Value of Parameters: $Query_{DEM}$

Finally, $Query_{DEM}$ was executed with user given accuracy preferences of 10%, 20%, ..., 100% on DEM files of sizes 125mb and 250mb. As seen in Figure 4.8, the sampling rates along with the workflow's corresponding execution times at each accuracy preference, increase as the user's accuracy preference increases. The figure clearly shows the benefits from using sampling, as the execution time is reduced polynomially despite some loss in accuracy.

The above experiments were repeated for the shoreline query ($Query_{SL}$) to obtain Table 4.3. Again, the results are consistent with the previous experiment, and moreover, our system offers slightly better parameter adjustments which results in tighter accuracies for this query. This can be explained again due to the fine-grained sensitivity of the error model for the previous query. We exhibit only a 0.07% average

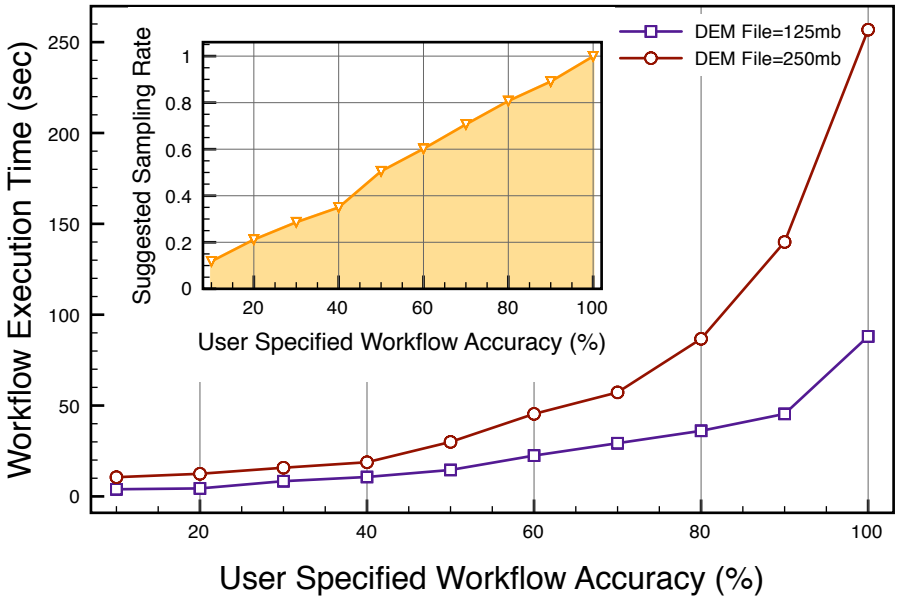


Figure 4.8: Meeting User-Specified Accuracy Constraints: $Query_{DEM}$

Ideal		**Suggested**	
Acc %	Error (meters)	Acc %	Error (meters)
10	61.1441	10.00	61.1441
20	30.7205	19.93	30.7204
30	20.4803	29.91	20.4798
40	15.3603	39.89	15.3599
50	12.2882	49.87	12.2892
60	10.2402	59.98	10.2392
70	8.7773	69.88	8.7769
80	7.6801	79.90	7.6803
90	6.8268	89.94	6.8266
100	6.1441	100	6.1441

Table 4.3: Suggested Value of Parameters: $Query_{SL}$

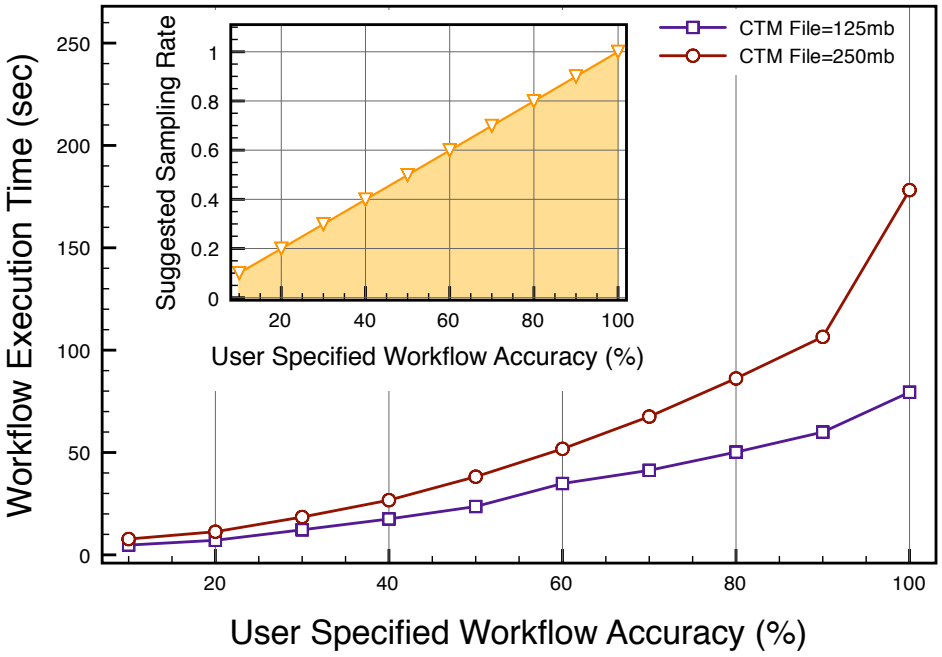


Figure 4.9: Meeting User-Specified Accuracy Constraints: $Query_{SL}$

and 0.13% worst case accuracy deviation from the expected values. CTMs of sizes 125mb and 250mb were used to run the actual experiments. The results, depicted in Figure 4.9 again show the consistency of our algorithm and the effects of sampling on both workflow accuracy and execution time.

Next, we discuss the evaluation of the parameter suggestion overhead. Recall that the parameter suggestion algorithm has two cases: (1) trivially invoke a predefined model for solving for the adjustment parameter (in this case, the sampling rate), or (2) if this model is not available, it solves for the parameter through binary search on the sampling rate by employing $E(w)$ or $T(w)$ per sampling rate at each iteration. For both queries, error estimation, i.e., the σ term in $E(w)$, involves a series of computations, and an inverse model cannot be easily derived. This forces the suggestParamValue algorithm to default to the latter case of binary search. The overhead (in msecs) to this approach is summarized in Figure 4.10.

Again, the sensitivity of the DEM query's error model observes a slightly longer time-to-convergence. This overhead, however, contributes negligible time to the overall enumeration time, shown earlier in Figure 4.3. A quick study was also carried out to compare these overheads to the trivial case of model invocation. Surprisingly, the best case time for model invocation (model contains no calculations and simply returns a constant) cost 0.024 msecs, which is significantly more expensive. This cost can be explained through the heavyweight implementation of our model — that is, we utilize an equation parser to offer users an intuitive interface for inputting complex calculations. This flexibility, of course, is not without the cost of data structure management, which undoubtedly contributes to the overhead.

We believe that our experimental results suggest that the system maintains robustness against user defined cost, and although not shown due to space limitations, parameter adjustment to meeting time constraints exhibited similar results.

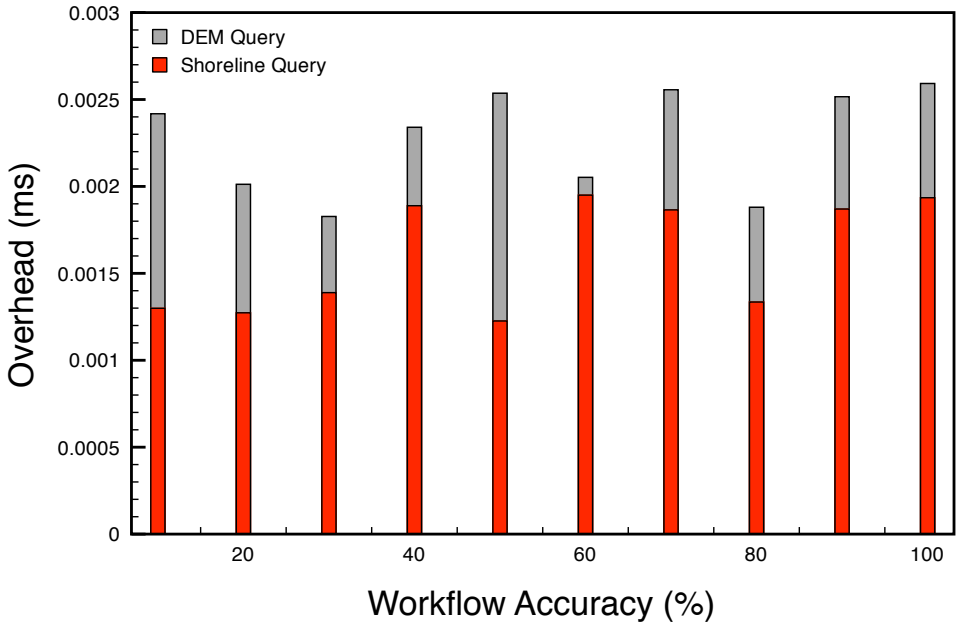


Figure 4.10: Overhead of Workflow Accuracy Adjustment

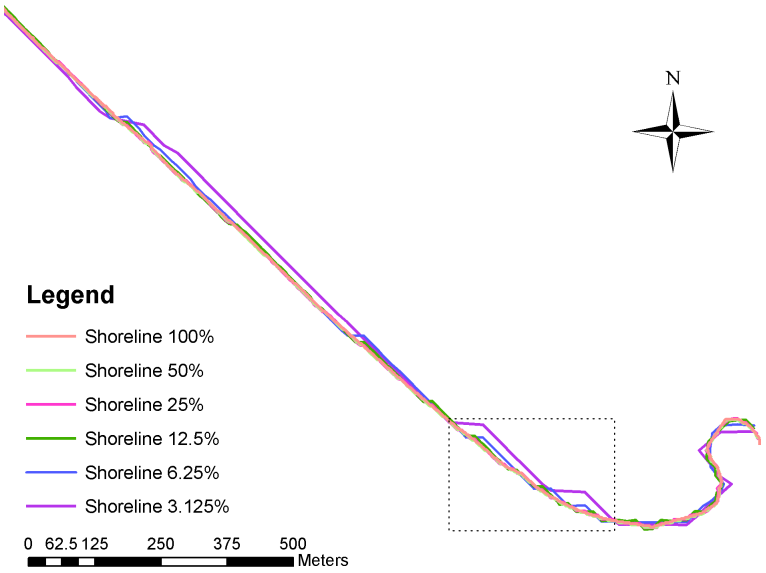
4.5.3 Shoreline Error Model Evaluation

Our final experiment evaluate only the shoreline error prediction model based on the availability of actual results for comparison. Recall that the time cost of shoreline extraction is dominated by retrieving and processing the CTM corresponding to the location. The sampling algorithm for DEMs and CTMs essentially skips $\lceil 1/r \rceil$ points per dimension, where r is the sampling rate. In our sampling algorithm, the CTM is reduced by eliminating data points at a regular interval. Clearly, the shorelines obtained from different sampling patterns would contain errors. By taking exponentially smaller samples (r = 100%, 50%, 25%, 12.5%, 6.25%, 3.125%), we effectively double the amount of points skipped per configuration.

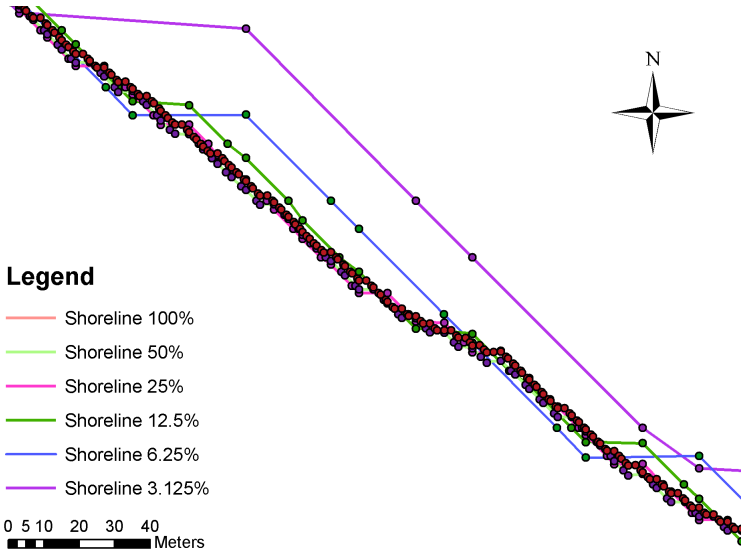
Accuracy (%)	**Real Error (meters)**	**Stddev**
100%	0	0
50%	1.36071	0.833924
25%	1.454593	1.050995
12.5%	2.651728	1.824699
6.25%	5.258375	4.06532
3.125%	15.03924	9.954839

Table 4.4: Actual Shoreline Errors

Given the sampled CTMs, we created a visualization of the resulting shoreline using ESRI ArcMap, depicted in Figure 4.11(a). Using the r = 100% setting as our baseline, it is visible that a slight deviation is associated with every downgraded sampling rate configuration. This becomes clearer in the zoomed region shown in Figure 4.11(b), which also makes visible the patterns of sampling and its deteriorating effects on the results. The actual errors shown in Table 4.4 are much less than predicted by our model (compare with Table 4.3). Admittedly, this suggests that our



(a) Overall Shoreline Region



(b) Focused Shoreline Region

Figure 4.11: Shoreline Extraction Results

model may be excessively conservative, at least for this particular shoreline. While the initial consequence is that a smaller sampling rate could have been suggested by our system for speeding up workflows involving extremely large data sets, it does, however, ultimately demonstrate that the actual results are no worse than what the model predicts and that our framework is overall safe to use.

We believe that our experimental results suggest that the system maintains robustness against user defined cost, and although not shown due to space limitations, parameter adjustment for meeting time-based QoS constraints exhibited similar results.

CHAPTER 5

HIERARCHICAL CACHES FOR WORKFLOWS

For years, the scientific community has enjoyed ample attention from the computing society as a result of new and compelling challenges that it poses. These issues, which fall under the umbrella of data intensive scientific computing problems, are largely characterized by the need to access, analyze, and manipulate voluminous scientific data sets. High-end computing paradigms, ranging from supercomputers to clusters and the heterogeneous Grid, have lent well to middlewares and applications that address this set of problems [84].

Among these applications, workflow management systems have garnered considerable interest because of their ability to manage a multitude of scientific computations and their interdependencies for deriving the resulting products, known as *derived data*. Although a substantial amount of effort in this area has been produced, great challenges for Grid-enabled scientific workflow systems still lie ahead. Recently, Deelman et al. outlined some of these challenges [50]. Among them, data reuse is one of particular interest, especially in the context of autonomous systems. While questions on how best to identify the existence of intermediate data as well as determining their benefits for workflow composition remain open, the case for providing an efficient scheme for intermediate data caching can certainly be made.

Historically, caching mechanisms have been employed as a means to speed up

computations. Distributed systems, including those deployed on the Grid, have relied on caching and replication to maximize system availability and to reduce both processing times and network bandwidth consumption [23]. In the context of scientific workflow systems, we could envision that intermediate data generated from previous computations could be stored on an arbitrary Grid node. The cached intermediate derived data may then be retrieved if a subsequent workflow calls for its use.

To exemplify, consider Figure 5.1, which depicts a workflow manager that is deployed onto some scientific (in this case, geospatial) data Grid. In this particular situation, a workflow broker maintains an overview of the physical Grid, e.g., an index of nodes, data sets, services, as well as their inter-relationships. The broker, when given a user query, generates workflow plans and schedules their execution before returning the data result back to the user.

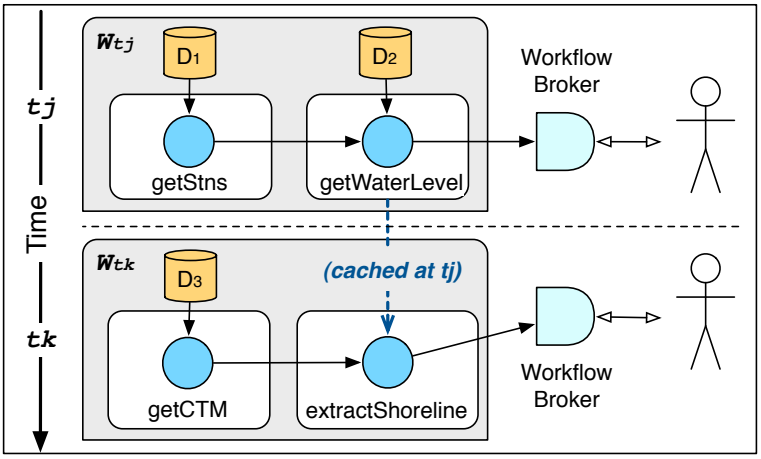


Figure 5.1: Example Workflow Sequence

Focusing on w_{tj}, this workflow initially invokes getStns(), which returns a list of

water gauge stations close to some area of interest. This list is input to another service, getWaterLevel(), which queries each station from the input list for their readings at the desired time. After a series of computations, getWaterLevel() eventually produces the desired data: average water level for some given region and time. Now, let's assume a second query is submitted at some later time $tk > tj$, involving shoreline extraction for those same time and region. The first half of w_{tk} invokes getCTM() to identify and retrieve spatiotemporally relevant CTM data. This is input into extractShoreline(), which also requires the water level. Having been processed earlier at tj, the water level redundant, and w_{tk}'s execution time can be reduced if our system can efficiently identify whether this data already exists.

The workflow redundancy exhibited above might seem a bit improbable in a spatiotemporal environment where space and time are vast and users' interests are disparate. Such situations, however, are not absent from *query intensive* circumstances. For instance, (i) an unexpected earthquake might invoke an onslaught of similar queries issued for a specific time and location for examination and satisfying piqued curiosities. (ii) Rare natural phenomena such as a solar eclipse might prompt a group of research scientists with shared interests to submit sets of similar experiments with repeated need for some intermediate data. Without intermediate data caching, a workflow system may not be able to adequately cope with the sudden surge in queries for the amount of data movement and analysis necessary. Managing a cache under these situations, however, is met with certain difficulties. In this chapter, we address approaches in handling several technical challenges towards the design of a Grid based cache-sensitive workflow composition system. These challenges, and our contributions, include:

- *Providing an efficient means for identifying cached intermediate data* — Upon

reception of a user query, our automatic workflow composition system immediately searches for paths to derive the desired intermediate data. It is within this planning phase that relevant intermediate data caches should be identified, extracted, and composed into workflows, thereby superseding expensive service executions and large file transfers. Clearly, the cache identification process must only take trivial time to ensure speedy planning.

- *Dealing with high-volumes of spatiotemporal data* — Large amounts of intermediate data can be cached at any time in a query intensive environment. But scientific qualifications, such as spatiotemporality, mixed in a potentially high update environment will undoubtedly cause rapid growth in index size. To this end, we describe an efficient index structure with an accompanying victimization scheme for size regulation.

- *Building a scalable system* — A large distributed cache system should leverage the Grid's versatility. Our cache structure is designed in such a way as to balance the index among available nodes. This consequently distributes the workload and reduces seek times as nodes are introduced to the system.

5.1 Enabling a Fast, Distributed Cache

Workflows, in general, involve a series of service executions to produce some set of intermediate data used to input into the next set of services. Caching these read-only intermediate results would clearly lead to significant speed up, particularly when replacing long running services with previously derived results. Returning back to Figure 1.2, the cache is logically positioned in the Planning and Execution Layer. Much of the challenges in its implementation is tied directly to the Planning Layer. For one, the existence of previously saved intermediate data must be quickly identified

so as to amortize the cache access overhead in the workflow enumeration phase. At first glance, it would then seem straightforward to place the intermediate data cache directly on the broker node. Several issues, however, argue against this justification:

1. Services are distributed onto arbitrary nodes within the Grid. Centralizing the cache would imply the need to transfer intermediate data to the broker after each service execution. Moreover, accessing the cache would further involve intermediate data transfer from the broker to the node containing the utilizing service. This would lead to an increase in network traffic on the broker, which should be avoided at all costs.

2. A centralized broker cache would scale poorly to large volumes of cached intermediate data. Due to the nature of our spatiotemporal intermediate data, multidimensional indices (e.g., R-Trees, its variants, and others [86, 143, 22]) can typically be employed. Some issues are at stake: (i) Cached intermediate data are read-only. In a high-insertion, query intensive environment, a centralized multidimensional index can quickly grow out of core [95]. (ii) To solve this issue, a cache replacement mechanism would be needed to contain the index within memory despite the fact that less intermediate data can be tracked.

In hopes of alleviating the challenges outlined above, we introduce a system of hierarchically structured caches, shown in Figure 5.2. Again, the existence of a cached result must be known at planning time to ensure speedy enumeration. For the Planning Layer to access this information efficiently, it is unavoidable that some form of cache index must still exist on the broker with the caveat being that its size must be regulated.

The *broker index* is organized in two tiers: (i) A table of domain concepts (specified within the Semantics Layer's ontology) summarizes the top tier. Placing concepts at

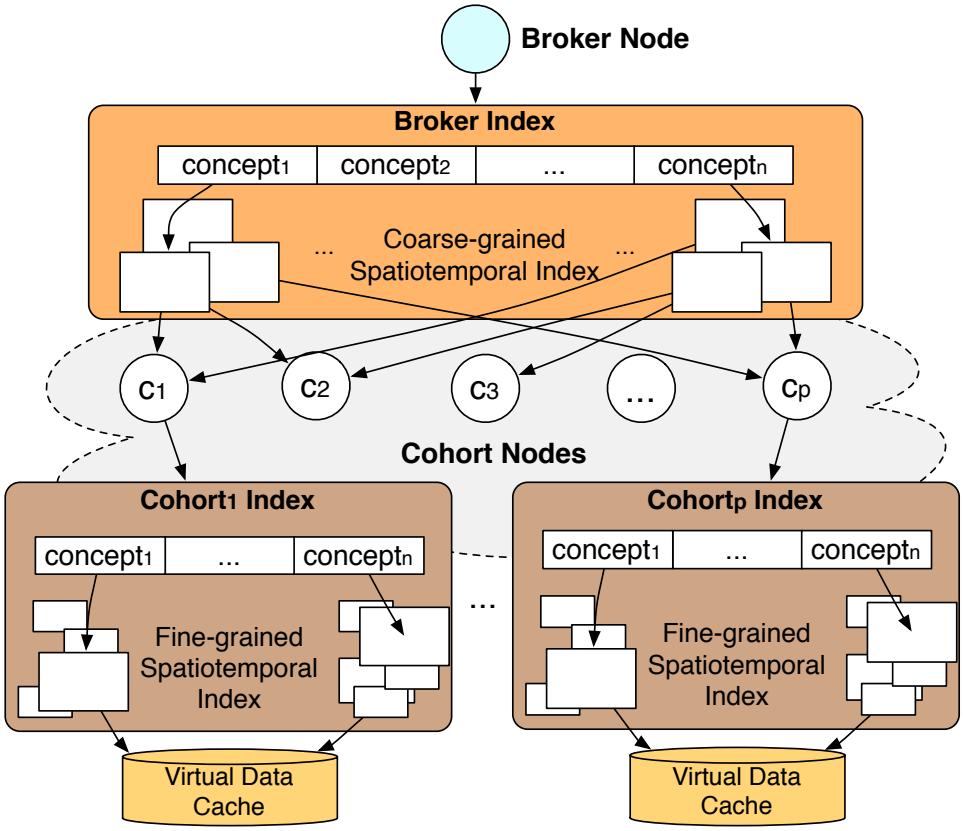


Figure 5.2: Hierarchical Index

the top enables the search function to prune significant portions of the index prior to initiating the costly spatiotemporal search. (ii) In the bottom tier of the broker index, each concept maintains a distinct spatiotemporal index tree. In each tree we want its spatiotemporal granularity to be coarse. By broadening the span of time and spatial coverage that each intermediate data element could hash into, we can dramatically reduce the broker index's size and thus reduce hit/miss times. Each broker index record contains pointers to any number of Grid nodes, i.e., cohorts, that might contain the desired information.

A *cohort index* exists locally on each cache-enabled node in the Grid. Its structure is not unlike that of the broker index, with the only difference being that it maintains a fine-grained spatiotemporal index tree. The logic is that, if enough nodes join the rank of cohorts, then each node can manage to cover increasingly finer spatiotemporal details. Moreover, the overall index size and load is balanced and shared. Each cohort index record contains the location of the intermediate data on the local disk. Together, the cohorts represent a massive distributed spatiotemporal index.

Direct consequences of this hierarchical structure are the hit and miss penalties. While recognizing a miss is trivially contained within the broker, a hit cannot be fully substantiated until hits on the cohort level are reported. Thus, three responses are possible: fast miss, slow miss, hit (slow). One of the design goals is to support our hypothesis that, in query intensive environments where centralized indices can quickly grow out of core, hits/misses can be realized significantly faster on the hierarchical index despite the overhead of cohort communications.

5.2 Bilateral Cache Victimization

The cost of maintaining a manageable broker index size is the ambiguity that leads to false broker hits (followed by cohort misses). With a large enough spatiotemporal

region defined in the broker, only a small hash function can be managed. This means that a true miss is only realized after a subsequent miss in the cohort level. Keeping a finer grained broker index is key to countering false broker hits. But in a high-insert environment, it is without question that the index's size must be controlled through victimization schemes, e.g., LRFU [107].

Algorithm 8 BrokerVictimization($brkIdx$, $V[\ldots]$, ϕ, τ)

```
1: while φ > τ do
2:     ▷ v is the victimized region key
3:     v ← V.pop()
4:     record ← brkIdx.get(v)
5:     ▷ broker records hold list of cohorts that may contain cached intermediate data
6:     ▷ broadcast delete to associated cohorts
7:     for all cohort ∈ record do
8:         cohort.sendDelete(v)
9:     end for
10:    brkIdx.delete(v)
11:    φ ← φ − 1
12: end while
```

Because of their direct ties, a broker record's victimization must be communicated to the cohorts, which in turn, deletes all local records within the victimized broker region. Cohort victimization, on the other hand, is not as straightforward. As each node can have disparate replacement schemes, a naïve method could have every cohort periodically send batch deletion requests to the broker. The broker deletes a region once it detects that all cohort elements have been removed from that entry. But this method is taxing on communications cost. To cut down on cost, we discuss the following bidirectional scheme: the top-down Broker Victimization and the bottom-up Cohort Victimization.

Broker Victimization (Algorithm 8) takes as input the broker index, $brkIdx$, a queue of victims, $V[\ldots]$, the current record size, ϕ, and the record size threshold, τ.

The algorithm is simple: as broker records are deleted to regulate index size back to τ, each involved cohort node must be communicated to delete its own records within the victimized region. This is repeated until ϕ is regulated down τ. The selection of an effective τ is largely dependent on system profiles (e.g., physical cache and RAM capacity, disk speed, etc), and can take some trial-and-error. For instance, we show in the experimental section that τ appears to be between 2 and 4 million records on our broker, which uses 1GB of RAM.

In solving for the complexity of Broker Victimization, we let C denote the set of cohorts in any hierarchical index. For some cohort node $c \in C$, we also define $t_{net}(c)$ to be the network transfer time from the broker to c. Finally, if we let $n = \phi - \tau$ be the amount of to-be-victimized records, the total time taken for Broker Victimization, $T_{bvic}(n)$, is:

$$T_{bvic}(n) = \sum_{i=1}^{n} (\max_{j=1}^{|C_i|}(t_{net}(c_j)) + \delta)$$

where $|C_i|$ denotes the number of cohorts that needs to be communicated to ensure the victimization of record i and δ is some trivial amount of local work on the broker (e.g., victim deletion). If we further let the slowest broker-to-cohort time be called t_m, i.e., $t_m = \max_{c \in C}(t_{net}(c))$, then the worst case bound is $T_{bvic}(n) = O(n(|C|t_m + 1))$.

Because the overall time is inevitably dominated by cohort communications, an asynchronous version which minimizes $|C|$ to 0 can be used. On behalf of the a priori principle: When broker records are removed, it implies that a multitude of cohort records has also not been recently accessed. Eventually, regardless of each cohort's individual replacement scheme, the unused records will be evicted due to its absence from the broker index. In effect, cohort communication can essentially be omitted, reducing the algorithm to $O(n)$, or an amortized $O(1)$ depending on the frequency of its invocation and due to the triviality of the constant time δ. This, of course, is at the expense of each cohort having to maintain some amount of deprecated records.

When used alone, Broker Victimization is insufficient. If only a few elements exist in the broker's larger regions, the entire coarse-grained record must still be kept while less frequently used records in cohort indices might have already been evicted in their own victimization schemes. This leads to an inconsistency between the two indices and causes false broker hits. To handle this issue, we employ the Cohort Victimization scheme (not shown due to space limitations). Each cohort maintains a copy of its own relevant subset of broker's spatiotemporal coverage. When a cohort victimizes a record, an eviction message is sent to the broker if region which empasses the victim is now empty. Upon reception of this message, the broker removes the pointer to the evicted cohort node from the indexed element. Only after all cohort pointers have been emptied from that broker record does the broker delete the respective region.

5.3 Fast Spatiotemporal Indexing

When facing query intensive environments, frequent cache index updates must be anticipated. We utilize a slightly modified version of the B^x-Tree [96] for fast spatiotemporal indexing. Originally proposed by Jensen et al. for indexing and predicting locations of moving objects, B^x-Trees are essentially B+Trees whose keys are the linearization of the element's location via transformation through space filling curves. The B^x-Tree further partitions its elements according to the time of the update: Each timestamp falls into a distinct partition index, which is concatenated to the transformed linear location to produce the record's key. The appeal of this index lies in its underlying B+Tree structure. Unlike most high dimensional indices, B+Trees have consistently been shown to perform exceptionally well in the high update environments that query intensive situations pose. But since B+Trees are intended to capture 1-dimensional objects, the space filling curve linear transformation is employed.

In the B^x-Tree, space filling curves (a variety of curves exist; the Peano Curve is

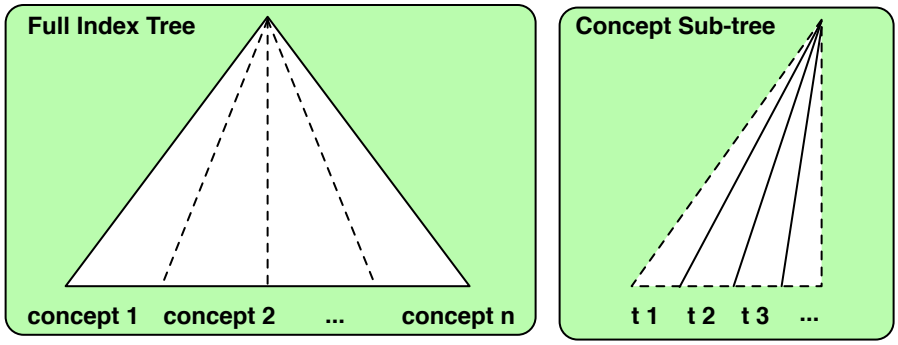


Figure 5.3: A Logical View of Our B^x-Tree

used in our implementation) [123] are used to map object locations to a linear value. In essence, these curves are continuous paths which visit every point in a discrete, multidimensional space exactly once and never crosses itself. The object's location, once mapped to a point on the space filling curve, is concatenated with a partition indexing the corresponding time.

Since our goal is not indexing moving objects and predicting their locations in present and future times, we made several adjustments to suit our needs. First, since the B^x-Tree tracks moving objects, their velocity is captured. In our implementation, we can simply omit this dimension. Second, our timestamps are not update times, but the physical times relevant to the intermediate data. Finally, recall from Figure 5.2, that the notion for the concept-first organization for the broker and cohort indices is a means to provide fast top level pruning. In practice, however, maintaining a separate spatiotemporal index per concept is expensive. We describe an alternate approach: We also linearize the domain concepts by mapping each to a distinct integer value and concatenating this value to the leftmost portion of the key. By attaching binary concept mappings to the most significant bit portions, we logically

partition the tree into independent concept sections, as shown in Figure 5.3. In the right side of the figure, we focus on a concept's sub-tree; each sub-tree is further partitioned into the times they represent, and finally, within each time partition lie the curve representations of the spatial regions.

We manipulate the key in this fashion because the B+Tree's native insertion procedure will naturally construct the partitioning without modifications to any B+Tree structures. This, due to the B+Tree's popularity, allows the B^x-Tree to be easily ported into existing infrastructures. The leftmost concatenation of concept maps also transparently enables the B+Tree search procedure to prune by concepts, again without modification of B+Tree methods. To clarify, if a intermediate data pertaining to concept k is located in (x, y) with t being its time of relevance, its key is defined as the bit string:

$$key(k, t, o) = [k]_2 \cdot [t]_2 \cdot [curve(x, y)]_2$$

where $curve(x, y)$ denotes the space filling curve mapping of (x, y), $[n]_2$ denotes the binary representation of n, and $\cdot$ denotes binary concatenation.

5.4 Experimental Evaluation

In this section, we present an evaluation of our cache-enabled workflow system. In our Grid environment, the broker node is a Linux machine running Pentium IV 3Ghz Dual Core with 1GB of RAM. The broker connects to a cluster of cohort nodes on a 10MBps link. Each cohort node runs dual processor Opteron 254 (single core) with 4GB of RAM. The cohort cluster contains 64 nodes with uniform intercluster bandwidths of 10MBps.

First, we pit our system against two frequently submitted geospatial queries to show the benefits of intermediate result caching. These are, Land Elevation

Change="return land elevation change at (x, y) from time u_{old} to time u_{new}" and Shoreline Extraction= "return shoreline for (x, y) at time u."

To compute the Land Elevation Change query, a readDEM() service is used to identify and extract Digital Elevation Model (DEM) files into intermediate objects corresponding to the queried time and location. This service is invoked twice for extracting DEMs pertaining to u_{old} and u_{new} into compressed objects. The compressed DEM objects are passed on to finish the workflow. We measured the overall workflow execution time for various sized DEMs and displayed the results in Figure 5.4 (top). The solid-square line, denoted *Total Time (original)*, is the total execution time taken to process this query without the benefits of caching. The dotted-square line directly underneath, denoted *readDEM() Time (original)*, shows the time taken to process the two readDEM() calls. Regardless of DEM size, readDEM() dominates, on average, 90% of the total execution time. If the intermediate DEM objects can be cached, the calls to readDEM() can simply be replaced by accesses to the compressed DEM objects in the cache. The triangular lines in Figure 5.4 (top) indicate the benefits from using the cache. Due to the reduction of readDEM() to cache accesses, the same workflow is computed in a drastically diminished time. The average speed up that caching provides over the original workflow is 3.51.

The same experiment was repeated for the Shoreline Extraction query. In the workflow corresponding to this query, a readCTM() service stands as its dominant time factor. Not unlike readDEM(), readCTM() extracts Coastal Terrain Models (CTM) from large data sets into compressed CTM objects. As seen in Figure 5.4 (bottom), we consistently attain average speed ups of 3.55 over the original, cacheless executions, from utilizing cached versions of CTM objects.

The next set of experiments looks at the effectiveness of our cache system over

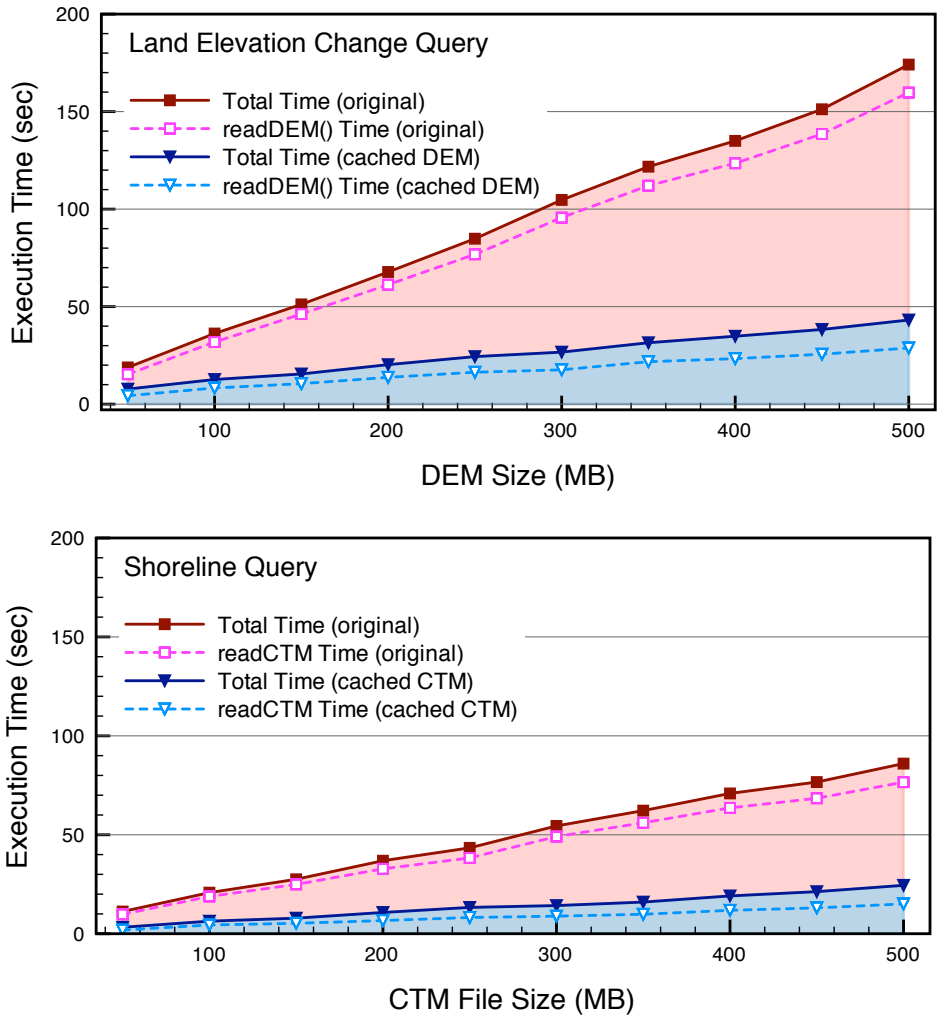


Figure 5.4: Effects of Caching on Reducing Workflow Execution Times

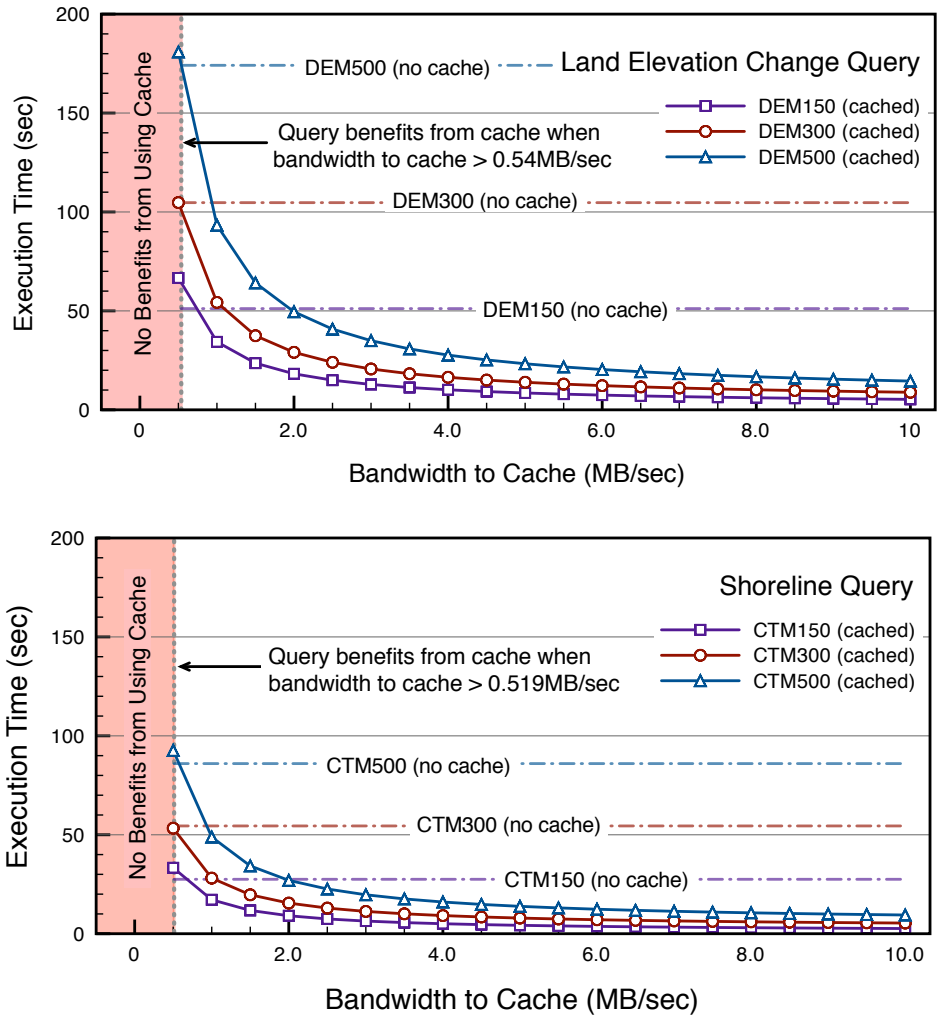


Figure 5.5: Varying Bandwidths-to-Cache

heterogeneous networking environments, expected in the Grid. We execute the previous workflows, this time with three fixed settings on intermediate data size. Here, an advantage of intermediate data sets is shown. Recall readDEM() and readCTM() both read large files into compressed objects. For original DEM and CTM files of size 150MB, 300MB, and 500MB, their respective intermediate object sizes are 16.2MB, 26.4MB, and 43.7MB. This is fortunate, as our system only needs to cache the compressed objects. In these experiments, we are interested in the point in broker-to-cache bandwidth where it becomes unreasonable to utilize the cache because it would actually be faster to execute the workflows in their original formats. Our hope is that the cache will provide enough speed up to offset the overhead induced by slow links. Figure 5.5 displays the results for this experiment on Land Elevation Change (top) and Shoreline Extraction (bottom). Among the three fixed DEM/CTM sizes (150MB, 300MB, and 500MB), we found that we will on average attain speed ups over broker-to-cache links greater than 0.54MBps for the Land Elevation Change workflow and 0.519MBps for Shoreline Extraction. In a typical scientific Grid or cluster environment we believe that it is reasonable to assume the existence of average bandwidths either at or above these values. Still, one can see how, by monitoring network traffic on the cache link and building a model around the results of these experiments, our system can decide whether or not the cache should be utilized for workflow execution. The bandwidth monitor, however, is not yet implemented in our system.

The last set of experiments provide insight into aspects of scalability. First, we investigate average seek times between our hierarchical structure and a centralized index. The centralized index is equivalent to a single broker index without cohorts. To facilitate this experiment, we simulated a query intensive environment by inserting an increasing amount of synthetic records into our index. In the centralized structure,

a massive B^x-Tree is used to index the entire intermediate data cache. Because cohort communications is avoided, we should expect far faster seek times for smaller index sizes. In Figure 5.6, the centralized index's seek time is illustrated by the lone solid line and follows the left y-axis.

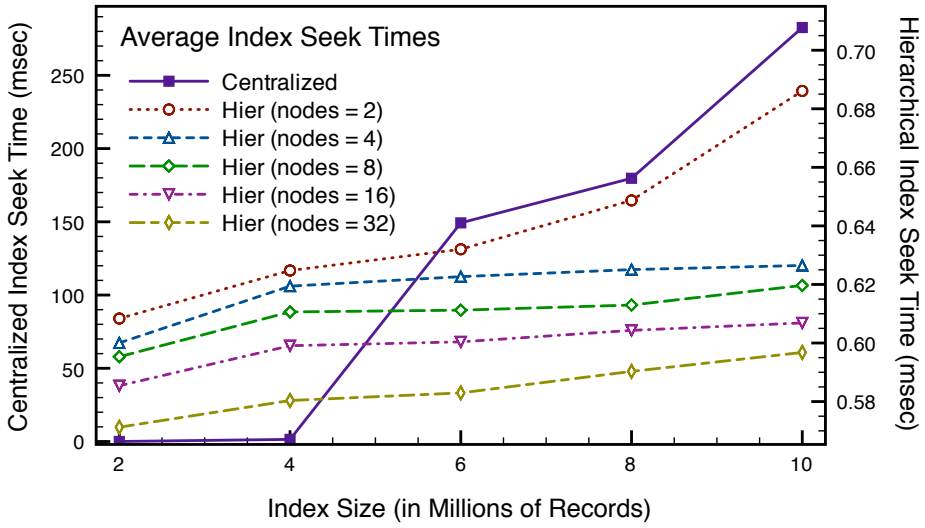


Figure 5.6: Broker and Cohort Seek Times

Not shown clearly in the graph, the centralized index queries are 0.024 msec and 1.46 msec respectively for 2×10^6 and 4×10^6 records. This supports the claim that the centralized version outperforms the hierarchical index if its record size is relatively small. But as the index grows into disk memory (in our case, around 6×10^6 records), queries on this index are subject to an exponential slowdown.

The hierarchical index was designed to elude this problem by partitioning the index's bulk into manageable parts shared by cohorts. The downside to this model, however, is that a hit in the broker index requires a subsequent hit on the cohort

level. Recall that a broker (fast) miss requires no cohort communications, and are thus omitted from our experiments because we are only interested in the penalties caused by cohort communications. The results, guided on the right-hand y-axis, tells us two things. First, the hierarchical scheme for small indices (less than 6×10^6 records) is expectedly slower than a centralized index. However, unlike the centralized scheme, average seek times do not suffer an exponential slowdown as records increase beyond 6×10^6 records because the load is split across the cohorts.

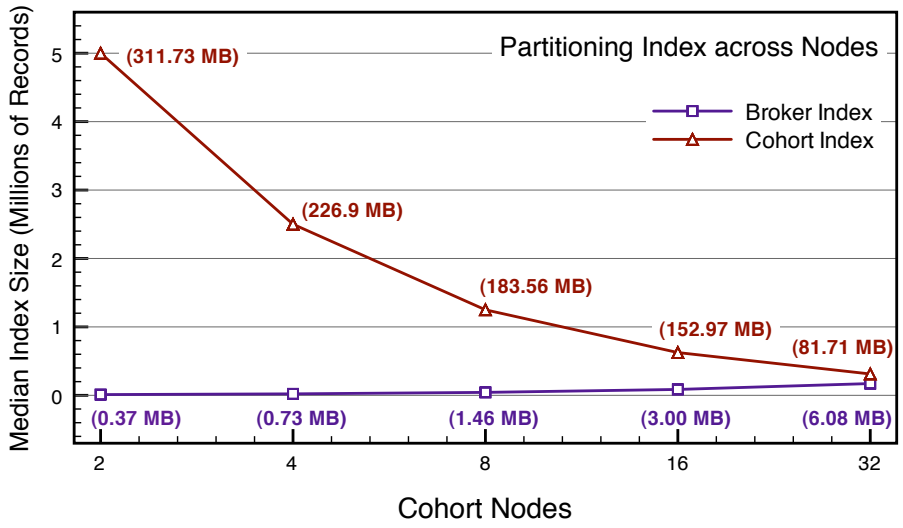


Figure 5.7: Broker and Cohort Index Sizes

The second observation is that, as cohort nodes are added, we notice a slight but consistent speed up in average seek times. This speed up is due to the cohorts being assigned smaller sets of records, and thus, faster querying. This is made clear in Figure 5.7, in which we used an index containing 10 million records and show its reduction once partitioned into $2, 4, \ldots, 32$ cohorts. While the decline in cohort size is obvious

due to index partitioning, the subtle increase in the broker index size can be explained by increased granularity. For example, when only two cohorts exist, the broker's spatiotemporal regions can only be split into one of the two cohorts. Most of the broker's records, in this case, simply hashes into the same regions, resulting in a small broker index. But as cohort nodes increase, the broker index becomes larger because records can now be hashed into far more regions — an increase in broker granularity. In summary, our results first show the potentials for intermediate data caching in scientific workflow systems. We show that our hierarchical cache scheme is effective even in low to medium bandwidth environments often resident in heterogeneous Grid environments. Most importantly, our results provide evidence supporting our case against strictly using a centralized broker index in a query intensive environment. The experiments reinforces our claim that a hierarchical structure is both efficient and scalable to Grid-scaled systems. Moreover, much more intermediate data can be indexed using this strategy over an aggressively victimized centralized index.

CHAPTER 6

COST AND PERFORMANCE ISSUES WITH WORKFLOW CACHING IN THE CLOUD

The diminishing cost of bandwidth, storage, and processing elements, together with advancements in virtualization technology, have allowed for the subsistence of *computing as a utility*. This utility computing model, the Cloud, ventures to offer users on-demand access to ostensibly unlimited computing and storage infrastructure, known as Infrastructure-as-a-Service (IaaS). The IaaS model has proved to be highly desirable for various stakeholders within the industry and the academe, as their localized data centers can now be outsourced to the Cloud to save on such costs as personnel, maintenance, and resource usage [12]. Providers, including (but certainly not limited to) Amazon AWS[1], Microsoft[2], and Google[3] have already made great strides toward ushering IaaS to the mainstream.

Particularly, scientific application users have begun harnessing the Cloud's *elastic* properties, i.e., on-demand allocation and relaxation of storage and compute resources [172, 88, 151]. Additionally, such applications have lately embraced the Web

[1]Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2

[2]Microsoft Azure, http://www.microsoft.com/windowsazure/

[3]Google App Engine, http://code.google.com/appengine

service paradigm [41] for processing and communications within distributed computing environments. Among various reasons, the interoperability and sharing/discovery capabilities are chief objectives for their adoption. Indeed, the Globus Toolkit [67] has been employed to support service-oriented science for a number of years [68]. These observations certainly do not elude scientific Cloud applications – indeed, some speculate that Clouds will eventually host a multitude of services, shared by various parties, that can be strung together like building-blocks to generate larger, more meaningful applications in processes known as *service composition*, *mashups*, and *service workflows* [78].

In this chapter, we first discuss an elastic caching mechanism that we have deployed over the Cloud, followed by an analysis of cost of deployment on Amazon Web Services (AWS), a popular Cloud platform.

6.1 Elastic Cloud Caches for Accelerating Service Computations

Situations within certain composite service applications often invoke high numbers of requests due to heightened interest from various users. In a recent, real-world example of this so-called *query-intensive* phenomenon, the catastrophic earthquake in Haiti generated massive amounts of concern and activity from the general public. This abrupt rise in interest prompted the development of several Web services in response, offering on-demand geotagged maps[4] of the disaster area to help guide relief efforts. Similarly, efforts were initiated to collect real-time images of the area, which are then composed together piecemeal by services in order to capture more holistic views. But due to their popularity, the availability of such services becomes

[4]e.g., http://apollopro.erdas.com/apollo-client

an issue during this critical time. However, because service requests during these situations are often related, e.g., displaying a traffic map of a certain populated area in Port-au-Prince, a considerable amount of redundancy among these services can be exploited. Consequently, their derived results can be reused to not only accelerate subsequent queries, but also to help reduce service traffic.

Provisioning resources for a cache storing derived data products involves a number of issues. Now consider a Software-as-a-Service (SaaS) environment where a cost is associated with every invocation of a service. By caching derived data products, a private lab or company can provide faster response, still charge the service users the same price, and save on processing costs. At the same time, if the data is cached, but left unused, it would likely incur storage costs that will not be offset by savings on processing costs. As demand for derived data can change over time, it is important to exploit the elasticity of Cloud environments, and dynamically provision storage resources.

In this section, we describe an approach to cache and utilize service-derived results. We implement a cooperative caching framework for storing the services' output data in-memory for facilitating fast accesses. Our system has been designed to automatically scale, and relax, elastic compute resources as needed. We should note that automatic scaling services exist on most Clouds. For instance, Amazon AWS allows users to assign certain rules, e.g., *scale up by one node if the average CPU usage is above 80%*. But while auto-scalers are suitable for Map-Reduce applications [48], among other easily parallelizable applications, in cases where much more distributed coordination is required, elasticity does not directly translate to scalability. Such is the case for our cache, and we have designed and evaluated specific scaling logic for our system. In the direction of the cost-incentivized down-scaling, a *decay*-based cache eviction scheme is implemented for node deallocation. Depending upon the

nature of data and services, security and authentication can be important concerns in a system of this nature [78]. Our work targets scenarios where all data and services are shared among users of that particular Cloud environment, and these issues are thus not considered here.

Using a real service to represent our workload, we have evaluated many aspects of the cache extensively over the Amazon EC2 public Cloud. In terms of utilization, the effects of the cache over our dynamic compute node allocation framework has been compared with static, fixed-node models. We also evaluate our system's resource allocation behavior. Overall, we are able to show that our cache is capable obtaining minimal miss rates while utilizing far less nodes than statically allocated systems of fixed sizes in the span of the experiment. Finally, we run well-designed experiments to show our cache's capacity for full elasticity — its ability to scale up, and down, amidst varying workloads over time.

The high-level contributions of this work are as follows. Our cache was originally proposed to speed up computations in our scientific workflow system, *Auspice* [39, 40]. Thus, the cache's API has been designed to allow for transparent integration with Auspice, and other such systems, to compose derived results directly into workflow plans. Our system is thus easily adaptable to many types of applications that can benefit from data reuse. We are furthermore considering cooperative caching in the context of Clouds, where resource allocation and deallocation should be coordinated to harness elasticity. To this end, we implement a sliding window view to capture user interest over time.

6.1.1 System Goals and Design

In this subsection, we identify several goals and requirements for our system, and we also discuss some design decisions to implement our data cache.

Provisioning Fast Access Methods:

The ability to store large quantities of precomputed data is hardly useful without efficient access. This includes not only identifying which cooperating cache node contains the data, but also facilitating fast hits and misses within that node. The former goal could be achieved through such methods as hashing or directory services, and the latter requires some considerations toward indexing. Although the index structure is application dependent, we utilize well-supported spatial indices [96, 86] due to the wide range of applications that they can accommodate and also their de facto acceptance into most practical database systems. This implies an ease of portability, which relates to the next goal.

Transparency and High-Level Integration with Existing Systems:

Our cache must subscribe to an intuitive programming interface that allows for nonintrusive integration into existing systems. Like most caches, ours should only present high-level *search* and *update* methods while hiding internal nuances from the programmer. These details might include victimization schemes, replacement policies, management of underlying compute resources, data movement, etc. In other words, our system can be viewed as a Cloud service, from the application developer's perspective, for indexing, caching, and reusing precomputed results.

Graceful Adaptation to Varying Workloads:

An increase in service request frequency implies a growing amount of data that must be cached. Taking into consideration the dimensionality of certain data sets, it is easy to predict that caches can quickly grow to sizes beyond main memory as query intensive situations arise. In-core containment of the index, however, is imperative

for facilitating fast response times in cache systems. The elastic resource allocation afforded by the Cloud is important here; in these cases, our system should also increase its available main memory to guarantee in-core access. Similarly, a decrease in request frequency should invoke a contraction of currently allocated resources.

Design Decisions

First, our cache has been designed under a cooperative scheme, where cache nodes are distributed over the Cloud, and each node stores only a portion of the entire cache. Upon a cache overflow, our system splits the overflown node and migrates its data either to a new allocated Cloud node, or an existing cooperating node. Similarly, our cache should understand when to relax and merge compute nodes to save costs. This approach is somewhat akin to distributed hashtables (DHT) and web proxies.

Each node in our system employs a variant of B+-Trees [20] to index cached data due to its familiar and pervasive nature. Because B+-Trees are widely accepted in today's database systems, its integration is simplified. Due to this fact, many approaches have been proposed in the past to extend B+-Trees to various application domains, which makes it extremely portable. Because our specific application involves spatiotemporal data sets, we utilize B^x-Trees [96] to index cached data. These structures modify B+-Trees to store spatiotemporal data through a linearization of time and location using space-filling curves, and thus, individual one-dimensional keys of the B+-Tree can represent spatiotemporality.

Another design decision addresses the need to handle changes in the cache's underlying compute structure. The B+-Tree index previously discussed is installed on each cache server in the cooperating system. However, as we explained earlier, due to memory overflow/underflow, the system may have to dynamically expand/contract. Adding and removing cache nodes should take minimal effort, which is a deceptively

hard problem. To illustrate, consider an n node cooperative cache system, and each node is assigned a distinct *id*: $0, \ldots, n-1$. Identifying the node responsible for caching some data identified by key, k, is trivial with static hashing, i.e., $h(k) = (k \mod n)$ can be computed as node *id*. Now assume that a new node is allocated, which effectively modifies the hash function to $h(k) = (k \mod n+1)$. This ostensibly simple change forces most currently keyed records to be rehashed and, worse, relocated using the new hash. Rehashing and migrating large volumes of records after each node acquisition is, without saying, prohibitive.

To handle this problem, also referred to as *hash disruption* [135], we implement consistent hashing [99]. In this hashing method, we first assume an auxiliary hash function, e.g., $h'(k) = (k \mod r)$, for some fixed r. Within this range exists a sequence of p buckets, $B = (b_1, \ldots, b_p)$, with each bucket mapped to a single cache node. Figure 6.1 (top) represents a framework consisting two nodes and five buckets. When a new key, k, arrives, it is first hashed via the auxiliary hash $h'(k)$ and then assigned to the node referenced by $h(k)$'s closest upper bucket. In our figure, the incoming k is assigned to node n_2 via b_4. Often, the hash line is implemented in a circular fashion, i.e., a key $k \mid b_5 < h'(k) \leq r-1$ would be mapped to n_1 via b_1.

Because the hash map is fixed, consistent hashing reduces hash disruption by a considerable factor. For instance, let us consider Figure 6.1 (bottom), where a new node, n_3, has been acquired and assigned by some bucket $b_6 = r/2$ to help share the load between b_3 and b_4. The introduction of n_3 would only cause a small subset of keys to be migrated, i.e., $k \mid b_3 < h'(k) \leq b_6$ (area within the shaded region) from n_2 to n_3 in lieu of a rehash of all records. Thus, we can implement the task of supporting elastic Cloud structures without hash disruption.

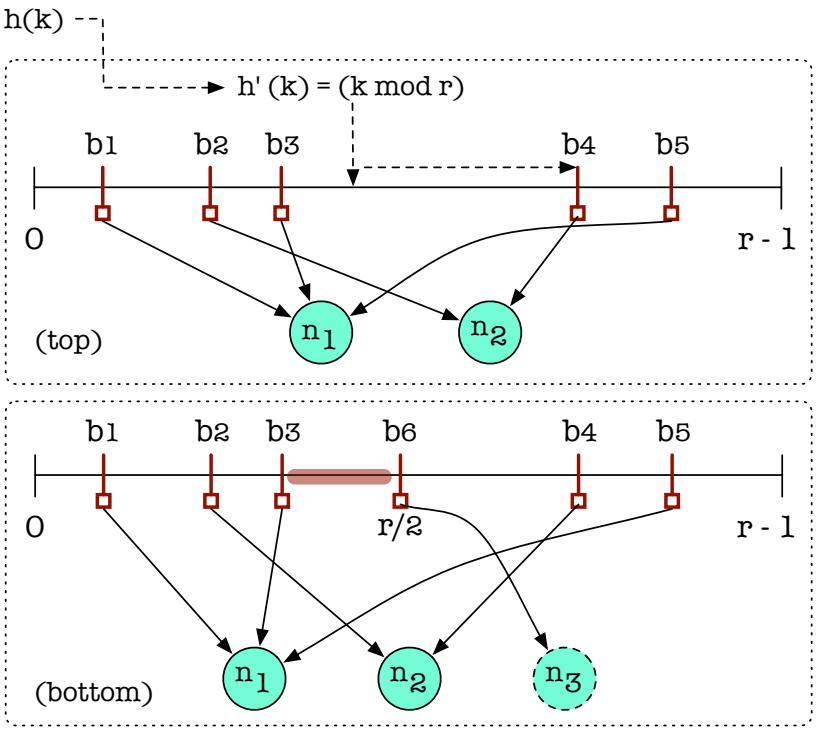


Figure 6.1: Consistent Hashing Example

6.1.2 Cache Design and Access Methods

Before presenting cache access methods, we first state the following definitions. Let $N = \{n_1, \ldots, n_m\}$ denote the currently allocated cache nodes. We define $||n||$ and $\lceil n \rceil$ to be the current space used and capacity respectively on cache node n. We further define the ordered sequence of allocated buckets as $B = (b_1, \ldots, b_p)$ such that $b_i \in [0, r)$ and $b_i < b_{i+1}$. Given an auxiliary, fixed hash function, $h'(k) = (k \mod r)$, in a circular implementation, our hash function is defined,

$$h(k) = \begin{cases} b_1, & \text{if } h'(k) > b_p \\ \underset{b_i \in B}{\arg\min}\ b_i - h'(k) : b_i \geq h'(k), & \text{otherwise} \end{cases}$$

For reading comprehension, we have provided a summary of identifiers in Table 6.1.

We can now focus on our algorithms for cache access, migration, and contraction over the Cloud. Note that we will not discuss the cache search method, as it is trivial, i.e., by running a B+-Tree search for k on the node referenced by $h(k)$.

Identifier	Description
k	A queried key
$B = (b_1, \ldots, b_p)$	The list of all buckets on the hash line
N	The set of all nodes in the cooperative cache
$n \in N$	A cache node
$\|\|n\|\|$	Current size of index on node n
$\lceil n \rceil$	Overall capacity on node n
$T = (t_1, \ldots, t_m)$	Sliding window of size m
$t_i \in T$	A single time slice in the sliding window, which records all keys that were queried in that period of time
α	The decay, $0 < \alpha < 1$, used in the calculation of $\lambda(k)$
$\lambda(k)$	Key k's likelihood of being evicted
T_λ	Eviction threshold, i.e., $k \mid \lambda(k) < T_\lambda$ are designated for eviction

Table 6.1: Listing of Identifiers

Insertion and Migration

The procedure for inserting into the cache could invoke migration, which complicates the otherwise simple insertion scheme. In Algorithm 9, the insert algorithm is defined with a pair of inputs, k and v, denoting the key and value object respectively. The Greedy Bucket Allocation (GBA) Insert algorithm is so named as to reflect that, upon node overflows, we greedily consider preexisting cache nodes as the data migration destination. In other words, node allocation is a last-resort option to save cost.

Algorithm 9 GBA-insert(k, v)

```
1: static NodeMap[...]
2: static B = (...)
3: static h' : K → [0, r)
4: n ← NodeMap[h'(k)]
5: if ||n|| + sizeof(v) < ⌈n⌉ then
6:     n.insert(k, v)     ▷ insert directly on node n
7: else
8:     ▷ n overflows
9:     ▷ find fullest bucket referencing n
10:    b_max ← argmax_{b_i ∈ B} ||b_i|| ∧ NodeMap[b_i] = n
11:    k^μ ← μ(b_max)
12:    n_dest ← n.sweep-migrate(min(b_max), k^μ)
13:    ▷ update structures
14:    B ← (b_1, ..., b_i, h'(k^μ), b_{i+1}, ..., b_p) | b_i < h'(k^μ) < b_{i+1}
15:    NodeMap[h'(k^μ))] ← n_dest
16:    GBA-insert(k, v)
17: end if
```

On Line 1, the statically declared inverse hash map is brought into scope. This structure defines the relation $NodeMap[b] = n$ where n is the node mapped to bucket value b. The ordered list of buckets, B, as well as the auxiliary consistent hash function, h', are also brought into scope (Lines 2-3). After identifying k's bucket and node (Line 4), the (k, v) pair is inserted into node n if the system determines that its insertion would not cause a memory overflow on n (Lines 5-6). Since cache indices

expanding into disk memory would become prohibitively slow, when an overflow is detected, migration of portions of the index must be invoked to make space (Line 7).

The goal of migration is to introduce a new bucket into the overflown interval that would reduce the load of about half of the keys from the overflown bucket. However, the fullest bucket may not necessarily be b. On (Line 10), we identify the fullest bucket which references n, then invoke the migration algorithm on a range of keys, to be described shortly (Line 11-12). As a simple heuristic, we opt to move approximately half the keys from bucket b_{max}, starting from the lowest key to the median, k^{μ}. The sweep-and-migrate algorithm returns a reference to the node (either preexisting or newly allocated), n_{dest}, to which the data from n has been migrated. On (Lines 13-15), the buckets, B, and node mapping data structures, $NodeMap[\ldots]$, are updated to reflect internal structural changes. Specifically, a new bucket is created at $h'(k^{\mu})$ and it references n_{dest}. The algorithm is finally invoked recursively to attempt proper insertion under the modified cache structure.

The *Sweep-and-Migrate* function, shown in Algorithm 10, resides on each individual cache server, along with the indexing logic. As an aside, in our implementation, the cache server is automatically fetched from a remote location on the startup of a new Cloud instance. The algorithm inputs the range of keys to be migrated, k_{start} and k_{end}. The least loaded node is first identified from the current cache configuration (Line 1). If it is projected that the key range cannot fit within n_{dest}, then a new node must be allocated from the Cloud (Lines 2-5). The aggregation test (Line 2) can be done by maintaining an internal structure on the server which holds the keys' respective object size.

Once the destination node has been identified we begin the transfer of the key range. We now describe the approach to find and *sweep* all keys in the specified range from the internal B+-Tree index. The B+-Tree's linked leaf structure simplifies the

Algorithm 10 sweep-migrate(k_{start}, k_{end})

1: $n_{dest} \leftarrow \underset{n_i \in N}{\text{argmin}} ||n_i||$
2: ▷ stolen keys and values will overflow n_{dest}
3: **if** $||n_{dest}|| + \sum_{k=k_{start}}^{k_{end}} sizeof(k, v) > \lceil n_{dest} \rceil$ **then**
4: $n_{dest} \leftarrow$ nodeAlloc()
5: **end if**
6: ▷ manipulate B+-Tree index and transfer to n_{dest}
7: $end \leftarrow false$
8: ▷ L = leaf initially containing k_{start}
9: $L \leftarrow btree$.search(k_{start})
10: **while** ($\neg end \wedge L \neq NULL$) **do**
11: ▷ each leaf node contains multiple keys
12: **for all** $(k, v) \in L$ **do**
13: **if** $k \leq k_{end}$ **then**
14: n_{dest}.insert(k, v)
15: $btree$.delete(k)
16: **else**
17: $end \leftarrow true$
18: break
19: **end if**
20: **end for**
21: $L \leftarrow L$.next()
22: **end while**
23: **return** n_{dest}

record sweep portion of our algorithm. First, a search for k_{start} is invoked to locate its leaf node (Line 9). Then, recalling that leaf nodes are arranged as a key-sorted linked list in B+-Trees, a sweep (Line 10-22) on the leaf level is performed until k_{end} has been reached. For each leaf visited, we transfer all associated (k, v) record to n_{dest}.

Analysis of GBA-Insert

GBA-insert is difficult to generalize due to variabilities of the system state, which can drastically affect the runtime behavior of migration, e.g., number of buckets, migrated keys, size of each object, etc. To be succinct in our analysis, we make the simple assumption that $sizeof((k, v)) = 1$ to normalize cached records. This simplification also allows us to imply an even distribution over all buckets in B and nodes in N. In the following, we only consider the worst case.

We begin with the analysis of sweep-and-migrate (Algorithm 10), whose time complexity is denoted $T_{migrate}$. First, the maximum number of keys that can be stolen from any node is half of the record capacity of any node: $\lceil n \rceil/2$. This is again due to our assumption of an even bucket/node distribution, which would cause Algorithm 9's calculation of $min(b_{max})$ and k^{μ} to be assigned such that $min(b_{max}) - k^{\mu} \approx \lceil n \rceil/2$, and thus the *sweep* phase can be analyzed as having an $O(\log_2 ||n||)$-time B+-Tree search followed by a linear sweep of $\lceil n \rceil/2$ records, i.e., $\log_2 ||n|| + \lceil n \rceil/2$. The complexity of $T_{migrate}$, then, is the sum of the above sweep time and the time taken to move the worst case number of records to another node. If we let T_{net} denote the time taken to move one record,

$$T_{migrate} = \log_2 ||n|| + \lceil n \rceil/2(T_{net} + 1)$$

We are now ready to solve for T_{GBA}, the runtime of Algorithm 9. As noted previously, $h(k)$ can be implemented using binary search on B – the ordered sequence of p buckets, i.e., $T(h(k)) = O(\log_2 p)$. After the initial hash function is invoked, the algorithm

enters the following cases: (i) the record is inserted trivially, or (ii) a call to migrate is made before trivially inserting the record (which requires a subsequent hash call). That is,

$$T_{GBA} = \begin{cases} \log_2 p, & \text{if } ||n|| + 1 < \lceil n \rceil \\ 2\log_2 p + T_{migrate}, & \text{otherwise} \end{cases}$$

Finally, after substitution and worst case binding, we arrive at the following conditional complexity due to the expected dominance of record transfer time, T_{net},

$$T_{GBA} = \begin{cases} O(1), & \text{if } ||n|| + 1 < \lceil n \rceil \\ O((\lceil n \rceil / 2) T_{net}), & \text{otherwise} \end{cases}$$

Although T_{net} is neither uniform nor trivial in practice, our analysis is sound as actual record sizes would likely increase T_{net}. But despite the variations on T_{net}, the bound for the latter case of T_{GBA} remains consistent due to the significant contribution of data transfer times.

Cache Eviction

Consider the situation when some interesting event/phenomenon causes a barrage of queries in a very short amount of time. Up till now, we have discussed methods for scaling our cache system *up* to meet the demands of these query-intensive circumstances. However, this demanding period may abate over time, and the resources provisioned by our system often become superfluous. In traditional distributed (e.g., cluster and grid) environments, this was less of an issue. For instance, in advance reservation schemes, resources are reserved for some fixed amount of time, and there is little incentive to scale back down. In contrast, the Cloud's usage costs prompts an important motivation to scale our system down.

We implement a cache contraction scheme to merge nodes when query intensities are lowered. Our scheme is based on a combination of exponential decay and a

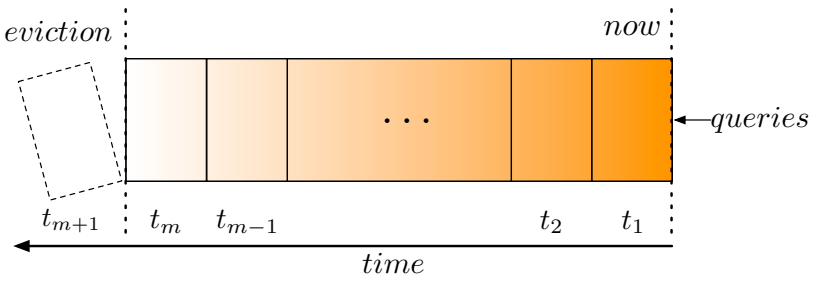


Figure 6.2: Sliding Window of the Most Recently Queried Keys

temporal sliding window. Because the size of our cache system (number of nodes) is highly dependent on the frequency of queries during some timespan, we describe a global cache eviction scheme that captures querying behavior. In our contraction scheme, we employ a streaming model, where incoming query requests represent streaming data, and a global view of the most recently queried keys is maintained in a sliding window. Shown in Figure 6.2, our sliding window, $T = (t_1, \ldots, t_m)$, comprises m time slices of some fixed real-time length. Each time slice, t_i, associates a set of keys queried in the duration of that slice. We argue that, as time passes, older unreferenced keys (i.e., those in the lighter shaded region, t_i nearing t_m) should have a lower probability of existing in the cache. As these less relevant keys become evicted, the system makes room for newer, incoming keys (i.e., those in the darker shaded region, t_i nearing t_1) and thus capturing temporal locality of the queries.

Cache eviction occurs when a time slice has reached t_{m+1}, and at this time, an eviction score,

$$\lambda(k) = \sum_{i=1}^{m} \alpha^{i-1} |\{k \in t_i\}|$$

is computed for every key, k, within the expired slice. The ratio, $\alpha : 0 < \alpha < 1$, is a *decay* factor, and $|\{k \in t_i\}|$ returns the number of times k appears in some slice

t_i. Here, α is passive in the sense that a higher value corresponds to a larger amount of keys that is kept in the system. After λ has been computed for each key in t_{m+1}, any key whose λ falls below the threshold, T_λ, is evicted from the system. Notice that α is amortized in the older time slices, in other words, recent queries for k are rewarded, so k is less likely to be evicted. Clearly, the sliding window eviction method is sensitive to the values of α and m. A baseline value for T_λ would be α^{m-1}, which will not allow the system to evict any key if it was queried even just once in the span of the sliding window. We will show their effects in the experimental section.

Due to the eviction strategy, a set of cache nodes may eventually become lightly loaded, which is an opportunity to scale our system down. The nodes' indices can be merged, and subsequently, the superfluous node instances can be discarded. When a time slice expires, our system invokes a simple heuristic for *contraction*. Our system monitors the memory capacity on each node. After each interval of ϵ slice expirations, we identify the two least loaded nodes and check whether merging their data would cause an overflow. If not, then their data is migrated using methods tantamount to Algorithm 10.

Analysis of Eviction and Contraction

The contraction time is the sum of eviction time and node merge time, $T_{contract} = T_{evict} + T_{merge}$. To analyze merge time, we first note that it takes $O(1)$ time to identify the two least loaded nodes, as we can dynamically maintain a list of nodes sorted by capacity. If the data merge is determined to be too dangerous to perform, the algorithm simply halts. On the other hand, it executes a slight variant of the *Sweep-and-Migrate* algorithm to move the index from one node to another, which, combined with our previous analysis of $T_{migrate}$, is $||n_{min}||(T_{net} + 1)$ where $||n_{min}||$ is the size of the migrated index. If we ignore the best case $O(1)$ time expended when contraction

is infeasible, then the time taken by T_{merge} can be summarized as follows,

$$T_{merge} = ||n_{min}||(T_{net} + 1)$$

The contraction method is invoked every ϵ time slices' expiration from the sliding window. By itself, the sliding window's slice eviction method, T_{evict} can be summarized by $T_{evict} = mK$ where m is the size of the sliding window, and $K = |\{k \in t_{m+1}\}|$ is the total number of keys in the evicted time slice, t_{m+1}. However, since T_{evict} again pale against network traffic time, T_{net}, its contribution can be assumed trivial. Together, the overall eviction and contraction method can be bound $T_{contract} = O(||n_{min}||T_{net})$.

6.1.3 Experimental Evaluation

n this section, we discuss the evaluation of our derived data cache system. We employ the Amazon Elastic Compute Cloud (EC2) to support all of our experiments.

Experimental Setup

Each Cloud node instance runs an Ubuntu Linux image on which our cache server logic is installed. Each image runs on a *Small EC2 Instance*, which, according to Amazon, comprises 1.7 GB of memory, 1 virtual core (equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor) on a 32-bit platform. In all of our experiments, the caches are initially cold, and both index and data are stored in memory.

As a representative workload, we executed repeated runs of a *Shoreline Extraction* query. This is a real application, provided to us by our colleagues in the Department of Civil and Environmental Engineering and Geodetic Science here at Ohio State University. Given pair of inputs: location, L, and time of interest, T, this service first retrieves a local copy of the Coastal Terrain Model (CTM) file with respect to (L, T). To enable this search, each file has been indexed via their spatiotemporal metadata. CTMs contain a large matrix of a coastal area where each point denotes a

depth/elevation reading. Next, the service retrieves actual water level readings, and finally given the CTM and water level, the coast line is interpolated and returned. The baseline execution time of this service, i.e., when executed without any caching, typically takes approximately 23 seconds to complete, and the derived shoreline result is $<$ 1kb.

We have randomized inputs over 64K possibilities for each service request, which emulates the worst case for possible reuse. The 64K input keys represent linearized coordinates and date (we used the method described in B^x-Trees [96]). The queries are first sent to a coordinating compute node, and the underlying cooperating cache is then searched on the input key to find a replica of the precomputed results. Upon a hit, the results are transmitted directly back to the caller, whereas a miss would prompt the coordinator to invoke the shoreline extraction service.

In the following experiments, in order to regulate the integrity in querying rates, we submitted queries with the following loop:

```
for time step i ← 1 to ... do
  R ← current query rate(i)
  for j ← 1 to R do
    invoke shoreline service(rand_coordinates())
  end for
end for
```

Specifically, we invoke R queries per *time step*, and thus each time step does not reflect real time. Note that the granularity of a time step in practice, e.g., t seconds, minutes, or hours, does not affect the overall hit/miss rates of the cache. At each time step, we observed and recorded the average service execution time (in number

of seconds real time), the number of times a query reuses a cached record (i.e., hits), and the number of cache misses.

Evaluating Cache Benefits

The initial experiment evaluates the effects of the cache without node contraction. In other words, the length of our eviction sliding window is ∞. Under this configuration, our cache is able to grow as large as it needs to handle the size of the cache. We run our cache system over static, fixed-node configurations (*static-2, static-4, static-8*), comparable to current cluster/grid environments, where the amounts of nodes one can allocate is typically fixed. We then compare these static versions against our approach, Greedy Bucket Allocation (*GBA*), which runs over the EC2 public Cloud. For these experiments, we submitted one query per time step, i.e., the query submission loop is configured $R = 1$ over 2×10^5 time steps.

We executed the shoreline service repeatedly with varying inputs. Figure 6.3 displays the miss rates over repeated service query executions. Notice that the miss rates (shown against the left y-axis) for *static-2*, *static-4*, and *static-8* converge at relatively high values early into the experiment due to capacity misses. The *static-2* version obtained minimum miss rate of 86.9%. The *static-4* version converged at 74.4%, and *static-8* converged at 50.25%. Because we are executing *GBA* with an infinite eviction window, we do not encounter the capacity issue since our eviction algorithm will never be invoked. This, however, comes at the cost of requiring more nodes than the static configuration. Toward the end of the run, *GBA* is capable of attaining miss rates of only 5.75%.

The node allocation behavior (shown against the right y-axis) shows that *GBA* allocates 15 nodes in the end of the experiment. But since allocation was only invoked as a last resort on-demand option, $\lceil 12.6 \rceil = 13$ nodes were utilized, if averaged over

the lifespan of this experiment. This translates to less overall EC2 usage cost per performance over static allocations. The growth of nodes is also not unexpected, though, at first glance it appears to be exponential. Early into the experiment, the cooperating cache's overall capacity is initially too small to handle the query rate, until stabilizing after $\sim$ 75000 queries have been processed.

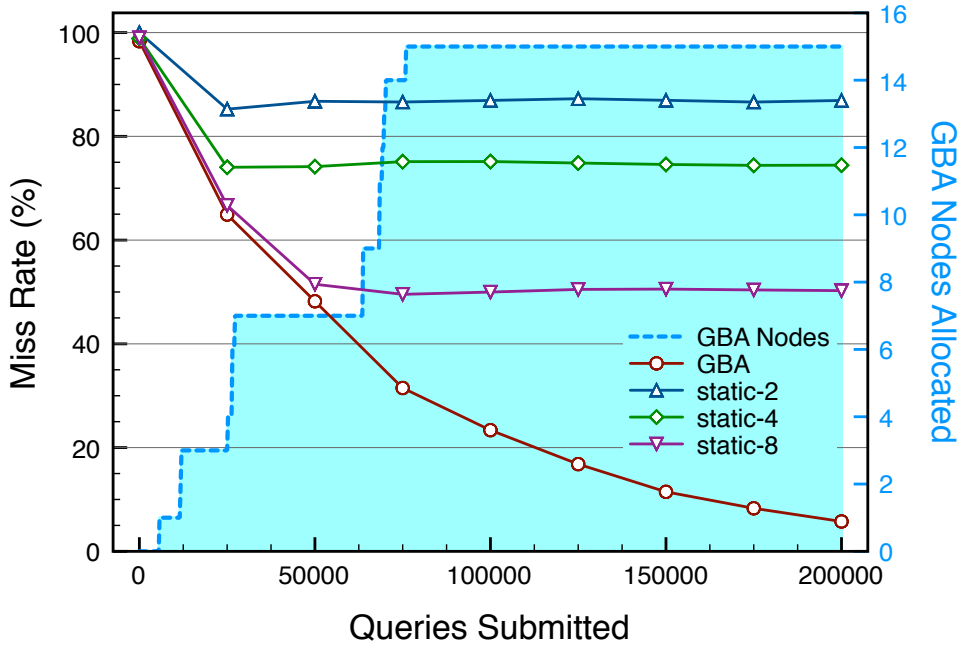


Figure 6.3: Miss Rates

Figure 6.4, which shows the respective relative speedups over the query's actual execution time, corresponds directly to the above miss rates. We observed and plotted the speedup for every $I = 25000$ queries elapsed in our experiment. Expectedly, the speedup provided by the static versions flatten somewhat quickly, again due to the nodes reaching capacity. The relative speedups converge at 1.15× for *static-2*, 1.34×

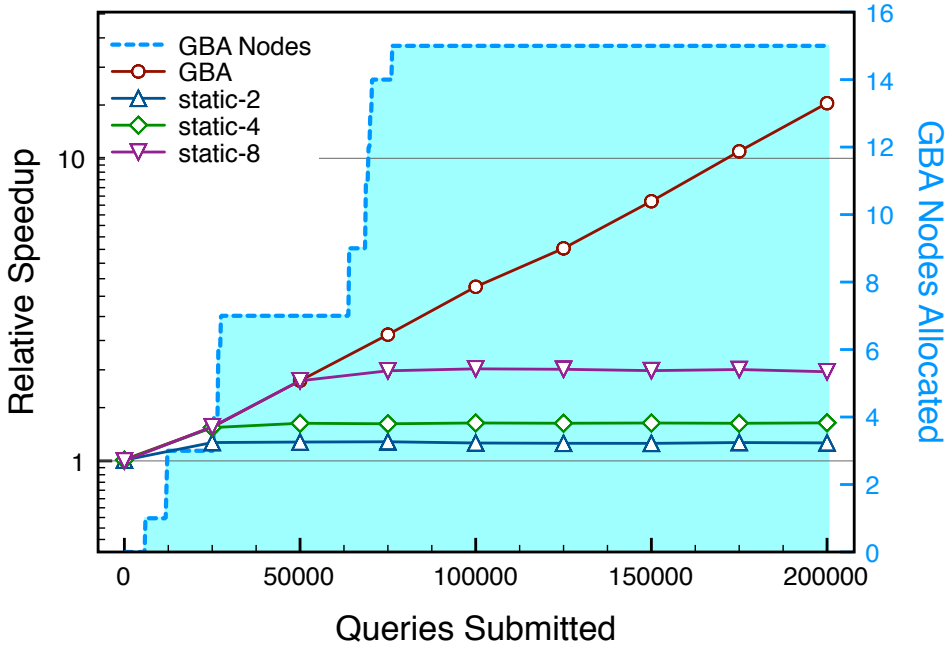


Figure 6.4: Speedups Relative to Original Service Execution

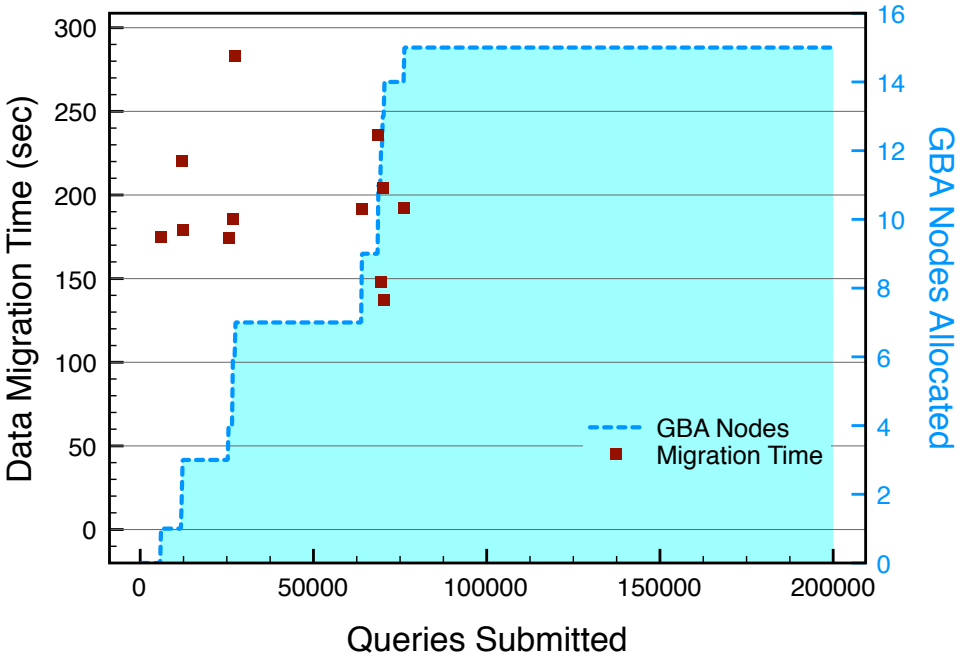


Figure 6.5: *GBA* Migration Times

for *static-4*, and 2× for *static-8*. *GBA*, on the other hand, was capable of achieving a relative speedup of over 15.2×. Note that Figure 6.4 is shown in $\log_{10}$-scale.

Next, we summarize in Figure 6.5 the overhead of node splitting (upon cache overflows) as the sum of node allocation and data migration times for *GBA*. It is clear from this figure that this overhead can be quite large. Although not shown directly in the figure, we note it is the node allocation time, and not the data movement time, which is the main contributor to this overhead. However, these penalties are amortized because node allocation is only seldom invoked. We also posit that the demand for node allocation diminishes as the experiment proceeds even with high querying rates due to system stabilization. Moreover, techniques, such as asynchronous preloading of EC2 instances and replication, can also be used to further minimize this overhead, although these have not been considered in this paper.

Evaluating Cache Eviction and Contraction

Next, we evaluate our eviction and contraction scheme. Two separate experiments were devised to show the effects of the sliding window and to show that our cache is capable of relaxing resources when feasible. We randomize the query inputs points over 32K possibilities, and we generated a workload to simulate a *query intensive situation*, such as the one described in the introduction in the following manner. Recall, from the query submission loop we stated early in this section, that a *time step* denotes an iteration where R queries are submitted. Specifically, in the following experiments, for the first 100 time steps, the querying rate is fixed at $R = 50$ queries/time step. From 101 to 300 time steps, we enter an intensive period of $R = 250$ queries/time step to simulate heightened interest. Finally, from 400 time steps onward, the query rate reduced back down to $R = 50$ queries/time step to simulate waning interest.

We show the relative speedup for varying sliding window sizes of $m = 50$ time

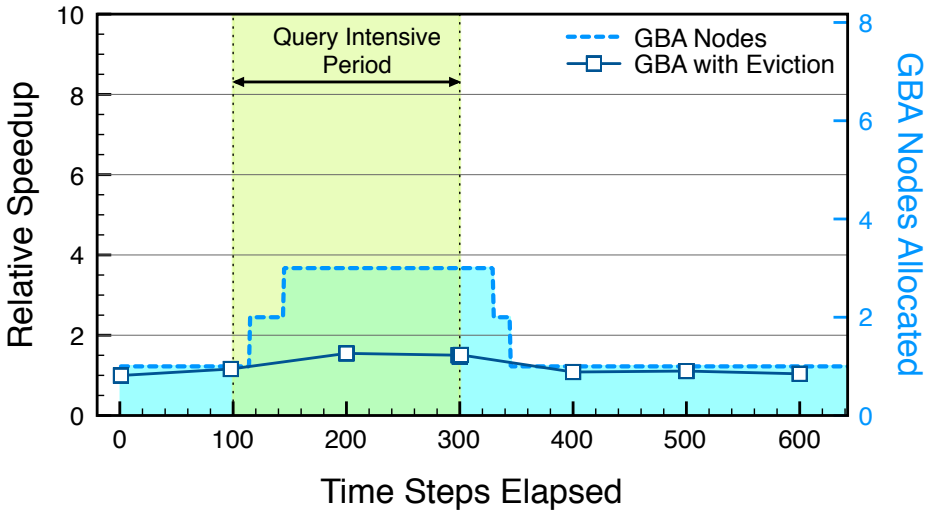


Figure 6.6: Speedup: Sliding Window Size = 50 time steps

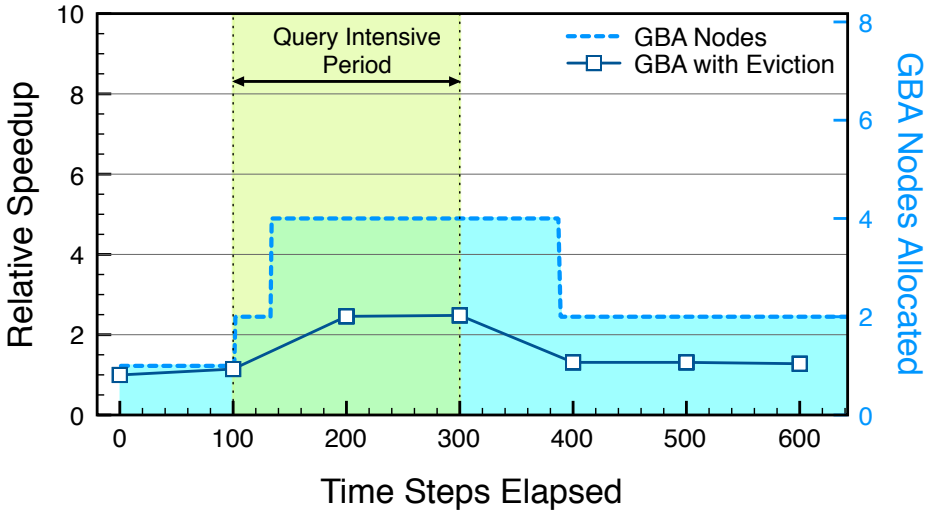


Figure 6.7: Speedup: Sliding Window Size = 100 time steps

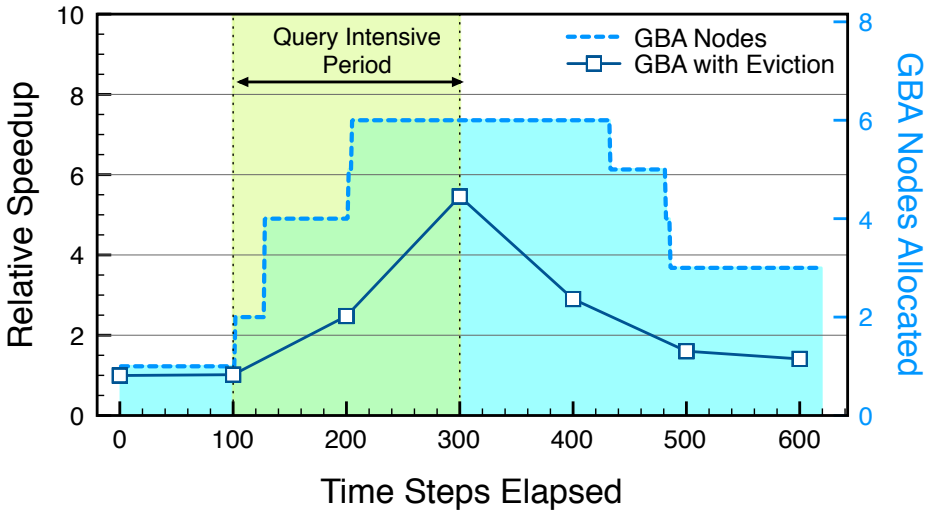


Figure 6.8: Speedup: Sliding Window Size = 200 time steps

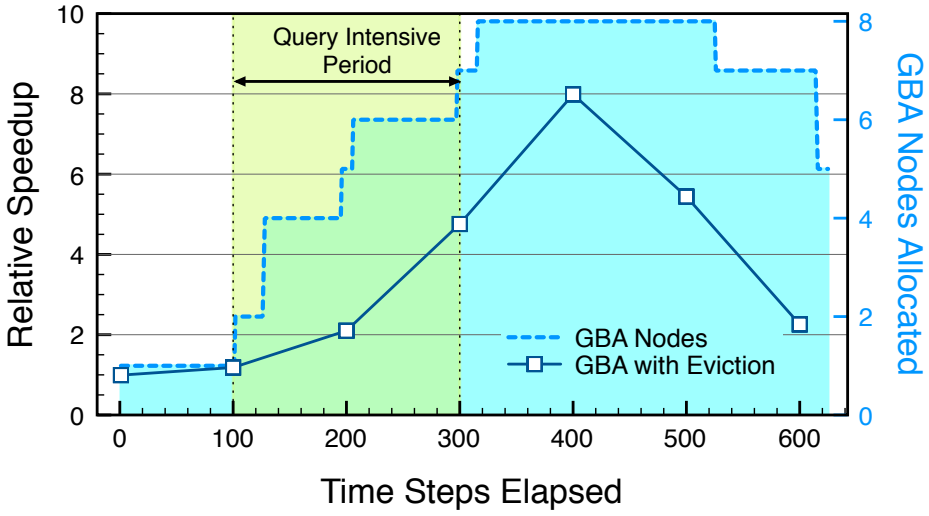


Figure 6.9: Speedup: Sliding Window Size = 400 time steps

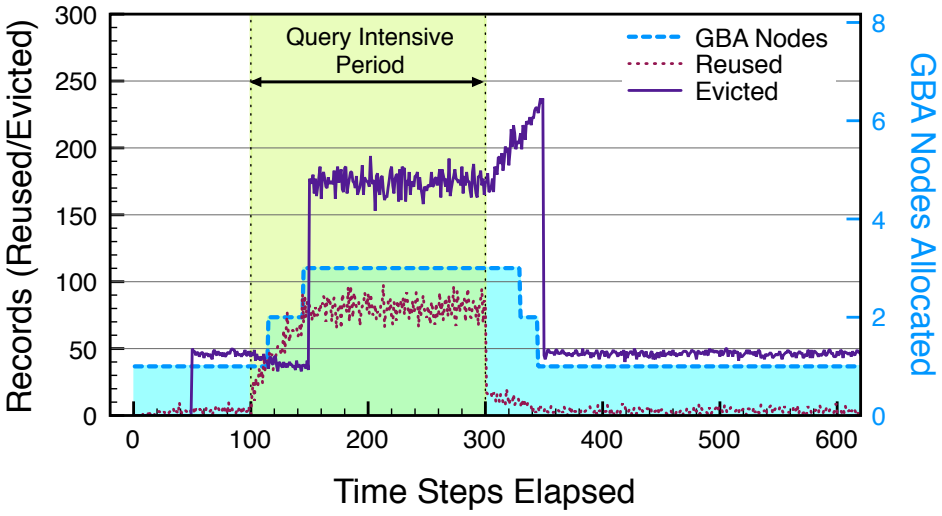


Figure 6.10: Reuse and Eviction: Sliding Window Size = 50 time steps

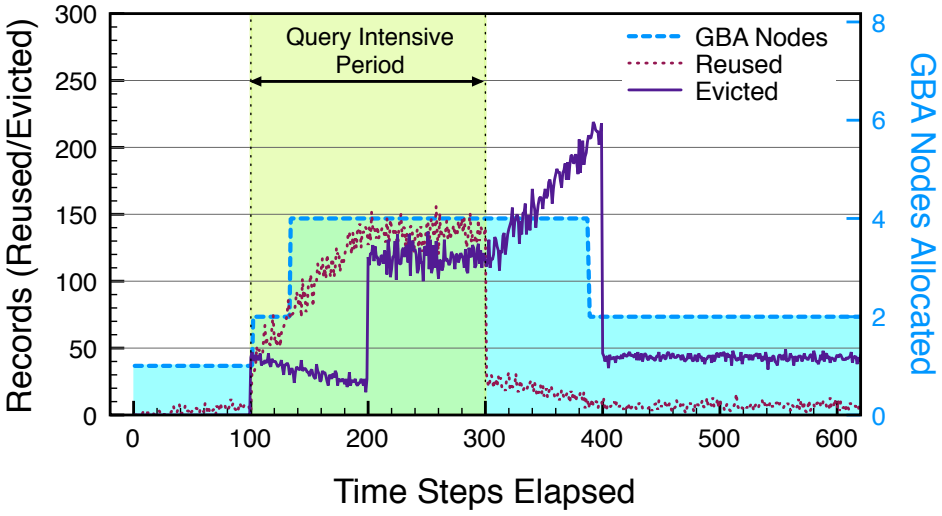


Figure 6.11: Reuse and Eviction: Sliding Window Size = 100 time steps

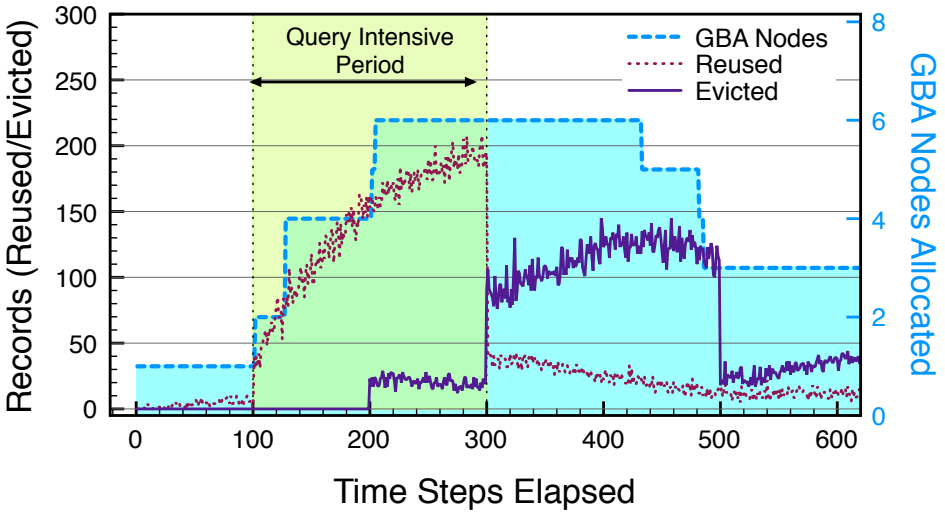


Figure 6.12: Reuse and Eviction: Sliding Window Size = 200 time steps

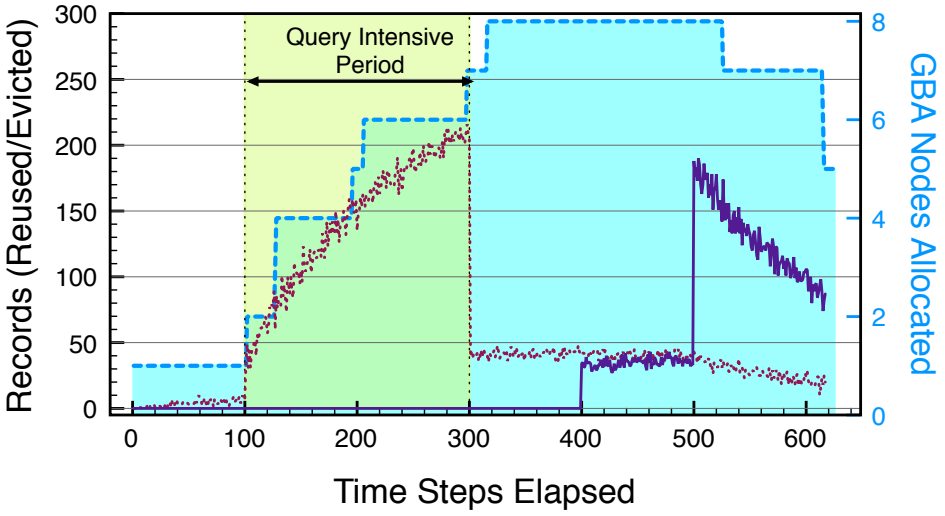


Figure 6.13: Reuse and Eviction: Sliding Window Size = 400 time steps

steps, $m = 100$ time steps, $m = 200$ time steps, and $m = 400$ time steps in Figures 6.6, 6.7, 6.8, and 6.9 respectively. Recall that the sliding window will attempt to maintain, with high probability, all records that were queried in the m most recent time steps. To ensure this probability, the decay has been fixed at $\alpha = 0.99$ for these experiments, and the eviction threshold is set at the baseline $T_\lambda = \alpha^{m-1} \approx 0.367$ to avoid evicting any key which had been queried even just once within the window.

From these figures, we can observe that our cache elastically adapts to the query-intensive period by improving overall speedup, albeit to varying degrees depending on m. For example, the maximum observable speedup achieved with the smaller sized window in Figure 6.6 is approximately 1.55×, with an average node allocation of $\lceil 1.7 \rceil = 2$ nodes. In contrast, the much larger sliding window of 400 in Figure 6.9 offers a maximum observable speedup of 8×, with an average use of $\lceil 5.6 \rceil = 6$ nodes. We can also observe that, after the query intensive period expires at 300 time steps, the sliding window will detect the normal querying rates and remove nodes as they become superfluous. This trend can also be seen in all cases — nodes do not decrease back down to 1 because our contraction algorithm is quite conservative. We have set our node-merge threshold to 65% of space required to store the coalesced cache to address *churn-avoidance*, i.e., repeated allocation/deallocation of nodes.

In terms of performance, our system benefits from higher querying rates, as it populates our cache faster within the window. The noticeable performance disparities among the juxtaposed figures also indicate that the size of the sliding window is a highly determinant factor on both performance and node allocation, i.e., cost. Compared with the ∞ sliding window experiments in Figure 6.4, we can observe that our eviction scheme affords us comparable results with lesser amounts of nodes, which translates to smaller cost of compute resource provisioning in the Cloud.

For these same experiments, we analyze the eviction and data reuse and eviction

behavior over time in Figures 6.10, 6.11, 6.12, and 6.13. One can see that, invariably, reuse expectedly increase over the query-intensive period, again to varying degrees depending on window size. After 300 time steps into the experiment, the query rate resumes to $R = 50$/time step, which means less chances for reuse. This allows aggressive eviction behaviors in all cases, except in Figure 6.13, where the window extends beyond 300 time steps.

There are several interesting trends that can be seen in these experiments. First, the eviction behavior in Figure 6.13 appears to oppose the upward trend observed in all other cases. Due to the size of this window, the decay becomes extremely small near the evicted time slice, and our cache removes records quite aggressively. At the same time, this eviction behavior decreases over time due to the evicted slices being a part of the query-intensive period, which accounted for more reuse, and thus, less probability for eviction. This trend simply was not seen in all other cases because the window size did not allow for such probability for reuse before records were candidates for eviction.

Another interesting observation can be made on node growth between Figures 6.12 and 6.13. Notice that node allocation continues to increase well after the intensive period in Figure 6.13 due to its larger window size. While this ensures more hits after the query-intensive period expires, justifying the tradeoff of allocation cost and the speedup of the queries after 300 time steps is questionable in this scenario. This implies that a dynamic window size can be employed here to optimize costs, which we plan to address in future works.

Finally, we present the effects of the decay, α, on cache eviction behavior. We used same querying configuration as in the above sliding window experiments, where normal querying rate is $R = 50$ queries/time step, and the intensive rate is $R = 250$ queries/time step. We evaluated the eviction mechanism under the $m = 100$ sliding

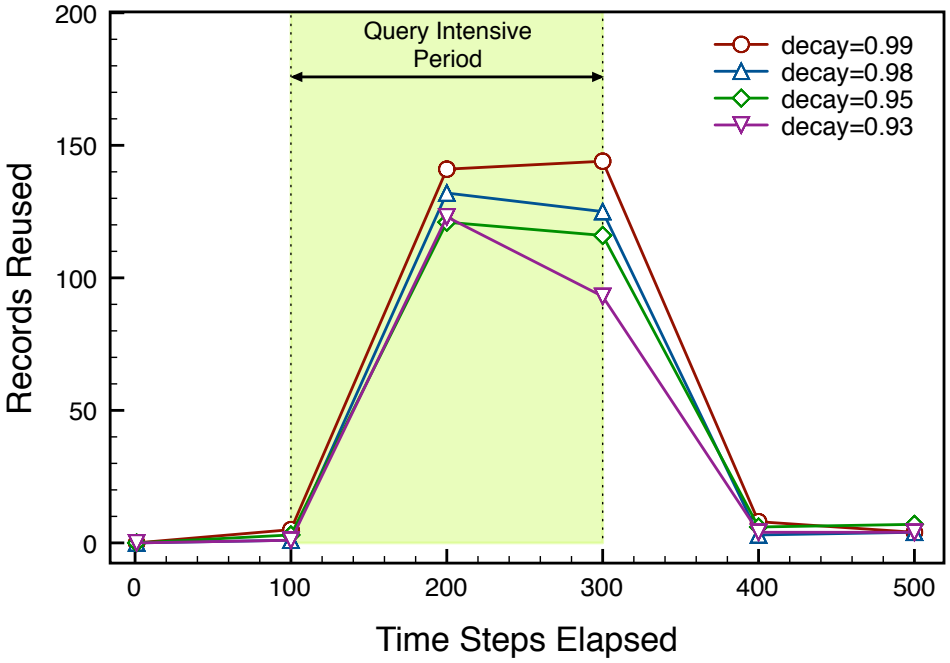


Figure 6.14: Data Reuse Behavior for Various Decay $\alpha = 0.99, 0.98, 0.95, 0.93$

window configuration on four decay values: $\alpha = 0.99, 0.98, 0.95, 0.93$. We would expect that a smaller decay value would lead to more aggressive eviction, which can be inferred from Figure 6.14. Also note the sensitivity of α due to its exponential nature.

When decay is small, a certain record must be reused many more times to be kept cached in the window. However, the benefit of this can also be argued from the perspective of cost – the cache system pertaining to a smaller α grows much slower and, according to Figure 6.14, the number of actual cache hits over this execution does not seem to vary enough to make any extraordinary contribution to speedup.

Summary and Discussion

We have evaluated our cooperative cache system from various perspectives. The relative performance gains from the infinite eviction window experiments show that caching service results over the Cloud is a fruitful endeavour, but it comes at the expense of high node allocation for ensuring cache capacity. We showed that the overhead of node splitting can be quite high, but is so seldom invoked that its penalties are amortized over the sheer volume of queries submitted. We also argue that it is rarely invoked once the cache's capacity stabilizes. However, this prompts a need for more intelligent strategies for reducing node allocation penalties.

Our sliding window-based eviction strategy appears to offer a good compromise between performance and cost tradeoffs, and captures situations with heightened (and waning) query intensities. For instance, the larger $m = 400$ sliding window, shown in Figure 6.13, achieves an $8\times$ speedup at the peak of the query intensive period, while only requiring a maximum of 8 nodes, which further reduces down to 5 nodes toward the end of the experiment.

Finally, through a study of eviction decay, we are able to conclude that both

system parameters, α and sliding window size m, account for node growth (and thus, cost) and performance. However, it is m that contributes far more significantly to our system. A dynamically changing m can thus be very useful in driving down cost.

6.2 Evaluating Caching and Storage Options on the Amazon Web Services Cloud

The mounting growth of scientific data has spurred the need to facilitate highly responsive compute- and data-intensive processes. Such large-scale applications have traditionally been hosted on commodity clusters or grid platforms. However, the recent emergence of on-demand computing is causing many to rethink whether it would be more cost-effective to move their projects onto the Cloud. Several attractive features offered by Cloud providers, after all, suit scientific applications nicely. Among those, elastic resource provisioning enables applications to expand and relax computing instances as needed to scale and save costs respectively. Affordable and reliable persistent storage are also amenable to supporting the data deluge that is often present in these applications.

A key novel consideration in Cloud computing is the pricing of each resource and the resulting costs for execution of an application. Together with considerations like wall-clock completion time, throughput, scalability, and efficiency, which have been metrics in traditional HPC environments, the cost of execution of an application is very important. Several recent studies have evaluated the use of Cloud computing for scientific applications with this consideration.

A key novel consideration in Cloud computing is the pricing of each resource and the resulting costs for the execution of an application. Together with considerations like wall-clock completion time, throughput, scalability, and efficiency, which have

been metrics in traditional HPC environments, the cost of execution of an application is very important. Several recent studies have evaluated the use of Cloud computing for scientific applications with this consideration. For example, Deelman, *et al.* studied the practicality of utilizing the Amazon Cloud for an astronomy application, Montage [51]. Elsewhere, researchers discussed the challenges in mapping a remote sensing *pipeline* onto Microsoft Azure [114]. In [103], the authors studied the cost and feasibility of supporting BOINC [9] applications, e.g., SETI@home, using Amazon's cost model. Vecchiola, *et al.* deployed medical imaging application onto Aneka, their Cloud middleware [172]. An analysis on using storage Clouds [132] for large-scale projects have also been performed. While other such efforts exist, the aforementioned studies are certainly representative of the growing interest in Cloud-supported frameworks. However, there are several dimensions to the performance and cost of executing an application in a Cloud environment. While CPU and network transfer costs for executing scientific workflows and processes have been evaluated in these efforts, several aspects of the use of Cloud environments require careful examination.

In this section, we focus on evaluating the performance and costs associated with a number of caching and storage options offered by the Cloud. The motivation for our work is that, in compute- and data-intensive applications, there could be considerable advantage in caching intermediate data sets and results for sharing or reuse. Especially in scientific workflow applications, where task dependencies are abundant, there could be significant amounts of redundancy among related processes [182, 38]. Clearly, such tasks could benefit from fetching and reusing any stored precomputed data. But whereas the Cloud offers ample flexibility in provisioning the resources to store such data, weighing the tradeoff between performance and usage costs makes for a compelling challenge.

The Amazon Web Service (AWS) Cloud [14], which is being considered in this

paper, offers many ways for users to cache and store data. In one approach, a cohort of virtual machine instances can be invoked, and data can be stored either on disk or in memory (for faster access, but with limited capacity). The costs of maintaining such a cache would also be more expensive, as users are charged a fixed rate per hour. This fixed rate is moreover dependent on the requested machine instances' processing power, memory capacity, bandwidth, etc. On the other hand, AWS's Simple Storage Service (S3) can also be used store cached data. It could be a much cheaper alternative, as users are charged a fixed rate per GB stored per month. Data are also persisted on S3, but because of this overhead, we might expect some I/O delays. However, depending on the application user's requirements, performance may well outweigh costs or vice versa.

We offer an in-depth view of the tradeoffs in employing the various AWS options for caching data to accelerate their processes. Our contributions are as follows. We evaluate performance and cost behavior given various average data sizes of an application. Several combinations of Cloud features are evaluated *vis à vis* as possible cache storage options, serving disparate requirements, including data persistence, cost, and high-performance needs. We believe that our analysis would be useful to the computing community by offering new insights into employing the AWS Cloud. Our experimental results may also generate ideas for novel cost-effective caching strategies.

6.2.1 Background

We briefly present the various Infrastructure-as-a-Service (IaaS) features offered by the Amazon Web Services (AWS) Cloud, which includes persistent storage and on-demand compute nodes.

6.2.2 Amazon Cloud Services and Costs

AWS offers many options for on-demand computing as a part of their Elastic Compute Cloud (EC2) service. EC2 nodes (*instances*) are virtual machines that can launch snapshots of systems, i.e., *images*. These images can be deployed onto various *instance types* (the underlying virtualized architecture) with varying costs depending on the instance type's capabilities.

AWS Feature	**Cost (USD)**
S3	\$0.15 per GB-month \$0.15 per GB-out \$0.01 per 1000 in-requests \$0.01 per 10000 out-requests
Small EC2 Instance	\$0.085 per allocated-hour \$0.15 per GB-out
Extra Large EC2 Instance	\$0.68 per allocated-hour \$0.15 per GB-out
EBS	\$0.10 per GB-month \$0.10 per 1 million I/O requests

Table 6.2: Amazon Web Services Costs

For example, a Small EC2 Instance (`m1.small`), according to AWS[5] at the time of writing, contains 1.7 GB memory, 1 virtual core (equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor), and 160 GB disk storage. AWS also states that the Small Instance has *moderate* network I/O. Another instance type we consider is the Extra Large EC2 instance (`m1.xlarge`), which contains 15 GB memory, 4 virtual cores with 2 EC2 Compute Units each, 1.69 TB disk storage with *high* I/O Performance. Their costs are shown in Table 6.2. We focus on these two highly

[5]AWS Instance Types, http://aws.amazon.com/ec2/instance-types

contrasting instance types in this paper because they offer a wide spectrum between capabilities, while also noting that several other instance types are in fact available in EC2.

Amazon's persistent storage framework, Simple Storage Service (S3), provides a key-value store with simple ftp-style API: `put`, `get`, `del`, etc. Typically, the unique keys are represented by a filename, and the values are themselves the data objects, i.e., files. While the objects themselves are limited to 5 GB, the number of objects that can be stored in S3 is unlimited. Aside from the simple API, the S3 architecture has been designed to be highly reliable and available. It is furthermore very cheap (see Table 6.2) to store data on S3.

Another option for persistent storage is to employ Elastic Block Storage (EBS) in conjunction with EC2 instances. The EBS service is a persistent disk volume that can be mounted directly onto a running EC2 instance. The size of an EBS volume is user defined and limited to 1 TB. Although an EBS volume can only be attached to one instance at any time, an instance can conversely mount multiple EBS volumes. From the viewpoint of the EC2 instance, the EBS volume can be treated simply as a local filesystem.

6.2.3 Tradeoffs

We now offer a brief discussion on the tradeoffs of deploying our cache over the aforementioned Cloud resources.

In-Instance-Core Option: There are several advantages in supporting our cache over EC2 nodes in terms of flexibility and throughput. Depending on the application, it may be possible to store all cached data completely directly in memory, which reduces access time. But because small instances consist only 1.7 GB of memory, we may need to dynamically allocate more instances to cooperate in establishing a

larger capacity. On the other hand, we could also allocate an extra large instance with much more memory capacity. However, the instance could overfit our cache needs, which would betray cost-effectiveness. Because of these reasons, we would expect a memory-based cache to be the most expensive, but possibly with the highest performance, especially for an abundance smaller data units.

In-Instance-Disk Option: In cases where larger amounts of data are expected to be cached, we could store on the instance's disk. Even small EC2 instances provide ample disk space (160 GB), which would save us from having to allocate new instances very frequently for capacity, as we would expect in the previous option. However, disk accesses could be very slow compared to an in-memory cache if request rates are high. Conversely, if the average data size is large, disk access overheads may be amortized over time. We can expect that this disk-based option should be cheaper than the memory-based, with slightly lower performance depending on the average unit-data size.

Persistent Options: The previous two configurations do not account for persisting data. That is, upon node failure, all data is presumed lost even if stored on disk. Moreover, it can be useful to stop and restart a cache, perhaps during peak/non-peak times, to save usage costs.

The simplest persistent method is to directly utilize S3 to store cached data. This avoids any indexing logic from the application developer, as we can subscribe to S3's API. It is very inexpensive to store data on S3 and more importantly, because S3 is independent from EC2, we further elude instance allocation costs. However, due to S3's reliability and availability guarantees, it implements an architecture which supports replication and consistency, which would likely impact performance. Also, although storage costs are low, the data transfer costs are equivalent to those of EC2

instances, which leads to the expectation that high-throughput environments may not benefit cost-wise from S3.

Another persistent method are EBS volumes. One difference between EBS and S3 is that EBS volumes are less accessible. They must first be mounted onto an EC2 instance. But because they are mounted, it alludes to the potential for higher throughput than S3, whose communications is only enabled through high-level SOAP/REST protocols that ride over HTTP. Also in contrast to S3, EBS volumes are not unlimited in storage, and their size must be predefined by users. In terms of cost, however, EBS invokes a storage and request overhead to the hourly-based EC2 instance allocation costs. In the next section we evaluate these options in depth.

6.2.4 Experimental Results

We now discuss the evaluation of various cache and storage schemes over the aforementioned AWS Cloud resources

Experimental Setup

We have run experiments over the following configurations:

- `S3`: Data stored as files directly onto the S3 storage service (persistent).
- `ec2-m1.small-mem`: Data stored in memory on Small EC2 instance (volatile, moderate I/O).
- `ec2-m1.small-disk`: Data stored as files on disk on Small EC2 instance (volatile, moderate I/O).
- `ec2-m1.small-ebs`: Data stored as files on a mounted Elastic Block Store volume on small EC2 instance (persistent, moderate I/O).

- `ec2-m1.xlarge-mem`: Data stored in memory on Extra Large EC2 instance (volatile, high I/O).
- `ec2-m1.xlarge-disk`: Data stored as files on disk on Extra Large EC2 instance (volatile, high I/O).
- `ec2-m1.xlarge-ebs`: Data stored as files on a mounted Elastic Block Store volume on Extra Large EC2 instance (persistent, high I/O).

Within the EC2 instances, we evaluate three disparate ways to store the cache: in-core (`*-mem`), on local disk (`*-disk`), and on Amazon's Elastic Block Storage, or simply EBS (`*-ebs`). In both `m1.small` (32-bit) and `m1.xlarge` (64-bit) systems, we employ the Ubuntu Linux 9.10 Server image provided by Alestic.[6]

Application: As a representative workload, we performed repeated execution on a *Land Elevation Change* process, a real application provided to us by our colleagues in the Department of Civil and Environmental Engineering and Geodetic Science here at Ohio State University. This process inputs a triple, (L, t, t'), where location, L, denotes a coordinate and $t, t' : t < t'$ represent the times of interest. The service locates two Digital Elevation Model (DEM) files with respect to (L, t) and (L, t'). DEMs are represented by large matrices of an area, where each point denotes an elevation reading. Next, the process takes the difference between the two DEMs, which derives a DEM of the same size, where each point now represents the change in elevation from t to t'. We do stress that, while all our experiments were conducted with this process, by changing certain parameters, we can capture a variety of applications.

We have fixed the parameters of this process to take in 500 possible input keys, i.e., 500 distinct (L, t, t') triples, and our experiment queries randomly over this range.

[6] Alestic, http://alestic.com/

These input keys represent linearized coordinates and date. The queries are first sent to a coordinating compute node, and the underlying cooperating cache is then searched on the input key to find a replica of the precomputed results. Upon a hit, the results are transmitted directly back to the caller, whereas a miss would prompt the coordinator to invoke the process.

DEM Size	**Execution Time (sec)**
1 KB	2.09
1 MB	6.32
5 MB	20.52
50 MB	75.39

Table 6.3: Baseline Execution Time for Land Elevation Change Service

To analyze cache and storage performance, which can be affected by memory size, disk speed, network bandwidth, etc., we varied the sizes of DEM files: 1 KB, 1 MB, 5 MB, 50 MB. One such file is output from one execution, and over time, we would need to store a series of such files in our cache. These sizes allow for scenarios from cases where all cached data can fit into memory (e.g., 1 KB, 1 MB) to cases where in-core containment would be infeasible (e.g., 50 MB), coercing the need for disk or S3 storage. The larger data will also amortize network latency and overheads, which increases throughput. The baseline execution times of the service execution are summarized for these input sizes in Table 6.3.

Performance Evaluation

In this subsection, we measure performance in terms of relative speedup to the baseline execution times shown in Table 6.3, as well as the overall throughput of the various system configurations. We randomly submitted 2000 queries over the 500 possible (L, t, t') inputs to the *Land Elevation Change* process. The querying strategy is not unlike most cache-aware systems. Each query submission first checks our cache, and on a miss, the process is executed and its derived result is transferred and stored in the cache for future reuse. Upon a hit, the result is transmitted immediately to the user. To ensure consistency across all experiments, we do not consider cache eviction here, and all caches start cold.

Relative Speedup: Let us first consider the relative speedup for 1 KB, 1 MB, 5 MB, and 50 MB DEM files, shown in Figure 6.15, 6.16, 6.17, and 6.18 respectively.

In Figure 6.15, it is somewhat surprising that the same speedup is observed in the across all configurations. Because the data size is small, we believe that this is due to internal memory caching mechanisms within the machine instances. Also, the network transmission time for 1 KB data is so insignificant as to not favor either *moderate* or *high* I/O. `S3`'s performance appears to drop relative to the instance-based settings toward the end of the experiment. Due to `S3`'s persistence features, this was not completely unexpected, and we posit that larger files may amortize `S3` invocation overhead. This becomes clear when we evaluate the average hit times per configuration later in this section.

The results for 1 MB and 5 MB DEMs are shown in Figures 6.16 and 6.17 respectively, with one caveat: The `ec2-m1.small-mem` configuration cannot fit all the data completely in its memory in the case of 5 MB. Fortunately, our cache had been designed to handle such cases (recall from the previous section that our cache can

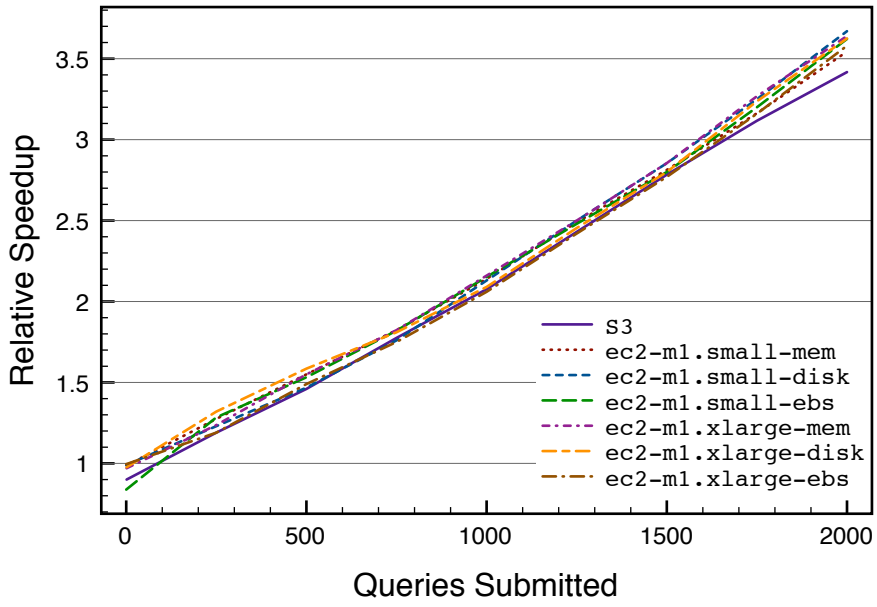


Figure 6.15: Relative Speedup: Unit-Data Size = 1 KB

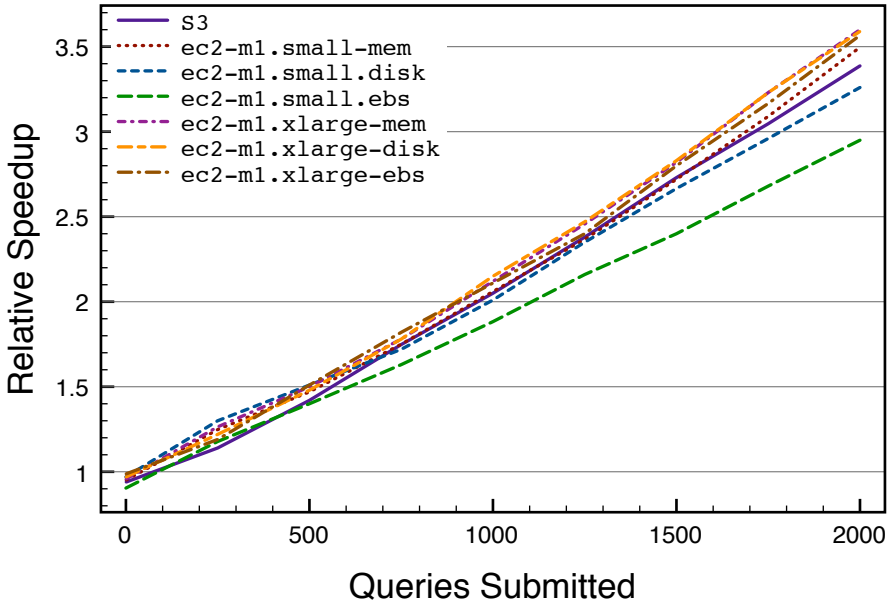


Figure 6.16: Relative Speedup: Unit-Data Size = 1 MB

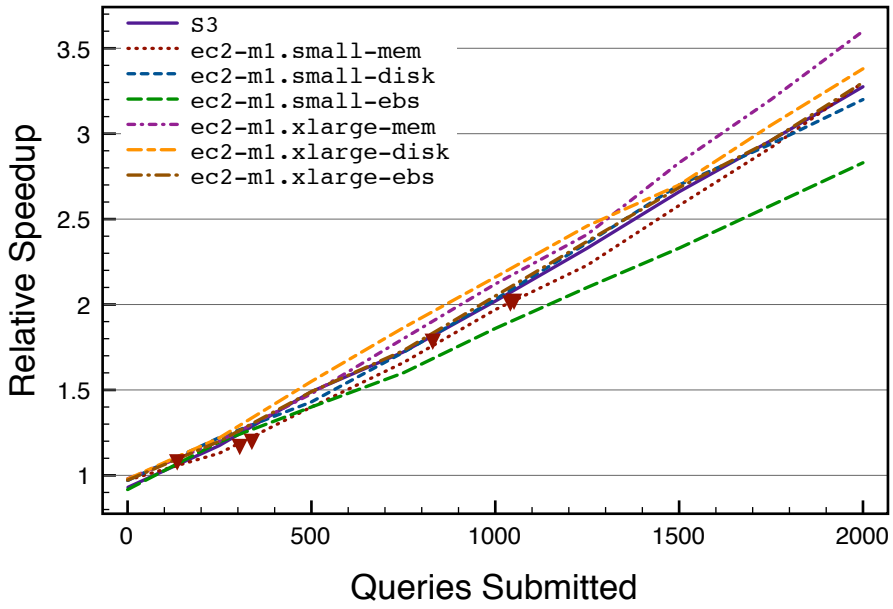


Figure 6.17: Relative Speedup: Unit-Data Size = 5 MB

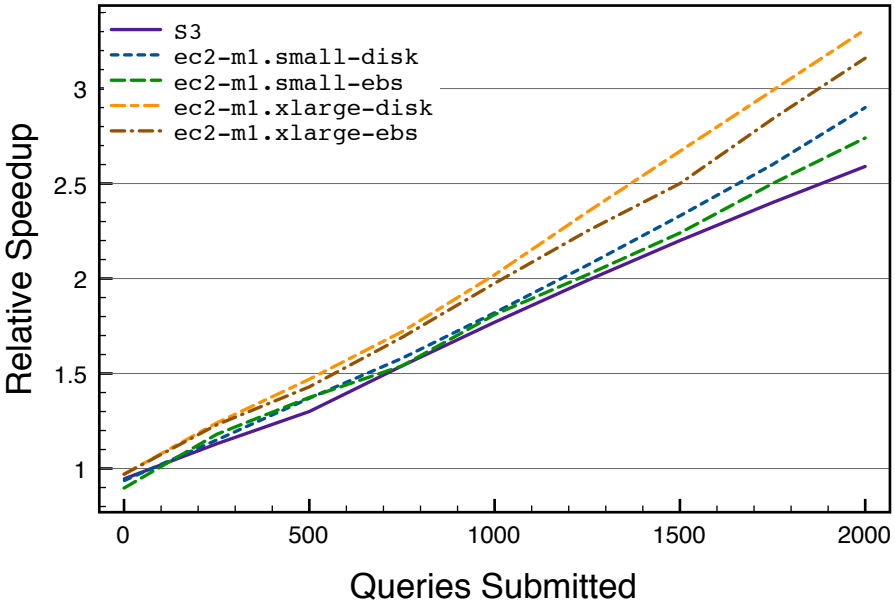


Figure 6.18: Relative Speedup: Unit-Data Size = 50 MB

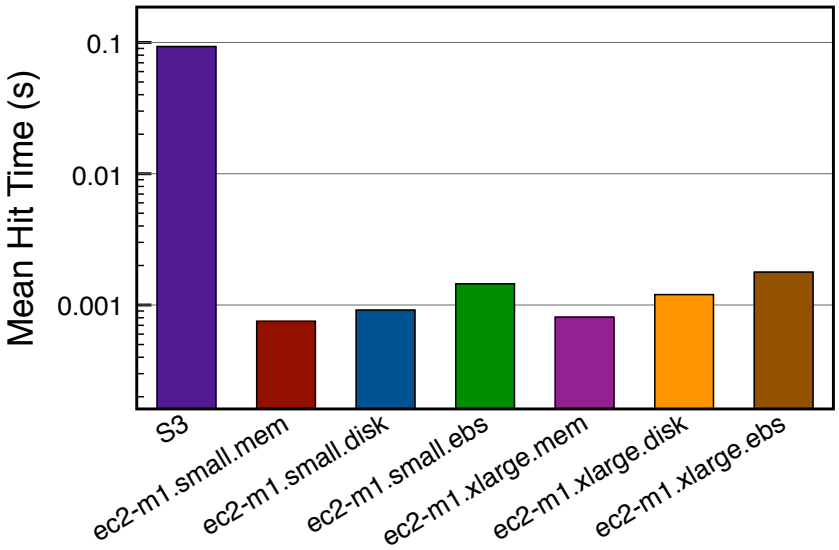


Figure 6.19: Mean Cache Hit + Retrieval Time: Unit-Data Size = 1 KB

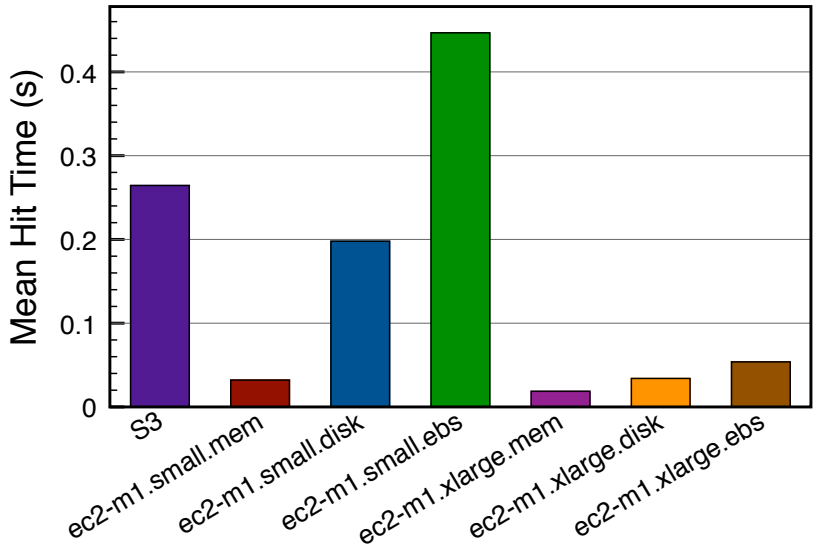


Figure 6.20: Mean Cache Hit + Retrieval Time: Unit-Data Size = 1 MB

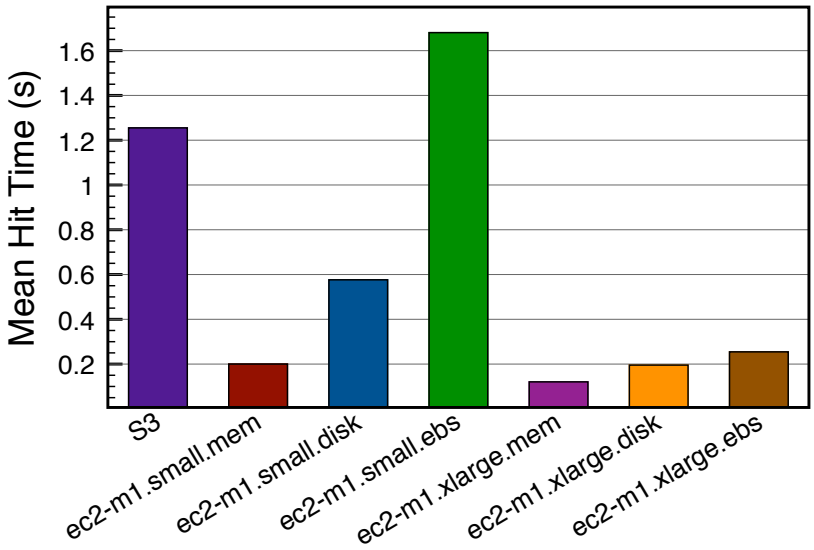


Figure 6.21: Mean Cache Hit + Retrieval Time: Unit-Data Size = 5 MB

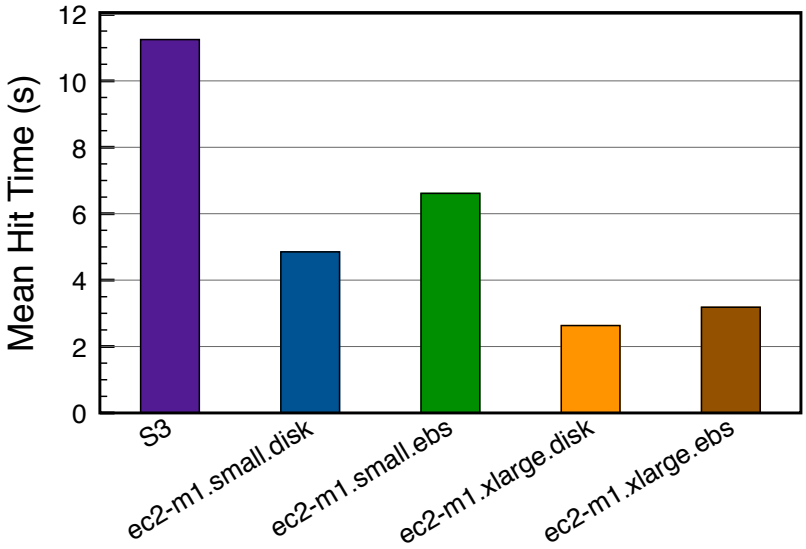


Figure 6.22: Mean Cache Hit + Retrieval Time: Unit-Data Size = 50 MB

add/remove cooperating nodes as needed. Every time a memory overflow is imminent, we allocate a new `m1.small` instance and migrate half the records from the overflown node to the new instance. In Figure 6.17, each instance allocation is marked as a triangle on `ec2-m1.small-mem`. Instance allocation is no doubt responsible for the performance slowdown.

In Figure 6.16, it becomes clear that in-memory containment is, in fact, beneficial for both small and extra large instance types. However, the *high* I/O afforded by the `m1.xlarge` instance marks the difference between the two memory-bound instances. This is justified by the fact that the persistent `ec2-m1.xlarge-ebs` eventually overcomes `ec2-m1.small-mem`. Also interesting is the performance degradation of the small disk-bound instances, `ec2-m1.small-disk` and `ec2-m1.small-ebs`, which performs comparably to `S3` during the first ~ 500 queries. Afterward, their performance decreases below `S3`. This is an interesting observation, considering that the first 500 queries are mostly cache misses (recall we start all caches out cold), which implies that the `m1.small` disk-based instance retrieves and writes the 1 MB files to disk faster than `S3`. However, when queries start hitting the cache more often after the first 500 queries, the dropoff in performance indicate that repeated random disk reads on the `m1.small` instances are generating significant overhead, especially in the case of the persistent `ec2-m1.small-ebs`.

Similar behavior can be observed for the 5 MB case, shown in Figure 6.17. The overhead of node allocation for `ec2-m1.small-mem` is solely responsible for its reduction in speedup. While the results are as expected, we do concede that our system's conservative instantiation of seven `m1.small` instances (that is, 1 to start + 6 over time) to hold a total of 500×5 MB of data in memory is indeed an overkill. Our instance allocation method was conservative here to protect against throttling, that

is, the possibility that an instance becomes overloaded and automatically stores on disk. Clearly, these cases would invalidate speedup calculations.

Finally, we experimented with 50 MB DEM data files. As a representative size for even larger sized data often seen in data analytic and scientific processes, we operated under the assumption that memory-bound caching would most likely be infeasible, and we experimented only with disk-bound settings. One interesting trend is the resurgence of `ec2-m1.small-disk` and `ec2-m1.small-ebs` over `S3`. One explanation may be that disk-bound caches favor larger files, as it would amortize random access latency. It may also be due to `S3`'s persistence guarantees – we noticed, on several occasions, that `S3` prompted for retransmissions of these larger files.

Cache Access Time: In all of the above experiments, the difference among speedups still seem rather trivial, albeit some separation can be seen toward the end of most experiments. We posit that, had the experiments run much longer (i.e., much more than only 2000 requests) the speedups will diverge greatly. To justify this, Figures 6.19, 6.20, 6.21, and 6.22 show the average *hit* times for each cache configuration. That is, we randomly submitted queries to full caches, which guarantees a hit on every query, and we are reporting the mean time in seconds to search the cache and retrieve the relevant file.

Here, the separation among the resource configuration becomes much clearer. Figure 6.19 shows that using `S3` for small files eventually exhibits slowdowns by 2 orders of magnitude. This fact eluded our observation previously in Figure 6.15 because the penalty caused by the earlier cache misses dominated the overall times. In the other figures, we again see justification for using memory-bound configurations, as they exhibit for the lowest mean hit times. Also, we observe consistent slowdowns for `ec2-m1.small-disk` and `ec2-m1.small-ebs` below `S3` in the 1 MB and 5 MB

cases. Finally, using the results from Figure 6.22, we can conclude that these results again support our belief that disk-bound configurations of the small instance types should be avoided for such mid-sized data files due to disk access latency. Similarly for larger files, `S3` should be avoided in favor of `ec2-m1.xlarge-ebs` if persistence is desirable. We have also ascertained from these experiments that the *high* I/O that is promised by the extra large instances contributes significantly to the performance of our cache.

Cost Evaluation

In this subsection, we present an analysis on cost for the instance configurations considered. The costs of the AWS features evaluated in our experiments are summarized in Table 6.2. While in-Cloud network I/O is currently free, in practice, we cannot assume that all users will be able to compute within the same Cloud. We thus assume that cache data is transferred outside of the AWS Cloud network in our analysis. We are repeating the settings from the previous set of experiments, so an average unit data size of 50 MB will yield a total cache size of 25 GB of Cloud storage (recall that there are 500 distinct request keys). We are furthermore assuming a fixed rate of $R = 2000$ requests per month from clients outside the Cloud. We have also extrapolated the costs (right side of table) for when request rate $R = 200000$, using the Mean Hit Times as the limits for such a large request rate R. Clearly, as R increases for a full cache, the speedup given by cache will eventually become denominated by the Mean Hit Times.

The cost, C, of maintaining our cache, the speedup S (after 2000 and 200000 requests), and the ratio C/S (i.e., the cost per unit-speedup), are reported under two requirements: volatile and persistent data stores. Again, volatile caches are less reliable in that, upon a node failure, all data is lost. The costs for sustaining a

volatile cache for one month is reported in Table 6.4. Here, the total cost can be computed as $C = (C_{Alloc} + C_{IO})$, where $C_{Alloc} = h \times k \times c_t$ denotes hours, h to allocate some k number of nodes using the said costs (from Table 6.2) for instance type t. $C_{IO} = R \times d \times c_{io}$ accounts for transfer costs, where R transfers were made per month, each involving d GB of data per transfer, multiplied by the cost to transfer per GB, c_{io}.

First, we recall that if the unit-data size, d, is very small (1 KB), we can obtain excellent performance for any volatile configuration. This is because everything easily fits in memory, and we speculate that, even for the disk-based options, the virtual instance is performing its own memory-based caching, which explains why performance is not lost. This is further supported by the speedup when $d = 1$ MB, under the disk-based option. When projected to $R = 200000$ requests, we observe lucrative speedups, which is not surprising, considering the fast access and retrieval times for such a small file. Furthermore, when $R = 2000$ requests, the `ec-m1.small-disk` option offers excellent C/S ratios, making it a very good option. Conversely, when request rate R is large, the I/O performance of the small instances accounts for too much of a slowdown, resulting in low speedups, and a low C/S ratio. This suggests that `m1.xlarge` is a better option for systems expecting higher throughput rates.

Next, we compiled the cost for persistent caches, supported by `S3` and `EBS` in Table 6.5. Here, C_S refers to the cost per 1 GB-month storage, C_R is the request cost, and C_{IO} refers to the data transfer costs per GB transferred out. Initially, we were surprised to see that `S3`'s C/S ratio is comparable to `EBS` (and even to the volatile options) when request rate R is low regardless of data size. However, for a large request rate, R, its overhead begins to slowdown its performance significantly compared to `EBS` options. Especially observed when unit-size, d, is very small, `S3`'s speedup simply pales in comparison to other options. Its performance expectedly

		2000 Requests			200000 Requests		
Unit-Size		S	$C = C_{Alloc} + C_{IO}$	C/S	S	C_{IO}	C/S
1 KB	`m1.small-mem`	3.54	$\$63.24 + \$0.00 = \$63.24$	\$17.84	2629.52	\$0.03	\$0.03
(500 KB total)	`m1.small-disk`	3.67	$\$63.24 + \$0.00 = \$63.24$	\$17.23	2147.681	\$0.03	\$0.03
	`m1.xlarge-mem`	3.64	$\$505.92 + \$0.00 = \$505.92$	\$138.99	2302.43	\$0.03	\$0.22
	`m1.xlarge-disk`	3.63	$\$505.92 + \$0.00 = \$505.92$	\$139.57	1823.215	\$0.03	\$0.28
1 MB	`m1.small-mem`	3.5	$\$63.24 + \$0.30 = \$63.54$	\$18.17	267.19	\$30.00	\$0.35
(500 MB total)	`m1.small-disk`	3.26	$\$63.24 + \$0.30 = \$63.54$	\$19.49	**28**	\$30.00	\$4.06
	`m1.xlarge-mem`	3.6	$\$505.92 + \$0.30 = \$506.22$	\$140.62	347.3	\$30.00	\$1.53
	`m1.xlarge-disk`	3.59	$\$505.92 + \$0.30 = \$506.22$	\$141.13	180.53	\$30.00	\$2.94
5 MB	`m1.small-mem`	3.3	$\$444.18 = \$442.68 + \$1.50$	\$96.27	109.47	\$150.00	\$4.26
(2.5 GB total)	`m1.small-disk`	3.2	$\$64.74 = \$63.24 + \$1.50$	\$20.24	33.84	\$150.00	\$6.30
	`m1.xlarge-mem`	3.6	$\$506.78 = \$505.92 + \$1.50$	\$140.78	174.42	\$150.00	\$3.76
	`m1.xlarge-disk`	3.38	$\$506.78 = \$505.92 + \$1.50$	\$149.94	111.71	\$150.00	\$5.87
50 MB	`m1.small-disk`	2.9	$\$63.24 + \$15.00 = \$78.74$	**\$27.16**	16.05	\$1500.00	**\$97.40**
(25 GB total)	`m1.xlarge-disk`	3.31	$\$505.92 + \$15.00 = \$520.92$	**\$152.85**	31.66	\$1500.00	**\$63.36**

Table 6.4: Monthly Volatile Cache Subsistence Costs

		2000 Requests			200000 Requests		
Unit-Size		S	$C_{S3} = C_S + C_R + C_{IO}$ $C_{EBS} = C_{Alloc} + C_S + C_R + C_{IO}$	C/S	S	C_{IO}	C/S
1 KB	`S3`	3.4	$\$0.0023 = \$0.00 + \$0.002 + \$0.0003$	\$0.0007	**24.79**	\$0.23	\$0.01
(500 KB total)	`m1.small-ebs`	3.62	$\$63.49 = \$63.24 + \$0.0002 + \$0.00 + \$0.0003$	\$17.54	1984.5	\$0.48	\$0.04
	`m1.xlarge-ebs`	3.58	$\$506.17 = \$505.92 + \$0.0002 + \$0.00 + \$0.0003$	\$141.39	2091.7	\$0.48	\$0.25
1 MB	`S3`	3.39	$\$0.38 = \$0.075 + \$0.002 + \$0.30$	\$0.12	29.98	\$30.01	\$1.01
(500 MB total)	`m1.small-ebs`	2.95	$\$63.59 = \$63.24 + \$0.05 + \$0.00 + \$0.30$	\$21.56	13.62	\$30.25	\$6.87
	`m1.xlarge-ebs`	3.57	$\$506.27 = \$505.92 + \$0.05 + \$0.00 + \$0.30$	\$142.00	133.96	\$30.25	\$4.01
5 MB	`S3`	3.27	$\$1.88 = \$0.375 + \$0.002 + \$1.50$	\$0.58	19.97	\$150.00	**\$7.53**
(2.5 GB total)	`m1.small-ebs`	2.83	$\$64.99 = \$63.24 + \$0.25 + \$0.00 + \$1.50$	\$22.97	11.84	\$150.27	\$18.04
	`m1.xlarge-ebs`	3.3	$\$507.67 = \$505.92 + \$0.25 + \$0.00 + \$1.50$	\$153.92	74.66	\$150.27	\$8.79
50 MB	`S3`	2.59	$\$18.75 = \$3.75 + \$0.002 + \$15.00$	**\$7.24**	6.43	\$1500.00	**\$233.87**
(25 GB total)	`m1.small-ebs`	2.74	$\$80.74 = \$63.24 + \$2.50 + \$0.00 + \$15.00$	**\$29.47**	11.09	\$1502.52	**\$142.70**
	`m1.xlarge-ebs`	3.16	$\$520.42 = \$505.92 + \$2.50 + \$0.00 + \$15.00$	**\$164.69**	22.66	\$1502.52	**\$88.63**

Table 6.5: Monthly Persistent Cache Subsistence Costs

increases as d becomes larger, due to the amortization of overheads when moving larger files. This performance gain of `S3`, however, drops sharply when $d = 50$ MB, resulting in only 6.43× speedup, making `EBS` better options in terms of cost per unit-speedup.

6.2.5 Discussion

The experiments demonstrate some interesting tradeoffs between cost and performance, the requirement for persistence, and the average unit-data size. We summarize these options below, given parameters d = average unit-data size, T = total cache size, and R cache requests per month.

For smaller data sizes, i.e., $d \leq 5$ MB, and small total cache sizes $T < 2$ GB, we posit that because of its affordability, `S3` offers the best cost tradeoff when R is small, even for supporting volatile caches. `m1.small.mem` and `m1.small.disk` also offer very good cost-performance regardless of the request rate, R. This is due to the fact that the entire cache can be stored in memory, together with the low cost of `m1.small` allocation. Even if the total cache size, T, is much larger than 2 GB, then depending on costs, it may still even make sense to allocate multiple small instances and still store everything in memory, rather than using one small instance's disk – we showed that, if request rate R is high, and the unit-size, d, is small, the speedup for `m1.small.disk` is eventually capped two orders of magnitude below the memory-bound option. If $d \geq 50$ MB, we believe it would be wise to consider `m1.xlarge`. While it could still make sense to use a single small instance's disk if R is low, we observed that performance is lost quickly as R increases, due to `m1.small`'s lower-end I/O.

If data persistence is necessary, `S3` is by far the most cost-effective option in most cases. However, it also comes at the cost of lower throughput, and thus `S3`

would be viable for systems with less expectations for high amounts of requests. The cost analysis also showed that storage costs are almost negligible for `S3` and `EBS` if request rates are high. If performance is an issue, it would be prudent to consider `m1.small-ebs` and `m1.xlarge-ebs` for smaller and larger unit-data sizes respectively, regardless of the total cache size. Of course, if cost is not an a pressing issue, `m1.xlarge` with or without EBS persistence should be used achieve the highest performance.

CHAPTER 7

CONCLUSION

In each of the following sections in this chapter, we discuss a summary of lessons-learned, concessions on limitations, as well as future outlook on our current set of contributions.

7.1 Enabling High-Level Queries with Auspice

Auspice is a system which supports simplified querying over low-level scientific datasets. This process is enabled through a combination of effective indexing over metadata information, a system and domain specific ontology, and a workflow planning algorithm capable of alleviating all tiers of users of the difficulties one may experience through dealing with the complexities of scientific data. Our system presents a new direction for users, from novice to expert, to share data sets and services. The metadata, which comes coupled with scientific data sets, is indexed by our system and exploited to automatically compose workflows in answering high level queries without the need for common users to understand complex domain semantics.

As evidenced by our experiments, a case can be made for supporting metadata registration and indexing in an automatic workflow management system. In our case study alone, comparing the overhead of workflow planning between linear search and index-based data identification methods, speedups are easily observed even for small

numbers of data sets. Further, on the medium scale of searching through 1×10^6 data sets, it clearly becomes counterproductive to rely on linear metadata search methods, as it potentially takes longer to plan workflows than to execute them. As evidenced, this scalability issue is easily mitigated with an indexed approach, whose planning time remains negligible for the evaluated sizes of data sets.

Although our system strives to support keyword queries, it is, admittedly, far from complete. For instance, despite metadata registration, ontology building, a process required for sustaining keyword queries and workflow planning, is a human cost. Moreover, our keyword queries currently only support ANDs, and we believe that, while it may not be very challenging to support other operators, they are certainly limitations to our interface.

We also alluded to including the quality of workflow as a factor in relevance calculations. In Chapter 4, we discussed a framework for estimating a workflow's execution time costs and its accuracy as a way for supporting QoS. Extensions to this work to improve our relevance metric by incorporating our cost models could be beneficial to users.

Furthermore, we are also planning to explore Deep Web integration with our system, as scientific data can often be found in backend data repositories. Exploring the integration of scientific Deep Web data sources into the Auspice querying and workflow planning framework add even more technical depth to the system. That is, just as Auspice is currently able to automatically compose data files with services, our goal would be to include the scientific Deep Web in this framework. To the best of our knowledge, this would be the first experience on seamlessly integrating the scientific Deep Web into a service workflow system.

7.1.1 QoS Aware Workflow Planning

The work reported herein discusses our approach to bring QoS awareness in the form of time and accuracy constraints to the process of workflow composition. Our framework, which allows users to express error and execution time prediction models, employs the a priori principle to prune potential workflow candidates. Our results show that the inclusion of such cost models contributes negligible overhead, and in fact, can reduce the overall workflow enumeration time through pruning unlikely candidates at an early stage. In addition, our dynamic accuracy parameter adjustment offers robustness by allowing workflows to be flexibly accurate for meeting QoS constraints under varying network speeds.

Auspice was evaluated against actual user constraints on time and the network bandwidth limitations. In its worst case, it maintained actual execution times that deviate no more than 14.3% from the expected values on average, and no worse than 12.4% from the ideal line when presented with varying network bandwidths. The evaluation also shows that, overall, the inclusion of such cost models contributes insignificantly to the overall execution time of our workflow composition algorithm, and in fact, can reduce its overall time through pruning unlikely candidates at an early stage. We also showed that our adaptive accuracy parameter adjustment is effective for suggesting relevant values for dynamically reducing the size of data.

As we seek to further our development of Auspice's execution engine, we are aware of features that have not yet been investigated or implemented. Computing QoS costs in the planning phase may facilitate workflow pruning well, but might not always be desirable. For instance, a user, who initially provisioned a maximum time of an hour for the task to finish, may change her mind halfway through the computation. Similarly, faster resources may become available during execution. To handle these issues, a dynamic rescheduling mechanism needs to be established.

This alludes to the area of scheduling on distributed heterogeneous resources. The problem, which is inherently NP-Hard, has received much recent attention. This problem is compounded by the novel dimension of cost in Cloud computing paradigms. We plan to investigate the support for these aspects and develop new heuristics for enabling an efficient and robust scheduler. Moreover, as Clouds allow for on-demand resource provisioning, compelling problems arise for optimizing scheduling costs while taking into account other QoS constraints such as time and accuracy.

7.2 Caching Intermediate Data

We have integrated Auspice with a hierarchical spatiotemporal indexing scheme for capturing preexisting virtual data. To maintain manageable indices and cache sizes, we set forth a bilateral distributed victimization scheme. To support a robust spatiotemporal index, a domain knowledge-aware version of the B^x-Tree was implemented. Our experimental results show that, for two frequently submitted geospatial queries, the overall execution time improved by a factor of over 3.5. The results also suggested that significant speedup could be achieved over low to medium bandwidth environments. Lastly, we showed that our indexing scheme's search time and index size can scale to the grid.

As the scientific community continues to push for enabling mechanisms that support compute and data intensive applications, grid workflow systems will experience no shortage of new approaches towards optimization. We are currently investigating a generalized version of our hierarchy. In large and fast networks, it may be worth maintaining multiple broker levels with increasingly granular regions before reaching the cohorts level. In this framework, as nodes are added or removed, a evolution of broker splits and cohort promotions will involve a detailed study of the effects of index partitioning and restructuring.

7.3 Caching and Storage Issues in the Cloud

Cloud providers have begun offering users at-cost access to on demand computing infrastructures. In this paper, we propose a Cloud-based cooperative cache system for reducing execution times of data-intensive processes. The resource allocation algorithm presented herein are cost-conscious as not to over-provision Cloud resources. We have evaluated our system extensively, showing that, among other things, our system is scalable to varying high workloads, cheaper than utilizing fixed networking structures on the Cloud, and effective for reducing service execution times.

A costly overhead is the node allocation process itself. Strategies, such as preloading and data replication can certainly be used to implement an *asynchronous* node allocation. Works on instantaneous virtual machine boots [42, 105] have also been proposed and can be considered here. However, with the current reliance on commercial-grade Clouds, we should seek unintrusive schemes. Modifications to current approaches, like the Falkon Framework [137], where ad hoc resource pools are preemptively allocated from remote sites, may be also employed here. Record prefetching from a node that is predictably close to invoking migration can also be considered to reduce migration cost. As discussed in Chapter 6, although our sliding window size for eviction is a parameter to the system, there may be merit in managing this value dynamically to reduce unnecessary (or less cost-effective) node allocation. Predictive eviction methods could be well worth considering.

Another major issue we discussed involved the cost that comes coupled of utilizing several mixed options to support a cache. Depending on application parameters and needs, we have shown that certain scenarios call for different Cloud resources. In the future, we hope to use our study to initiate the development of finer-grained cost models and automatic configuration of such caches given user parameters. We will

also develop systematic approaches, including hybrid cache configurations to optimize cost-performance tradeoffs. For example, we could store highly critical/popular cached data in EC2 nodes while evicting records into S3. Interesting challenges include, when to evict from EC2? From S3? How can we avoid the network delays of bringing data from S3 back into EC2 with preemptive scheduling?

We could also fuse these ideas with other aspects of our system. For instance, we could exploit the cheap costs of Amazon's S3 persistent store to hold the most accurate, precomputed data sets. Then, depending on the prospect of users' QoS requirements, we retrieve and compress the data as needed. This will cause us to rethink issues on data accuracy, versus time, versus cost.

BIBLIOGRAPHY

[1] Ian F. Adams, Darrell D.E. Long, Ethan L. Miller, Shankar Pasupathy, and Mark W. Storer. Maximizing efficiency by trading storage for computation. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[2] Ali Afzal, John Darlington, and Andrew Stephen McGough. Qos-constrained stochastic workflow scheduling in enterprise and scientific grids. In *GRID*, pages 1–8, 2006.

[3] Sanjay Agrawal. Dbxplorer: A system for keyword-based search over relational databases. In *In ICDE*, pages 5–16, 2002.

[4] Nadine Alameh. Chaining geographic information web services. *IEEE Internet Computing*, 07(5):22–29, 2003.

[5] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows, 2004.

[6] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. *Provenance and Annotation of Data*, pages 118–132, 2006.

[7] Fatih Altiparmak, David Chiu, and Hakan Ferhatosmanoglu. Incremental quantization for aging data streams. In *ICDM Workshops*, pages 527–532, 2007.

[8] Fatih Altiparmak, Ertem Tuncel, and Hakan Ferhatosmanoglu. Incremental maintenance of online summaries over multiple streams. *IEEE Trans. Knowl. Data Eng.*, 20(2):216–229, 2008.

[9] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Active semantic caching to optimize multidimensional data analysis in parallel and distributed environments. *Parallel Comput.*, 33(7-8):497–520, 2007.

[11] ANZLIC. Anzmeta xml document type definition (dtd) for geospatial metadata in australasia, 2001.

[12] Michael Armbrust*, et al.* Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[13] The atlas experiment, http://atlasexperiment.org.

[14] Amazon web services, http://aws.amazon.com.

[15] Amazon elastic mapreduce, http://aws.amazon.com/elasticmapreduce/.

[16] Microsoft azure services platform, http://www.microsoft.com/azure.

[17] Jon Bakken, Eileen Berman, Chih-Hao Huang, Alexander Moibenko, Donald Petravick, and Michael Zalokar. The fermilab data storage infrastructure. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 101, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Adam Barker, Jon B. Weissman, and Jano I. van Hemert. The circulate architecture: Avoiding workflow bottlenecks caused by centralised orchestration. *Cluster Computing*, 12(2):221–235, 2009.

[19] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1998.

[20] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[21] Boualem Benatallah, Marlon Dumas, Quan Z. Sheng, and Anne H.H. Ngu. Declarative composition and peer-to-peer provisioning of dynamic web services. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, Washington, DC, USA, 2002. IEEE Computer Society.

[22] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[23] Wes Bethel, Brian Tierney, Jason lee, Dan Gunter, and Stephen Lau. Using high-speed wans and network data caches to enable remote and distributed visualization. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Dallas, TX, USA, 2000.

[24] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with datacutter. *Parallel Computing*, 27(11):1457–1478, Novembro 2001.

[25] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[26] Microsoft biztalk server, http://www.microsoft.com/biztalk.

[27] Jim Blythe, Ewa Deelman, Yolanda Gil, Carl Kesselman, Amit Agarwal, Gaurang Mehta, and Karan Vahi. The role of planning in grid computing. In *The 13th International Conference on Automated Planning and Scheduling (ICAPS)*, Trento, Italy, 2003. AAAI.

[28] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[29] Ivona Brandic, Siegfried Benkner, Gerhard Engelbrecht, and Rainer Schmidt. Qos support for time-critical grid workflow applications. *E-Science*, 0:108–115, 2005.

[30] Ivona Brandic, Sabri Pllana, and Siegfried Benkner. An approach for the high-level specification of qos-aware grid workflows considering location affinity. *Sci. Program.*, 14(3,4):231–250, 2006.

[31] Ivona Brandic, Sabri Pllana, and Siegfried Benkner. Specification, planning, and execution of qos-aware grid workflows within the amadeus environment. *Concurr. Comput. : Pract. Exper.*, 20(4):331–345, 2008.

[32] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[33] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous concurrent modeling and design in java (volume 2: Ptolemy ii software architecture). Technical Report 22, EECS Dept., UC Berkeley, July 2005.

[34] Yonny Cardenas, Jean-Marc Pierson, and Lionel Brunie. Uniform distributed cache service for grid computing. *International Workshop on Database and Expert Systems Applications*, 0:351–355, 2005.

[35] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eFlow. In *Conference on Advanced Information Systems Engineering*, pages 13–31, 2000.

[36] U. Çetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, and S. Zdonik. The Aurora and Borealis Stream Processing Engines. In M. Garofalakis, J. Gehrke, and R. Rastogi, editors, *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2007.

[37] Liang Chen, Kolagatla Reddy, and Gagan Agrawal. Gates: A grid-based middleware for processing distributed data streams. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 192–201, Washington, DC, USA, 2004. IEEE Computer Society.

[38] David Chiu and Gagan Agrawal. Hierarchical caches for grid workflows. In *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE, 2009.

[39] David Chiu, Sagar Deshpande, Gagan Agrawal, and Rongxing Li. Composing geoinformatics workflows with user preferences. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'08)*, New York, NY, USA, 2008.

[40] David Chiu, Sagar Deshpande, Gagan Agrawal, and Rongxing Li. Cost and accuracy sensitive dynamic workflow composition over grid environments. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid'08)*, 2008.

[41] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1.

[42] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[43] Condor dagman, http://www.cs.wisc.edu/condor/dagman.

[44] Shaul Dar, Gadi Entin, Shai Geva, and Eran Palmon. Dtl's dataspot: Database exploration using plain language. In *In Proceedings of the Twenty-Fourth International Conference on Very Large Data Bases*, pages 645–649. Morgan Kaufmann, 1998.

[45] Shaul Dar, Michael J. Franklin, Bjorn Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[46] Dublin core metadata element set, version 1.1, 2008.

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[48] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[49] Mike Dean and Guus Schreiber. Owl web ontology language reference. w3c recommendation, 2004.

[50] Ewa Deelman and Ann Chervenak. Data management challenges of data-intensive scientific workflows. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 687–692, Washington, DC, USA, 2008. IEEE.

[51] Ewa Deelman, Gurmeet Singh, Miron Livny, J. Bruce Berriman, and John Good. The cost of doing science on the cloud: the montage example. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*. IEEE/ACM, 2008.

[52] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia C. Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[53] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query language for xml. *Computer Networks*, 31(11-16):1155–1169, 1999.

[54] Liping Di, Peng Yue, Wenli Yang, Genong Yu, Peisheng Zhao, and Yaxing Wei. Ontology-supported automatic service chaining for geospatial knowledge discovery. In *Proceedings of American Society of Photogrammetry and Remote Sensing*, 2007.

[55] Liping Di, Peng Yue, Wenli Yang, Genong Yu, Peisheng Zhao, and Yaxing Wei. Ontology-supported automatic service chaining for geospatial knowledge discovery. In *Proceedings of American Society of Photogrammetry and Remote Sensing*, 2007.

[56] Flavia Donno and Maarten Litmaath. Data management in wlcg and egee. worldwide lhc computing grid. Technical Report CERN-IT-Note-2008-002, CERN, Geneva, Feb 2008.

[57] Prashant Doshi, Richard Goodwin, Rama Akkiraju, and Kunal Verma. Dynamic workflow composition using markov decision processes. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 576, Washington, DC, USA, 2004. IEEE Computer Society.

[58] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[59] David Martin (ed.). Owl-s: Semantic markup for web services. w3c submission, 2004.

[60] Johann Eder, Euthimios Panagos, and Michael Rabinovich. Time constraints in workflow systems. *Lecture Notes in Computer Science*, 1626:286, 1999.

[61] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. The MIT Press, 1998.

[62] Lin Guo Feng, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *In SIGMOD*, pages 16–27, 2003.

[63] Metadata ad hoc working group. content standard for digital geospatial metadata, 1998.

[64] Federal geospatial data clearinghouse, http://clearinghouse.fgdc.gov.

[65] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into xml query processing. In *BDA*, 2000.

[66] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[67] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pages 2–13, 2005.

[68] Ian Foster. Service-oriented science. *Science*, 308(5723):814–817, May 2005.

[69] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[70] Ian T. Foster, Jens S. Vockler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 37–46, Washington, DC, USA, 2002. IEEE Computer Society.

[71] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 7–9, San Francisco, California, August 2001.

[72] Keita Fujii and Tatsuya Suda. Semantics-based dynamic service composition. *IEEE Journal on Selected Areas in Communications (JSAC)*, 23(12), 2005.

[73] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: An approach to building large internet caches. *Workshop on Hot Topics in Operating Systems*, 0:93, 1997.

[74] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and Accessing Heterogenous Information Sources in TSIMMIS. In *Proceedings of the AAAI Symposium on Information Gathering*, 1995.

[75] gbio: Grid for bioinformatics, http://gbio-pbil.ibcp.fr.

[76] Bioinfogrid, http://www.bioinfogrid.eu.

[77] Biomedical informatics research network, http://www.nbirn.net.

[78] Roxana Geambasu, Steven D. Gribble, and Henry M. Levy. Cloudviews: Communal data sharing in public clouds. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[79] Cyberstructure for the geosciences, http://www.geongrid.org.

[80] The geography network, http://www.geographynetwork.com.

[81] Yolanda Gil, Ewa Deelman, Jim Blythe, Carl Kesselman, and Hongsuda Tangmunarunkit. Artificial intelligence and grids: Workflow planning and beyond. *IEEE Intelligent Systems*, 19(1):26–33, 2004.

[82] Yolanda Gil, Varun Ratnakar, Ewa Deelman, Gaurang Mehta, and Jihie Kim. Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. In Proceedings of the 19th Annual Conference on Innovative Applications of Artificial Intelligence (IAAI), Vancouver, British Columbia, Canada, July 22-26,, 2007.

[83] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployment of data-intensive applications on grids with moteur. *Int. J. High Perform. Comput. Appl.*, 22(3):347–360, 2008.

[84] Leonid Glimcher and Gagan Agrawal. A middleware for developing and deploying scalable remote mining services. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 242–249, Washington, DC, USA, 2008. IEEE Computer Society.

[85] Google app engine, http://code.google.com/appengine.

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

[87] Gobe Hobona, David Fairbairn, and Philip James. Semantically-assisted geospatial workflow design. In *GIS '07: Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[88] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good. On the use of cloud computing for scientific workflows. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 640–645. IEEE Computer Society, 2008.

[89] Christina Hoffa, Gaurang Mehta, Tim Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the use of cloud computing for scientific workflows. *Fourth IEEE International Conference on eScience*, pages 640–645, 2008.

[90] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[91] Yu Hua, Yifeng Zhu, Hong Jiang, Dan Feng, and Lei Tian. Scalable and adaptive metadata management in ultra large-scale file systems. *Distributed Computing Systems, International Conference on*, 0:403–410, 2008.

[92] Richard Huang, Henri Casanova, and Andrew A. Chien. Automatic resource specification generation for resource selection. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[93] Yannis E. Ioannidis, Miron Livny, S. Gupta, and Nagavamsi Ponnekanti. Zoo: A desktop experiment management environment. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 274–285, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[94] Arun Jagatheesan, Reagan Moore, Arcot Rajasekar, and Bing Zhu. Virtual services in data grids. *International Symposium on High-Performance Distributed Computing*, 0:420, 2002.

[95] Christian S. Jensen. Towards increasingly update efficient moving-object indexing. *IEEE Data Eng. Bull*, 25:200–2, 2002.

[96] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient b+tree-based indexing of moving objects. In *Proceedings of Very Large Databases (VLDB)*, pages 768–779, 2004.

[97] Song Jiang and Xiaodong Zhang. Efficient distributed disk caching in data grid management. *IEEE International Conference on Cluster Computing (CLUSTER)*, 2003.

[98] Gideon Juve and Ewa Deelman. Resource provisioning options for large-scale scientific workflows. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 608–613, Washington, DC, USA, 2008. IEEE Computer Society.

[99] David Karger*, et al.* Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[100] David Karger*, et al.* Web caching with consistent hashing. In *WWW'99: Proceedings of the 8th International Conference on the World Wide Web*, pages 1203–1213, 1999.

[101] Jihie Kim, Ewa Deelman, Yolanda Gil, Gaurang Mehta, and Varun Ratnakar. Provenance trails in the wings-pegasus system. *Concurr. Comput. : Pract. Exper.*, 20(5):587–597, 2008.

[102] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Meeting of the Association for Computational Linguistics*, pages 423–430, 2003.

[103] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Cappello, and David P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[104] Vijay S. Kumar, P. Sadayappan, Gaurang Mehta, Karan Vahi, Ewa Deelman, Varun Ratnakar, Jihie Kim, Yolanda Gil, Mary Hall, Tahsin Kurc, and Joel Saltz. An integrated framework for performance-based optimization of scientific workflows. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 177–186, New York, NY, USA, 2009. ACM.

[105] H. Andres Lagar-Cavilla, Joseph Whitney, Adin Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *3rd European Conference on Computer Systems (Eurosys)*, Nuremberg, Germany, April 2009.

[106] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. *Lecture Notes in Computer Science*, 1832, 2000.

[107] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, 2001.

[108] Rob Lemmens, Andreas Wytzisk, Rolf de By, Carlos Granell, Michael Gould, and Peter van Oosterom. Integrating semantic and syntactic descriptions to chain geographic services. *IEEE Internet Computing*, 10(5):42–52, 2006.

[109] Isaac Lera, Carlos Juiz, and Ramon Puigjaner. Performance-related ontologies and semantic web applications for on-line performance assessment intelligent systems. *Sci. Comput. Program.*, 61(1):27–37, 2006.

[110] The atlas experiment, http://atlas.ch/.

[111] Worldwide lhc computing grid (wlcg), http://lcg.web.cern.ch/lcg/.

[112] Lhc computing grid, http://public.web.cern.ch/public/en/lhc/computing-en.html.

[113] Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 331–339, New York, NY, USA, 2003. ACM Press.

[114] Jie Li, *et al.* escience in the cloud: A modis satellite data reprojection and reduction pipeline in the windows azure platform. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.

[115] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[116] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 563–574, New York, NY, USA, 2006. ACM.

[117] Large synoptic survey telescope, http://www.lsst.org.

[118] Wenjing Ma and Gagan Agrawal. A translation system for enabling data mining applications on gpus. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 400–409, New York, NY, USA, 2009. ACM.

[119] Shalil Majithia, Matthew S. Shields, Ian J. Taylor, and Ian Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 514–524. IEEE Computer Society, 2004.

[120] Frank Manola and Eric Miller. Resource description framework (rdf) primer. w3c recommendation, 2004.

[121] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing web services on the semantic web. *The VLDB Journal*, 12(4):333–351, 2003.

[122] D. Mennie and B. Pagurek. An architecture to support dynamic composition of service components. In *Proceedings of the 5th International Workshop on Component -Oriented Programming*, 2000.

[123] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13:124–141, 2001.

[124] Peter Muth, Dirk Wodtke, Jeanine Weissenfels, Angelika Kotz Dittrich, and Gerhard Weikum. From centralized workflow specification to distributed workflowexecution. *Journal of Intelligent Information Systems*, 10(2):159–184, 1998.

[125] Biological data working group. biological data profile, 1999.

[126] Jaechun No, Rajeev Thakur, and Alok Choudhary. Integrating parallel file i/o and database support for high-performance scientific data management. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 57, Washington, DC, USA, 2000. IEEE Computer Society.

[127] Open geospatial consortium, http://www.opengeospatial.org.

[128] Seog-Chan Oh, Dongwon Lee, and Soundar R. T. Kumara. A comparative illustration of ai planning-based web services composition. *SIGecom Exch.*, 5(5):1–10, 2006.

[129] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[130] Open science grid, http://www.opensciencegrid.org/.

[131] Ekow J. Otoo, Doron Rotem, Alexandru Romosan, and Sridhar Seshadri. File caching in data intensive scientific applications on data-grids. In *First VLDB Workshop on Data Management in Grids*. Springer, 2005.

[132] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64, New York, NY, USA, 2008. ACM.

[133] Shankar R. Ponnekanti and Armando Fox. Sword: A developer toolkit for web service composition. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, 2002.

[134] Jun Qin and Thomas Fahringer. A novel domain oriented approach for scientific grid workflow composition. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[135] Michael Rabinovich and Oliver Spatschek. *Web caching and replication*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[136] Prabhakar Raghavan. Structured and unstructured search in enterprises. *IEEE Data Eng. Bull.*, 24(4):15–18, 2001.

[137] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: a fast and light-weight task execution framework. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[138] Rajesh Raman, Miron Livny, and Marv Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2(2):129–138, 1999.

[139] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *SWSWPC*, pages 43–54, 2004.

[140] Rob Raskin and Michael Pan. Knowledge representation in the semantic web for earth and environmental terminology (sweet). *Computer and Geosciences*, 31(9):1119–1125, 2005.

[141] Daniel A. Reed. Grids, the teragrid, and beyond. *Computer*, 36(1):62–68, 2003.

[142] Qun Ren, Margaret H. Dunham, and Vijay Kumar. Semantic caching and query processing. *IEEE Trans. on Knowl. and Data Eng.*, 15(1):192–210, 2003.

[143] H. Samet. The quadtree and related hierarchical structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[144] Mayssam Sayyadian, Hieu LeKhac, AnHai Doan, and Luis Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE*, pages 346–355, 2007.

[145] Sloan digital sky survey, http://www.sdss.org.

[146] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[147] Srinath Shankar, Ameet Kini, David J. DeWitt, and Jeffrey Naughton. Integrating databases and workflow systems. *SIGMOD Rec.*, 34(3):5–11, 2005.

[148] Q. Sheng, B. Benatallah, M. Dumas, and E. Mak. Self-serv: A platform for rapid composition of web services in a peer-to-peer environment. In *Demo Session of the 28th Intl. Conf. on Very Large Databases*, 2002.

[149] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[150] L. Shklar, A. Sheth, V. Kashyap, and K. Shah. InfoHarness: Use of Automatically Generated Metadata for Search and Retrieval of Heterogeneous Information. In *Proceedings of CAiSE*, 1995.

[151] Yogesh Simmhan, Roger Barga, Catharine van Ingen, Ed Lazowska, and Alex Szalay. Building the trident scientific workflow workbench for data management in the cloud. *Advanced Engineering Computing and Applications in Sciences, International Conference on*, 0:41–50, 2009.

[152] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. Karma2: Provenance management for data-driven workflows. *International Journal of Web Service Research*, 5(2):1–22, 2008.

[153] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 117–126, New York, NY, USA, 2007. ACM.

[154] Gurmeet Singh*, et al.* A metadata catalog service for data intensive applications. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 33, Washington, DC, USA, 2003. IEEE Computer Society.

[155] Evren Sirin, Bijan Parsia, and James Hendler. Filtering and selecting semantic web services with interactive composition techniques. *IEEE Intelligent Systems*, 19(4):42–49, 2004.

[156] Warren Smith, Ian Foster, and Valerie Taylor. Scheduling with advanced reservations. In *In Proceedings of IPDPS 2000*, pages 127–132, 2000.

[157] Soap version 1.2 part 1: Messaging framework (second edition), w3c recommendation 27 april 2007, http://www.w3.org/tr/soap12-part1.

[158] Borja Sotomayor, Kate Keahey, and Ian Foster. Combining batch execution and leasing using virtual machines. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 87–96, New York, NY, USA, 2008. ACM.

[159] Sparql query language for rdf, w3c recommendation, 15 january, 2008. http://www.w3.org/TR/rdf-sparql-query.

[160] Heinz Stockinger, Asad Samar, Koen Holtman, Bill Allcock, Ian Foster, and Brian Tierney. File and object replication in data grids. *10th International Symposium on High Performance Distributed Computing (HPDC 2001)*, 2001.

[161] Michael Stonebraker, Jacek Becla, David Dewitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stan Zdonik. Requirements for science data bases and scidb. In *Conference on Innovative Data Systems Research (CIDR)*, January 2009.

[162] Qi Su and Jennifer Widom. Indexing relational database content offline for efficient keyword-based search. In *IDEAS '05: Proceedings of the 9th International Database Engineering & Application Symposium*, pages 297–306, Washington, DC, USA, 2005. IEEE Computer Society.

[163] Biplav Svrivastava and Jana Koehler. Planning with workflows - an emerging paradigm for web service composition. In *Workshop on Planning and Scheduling for Web and Grid Services*. ICAPS, 2004.

[164] Ian Taylor, Andrew Harrison, Carlo Mastroianni, and Matthew Shields. Cache for workflows. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 13–20, New York, NY, USA, 2007. ACM.

[165] D.G. Thaler and C.V. Ravishankar. Using name-based mappings to increase hit rates. *Networking, IEEE/ACM Transactions on*, 6(1):1–14, Feb 1998.

[166] B. Tierney*, et al.* Distributed parallel data storage systems: a scalable approach to high speed image servers. In *MULTIMEDIA '94: Proceedings of the second ACM international conference on Multimedia*, pages 399–405, New York, NY, USA, 1994. ACM.

[167] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *3rd International Semantic Web Conference*, 2004.

[168] Rattapoom Tuchinda, Snehal Thakkar, A Gil, and Ewa Deelman. Artemis: Integrating scientific data on the grid. In *Proceedings of the 16th Conference on Innovative Applications of Artificial Intelligence (IAAI)*, pages 25–29, 2004.

[169] Universal description discovery and integration, http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm.

[170] Vagelis Hristidis University and Vagelis Hristidis. Discover: Keyword search in relational databases. In *In VLDB*, pages 670–681, 2002.

[171] S.S. Vazhkudai, D. Thain, Xiaosong Ma, and V.W. Freeh. Positioning dynamic storage caches for transient data. *IEEE International Conference on Cluster Computing*, pages 1–9, 2006.

[172] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyya. High-performance cloud computing: A view of scientific applications. *Parallel Architectures, Algorithms, and Networks, International Symposium on*, 0:4–16, 2009.

[173] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, volume 39, pages 148–162, New York, NY, USA, December 2005. ACM Press.

[174] Jon Weissman and Siddharth Ramakrishnan. Using proxies to accelerate cloud applications. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[175] D. Wessels and K. Claffy. Internet cache protocol (icp), version 2, 1997.

[176] Wolfram Wiesemann, Ronald Hochreiter, and Daniel Kuhn. A stochastic programming approach for qos-aware service composition. *The 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'08)*, pages 226–233, May 2008.

[177] Web services business process execution language (wsbpel) 2.0, oasis standard.

[178] D. Wu, E. Sirin, J. Hendler, D. Nau, and B. Parsia. Automatic web services composition using shop2. In *ICAPS'03: International Conference on Automated Planning and Scheduling*, 2003.

[179] Extensible markup language (xml) 1.1 (second edition).

[180] Xml path language (xpath) 2.0. w3c recommendation 23 january 2007. http://www.w3.org/tr/xpath20.

[181] Jinxi Xu and W. Bruce Croft. Corpus-based stemming using cooccurrence of word variants. *ACM Transactions on Information Systems*, 16(1):61–81, 1998.

[182] Dong Yuan, Yun Yang, Xiao Liu, and Jinjun Chen. A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.

[183] Peng Yue, Liping Di, Wenli Yang, Genong Yu, and Peisheng Zhao. Semantics-based automatic composition of geospatial web service chains. *Comput. Geosci.*, 33(5):649–665, 2007.

[184] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

[185] Jia Zhou, Kendra Cooper, Hui Ma, and I-Ling Yen. On the customization of components: A rule-based approach. *IEEE Transactions on Knowledge and Data Engineering*, 19(9):1262–1275, 2007.

[186] Qian Zhu and Gagan Agrawal. Supporting fault-tolerance for time-critical events in distributed environments. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2009. ACM.