

Automated Data Management in Cloud Computing

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Arkaitz Ruiz Álvarez

May 2012

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy (Computer Science)



Arkaitz Ruiz Álvarez

This dissertation has been read and approved by the Examining Committee:



Marty Humphrey, Adviser



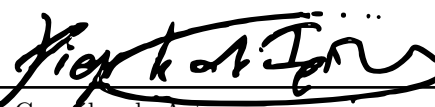
Kim Hazelwood, Committee Chair



Wes Weimer

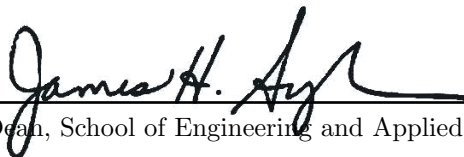


Brian Smith



Diego López de Ipiña Gonzalez de Artaza

Accepted for the School of Engineering and Applied Science:



James H. Aylor, Dean, School of Engineering and Applied Science

May 2012

Abstract

Scientists are increasingly relying on computational resources, both compute and storage, to expand scientific knowledge. For example, the *data deluge* is quickly overcoming the capacity of storage systems and the increasing use of simulation requires large compute capabilities. Thus, scientists need to expand their local resources with highly available and scalable systems. We consider cloud computing to be the solution that provides scientific applications with the computational resources needed.

However, the services offered by the cloud providers do not address several important issues: how to meet the data requirements with the storage systems available, and how to optimize cost and other performance metrics. The variety of storage and compute choices with different characteristics and prices, the growth of the data stored in terms of size and number and the data management requirements make these tasks overwhelmingly complex for individual users.

To address these challenges, we focus on four key elements of data management: the analysis of current storage services, the expression of data requirements and storage capabilities, data management algorithms and data-aware scheduling algorithms. We combine the information from our analysis of the storage services with their capabilities in a machine-readable format that can be processed by our implementation of the user’s data requirements. Thus, we can obtain within a few milliseconds a list of storage services per application dataset that meet the user’s requirements, and provide cost and performance estimates. Our unique approach to data management generates an integer linear programming problem with this list. The solution to this problem is an optimal assignment of the application’s data to cloud services. Our implementation can provide optimal

solutions for our use cases in less than one second. We have also created new scheduling algorithms for two types of cloud applications (MapReduce and watershed model calibration) that balance cost and execution time. The scheduling decisions are Pareto optimal and, therefore, superior to other strategies. We believe that these four elements can provide the users with a comprehensive solution to the data management problem, and allow them to take advantage of the new opportunities that cloud computing offers.

To my parents, Jose Miguel and Paz, my sister Aitziber and my wife Rebecca

Acknowledgments

I would like to first thank my advisor, Marty Humphrey, for his guidance during the last six years. I have learned a lot of things under his supervision and he has helped me enormously in my research projects, our joint publications and this dissertation.

I would like to thank also my dissertation committee members Kim Hazelwood, Wes Weimer, Brian Smith and Diego López de Ipiña González de Artaza, for their effort on improving the quality of this work.

I am very grateful to have shared the last few years with the other members of Marty's research group: Zach Hill, Sang-Min park, Jie Li, Ming Mao, Norm Beekwilder and Glenn Wasson. I have enjoyed listening to their research ideas and collaborating with them in multiple projects.

My wife, Rebecca, has always supported me during my degree. I have deeply enjoyed our time in Charlottesville together and I look forward to starting a family and opening a new chapter in our lives.

I cannot express how much I have missed my parents, Jose Miguel and Paz, my sister Aitziber, the rest of my family and my friends. They have helped me greatly throughout my studies and in my personal life. I know my move to the United States has been hard for them; being so far away has been hard for me too. I hope that in the future I will be able to spend more time with them.

I would also like to thank Fundación Caja Madrid for their very generous economic support.

Contents

Contents	vi
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Research issues	6
1.2 Research contributions	7
1.3 Research overview	9
1.3.1 Observations on the performance of Windows Azure	10
1.3.2 Storage capabilities and data requirements	11
1.3.3 Data management algorithms	12
1.3.4 Data-aware scheduling	13
1.4 Dissertation organization	14
2 Related work	15
2.1 Cloud computing	18
2.2 Storage services and data management	19
2.3 Scheduling	21
3 Observations on the performance of Windows Azure	23
3.1 Windows Azure storage services	25
3.1.1 Blob	25
3.1.2 Table	27
3.1.3 Queue	30
3.2 Windows Azure computing services	32
3.2.1 Dynamic scalability	32
3.2.2 TCP communication	35
3.3 SQL Azure database	36
3.3.1 Single thread client performance	38
3.3.2 Scalability	40
3.3.3 Availability	42
3.4 Recommendations	44
3.4.1 Windows Azure storage services	44
3.4.2 Dynamic scaling	45
3.4.3 Windows Azure SQL services	46
3.4.4 Testing and development	46
3.5 Conclusion	47
4 Cloud storage capabilities and data requirements	48
4.1 Targeted storage systems	50
4.2 XML Schema	51
4.3 Descriptions of storage systems	55
4.4 Use cases	57

4.4.1	Design of an application	57
4.4.2	Cost savings analysis	59
4.4.3	Cost and performance estimation	60
4.4.4	Amazon EC2 to Eucalyptus	61
4.5	Evaluation	63
4.5.1	Extensibility of schema and algorithms	63
4.5.2	Use cases	65
4.6	Conclusion	67
5	Cloud data management	69
5.1	Integer linear programming	71
5.2	Resource allocation problem model	72
5.3	Resource allocation problem solver	76
5.3.1	Basic examples	77
5.3.2	Scalability of the solver	78
5.3.3	Sensitivity of the solution	81
5.4	Use cases	83
5.4.1	BLAST	84
5.4.2	MODIS Azure cloud bursting	85
5.5	Limitations	87
5.6	Conclusion	88
6	Cloud data-aware scheduling	89
6.1	Integer linear programming solver	91
6.2	Applicability to MapReduce	92
6.2.1	Problem formulation	93
6.2.2	Use case	96
6.3	Applicability to watershed model calibration	98
6.3.1	Problem formulation	98
6.3.2	NP hardness	101
6.3.3	Use case with hourly billing	103
6.3.4	Use case with multiple watershed models and data-aware scheduling	105
6.4	Conclusion	109
7	Conclusions and future work	111
7.1	Summary of results	111
7.2	Conclusions	113
7.3	Future work	115
	Bibliography	117

List of Tables

3.1	Windows Azure Table throughput	29
3.2	Windows Azure Blob/Table throughput comparison	30
3.3	Windows Azure Queue throughput	32
3.4	Windows Azure worker role request time	33
3.5	Windows Azure web role request time	33
4.1	Length of the XML description per storage system and per schema elements	56
4.2	Storage recommendations for example A	59
4.3	Storage recommendations for example B	59
4.4	Storage recommendations for use case 4	62
5.1	Data management model variables	74
5.2	Running BLAST on a budget	85
6.1	Problem model variables for MapReduce.	95
6.2	Problem model variables for the watershed model calibration.	98

List of Figures

1.1	Overview of our system	10
3.1	Windows Azure Blob client performance	26
3.2	Windows Azure Table client performance	28
3.3	Windows Azure Queue client performance	31
3.4	Windows Azure TCP latency	36
3.5	Windows Azure TCP bandwidth	37
3.6	Windows Azure TCP bandwidth over time	38
3.7	Windows Azure TPC-E client performance	39
3.8	Windows Azure TPC-E slowdown	40
3.9	Windows Azure TPC-E transactions committed	41
3.10	Windows Azure TPC-E cloud performance over time	42
3.11	Windows Azure TPC-E local performance over time	43
4.1	Main elements of the XML Schema	52
4.2	Object type definition in the XML Schema	52
4.3	Windows Azure Blob description in XML	53
4.4	Lines of code for each data requirement	64
4.5	Storage matching process performance	66
5.1	Solving time vs. number of storage services	79
5.2	Solving time vs. number of problem variables	80
5.3	Sensitivity of the solution	82
5.4	Monthly budget and computation speed tradeoff	86
6.1	Scheduling cost and turnaround time for MapReduce	97
6.2	Scheduling cost and turnaround time for the watershed model calibration	104
6.3	Scheduling cost and turnaround time for multiple watershed models	108

Chapter 1

Introduction

Scientists are increasingly relying on computational resources, both compute and storage, to expand scientific knowledge. The amount of data gathered from scientific instruments is quickly overcoming the capacity of storage systems; a phenomenon called *data deluge* [1]. Another trend is the increasing use of simulation, which is now a key component of the scientific method, but it requires compute capabilities beyond the single workstation. Thus, scientists are looking to expand their local resources (workstations and clusters) with highly available and scalable systems.

Recently, several companies like Amazon, Google, and Microsoft have started providing remote computational resources on a pay-as-you-go basis, which has been introduced as cloud computing. Computational resources are allocated in their huge datacenters and provide the users with great flexibility: resource usage can scale up or down, resource capacity appears to be unlimited to individual users, there are no upfront costs, and the pricing is competitive with local resources. One of the main motivations for the introduction of cloud computing is the low average utilization of datacenters. Datacenters are sized to handle peak loads (e.g. customer orders during the holiday season for Amazon), which leads to resources being underutilized most of the time since the average load is much smaller than peak loads. If there are enough idle resources in Amazon's datacenters then it makes economic sense to rent some of the excess capacity to external users. Also, cloud providers can take advantage of the economies of scale to build datacenters that support the computational needs

of many users. The aggregate workload of many cloud users leads to a more efficient use of resources, a lower cost for cloud users and a profit margin for cloud providers. Each cloud provider offers different abstractions for accessing computational resources; cloud providers are usually classified in three different categories:

- **Infrastructure as a Service, IaaS:** cloud providers may offer the lowest level of abstraction possible, the one that is closest to the actual hardware. Here a cloud user can request Virtual Machines whose interface is the same as a physical machine. Therefore the user needs to manage the operating system and other configurations. This is the most flexible configuration for the user since any operating system and application can be run. An example of this type of cloud is Amazon Web Services [2]. One of the main components of Amazon Web Services is the Elastic Compute Cloud (EC2), which offers a wide range of compute resources.
- **Platform as a Service, PaaS:** instead of the Virtual Machine abstraction, the PaaS cloud provider offers a particular Application Programming Interface (API) against which to program their application. The configuration of the operating system is transparent to the user. Examples of PaaS providers are Force ¹ and Windows Azure [3] (based on the Microsoft .NET platform). In Windows Azure the user develops a .NET application with Visual Studio, which produces a package that can be uploaded and run directly in the cloud. Application development and deployment can be much less complex and there are other benefits (e.g. easier scalability), but PaaS ties the user to a particular platform.
- **Software as a Service, SaaS:** a cloud provider may offer a complete application as a service. Two examples of this approach are GMail ² for e-mail and Salesforce ³ for Customer Relationship Management (CRM). This approach is not limited to applications with a large customer base, for example Microsoft Research supports the efforts of certain teams to provide scientific applications like MODIS Azure [4] on top of the Windows Azure Platform. These applications

¹<http://www.force.com>

²<http://www.gmail.com>

³<http://www.salesforce.com>

are remotely accessible, commonly via the Web browser. With this model it is possible to meet the computational needs of scientists from all over the world.

In our work we focus on IaaS and PaaS clouds; Amazon Web Services and Windows Azure are the largest cloud providers for each of these types. EC2 is a service offered by Amazon where users can rent Virtual Machines by the hour. Currently there are 13 different machine configurations available, which range from micro instances to cluster compute instances with 88 cores. There is also a wide range of storage services. Instances can mount virtual hard drives through the Elastic Block Storage (EBS) or access individual objects through the Simple Storage Service (S3). S3 objects are stored in buckets and are analogous to files in a shared folder. Amazon also offers the SimpleDB and DynamoDB services; these services offer support for more structured data than S3, but they do not support the traditional SQL interface. These two services have many advantages such as ease of use, performance, and scalability compared to traditional SQL databases. The traditional SQL interface is offered by the Relational Database Service (RDS). There are other services available that offer more storage options, services for the application, networking capabilities, and development utilities and services.

A very important feature of cloud computing is its elasticity and the pay-as-you-go model. For example an application may require 10 Virtual Machines running for most of the time but during peaks may require more than 50. The user only pays for the resources used by the hour; in this example the user no longer requires buying and maintaining a small cluster with 50 machines since the Amazon cloud can provide this capacity in minutes and at a very low price (e.g. a medium instance costs 0.16 dollars per hour). This elasticity not only applies to computing, but also to other resources such as data storage. A small application storing a few hundred megabytes pays 0.125 dollars per gigabyte per month to store the data in S3; if the application suddenly becomes highly used and the storage grows into the dozens of gigabytes the price per gigabyte remains the same and extra capacity is always available. S3 can provide storage for thousands of terabytes of data.

Local resources can not easily scale and are costly to maintain, but that does not mean that they can not benefit from cloud technologies. It is common for scientists or other groups to have some local

resources (clusters, data storage networks) available. Some software systems, such as Eucalyptus [5] and Open Nebula [6], can turn a local compute cluster into a local private cloud whose interface is compatible with Amazon's cloud. The advantages here are two-fold: local applications can use virtualization and other cloud technologies, and local resources can be extended with cloud resources. This way a local cluster can be sized to handle the average workload (leading to high utilization and cost-effectiveness) and when a workload peak occurs the user can obtain some additional resources from a public cloud provider. This use case is commonly referred to as *cloud bursting*.

As we have mentioned before, Windows Azure follows the Platform as a Service model. The compute services, similar to Amazon's, include several sizes of virtual machines, as well as types: web, worker and VM. The application functionality assigned to web and worker roles is defined during the development phase. VM roles are more flexible and more similar to instances in IaaS providers since it allows the user to make configuration changes to the operating system. Similar to Amazon's storage services, there are SQL services (RDS), Blobs (S3), Tables (SimpleDB and DynamoDB) and Azure Drives (EBS). There also exist multiple services at the application level and utilities for development (through Microsoft's Visual Studio). In summary, even if Amazon Web Services and Windows Azure fit in two different categories there are multiple comparable services for storage, networking, application, and development. Even in the services where they differ, mainly compute, there is enough flexibility (VM roles in Windows Azure, Windows instances in Amazon) to consider both clouds for the development and deployment of almost any cloud application.

Although primarily targeted to a business audience, we consider cloud computing to be the solution that provides scientific applications with the computational resources needed. The current state of the services offered and the APIs does not, however, address several important issues. Let us consider the following example: scientists from different institutions are interested in sharing climate data obtained locally (or via satellite) and feeding these data into their climate models to gain new insights. One of the problems that this group faces is deciding where to store each datum: they cannot afford to lose original data (measurements) so it must be stored in durable storage; intermediate data must be saved in scalable storage since computational climate models

could potentially involve hundreds of processes; etc. Thus, we need to match each type of data to the storage system that satisfies the requirements. This problem is not trivial when, as in our example, the datasets originate from different sources, are stored in different sites and even in different storage systems/services within the same site (where there are many different storage options).

A central issue of cloud computing is cost: every hour of computation, GB stored or transferred over the network has an explicit price tag. Prior to the cloud, the price to pay for computing has been implicit because shared local clusters generally hide the cost of hardware to individual users and universities or research centers assume the cost of electricity and IT staff. Thus, users of cloud computing are faced with the task of optimizing cost, but this task is usually daunting without an automated approach to data management: datasets are large in terms of size and number of files (scalability), there exist multiple cloud providers with different storage options and prices (accurate cost models), data growth over time requires reevaluation of the storage choices (adaptation). Finally, scheduling compute jobs also becomes a complex problem because of the latency and bandwidth limitations for staging data in/out and the additional hourly cost of cloud computing. We propose that scheduling algorithms should be made aware of the location of datasets and the prices and network interconnects between the possible sites that can host the computational jobs. For example, users of scientific applications may take advantage of the local computing infrastructure at their university or research center and optionally offload some workload to a cloud when the available resources are limited.

Thus, expanding scientific computation from the local environment to the cloud presents several challenges: how to meet the data requirements with the storage systems available, how to optimize the storage cost and access bandwidth and which cloud platform (from a limited choice of cloud providers) should be chosen to schedule computational jobs that access the data. The variety of storage and compute choices (both locally and in clouds) with different characteristics and prices, the growth of the data stored in terms of size and number, and the long term management requirements make these tasks overwhelmingly complex for individual users. A manual solution is neither scalable nor adapts easily to changing conditions and it is overly complex because of the multiple variables

(price, latency, durability, availability, etc.).

1.1 Research issues

In this section we enumerate the key issues that we address in our research:

- **Cloud platform performance:** users need performance information to effectively use cloud platforms, but which performance metrics are the important ones for developing cloud applications? Which parameters should be chosen to represent the multiple dimensions of the performance of cloud services? Some examples of these dimensions may be scalability (some services can crash under 32 concurrent instances while some do not), the amount of data per request as well as the type of request, and variability over time (TCP communication bandwidth has a stable average but high variability). In general we are concerned on how to analyze and represent the performance of cloud platforms so that cloud applications can be built successfully.
- **Cloud storage:** given that there are different cloud providers, each cloud provider may offer services in many datacenters across the globe, and each datacenter can host different storage services, how can users choose storage services for each application dataset? Is the same approach going to work in the future, since the number of different storage services is going to increase with new cloud providers, datacenters (e.g. Amazon in Oregon) and services (e.g. Amazon's new DynamoDB)? A dimension that multiplies this problem further is the number of datasets in a cloud application since each dataset will have a different set of requirements. Thus, the user faces the problem on how to make sure that the chosen storage services for a cloud application meet the application's requirements. Additionally, cost becomes an important requirement for cloud applications, so how can solutions account for the cost dimension in cloud computing? Even if we can produce solutions that are sufficient (i.e. the solution meets the budget and the storage requirements); is it possible to produce better (e.g. cheaper and faster) decisions? How can we prove these decisions to be optimal? For example, only for

Amazon and only in the United States there are currently 21 storage services in three different zones; for each dataset in an application we have 21 possible choices, but a global solution for n datasets the number grows to 21^n possible choices. How can we make the best resource allocation decisions within this vast space, and for different applications? Furthermore, users should be able to define what constitutes a best solution based on performance parameters (latency, bandwidth, job turnaround time) and/or cost.

- **Cloud scheduling:** similar to the offer of storage services, there is a wide range of compute services available (Amazon currently offers 13 different instance types) for the cloud user and, for most applications, a seemingly unlimited supply of them. In this context, the scheduler's decision for each job request is composed by the following parameters: how many instances to use and/or boot, what is the type of each instance, where do we get the input/output data from/to, and how long to keep these instances running. Some of the factors and restrictions that come into play for this decision are: the size of the computation, the location of the input and output datasets, how much parallelism the application supports, the hourly billing model and the cost of data transfer, and the different types of instances available. The main metrics concerning the user are cost and time; so how can the scheduler make scheduling decisions that balance the cost and the job execution time (according to the user's preferences) by tuning the aforementioned parameters?

1.2 Research contributions

The hypothesis of this dissertation is that *cloud applications should automate data management in order to effectively utilize cloud resources*. We propose data management middleware as an essential part of the infrastructure of future distributed systems. The users will rely on this data management middleware to automatically map each dataset/file to one or several storage services in order to satisfy the user's requirements, such as capacity, durability, availability, consistency semantics and

scalability. This data management middleware should also optimize the variables of cost, latency, bandwidth and job turnaround time.

The contributions of this work are:

- We present the first comprehensive evaluation of the Windows Azure cloud platform. These experiments present different metrics such as bandwidth, latency, time, and percentage and number of successful actions for different services across different usage scenarios (different parameters for scale, size of data, size and type of request, and variability over time). We also present recommendations for every service for developers of the platform; for example the 2x slowdown of SQL Azure (although with better scalability) compared to a local SQL Server needs to be taken into account when developing a cloud application that needs a database.
- We introduce a machine-readable format for storage service description that is capable of fully describing cloud storage services such as the ones offered by the Amazon cloud, the Windows Azure cloud, and local clusters. Our approach is extensible as new services and new features are added to the changing cloud environment. We have also created a framework for implementing user's data requirements based on our service descriptions; this framework can process the user's requirements quickly (under 70 ms for every of our four use cases) and provide a list of possible storage services for each dataset along with performance and cost estimates. This list of possible storage services represents the set of solutions which are sufficient, that is, they all meet the user's requirements.
- Given the lists of possible storage services for all datasets in an application we have created an algorithm that can provide a globally optimal solution based on cost, job turnaround time, latency and/or bandwidth. Our algorithm transforms the lists of all possible storage services into a mathematical problem (integer linear programming, ILP) whose solution is the assignment of data to storage services. Our unique approach leverages the advances on ILP solvers; even if our problem is NP hard we show that the ILP solver can arrive to a solution in under a second for current problem sizes. Our solution can also accommodate an order of

magnitude increase in problem sizes since solving times would remain under one minute. The solutions presented by the ILP solver are guaranteed to be the optimal ones: for the MODIS Azure cloud application our approach is 52% faster than a local-only solution (if the user favors speed) or it can halve the cost of a cloud-only solution (if the users favors cost).

- In the final part of our work we combine our first three contributions into an end-to-end solution by creating new scheduling algorithms for cloud applications. We evaluate our approach with two different classes of emerging cloud applications (MapReduce and Monte Carlo type simulations), producing provably optimal scheduling decisions. In our model we take into account the data allocation and other cloud characteristics (hourly billing cost, instance type selection, number of instances); similar to our data management algorithm we generate an ILP problem. Our approach balances the trade-offs between the job's execution time and cost, and provides users with a set of choices that are Pareto optimal (for a given execution time no other solution is cheaper). In our studies, our algorithm's scheduling choices are always better than a naive approach, which can be up to 2.6 times more expensive and 5.46 times slower (for our MapReduce use case). More advanced algorithms can avoid the pitfalls of trivial strategies, but they are still up to 11% slower and up to 38% more expensive in our watershed model calibration use case.

1.3 Research overview

Figure 1.3 shows a general overview of our system. On the right side we have the different datacenters with storage and compute services that cloud providers and others (local clusters) provide. On the left side we have the user that provides the data, the data requirements, and information about the execution of the application (number of runs, duration, datasets accessed). Our work focuses on four key elements of the data management middleware for distributed cloud systems: the analysis of current storage services, the expression of data requirements and storage capabilities, data

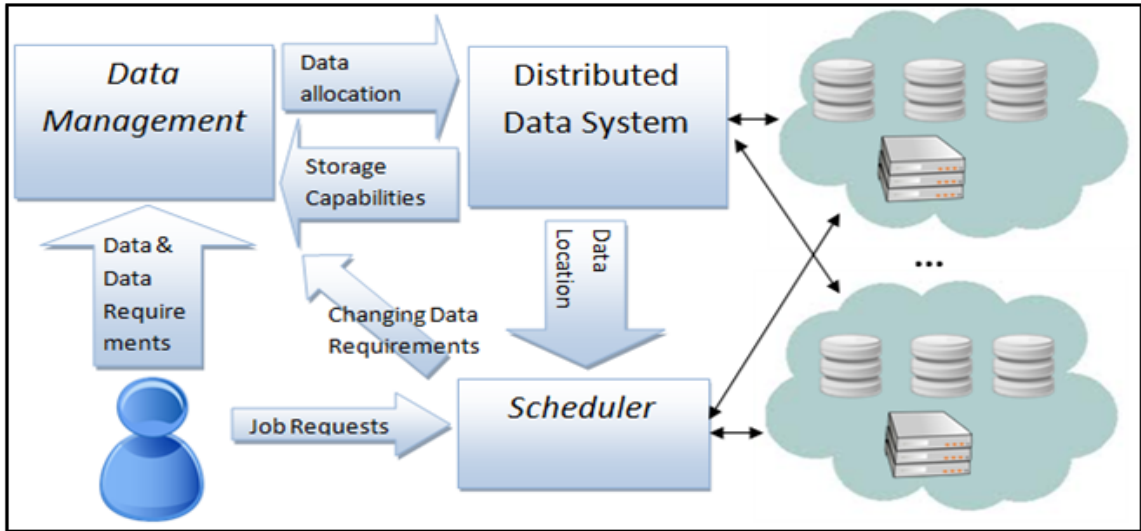


Figure 1.1: Overview of our system.

management algorithms and data-aware scheduling algorithms. We believe that these four elements can address the challenges described above.

1.3.1 Observations on the performance of Windows Azure

In order to make good data allocation decisions we first must analyze the performance behavior of the cloud; in our case we have targeted the Windows Azure Platform. In our work we focus on two elements of this platform: Windows Azure, which encompasses both compute resources and scalable storage services, and SQL Azure, which provides traditional SQL Server capabilities for databases. Our evaluation of the Windows Azure service begins with the performance of its compute resources and its three primary storage services: Blobs, Queues, and Tables. Because these are the basic storage components that scalable Windows Azure applications are built upon, it is important to understand their performance characteristics as scale increases. We then evaluate Virtual Machine instantiation time because instance acquisition and release times are critical metrics when evaluating the performance of applications that scale dynamically. We also present an evaluation of direct instance-to-instance TCP performance as this mechanism provides an alternative to the other storage services for communication between instances that has lower latency. For the SQL Azure service,

we present the performance implications of running a database server in the cloud, the influence of client location, scalability in terms of the number of clients and availability over time. We use the TPC-E benchmark, which is a database benchmark defined by the Transaction Processing Council that represents the operations of an OLTP system.

We summarize our experimental data into several specific recommendations for developers using Windows Azure Platform. In these recommendations we address virtual machine instances, the storage services, SQL Azure and our experience in testing and developing cloud applications. Although the discussion and analysis is tailored to scientific applications, the results are broadly applicable to the range of existing and future applications running in Windows Azure. This work appears in the *Proceedings of the 1st Workshop on Scientific Cloud Computing (ScienceCloud 2010)*, co-located with HPDC 2010 [7]. An extended version was published in the *Journal of Scientific Programming*, vol. 19 [8].

1.3.2 Storage capabilities and data requirements

Each cloud provider offers different storage abstractions: Amazon provides the S3, EBS, SimpleDB and Relational Database storage services; similarly Windows Azure provides the Blob, Table, Drive and SQL Azure services. For local storage resources users have traditionally used NFS drives and sometimes high performance distributed systems such as the GPFS and Hadoop. In this context, with multiple and very different options for storage, users are facing a daunting task when trying to select a storage service. There are multiple aspects that have to be taken into account: performance, cost, specific features, long term management issues, etc. Application performance is affected by the selection of the storage and several variables (number of clients, size of data and requests, etc.). Users would also like to select the cheapest storage service that meets the data requirements. To further complicate this selection, the requirements for existing data change over time and new datasets and applications are added to the cloud so choices need to be reconsidered periodically.

In order to automate these decisions we develop a machine readable description of storage services and a framework for expressing data requirements. First, we identify the multiple characteristics and

features of storage systems (such as the performance analysis from the previous section) and propose an XML schema that includes them. Second, we present examples of current storage systems that are fully described using our schema. Third, we take several use cases drawn from our experience in the field of scientific computing and algorithmically match their storage needs to concrete storage services. The final result of our system is a match of services and requirements, which ensures that data storage meets users' requirements (durability, availability, etc.), performance expectations (latency and throughput at scale) and provides cost estimates. We evaluate two aspects of our approach: the extensibility of our approach, and the wall clock time spent processing each use case. This part of our work appears in *Proceedings of the 2nd Workshop on Scientific Cloud Computing (ScienceCloud 2011)* [9], co-located with HPDC 2011.

1.3.3 Data management algorithms

The output of the previous part is, for each dataset, a list of storage services that meet the user's requirements, along with cost and performance estimates. The user would then choose a data allocation from these options, mainly by selecting the cloud provider with the best (based on cost or performance) storage options for all the datasets. Although this choice may not be optimal, it meets all the user's data, cost and performance requirements. In this part we address three important limitations: we rely on the user to make a choice from the list of storage options for each dataset; each dataset is analyzed in isolation so it is not obvious what the best global solution is; and the computational side (number of application runs, cost per hour, machine speed, etc.) is not taken into account. Our goal here is to produce a global data allocation solution that balances cost and performance of both storage and computation. If the best storage service for a single dataset resides in a cloud that does not have good choices for the rest of the application data, then we may arrive to a sub-optimal data allocation. As we will show, trying to find an optimal solution increases the complexity of the problem to NP hard. We provide a model for this data allocation problem and a software implementation that is both fast and scalable.

We first present our problem model, whose solution represents the data allocation decision. Factors

such as storage cost, compute cost, job turnaround time, latency, and bandwidth are combined into an objective function that has to be minimized. The combination of this function and additional restrictions (linear constraints) forms an integer linear programming problem. In order to solve this computationally hard problem we use a modern ILP solver (lpsolve); we show that, for simple use cases, a couple hundred milliseconds are enough to solve the problem instance; more complex problems can still be solved under 1 second.

We present two use cases for our system: BLAST and MODIS Azure. For BLAST we take our standard model and include one additional restriction: a monthly budget. Thus, the user may ask for the best data and compute allocation which fits her budget. For MODIS Azure we try to make the allocation decisions that will minimize job turnaround time the most, while still meeting the data requirements and the budget. This work will be presented in the *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, (CCGRID 2012)* [10].

1.3.4 Data-aware scheduling

Traditionally, a scheduler assigns some shared resources (e.g. a single CPU or a Beowulf cluster) under its control to processes, threads or parallel applications in such a way that optimizes throughput, latency, fairness and/or other metrics. However, in a cloud environment, applications within a reasonable range of computing requirements operate under the abstraction that computational resources are unlimited and almost immediately attainable. A scheduler which simply replies to requests composed by a number of machines and the job's execution time seems trivial to implement in cloud computing since we do not have to worry about resource sharing. In the last part of our work we explore the question of whether the scheduler should optimize some metric instead of blindly processing a request, what interface should be presented to the user, and how we can take advantage of the data allocation decisions made in the previous part of our work.

We believe that there are new requirements for scheduler algorithms in the context of certain cloud applications. Thus, we are not looking for a general scheduler, but for a scheduler that takes advantage of the characteristics of a certain class of applications in order to produce a scheduling plan

that is cheaper and/or faster. The first class of applications that we consider is MapReduce jobs. The second one is applications whose running time can be estimated (similar to Monte Carlo simulations). For each of these two classes we generate an Integer Linear Programming (ILP) problem based on the information from the cloud providers (the processing capacity of each instance and its cost), the local cluster, the application (the amount of work to be done and the data location) and a constraint on the job's execution time (provided by the user at submission time). The solution to this ILP problem will be the optimal assignment of tasks to compute resources based on cost or execution time. The scheduling decisions made by this strategy form a Pareto efficient frontier that are faster or cheaper than any alternatives. For example, the scheduler for the watershed model calibration avoids the worst possible scenarios in a naive strategy (which costs almost twice as much), and is also superior to other strategies (which are 11% slower for the same amount of money). This part is currently being considered for submission.

1.4 Dissertation organization

The rest of the dissertation is organized as follows: we review related work in Chapter 2. In Chapter 3 we introduce an in-depth look at the performance of the storage services and other characteristics of the Windows Azure cloud. Chapter 4 introduces the description of cloud storage services in a machine-readable format as well as the mechanism to match storage services to the data requirements of the application. We present storage descriptions for Windows Azure, AmazonWeb Services and a local cluster. Chapter 5 introduces the algorithms that leverage the approach outlined in Chapter 4 to select the optimal assignment of application datasets to storage services. In Chapter 6 we present new scheduling algorithms that take advantage of the data allocation decisions and the performance of cloud resources. We conclude the dissertation with Chapter 7.

Chapter 2

Related work

We find today many examples of scientific applications with complex data requirements that rely on distributed systems to provide the computing capabilities needed to support them. On one hand we find applications with very large data storage needs, such as DZero [11] for high-energy physics and LEAD for weather forecasting [12]. These applications store great amounts of input data (from sensors, satellites, etc.) or computed data, and usually several geographically dispersed organizations collaborate in the research effort. Previous work has addressed the possibility of meeting the large resource needs with cloud computing [11], but in an application-specific context. On the other hand there is an emerging market for supporting applications that work most of the time at small scale in individual workstations but can scale up in distributed systems if needed [13, 14]. Again, these efforts are either application-dependent transitions to cloud systems or they focus on the computational design of new applications that run exclusively in one cloud platform. The complex problem of data management is getting worse because of the increasing amount of data in digital form and the variety of data sources: a phenomenon commonly referred to as the *data deluge* [1].

There exists a variety of options to provide resources for scientific applications: local clusters [15], supercomputer centers [16], datacenters [17] and a combination of several sites forming a grid [18] or a cloud [19]. Local clusters are generally locally available to researchers at universities or other institutions, but their storage capacity is limited and other requirements, such as high availability

or data durability, are difficult to achieve. Supercomputer centers are very expensive to build and maintain; they also quickly become obsolete. Currently few institutions can afford a supercomputer center. A group of institutions may share a supercomputer center, but utilization of this expensive resource becomes an issue. The same main issues are shared by datacenters. Peak utilization can overload all the resources and utilization valleys are seen as a waste of money. Aggregating several workloads is the common solution to smooth the difference between peaks and valleys, which is one important advantage for grids and clouds. Teragrid [20] is the leading example of a distributed infrastructure based on grid technologies that makes high-performance computers and data resources available to scientists.

Grids that focus on data storage and management have been traditionally labeled as data grids [21]. The core data grid services commonly include data storage and access, and metadata management. Because of the heterogeneous nature of grid computing, designers of data grids services provide abstractions for accessing and storing data that are independent of the underlying storage systems and transfer protocols. Since different cloud providers have different APIs and data services, such storage and access abstractions can provide a building block for our work. However, there is an important limitation for applying data grid APIs to the current environment: data grids usually use the file abstraction as the basic unit of storage while cloud computing has introduced new data storage interfaces. A file instance in grids corresponds to the blob instance in clouds, but the queue and table cloud abstractions have no corresponding counterpart in grids. Thus, we need to extend previous work on these storage abstractions such as the GLUE schema [22] in order to accommodate the new cloud services. One of the new characteristics of cloud services is cost: every hour of computation, GB stored remotely or transferred over the network has an explicit price tag. Prior to the cloud, the price to pay for computing had been hidden from users; in cloud computing users are faced with the task of optimizing cost, but this task is usually daunting without an automated approach since there exists multiple cloud providers with different storage options and prices.

Together with the data storage and access APIs the metadata management is a key component of the data grid infrastructure. From the perspective of our work, we are mostly interested with

the replica metadata and the system configuration metadata. Since data is commonly replicated in a distributed system, applications need access to a replica catalog to locate the actual data [23]. One advanced strategy for replica selection is relying on the matchmaking algorithm of Condor to select a replica that fits the application requirements. The requirements are specified as expressions over attribute-value pairs, for example “reqdSpace = 5G”. Matching replicas are ranked by a certain attribute, for example available space. The matchmaking framework [24] introduces the matchmaking algorithm and protocols (for advertising, matchmaking and claiming) to assign each user’s request to a resource provider and process it. This framework certainly provides the foundation for a solution to one of the problems that we address in our work. In the current context, with the introduction of cloud computing, we believe that this matchmaking framework (as well as the underlying language, the ClassAds language [24]) should be taken a step further to support expressing high level user requirements, such as scalability, and matching them to the specifics of cloud storage systems. Our approach shares some similarities with Condor’s although instead of selecting replicas we select storage services. We also present an extensible interface to execute arbitrary code instead of doing an attribute-value match: this way we can take into account the particularities of cloud computing and match higher level requirements against the different cloud storage services.

Computer scientists that developed data grid technologies often built them upon previous work on distributed file systems. Data grids services offer a high level interface and rely on a low level distributed file system. iRODS [25], the Storage Resource Broker [26], Armada [27], the Legion FS [28] are important examples of distributed file systems that have been used in conjunction with grid technologies. These systems provide the data grid with the necessary tools to store data, replicate data across several sites and manage these replicated datasets. Stork [29] is a project that aims to improve data management for large data sets that are stored on distributed systems. Users can create data placement jobs with Stork, which are essentially managed, fault-tolerant data transfers between different storage systems (local disk, FTP, GridFTP, iRODS, SRB, etc.); each data placement can be associated with its corresponding computing job. In our work we see these distributed file systems and data transfer management systems as necessary building blocks.

2.1 Cloud computing

Cloud computing has burst onto the high performance computing scene in recent years and has established itself as a viable alternative to customized HPC clusters for many users who do not have the resources (either time or money) to build, configure, and maintain a cluster on their own. Many eScience developers are increasingly looking to create data-intensive applications [13, 14] that can take advantage of the pay-as-you-go cost model. Amazon’s pioneering effort in the area [2], API compatibility with other projects (Eucalyptus [5], OpenNebula [30]), wide customer base and the widest and most mature offer of cloud services make Amazon EC2 our primary target for cloud providers. We also consider Microsoft’s Windows Azure [3] cloud which, although it mainly provides the Platform as a Service (PaaS) abstraction, gives much flexibility in terms of data storage and program execution (any Windows binary). Other clouds, such as the Google App Engine [31], provide a very rigid programming and data storage model that will not be valuable for many applications. Thus, we consider this last type of clouds as out of scope in our work.

In the research community, there is an increasing recognition of performance concerns of clouds and their underlying technologies. For example, Menon et al.[32] and others [33] evaluated the performance overhead of Xen [34], a software virtualization technology which is a popular choice as the low level virtual machine manager by several cloud providers. Xen has been shown to impose negligible overheads in both micro and macro benchmarks [33]. A higher level analysis is provided by Garfinkel [35], who evaluates some of the cloud services that Amazon provides. Earlier work within our research group has compared the performance of cloud platforms with local HPC clusters for scientific applications [36]. Another report examines the feasibility of using EC2 for HPC in comparison to clusters at NCSA [37]. This comparison pits EC2 against high-end clusters utilizing Infiniband interconnects.

There are also studies that focus on a specific scientific application, such as DZero [11] or Montage [38], to evaluate the possibility of migrating existing applications and data to the cloud, based on performance and cost parameters. Workflows [39] and service-oriented applications [40] have also been the object of study. Another study reports on the possibility of running coupled

ocean-atmosphere simulations on EC2 [41]. Our research complements this earlier work by providing a direct measurement of the mechanisms and APIs of a specific cloud, Windows Azure.

2.2 Storage services and data management

Recent work in the cloud computing area has focused on the storage APIs and underlying systems. We can find in the literature evaluations of current cloud storage systems such as Amazon S3 [11] and Windows Azure [7]. Additionally, new storage systems such as an elastic transactional data store have been suggested [42]; other papers focus on the storage stack [43] and moving from the file system interface to scalable cloud storage [44]. The focus of our work is not introducing new storage systems or APIs, but rather developing a higher level process that will use these underlying systems. Thus, our aim is to express the characteristics and features of current storage systems (while being flexible enough to accommodate future ones) in a machine readable format. These descriptions allow us to automate processes such as storage selection and cost analysis; we could implement past manual work that have evaluated specific applications such as Montage [38] or platforms such as grids for volunteering computing [45].

Currently high level data management is limited to certain strategies such as trading storage for computation [46] or using proxies [47]. These optimizations could save money by not storing data and generating it on demand or by using proxies that compute data operations locally to save bandwidth costs. Although current cloud middleware has yet to implement a comprehensive strategy for data management (other than virtual machine images), we can find in the literature several proposed algorithms for managing data replicas and job scheduling in Grid computing. An early example is the Close-to-Files algorithm [48]. The authors used this algorithm in the KOALA grid scheduler [49]. In this algorithm, a central scheduler allocates jobs in a multicluster system by selecting for each possible pairing of execution site and file site the one that minimizes transaction time. In their model, each job is composed by several components where all the job components require the same single input file. This algorithm only schedules jobs and does not make any decisions regarding the data: both the number and the location of replicas are an input to the algorithm. The work of R.

Chang et. al. proposes two algorithms, Hierarchical Cluster Scheduling (HCS) and Hierarchical Replication Strategy (HRC) [50], and implements them for the Taiwan Unigrid environment. HRC simply selects the data replica closer to the machine running the job and stores a new copy of it, possibly deleting some unused replicas to free storage resources. HRC does not provide any replica consistency semantics. HCS minimizes the inter-cluster-communication cost by selecting the cluster with the fastest access to data replicas. This algorithm, however, does not take into account available compute resources in each cluster and works exclusively at the cluster level. Additionally, this strategy minimizes computation time but cannot be extended to account for many data requirements (durability, availability, bandwidth, storage cost, etc.). Another approach to the scheduling problem is utilizing the replica selection algorithms mentioned previously (for example, the matchmaking algorithm used by Condor [51]).

The theoretical foundation of the file allocation problem was formalized by Chu several decades ago [52]. In this problem we have a set of computers interconnected by a network which provides storage for a set of files. Each file can be stored in multiple computers, and the problem model takes into account the cost of storage, the cost of data transmission, the storage capacity of each computer, file sizes, the requests rates for reading/updating each file and the maximum latency allowed for each access. The optimal solution is the one that minimizes the costs of storage and transmission. Chu's work formulates the problem as a zero-one integer linear programming problem, which is NP hard. This problem model, however, does not address some of the users requirements outside cost and maximum latency; it also does not take into account the possibility of multiple sites. Thus, it does not apply directly to data management in cloud computing. Other similar work by Casey has formulated a problem model for allocating multiple replicas of a file in a distributed system [53] taking into account the cost of storage and data transmission and the read and update queries: this problem is still NP hard. Reliability has also been considered the variable to optimize in a variation of the file allocation problem approached with genetic algorithms [54]. Subsequent work on the file allocation problem has addressed the complexity of these models by filtering sites that participate or not in an optimal solution [55], devising polynomial-time approximation algorithms

based on a reduction of the file allocation problem to the Knapsack problem [56] or other heuristics that iteratively refine feasible initial solutions [57]. Dowdy and Foster [58] identified 12 different models of the file allocation problem which differ on several parameters: minimizing cost, execution time, access time, response time or maximizing throughput; considering single files, multiple files or data and program files; etc. Later variations of the problem [59, 60] considered also a dynamic approach as the storage needs change over time and also the location of program files associated with data files [61]. To the best of our knowledge, the most common software for managing data grids (SRB, iRODS, GPFS, HPSS) does not implement these file allocation algorithms. In data grids we can usually find a dedicated part of a site to storage, and the rest of the nodes access data through the network. Thus, there is not a concern for optimizing data storage costs at the individual computer level and the majority of the access to the data occurs within the site. However, with the introduction of cloud computing there is the possibility of renting storage space at different sites, making the file allocation problem relevant again. The unique characteristics of our approach are: our problem model was built specifically for data management in cloud computing (as opposed to within a local cluster), and we present an implementation that is fast enough to provide an optimal solution. Our implementation relies on recent advances that have led to the development of efficient boolean satisfiability solvers [62] and integer linear programming solvers [63].

2.3 Scheduling

The introduction of the concept of *cloud bursting* seems like a logical extension of the already existing local resources, augmented with the capabilities of the cloud. This type of hybrid computing [64] has been examined as a potential solution for taking advantage of the scalability benefits of the cloud while keeping the performance benefits of the local cluster. These Elastic Clusters are based on software such as OpenNebula [6], Eucalyptus [5] or on application specific middleware [65] (Windows HPC and Azure integration). Interesting algorithms for a cloud bursting scheduler have also been suggested [66]; our approach differs in that we do not consider a job queue but rather make a decision on each job as they come and we are focusing on application specific jobs rather on a general scheduler

(albeit the information required by both approaches is similar). This last approach from S. Kailasam et al. also focuses on data intensive computing, similar to the approach taken by T. Bicer et al. [67] where they suggest middleware to expand the resources available to MapReduce jobs while taking into account the transfer of data for data intensive jobs.

In our present work we focus more on the one datacenter approach for one of the use cases (MapReduce applications) and the choices related to the computational resources. In other use cases (a watershed model calibration), we do take into account data transfer. At job submission our algorithm will consult the input data location (or locations) and account for data transfer times and costs. We believe that the data allocation should be planned in advance [10]; we also believe that provably optimal algorithms should be preferred if the scheduler can be shown to be fast enough.

Chapter 3

Observations on the performance of Windows Azure

As more cloud providers and technologies enter the market, developers are faced with an increasingly difficult problem of comparing various offerings and deciding which vendor to choose for deploying an application. One critical step in the process of evaluating various cloud offerings can be to determine the performance of the services offered and how those match the requirements of the application. Even in situations where other factors ultimately dominate the choice regarding potential cloud platforms (e.g., cost per unit time of a virtual machine or application hosting environment in the particular cloud), it is important to consider performance ramifications of design decisions to ensure maximum value of cloud applications.

On February 1st, 2010, Microsoft announced the commercial availability of their cloud offering, the Windows Azure Platform [3]. As evidenced by many of the presentations at Microsoft's Professional Developers Conference (PDC), Microsoft has made a significant investment into Windows Azure to make it attractive to the Information Technology community. This investment is both the hardware/software necessary to run the Windows Azure cloud and also enhancements into the Microsoft software development tools (i.e. Visual Studio) to address the challenges of writing, deploying, and managing a cloud application. While the target audience of Windows Azure has

understandably been the broad business community, it is worth noting that Microsoft and the National Science Foundation have a partnership agreement for providing resources to selected scientific projects.

Shortly after the introduction of Windows Azure, we designed and performed the first quantitative analysis of the performance of the Windows Azure Platform. Through our existing collaborative partnership with Microsoft Research, we were able to evaluate the services at scales not readily available to users of the early Community Technology Preview release (up to 192 concurrent instances). This partnership, however, does not imply interior knowledge of the Azure infrastructure, and throughout this chapter our perspective is the same as other users of the Windows Azure platform. This platform is composed of three services: Windows Azure, SQL Azure, and AppFabric. This chapter focuses on Windows Azure, which encompasses both compute resources and scalable storage services, and SQL Azure, which provides traditional SQL Server capabilities for databases. AppFabric, however, was too early in its development for a comprehensive evaluation.

Our evaluation of the Windows Azure service begins with the performance of its compute resources and its three primary storage services: Blobs, Queues, and Tables. Because these are the basic storage components that scalable Windows Azure applications are built upon, it is important to understand their performance characteristics as scale increases. We then evaluate virtual machine instantiation time because instance acquisition and release times are critical metrics when evaluating the performance of dynamic scalability for applications. We also present an evaluation of direct instance-to-instance TCP performance as this mechanism provides an alternative to the other storage services for communication between instances that has lower latency. For the SQL Azure service, we present the performance implications of running a database server in the cloud, the influence of client location, scalability in terms of the number of clients and availability over time. We use the TPC-E benchmark, which is a database benchmark defined by the Transaction Processing Council that represents the operations of an OLTP system.

In general, we have observed good performance of the Windows Azure mechanisms, although the average 10 minute VM startup time and the worst-case 2x slowdown for SQL Azure in certain situations (relative to commodity hardware within the enterprise) must be accounted for in application

design.

Finally, we summarize our experimental data into several specific recommendations for developers using Windows Azure Platform. In these recommendations we address virtual machine instances, the storage services, SQL Azure and our experience in testing and developing cloud applications. Although the discussion and analysis is tailored to scientific applications, the results are broadly applicable to the range of existing and future applications running in Windows Azure.

3.1 Windows Azure storage services

In this section we start our analysis with the performance tests of all three Windows Azure Storage Services: Blob, Table and Queue. For each service we measure maximum throughput in operations/sec or MB/sec and the scalability of the service as a function of the number of concurrent clients, among other service-related metrics. For all our tests we use from 1 to 192 concurrent clients.

In general, the scalability of these storage services may be a problem for large-scale applications with multiple concurrent clients if the data is not adequately partitioned or replicated (all of the storage services are automatically and transparently triple-replicated). The performance penalty caused by increased concurrency must be taken into account in order to meet the application's requirements, and we provide several data points to help the software developer make decisions about the application's architecture and scale.

3.1.1 Blob

The Blob service in Windows Azure establishes a storage hierarchy: storage accounts have multiple containers, which hold one or more blobs. Each individual blob can store one terabyte of data and some associated metadata. Here we analyze the performance of both the download and upload of data onto blobs. For our blob download test we use a single 1 GB blob that is stored in Windows Azure. Then, we start a number of worker roles (from 1 to 192) that download the same 1 GB blob simultaneously from the blob storage to their local storage. For the upload test, since the different worker roles can't upload the data to the same blob in Windows Azure storage, a different test is used:

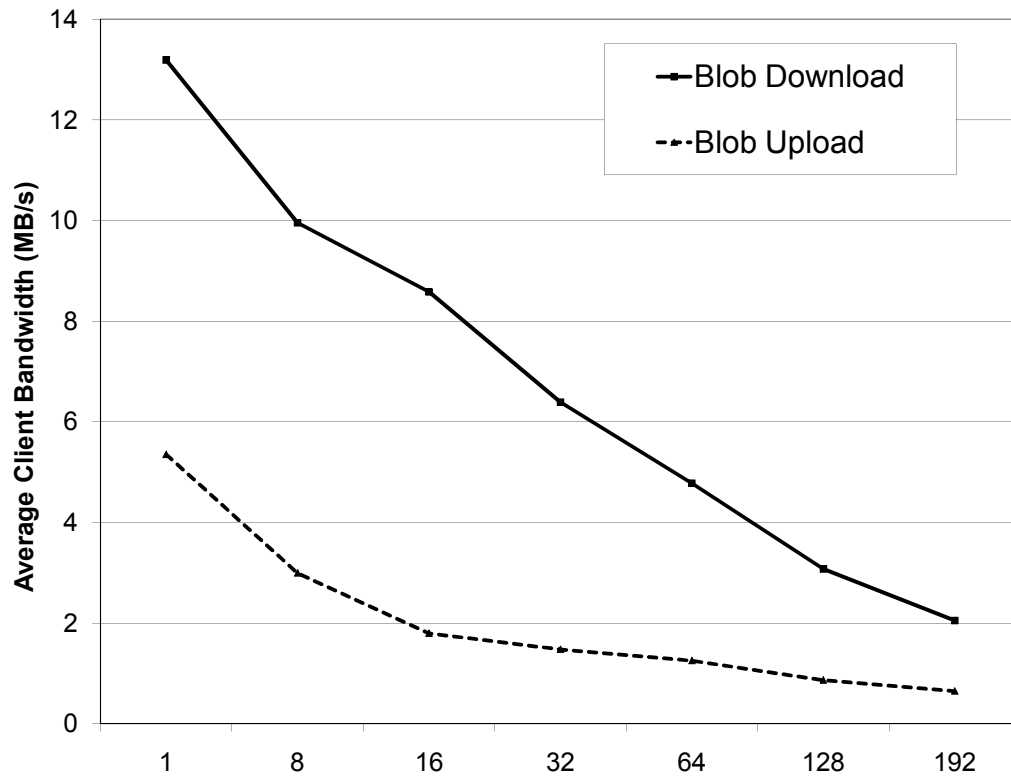


Figure 3.1: Average client blob download bandwidth as a function of the number of concurrent clients.

the worker roles will upload the same 1 GB data to the same container in the blob storage, using different blob name. We run the same test three times each day. Although we have collected data for several days at different times, the variation in performance is small and the average bandwidth is quite stable across different times during the day, or across different days.

The maximum server throughput for the blob download operation was 393.4 MB/s, which was achieved by using 128 clients. For the blob upload operation, the maximum throughput was 124.25 MB/sec, which was observed in our experiments with 192 concurrent clients. Figure 3.1 shows the average client bandwidth as a function of the number of concurrent clients attempting to download the same blob. The bandwidth for 32 concurrent clients is half of the bandwidth that a single client achieves. Using more concurrent clients (up to 128) increases the total aggregate bandwidth, although this comes with the price of much slower clients. Figure 3.1 also shows the performance of the upload blob operation. The scalability test results for the upload operation show similar results

to the download ones, as we can see the similarities between both curves in the graph. However, because we are now only uploading the 1 GB file to the same container as different blobs, we are not accessing the same blob object as in the download operation. We conclude that the operations “uploading blobs to the same container” and “downloading the same blob” both suffer from the same concurrency constraints.

The overall blob upload speed (including the baseline) is much slower than the download speed. For example, average upload speed is only 0.65 MB/s for 192 VMs, and 1.25 MB/s for 64 VMs. This lower speed may be linked to the internal network upload policies, but also to constraints on write operations. The upload speed may be limited by the complexities involved in infrastructure: level operations for creating new blob objects, such as record generations, replication operations, etc.

3.1.2 Table

A table in Windows Azure is a set of entities with properties, where each property can have various types and the table has no defined schema. In this sense, it is very different from a table in a relational database. A single table has the capability to store a large number (billions) of entities holding terabytes of data. We have examined the performance of 4 operations from the Windows Azure Table API: Insert, Query, Update and Delete. We have run our experiments with different entity sizes: 1 KB, 4 KB, 16 KB and 64 KB. We used from 1 to 192 concurrent clients to study the scalability of each type of table operation. We have found that the shape of the performance curves for different entity sizes are similar, except for some exceptions which are noted below.

For each test case, our experiments performed the following steps: we start with the Insert experiment using different number of concurrent clients, where each client inserts 500 new entities into the same table partition; after the Insert experiment the table partition includes 220K same-size entities. Next, we perform Query operations over the same partition by using a partition key and row key, and each client queries the same entity 500 times. Since currently Windows Azure tables are indexed on PartitionKey and RowKey only, this key-based query is the fastest query option. Windows Azure Table also supports query on table properties other than the keys, but we didn't

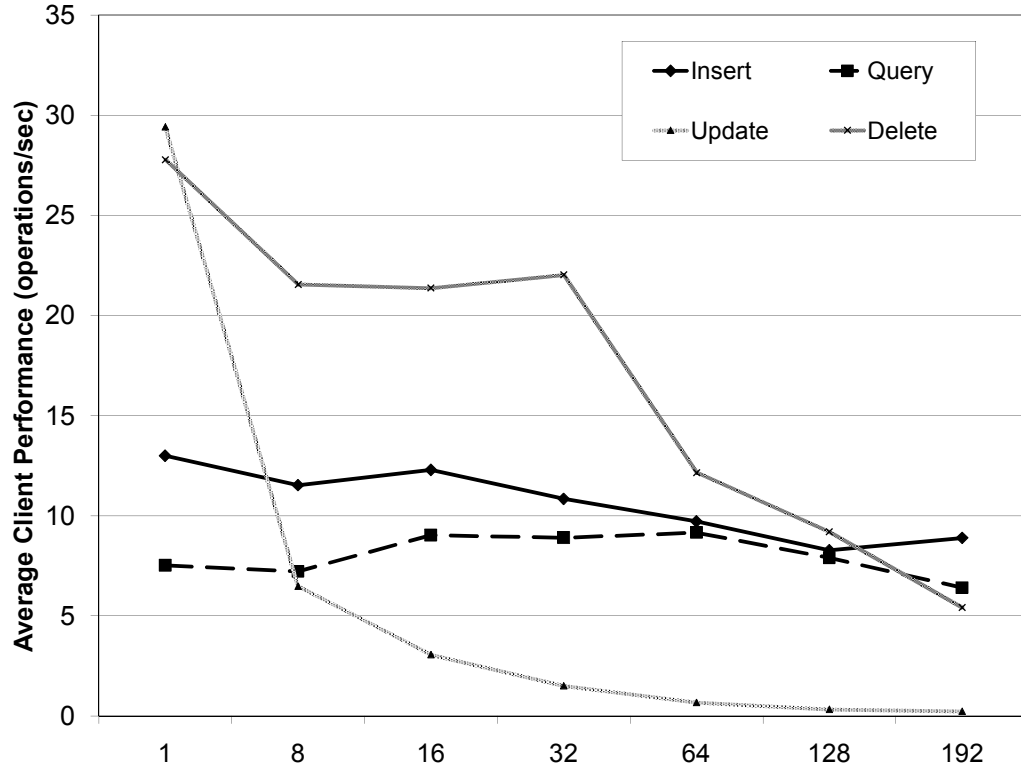


Figure 3.2: Average client Table performance as a function of the number of concurrent clients. Entity size is 4 KB.

evaluate their performance here. For the Update experiment, each concurrent client updates the same entity in the partition and repeats the operation for 100 times. Here we only tested with the unconditional updates option as it doesn't enforce atomicity of each update request, so that different clients can issue update requests to the same table entity at the same time. Finally, in the Delete experiment, each client removes the same 500 entities it inserted in the first step of our experiment.

The result of our experiments is summarized in Figure 3.2 and Table 1. Figure 3.2 presents the data from the point of view of the client, that is, how many operations per second can concurrent clients sustain? For both Insert and Query, the performance of the clients decreases as we increase the level of concurrency. However, we think that even with 192 concurrent clients we have not hit the maximum server throughput for these two operations. The Update and Delete tests show more drastic performance declines as we increase the number of clients, though. These two operations have high initial throughput with only 1 client, but then slow down drastically as the number of concurrent

Table 3.1: Maximum observed Table throughput in ops/sec as a function of entity size. The numbers in parentheses are the number of concurrent clients.

Operation	1 KB	4 KB	16 KB	64 KB
Insert	2237 (192)	1706 (192)	520 (128)	165 (128)
Query	1525 (192)	1011 (192)	1153 (192)	1139 (192)
Update	85 (16)	52 (8)	59 (8)	37 (16)
Delete	2227 (192)	1178 (128)	1281 (192)	622 (64)

clients increases. The maximum throughput for these two services is reached at 8 concurrent clients for the Update operation and 128 for the Delete operation.

Our experiments show that the performance curves for other entity sizes are similar as Figure 3.2, except for the following exceptions during the Insert and Delete tests: for the Insert test on 64 KB entities with 192 concurrent clients, only 89 clients successfully finished all 500 insert operations, and the other 103 client have encountered timeout exceptions from the server. With 128 concurrent clients inserting 64KB entities, only 94 clients successfully finished all 500 operations. This indicates that we may hit the table service capability limit on the above two combinations with large entity size and high concurrency. We have also observed similar behaviors during the Delete tests.

Table 3.1 summarizes the observed maximum throughput for the Table service for different entity sizes. The number in the parentheses indicates the number of concurrent clients at which point the maximum throughput is achieved. As we can see, Insert is the most sensitive operation to the size of entities, whose effect can be as high as 13 times less operations per second. On the other hand, the throughput of Query operations is the least sensitive to different entity sizes.

Finally, we have run experiments that compare the performance of the Table and Blob services. That is, for storing information that does not need to be queried (so it could be stored as a blob) and that does not exceed the table entity size limit (so it could be stored in a table too), we want to find out what is the best option for the programmer. Table 3.2 answers this question. We run the same experiment: 1, 8 and 16 concurrent clients performing insertions of objects from 1 KB to 64 KB. We have used both a blob container and a table as the backend storage system. The results show that

Table 3.2: Average client Blob insert performance in ops/sec compared to Table insert performance.

Number of Clients	1 KB	4 KB	16 KB	64 KB
1	0.22x	1.35x	1.43x	1.44x
8	0.29x	2.05x	2.86x	2.60x
16	0.36x	2.13x	2.56x	2.79x

Blob is preferable if the amount of data to insert is between 4 KB and 64 KB. The Blob performance can be as low as a fourth of the Table performance for small objects of 1 KB: it takes around 200 ms to create a new blob of 1 KB versus 40 to 70 ms to insert a 1 KB table entity. Depending on the number of concurrent clients and the size of the data objects (if 4 KB or greater) to be stored the performance of Blob is between 35% to 3 times faster than Table. Surprisingly, it takes between 40 to 60 ms to insert a blob of 4 KB, which means that it is 5 times faster to create a 4 KB blob than a 1 KB blob. Due to the black box nature of the cloud services we do not have an explanation for this behavior.

3.1.3 Queue

The main purpose of Queues in Windows Azure is to provide a communication facility between web roles and worker roles. For our queue test we use one queue that is shared among several worker roles (from 1 to 192). We examine the scalability of three Queue operations: Add, Peek and Receive. For each operation we run the test with different message sizes: 512 bytes, 1 KB, 4 KB and 8 KB. As it was the case with Table, the shape of the performance curves for each message size is similar and we choose to show the results for 512 bytes.

Figure 3.3 shows our results. In general, the operations Add and Receive display similar trends. Peek message is the fastest operation, since it does not alter the visibility of the message. Thus, the different replicas of the queue do not need to be synchronized on a Peek operation and all of these requests can be run in parallel. Add and Receive require a different implementation, though. Both of these operations require synchronization among distributed objects: the Add operation needs to add the message to the same places in each replica of the queue; the Receive operation needs to assign the triple-replicated message to only one client.

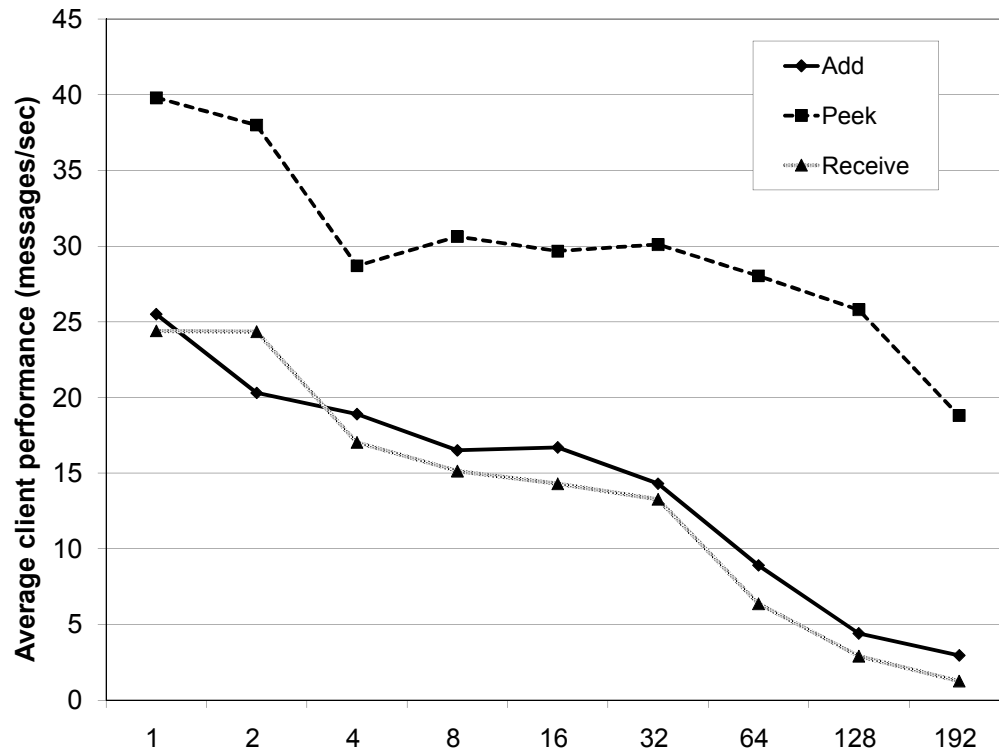


Figure 3.3: Average Queue client performance as a function of the number of concurrent clients. Message size is 512 bytes.

Table 3.3 presents the maximum throughput of each operation based on the size of the message. For the Add and Receive operations, the maximum performance peaks at 64 concurrent clients. For Peek, however, we present the throughput that was achieved with 192 concurrent clients. We believe that we have not exercised this operation on the Queue service to the maximum of its capacity, albeit the throughput increase from 128 instances to 192 instances starts showing diminishing returns. Limitations on the number of virtual machines that we can start on Windows Azure prevent us from running this experiment at a higher scale.

We have also run some experiments to test the influence of the size of the queue on the performance of the mentioned operations. We have found that there is not much variation at all as the queue grows in size from 200 thousand to 2 million messages.

Table 3.3: Maximum observed Queue throughput in ops/sec as a function of on message size. The numbers in parentheses are the number of concurrent clients.

Operation	512 B	1 KB	4 KB	8 KB
Add	569 (64)	565 (64)	556 (128)	502 (64)
Peek	3878 (192)	3609 (192)	3388 (192)	3369 (192)
Receive	424 (64)	407 (64)	396 (64)	393 (64)

3.2 Windows Azure computing services

In this section, we discuss computing instance acquisition and release time in Windows Azure and TCP communication between different virtual machines instances.

3.2.1 Dynamic scalability

We believe that computing instance acquisition time is a critical metric to evaluate the efficiency of dynamic scalability for cloud applications. We have written a test program that uses the Windows Azure management API to collect timing information about each possible action on Windows Azure Virtual Machine instances. We manage two types of Virtual Machines: web roles and worker roles. In addition, Windows Azure offers four types of VM size: small, medium, large and extra large. By combining these two parameters for each test case we create a new Windows Azure cloud deployment.

For every run of our test program, it randomly picks a role type and a VM size, and creates a new deployment. We choose the number of instances in each deployment based on the VM size: 4 instances for small, 2 for medium and one for large and extra large. Then our test program measures the time spent in all five phases (create, run, add, suspend and delete). These phases are divided based on Windows Azure deployment and instance status.

1. Create: in this phase, we record the wall clock time from application deploy request initiation to the time when Windows Azure indicates the deployment is ready to use. In our test, all the deployment packages are stored in Windows Azure Blob storage services.

Table 3.4: Worker role instance request time in seconds.

Size	Statistic	Create	Run	Add	Suspend	Delete
Small	AVG	86	533	1026	40	6
	STD	27	36	355	30	5
Medium	AVG	61	591	740	37	5
	STD	10	42	176	12	3
Large	AVG	54	660	774	35	6
	STD	11	91	137	8	6
Extra Large	AVG	51	790	N/A	42	6
	STD	9	30	N/A	19	5

Table 3.5: Web role instance request time in seconds.

Size	Statistic	Create	Run	Add	Suspend	Delete
Small	AVG	86	594	1132	86	6
	STD	17	32	478	14	2
Medium	AVG	61	637	789	92	6
	STD	10	77	181	17	6
Large	AVG	52	679	670	94	5
	STD	9	40	155	14	3
Extra Large	AVG	55	827	N/A	96	6
	STD	16	40	N/A	3	8

2. Run: when the deployment is successful, the test program initiates a “Run” request to start the VM instances in the deployment. We measure the time from the start of the request to the time when all VM instances are ready to use status changes from “stopped” to “ready”.
3. Add: after the instances are started running, the test program initiates a “Change” request and doubles the number of running instances. We measure the time that takes these newly added instances to become ready status changes to ready.
4. Suspend: when all the VM instances are running, we suspend all the running instances in the deployment and measure the time spent to terminate each Windows Azure VM instance (status changes from “ready” to “stopped”).
5. Delete: After all the running instances are suspended, our test program initiates a “Delete” request and removes the current deployment.

From Dec 17th, 2009 to Jan 9th, 2010, we collected data from 431 successful runs. The VM startup failure rate, taking into account all of our test cases, is 2.6%. Starting from Jan 1st, 2010,

Windows Azure changed from CTP to commercial platform. Our observations did not find clear performance differences between these two periods. The output data is shown in Table 3.4 and Table 3.5. From these tables we draw the following observations:

1. Web role VM instances need longer time to startup than worker role instances. For all VM sizes, web role takes 20 to 60 seconds longer than Work role VMs. Such observation is consistent with our expectation, since each web role requires a more complex initialization than a worker role (e.g. IIS support and Windows Azure load balancer registration). Also, large VMs take longer time to startup than small VMs.
2. The average time to start a worker role small instance is around 9 min., while the average time to start a web role instance is around 10 min. The quickest time we observe is 7.5 min. for worker role and 9 min. for web role. For 85% of our test runs, the first small worker role instance becomes ready within 9 min. and for 95%, the first small worker role instance becomes ready within 10 min. For 80%, the first small web role instance becomes ready within 10 min and for 90%, the first small web role instance becomes ready within 11 min.
3. Windows Azure does not serve a request for multiple VMs at the same time. That is, there is a lag between the time the first instance becomes available and the following ones. For both worker role and web role small instances, we have observed a 4 min. lag between the 1st instance and the 4th instance of our deployment.
4. Adding more instances to existing deployment takes much longer than requesting the same number of instances at the beginning.
5. Application deployment performance (create phase) is largely a function of the application size. A 1.2MB application starts 30 seconds faster than a 5 MB application. Note that our test deployment is stored in Windows Azure storage service. If the application package is stored locally, the deployment time could takes much longer because of the local network bandwidth limit.

6. Windows Azure shows consistent performance for deployment deletion, around 6 seconds for all test cases.

3.2.2 TCP communication

Windows Azure allows the programmer to define TCP or HTTP internal endpoints for the virtual machine instances in the deployment. This type of communication is highly coupled, works only in a point to point fashion and the application needs to define the protocol. However, it is a good complement to the Queue (low coupling, multiple readers/writers, defined API) since these internal ports allow the VMs to talk directly with each other using a low-latency, high-bandwidth TCP/IP port. Therefore, we have measured the performance of this feature of Windows Azure VMs based on three metrics: latency, bandwidth and bandwidth variability over time.

In our first experimental setup, we create a deployment with 20 small VMs. 10 of these VMs measure latency, and the rest measure bandwidth. Each virtual machine is paired with another one; each pair contains one server and one client. In order to measure the latency, the client measures the roundtrip time of 1 byte of information sent on the TCP channel, after communication has been established. For the bandwidth measurement the client sends 2 GB of information to the server; each run of this bandwidth test usually takes around 30 seconds.

Figure 3.4 and Figure 3.5 present our results. We have collected for these two graphs a total of 10,000 measurements. Both figures present the histogram of our samples. Figure 3.4 shows that approximately 50% of the time the latency is equal to 1 ms; 75% of the time the latency is 2 ms or better. In general, the most common case is to find in the datacenter latency that is similar to our LAN. Figure 3.5 summarizes the bandwidth measurements. 50% of the time we find the bandwidth to be 90 MB/s or better. We assume that the physical hardware is Gigabit Ethernet, which has a limit of 125 MB/s. So in our experiments we have seen that is rather common for the VMs to have good bandwidth. However, for the lower end of the sample (15%) the performance drops to 30 MB/s or worse. In our second experimental setup, we start a deployment with two VMs, one client and one server. These two VMs transfer the same 2 GB of data, but they do that every half

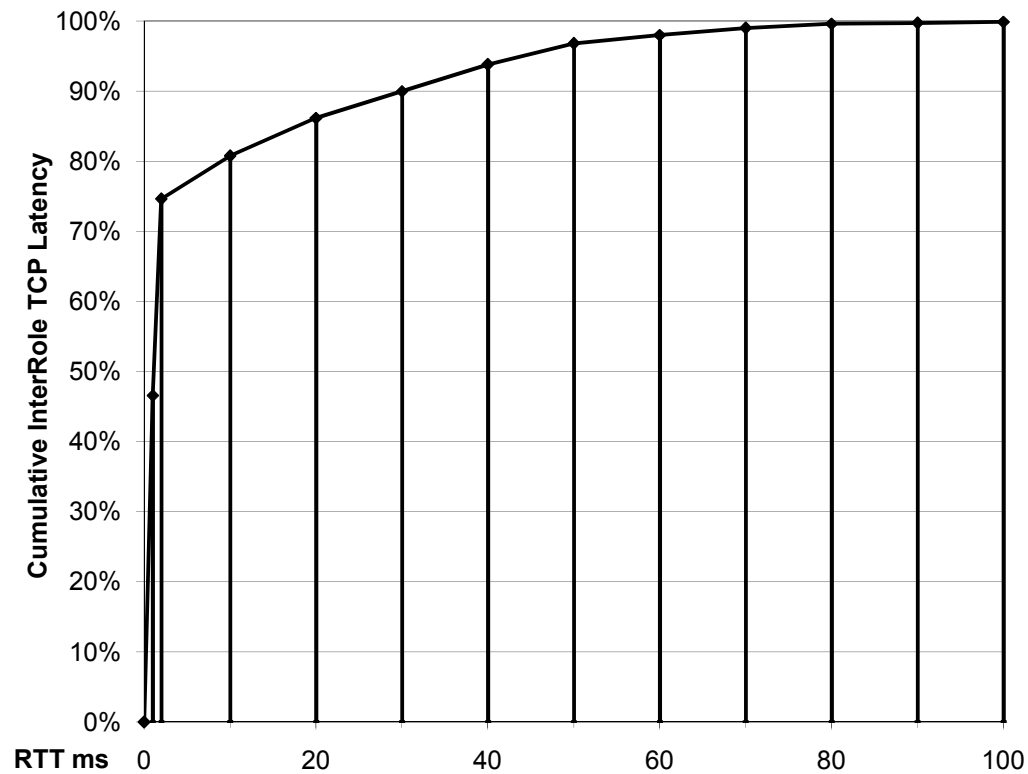


Figure 3.4: Cumulative TCP latency (RTT ms) between two small VMs communicating through TCP internal endpoints.

hour for several days. We have plotted in Figure 3.6 our bandwidth measurements in MB/s. For the most part the bandwidth exceeds 80 MB/s, but there are times in which it can be as low as 10MB/s. Since we are using small instances, the resource manager will most likely allocate other instances from other deployments in the same physical hardware. We believe that the activity of these other neighboring instances can greatly influence the bandwidth observed by the application. Although the common case is still similar to our LAN, developers are warned that the variability is high. TCP is the feature in which we have observed the highest degree of variability for the Windows Azure platform.

3.3 SQL Azure database

After an initial attempt to provide new data services models designed to be scalable, but with limited functionality, the Windows Azure cloud platform started offering a full SQL product as an integral

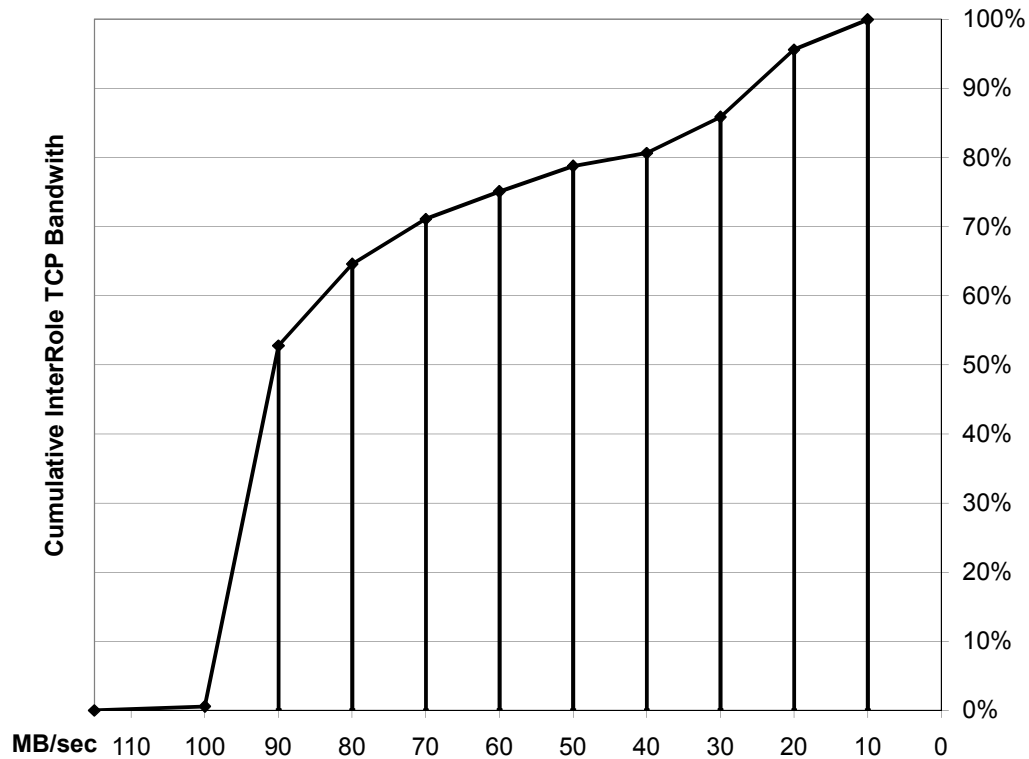


Figure 3.5: Cumulative TCP bandwidth (MB/sec) between two small VMs sending 2 GB of data through TCP internal endpoints.

part of their service. The widespread use of the relational model in all kind of applications makes the SQL Azure Database a crucial part of the cloud infrastructure. Essentially, this product is a modified version of SQL Server that runs on the Windows Azure platform and is compatible with Microsofts database protocol Transact-SQL.

Similar to other Windows Azure products, SQL Azure focuses on high availability, scalability, ease of deployment and automatic management. In this section, we analyze these aspects together with the performance of SQL Azure to offer a complete picture of what this platform offers to new developers. Our test application is the TPC-E benchmark, whose specification is published by the Transaction Processing Council. The TPC-E benchmark simulates an OLTP workload, and is designed to replace the old TPC-C benchmark.

Our experimental setup is the following one: in the Windows Azure cloud we have deployed a SQL Azure server with our 3 GB TPC-E database and we deploy up to 8 extra large instances (with

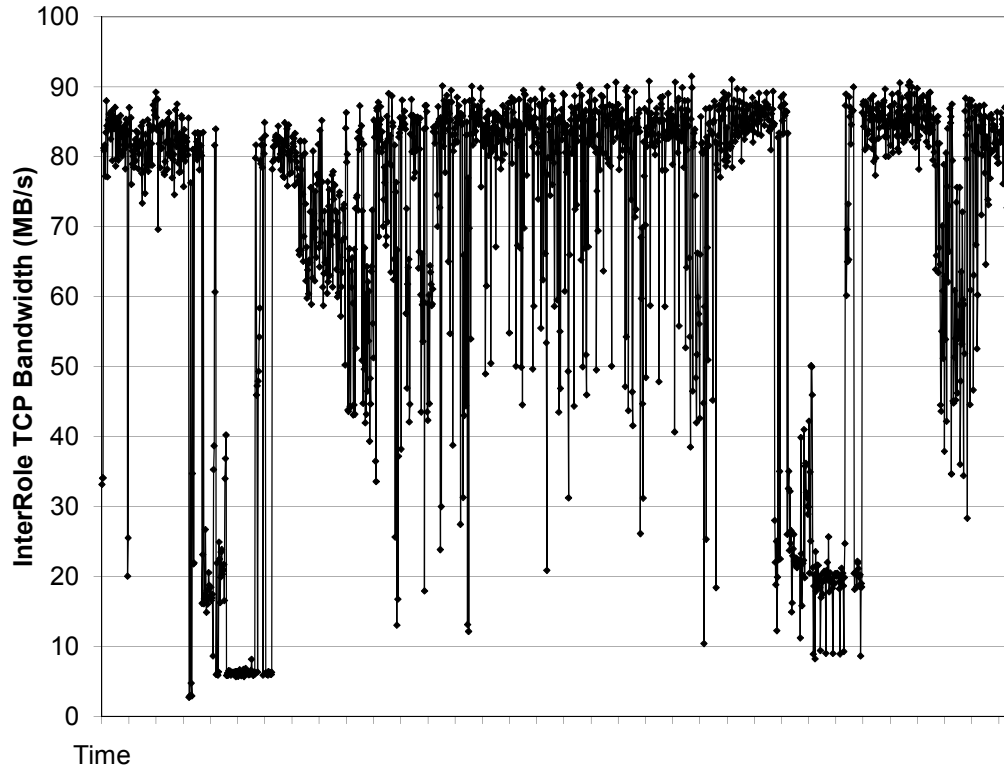


Figure 3.6: Variation of the InterRole TCP Bandwidth (MB/sec) between two small VMs.

eight cores and 8GB of RAM each) to function as TPC-E clients. Locally, we have a Windows Server 2008 in a quad core with 8 GB of RAM running SQL Server 2008, and 3 TPC-E client machines (dual cores) connected by a LAN.

3.3.1 Single thread client performance

Graphs in Figure 3.7 show the performance comparison of the different SQL servers with clients that run in the same machine (Local client to local server), across a LAN (Local client to server in LAN) and across the datacenter (Windows Azure client to Windows Azure server). The TPC-E client runs a single thread of execution for four hours, including a ramp-up period of 30 minutes. The TPC-E benchmark consists of 10 different transactions, each of them consists of different SQL queries that include selections, updates, inserts and deletes across the 33 tables that compose the TPC-E database. We measure the time it takes to complete the transaction from the client side and

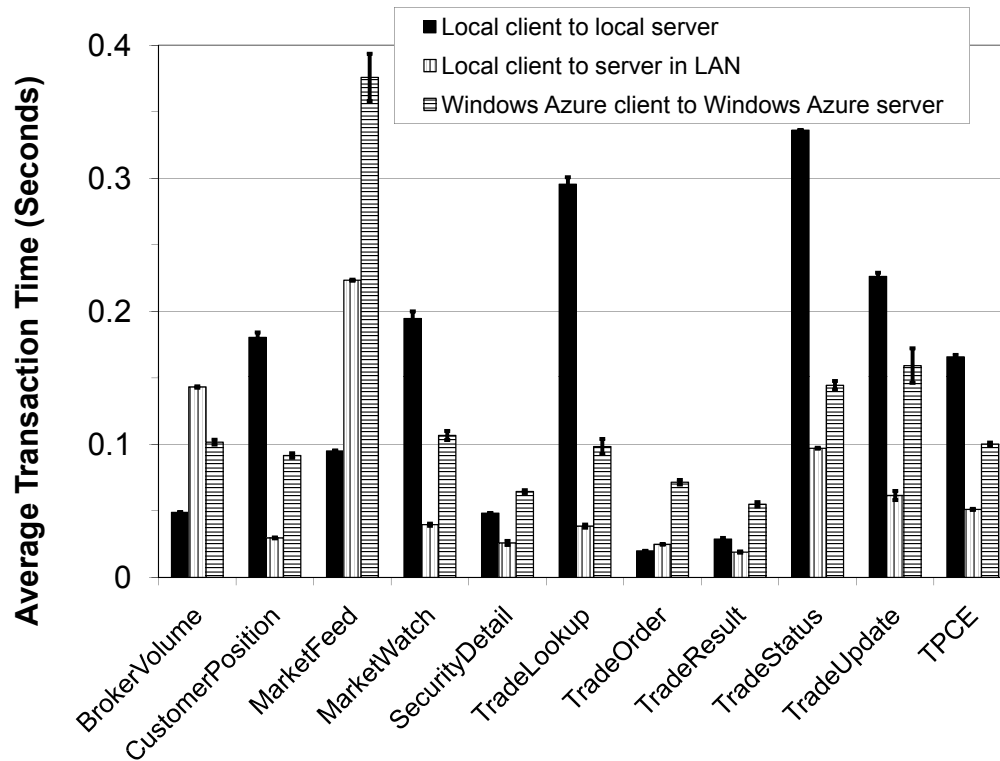


Figure 3.7: TPC-E transaction times in seconds for each client and server location.

show the average of each type of transaction (including a 95% confidence interval on the population mean as the error bar) and the general average, labeled as TPC-E.

There is, on average, almost a 2x difference in speed when we compare an Windows Azure VM client querying a SQL Azure database with a local client querying a SQL Server across our LAN. The slowdown for “Local client to local server” is caused by SQL Server taking too many resources, which slows down the client. If the user wishes to locate the Windows Azure VM client outside the cloud (mobile clients, saving costs, collaboration across multiple sites, etc.) we have observed that there is an order of magnitude of slowdown (not included in this graph). This additional overhead is caused by latency of communications between our client in Virginia and the Windows Azure server in the Southern United States datacenter.

This graph shows that, based on performance, a local deployment is preferable than a purely cloud deployment. In the next sections, however, we will discuss other aspects that developers should take into account.

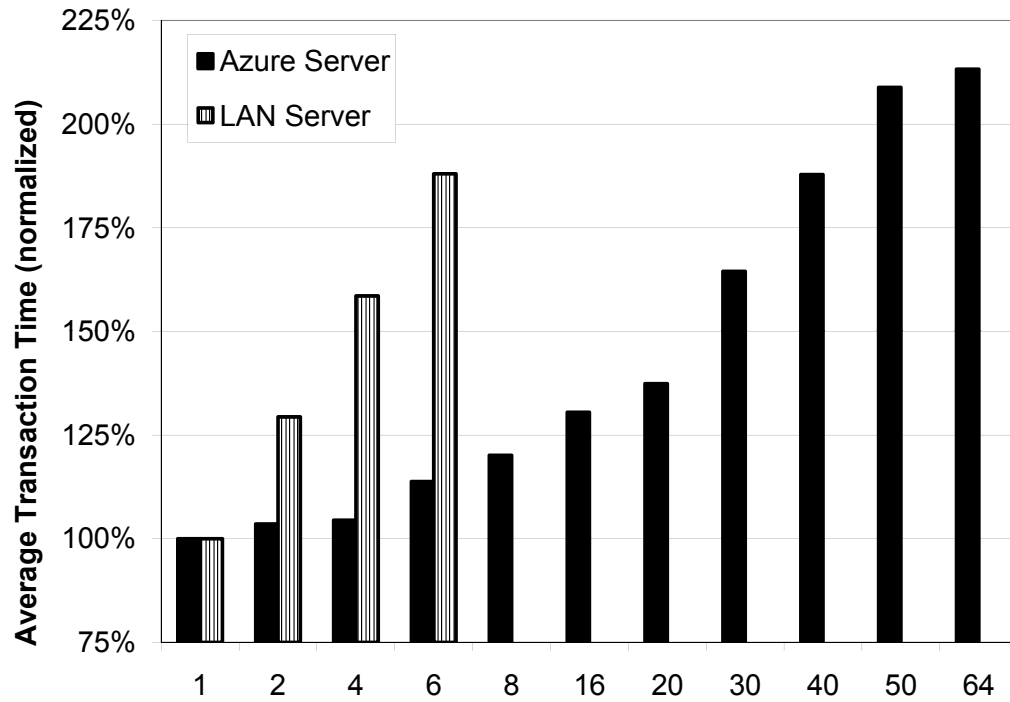


Figure 3.8: Average transaction slowdown as a function of the number of concurrent TPC-E clients.

3.3.2 Scalability

In order to evaluate the scalability of both local and Windows Azure SQL servers, we start several TPC-E clients that query a single server at the same time. We examine two cases, the first one being several Windows Azure VMs running clients against Windows Azure SQL Database (up to 64) and the second one includes several clients (up to 6) querying our local SQL Server 2008 across the LAN. We compare the average transaction time for each client thread running concurrently with several others with the single thread client performance. Our first result is Figure 3.8. The graph shows that the Windows Azure database does not take such a big performance hit as the local one. 6 concurrent local clients achieve a similar slowdown as 40 cloud clients. In this case, the cloud service performs better than our local SQL Server when we scale the number of concurrent clients.

We can also notice that the slowdown plateaus around 50 clients in the cloud. This does not indicate that the scalability is unlimited, since we are taking into account for this graph only the transactions that successfully commit. Thus, for a high number of threads trying to access the

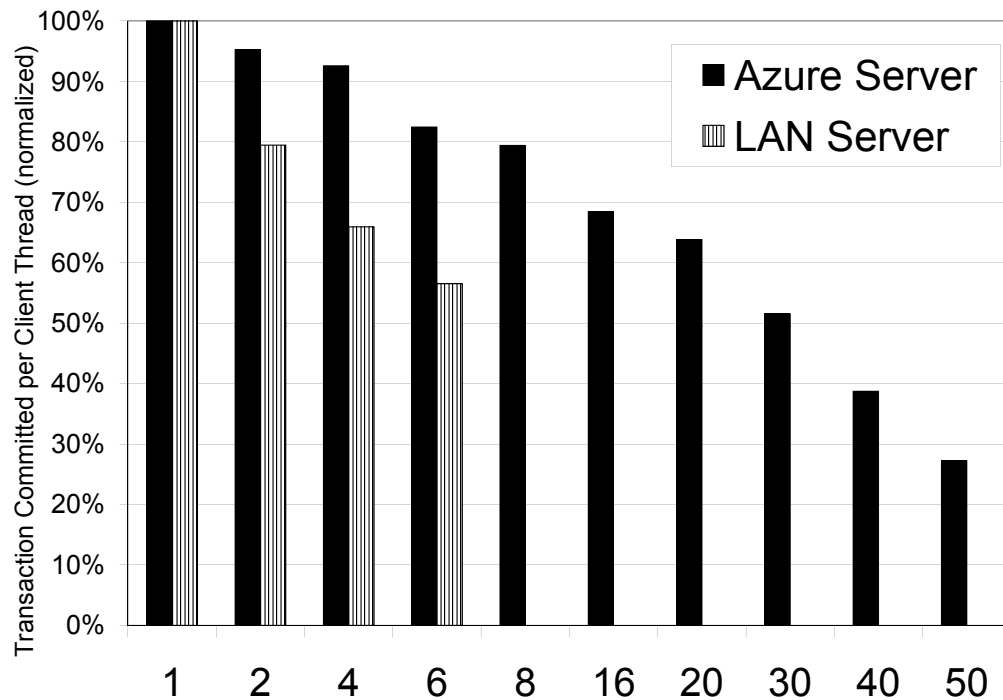


Figure 3.9: Transactions successfully executed as a function of the number of concurrent TPC-E clients.

database concurrently, both SQL Azure Database and SQL Server 2008 start rejecting the incoming connections to the database. We present some data related to this phenomenon in Figure 3.9. Here we present, for each thread, the normalized number of transactions that successfully commit. For example, each of the 64 concurrent cloud clients commits 17% of the transactions that 1 single client commits in the same time interval. Thus, the amount of work done is equivalent to 11 ideal concurrent clients ($64 * 17\% = 1088\%$). Again, the local server performs worse than the cloud server in this metric too, where 6 concurrent local clients show the same performance degradation as 20 to 30 concurrent cloud clients.

With these results in hand, one may argue that such performance degradation even at a relatively low number of clients (64) invalidates the scalability promises of the cloud. We should factor into account that every application that relies on a central server is going to hit a limit. Furthermore, for complex OLTP such as the TPCE benchmark this limit is going to be lower, as we have seen.

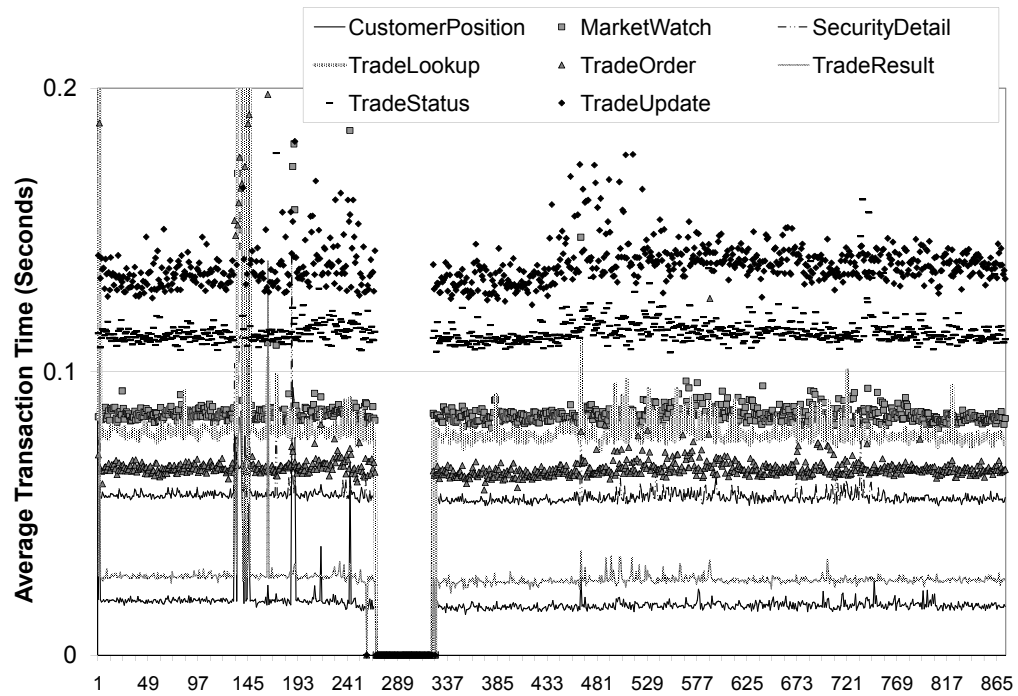


Figure 3.10: Performance of each TPC-E transaction over time in seconds, where the TPCE client runs in Windows Azure.

The system designer should take those limits into account, and design the application and databases accordingly. For example, we could have had partitioned our database into different databases, so several servers would share the load. Data partition is completely application dependent, though, so it falls outside the scope of our analysis.

3.3.3 Availability

One of the central promises of the SQL Azure Database is high availability. It is a common claim that the automatic management and replication of the database server in Windows Azure is a superior alternative to a local in-house solution. We can automatically deploy new instances of databases in the cloud in a matter of seconds, which do not require setups, patches or update downtimes. Although these instances are available, there are no claims about the performance of the database server over time. Thus, we have scheduled a modified TPC-E client which runs locally and queries the SQL Azure

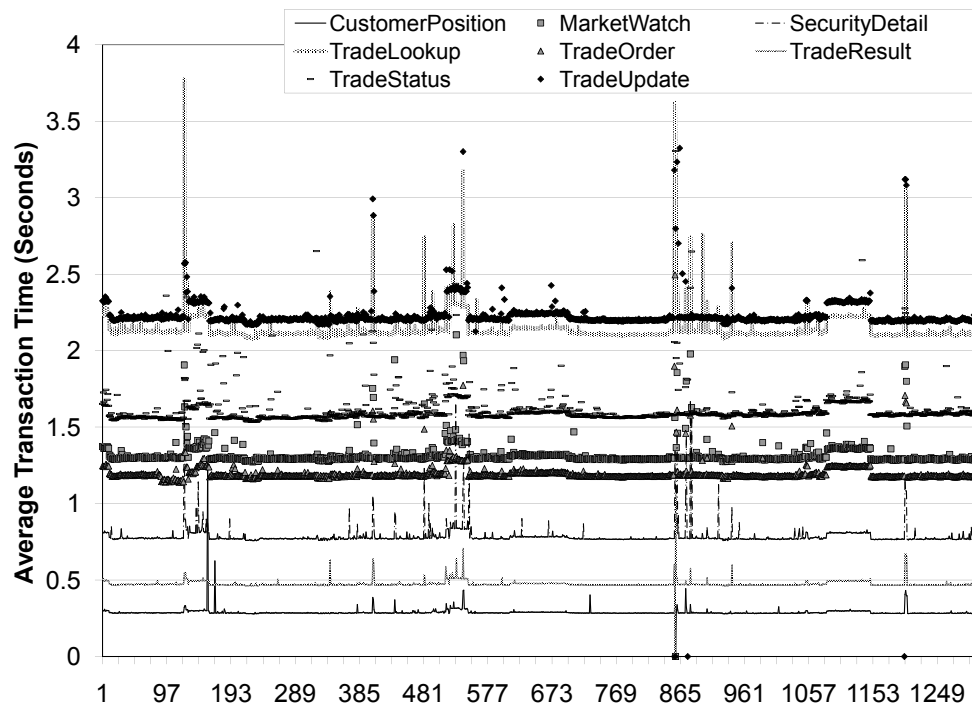


Figure 3.11: Performance of each TPC-E transaction over time in seconds, where the TPCE client runs in our LAN.

Database for 20 minutes every hour. This client has been modified to be completely deterministic, because in such a short amount of time the probabilistic nature of the TPC-E benchmark distorts the results. Thus, we perform the exact same read only queries every hour. Figure 3.10 and Figure 3.11 summarize the results over several weeks: we started a TPCE client on our LAN (Figure 3.11) on November 25th, 2009 that run till January 23rd, 2010; we also started a TPCE client on a worker Role in Windows Azure (Figure 3.10) on December 18th, 2009 which run till January 31st, 2010.

The results are, in general, stable. There are, however, some spikes in the graph that represent instances in which SQL Azure Database has been slower than usual to respond to our client, although these peaks are rather unusual. Also, our client in Windows Azure failed to contact the database server, and for several days the Windows Azure fabric was unable to recycle it correctly. This period of time can be seen in Figure 3.10 between the 260 (December 29th) and 330 data points. Manual redeployment of the TPCE client fixed this error, but the developers are advised to monitor

continuously the health of each instance running in Windows Azure, even if it is apparently running correctly (i.e., generating output blobs).

3.4 Recommendations

In this section, we present our recommendation for developers and users of the Windows Azure cloud. These recommendations are based upon our experimental results and our experience developing scientific applications for Windows Azure. These results are consistent with the findings of a independent study done on Windows Azure by Microsoft Research [68].

3.4.1 Windows Azure storage services

In order to improve performance users should choose a blob storage hierarchy so that accesses to the blobs are spread into as many storage partitions as possible. The data transfer throughput is sensitive to the number of concurrent accesses to a single partition. For example, with 32 clients concurrently accessing the blobs (no prefix used in the blob names) under the same container, the per-client data transfer rate can be degraded as much as 50% of that when only a single client is accessing the blobs.

The blob storage download bandwidth, when accessed from small instance types, is limited by the client's bandwidth for small numbers of concurrent clients (1 to 8); we saw a 100 Mbit/s (around 13 MB/s) limitation. We have observed a per-client bandwidth drop of approximately 1.5 MB/s when we doubled the number of concurrent clients. The maximum service-side bandwidth achievable against a single blob for a high number of concurrent clients is limited to approximately 400 MB/s (just about what we would expect from three 1 Gb/s links if a blob is triple-replicated). Therefore, we recommend using some extra data caching mechanisms on the worker role level to expand the per-client bandwidth limit, and creating data replications on the blob storage to expand the server-side bandwidth limit.

In order to get the best performance out of the Table service, the table entities should be accessed by using partition keys and row keys only. Particularly, users should avoid querying tables using

property filters under performance-critical or large concurrency circumstances. Currently, all tables are indexed on the PartitionKey and RowKey properties of each entity, but creating an index on any other properties cannot be specified. Under high concurrency circumstances, situations become even worse. In one of our experiments, over a half of the 32 concurrent clients got time-out exceptions instead of correct results when querying the same table partition (with 220000 entities pre-populated) using property filters.

Table storage is not the same as a relational database and schemas/designs that work well in an RDBMS are typically not efficient in the Table Storage architecture. Replicating data between tables rather than doing a “join” type operation in the client is often preferable as it minimizes calls to the Table Service.

Queues can be used for storage of many small objects as long as an approximate FIFO model is sufficient. We have not observed that queue performance for a single client is dependent on queue size; we found consistent Add, Peek, and Receive operation performance from queues ranging in size from 200K messages to 2M messages.

Multiple queues should be used for supporting many concurrent readers/writers as we found that performance degraded as concurrent readers and/or writers were added, but each client obtained on average more than 10 operations per second for message sizes of 512 bytes to 8 KB for up to 32 writers. With 16 or fewer writers each client obtained 15-20 ops/sec. We also found that message retrieval was more affected by concurrency than message put operations so users cannot assume similar scale at each end of the queue.

3.4.2 Dynamic scaling

Dynamically adding VMs to a deployment at runtime is a useful feature of Windows Azure enabling dynamic load matching, but you should be aware that it often takes on the order of 10 minutes from the time of the request until the instance is integrated into the deployment. If fast scaling out is important, hot-standbys may be required if a 10 minute delay is not acceptable, although this option would incur a higher economic cost. Additionally, web roles took, on average, 60 seconds longer to

come up for small instance types, and about 30 seconds longer for medium to extra large instance sizes.

3.4.3 Windows Azure SQL services

Users should expect a single Windows Azure-based client accessing SQL Azure to take about twice as long as a single local-enterprise client accessing SQL Server within the enterprise (commodity hardware). Therefore, if speed is a chief concern and the number of concurrent clients is expected to be small, a local deployment (with local clients) will obtain the better performance. The opaque and potentially changing nature of the Cloud prevents us from determining exactly why there was a 2X slowdown.

We found that the throughput of a single large database instance (10GB max size) peaks at 30 concurrent clients running continuous transactions tested using OLTP-style TCPE database benchmark. The corresponding peak was seen at 6 concurrent clients in our LAN testing. In that case we experience 50% transaction failure and the average transaction completion time, for those that did complete, was 65% longer than that of a single client. For 8 concurrent clients we found a reasonable 15% transaction completion time increase. Use these numbers and the graphs presented in Section 5 as a general guide to concurrency limits in SQL Azure.

We did, however, find that performance over time was consistent, although there are rare occasions (less than 1.6% occurrence for 50% slowdown or worse, 0.3% for 2x slowdown or worse) where performance degraded significantly. Therefore, we have seen no reason to provision assuming a worst-case behavior in SQL Azure that is significantly worse than average-case behavior.

3.4.4 Testing and development

Development on the Windows Azure Development Fabric & Development Storage should be approached with realistic expectations. The Windows Azure Development Fabric & Development Storage provides the ability to test connectivity and functional correctness of your own code, but only under limited concurrency. Additionally, the behavior of some SDK components may not match

the deployment environment. One example is the load balancing algorithm: the Development Fabric appears to load-balance based on each instance's outstanding requests and does round-robin on equivalently loaded instances. In the full deployment environment the load balancer appears to utilize random scheduling amongst web role instances. Differences like these can make debugging deployed applications difficult if users assume the same behavior as seen in the smaller-scale local SDK environment.

3.5 Conclusion

In this chapter we have presented the results from performance evaluation experiments, which we have conducted on Windows Azure. We have shown an exhaustive evaluation of each of the integral parts of the platform: virtual machines, storage services (Table, Blob and Queue) and SQL services. Based on these experiments, we also provide our performance-related recommendations for users of the Windows Azure platform.

These cloud services are the building blocks for cloud applications, and are usually presented to the user as a black box, with no performance guarantees. Our main focus is to provide the community with performance information and concrete recommendations that help the design and development of scalable cloud applications. We have used these results as the foundation of our description of the cloud storage capabilities, which is described in the next chapter.

Chapter 4

Cloud storage capabilities and data requirements

In this chapter we focus on the storage of data in cloud computing. Each cloud provider offers different storage abstractions: Amazon provides the S3, EBS, SimpleDB and Relational Database storage services [2]; similarly Windows Azure provides the Blob, Table, Drive and SQL Azure services [3]. For local storage users have traditionally used NFS drives and sometimes high performance distributed systems such as the GPFS and Hadoop. In this new context, with multiple and very different options for storage, users are facing a daunting task when trying to select a storage service. There are multiple aspects that have to be taken into account: performance, cost, specific features, long term management issues, etc.

For example, a user may decide to store some data in the Blob service of the Windows Azure cloud; read throughput is strongly dependent on the number of concurrent clients and going from 1 client to 11 concurrent ones the throughput per client decreases by 40% [7]. Application performance is affected by the selection of the storage and several variables (number of clients, size of data and requests, etc.). Users would also like to select the cheapest storage service that meets the data requirements: storing data in Amazon's Reduced Redundancy Storage (RRS) is cheaper than the regular service, but comes at the cost of reduced durability. Amazon's RRS is appropriate for

intermediate results and would reduce the storage bill for these data by 33%. Selecting a storage service can also be limited to specific characteristics, such as support for data versioning. To further complicate this selection, the requirements for existing data change over time and new datasets and applications are added to the cloud so choices need to be reconsidered periodically.

We believe that this complex decision should be automated, and we present our approach in this chapter. First, we create a novel XML schema to define the multiple characteristics and features of storage systems. This XML schema is capable of fully describing cloud and local storage systems. Our goal is to algorithmically process this information to match users' requirements. We present several descriptions of storage services, including the most commonly used services from Amazon and Windows Azure.

Second, we take several use cases drawn from our experience in the field of scientific computing and automatically match their storage needs to concrete storage services. The result of our system, at this stage, is a match of services and requirements, which ensures that data storage meets users' requirements (durability, availability, etc.), performance expectations (latency and throughput at scale) and provides cost estimates. In Chapter 5 we leverage this result to provide a globally optimal data allocation solution. However, the approach outlined in this chapter can also be useful on its own. For example, this approach could calculate cost savings by switching clouds or storage systems (with possible tradeoffs) or assist with storage system selection in private cloud deployments. A migration from the public Amazon cloud to a private Eucalyptus deployment would usually involve the movement of data out of the Amazon cloud. However, we find that for one of our use cases continuing using the SimpleDB service from our local cluster in Virginia offers good performance and the data transfer and storage cost is negligible.

We finish this chapter with the evaluation of two aspects of our approach: its extensibility, and the wall clock time spent processing each use case. Since cloud computing is an evolving field cloud providers will change existing storage systems and release new ones. We present the coding effort required for each data requirement that the user can check against the description of the storage systems; in our example *Datacenter location* is coded in under 25 lines of C# code. For each of our

use cases we present time measurements processing each cloud provider (Amazon, Windows Azure and local) and the total wall clock time; all measurements fall below 70 ms.

4.1 Targeted storage systems

Until now, scientists usually have had very limited options for data storage. A usual local cluster solution includes a small user directory (with backup), an NFS system that can hold several GB of data, and a big scratch temp folder. More advanced systems have also included a high performance parallel filesystem, such as GPFS [69]. In recent years cloud computing has burst onto the computing scene and has established itself as a viable alternative to customized HPC clusters for many users who do not have the resources (either time or money) to build, configure, and maintain a cluster on their own. Many eScience developers are increasingly looking to create data-intensive applications [13, 14] that can take advantage of the pay-as-you-go cost model. In this chapter we will focus mainly on cloud storage systems although we also discuss the traditional storage systems for local clusters.

The first cloud provider that we have examined is Amazon. Amazon's pioneering effort in the area [2], API compatibility with other projects (Eucalyptus [5], OpenNebula [30]), wide customer base and the widest and most mature offer of cloud services make Amazon EC2 our primary target for cloud providers. Within the Amazon cloud we consider the following storage services: S3, EBS, SimpleDB and Relational Database. S3 provides storage for objects of a wide range of sizes (up to 5 TB), which are organized into buckets. The most closely related interface to S3 is the traditional directory/file interface. EBS offers a device type interface, in which a virtual hard drive (formatted with a filesystem) can be attached to a Virtual Machine running on the Amazon cloud. SimpleDB is based on tables that store items composed by attribute/value pairs. Although SimpleDB does not support a rich SQL interface its potential for scalability is superior. Finally, the Relational Database Service is essentially a MySQL database running on the cloud. Each storage service is offered in different regions that include the United States, Europe and Asia. In summary, Amazon offers very different options for data storage; each option is best suited to certain tasks.

We have also considered Microsoft’s Windows Azure cloud which, although it provides the Platform as a Service (PaaS) abstraction, gives much flexibility in terms of data storage. The storage offer in Windows Azure [70] is very similar to Amazon: Blobs are similar to S3, Azure Drives to EBS, SQL Azure to Relational Database Service and Tables to SimpleDB. Even though the storage abstraction are essentially the same, the implementation details differ. For example, Amazon S3 offers both regular and reduced redundancy storage unlike Windows Azure Blob. Thus an application can make the explicit tradeoff of cost and durability in Amazon to save money; this tradeoff is not possible for Windows Azure Blob. On the other hand, Windows Azure Blob comes in two flavors: page and block. Streaming data are best kept in block blobs, while random accessible data in page blobs. On top of the differences between features there are performance and cost differences that, for some given application’s requirements, can tip the balance in favor of one of these cloud providers.

In addition to the Amazon EC2 and Windows Azure cloud platforms we have included several local storage systems: users directories and scratch folders mounted over NFS in a local cluster, and a local Hadoop deployment. We believe that the inclusion of storage systems from these three sources covers the vast majority of storage systems available (SQL, NoSQL -scalable tables-, block devices, files, etc.). We also take into account that storage systems are continuously evolving: different aspects related to extensibility are introduced in the next sections. In summary, we try to target a broad range of storage systems so that our system does not artificially limit the number of possibilities presented to the user.

4.2 XML Schema

In this section we present the general structure of the XML schema used to describe the storage systems supported by the different cloud providers. We also introduce the types declared and focus our attention on a couple of examples. The complete schema is readily available on our website¹: it currently features 54 complex types in over 500 lines. Figure 4.1 shows the most important elements; the global element is the `CloudProvider` and the second level is the `StorageServices` element. Each

¹<http://www.cs.virginia.edu/~ar5je/SCPaper.html>

```

<xsd:element name="CloudProvider" '
  type="tns:CloudProviderType"/>
<xsd:complexType name="CloudProviderType">
  <xsd:element name="StorageServices">
    <xsd:element name="StorageService">
      <xsd:element name="Regions">
        <xsd:element name="Cost">
          <xsd:element name="Performance">
            <xsd:element name="StorageAbstraction">
              <xsd:element name="Container">
                <xsd:element name="Object">
<xsd:/complexType>

```

Figure 4.1: The hierarchical organization of the most important elements in the XML schema. The global element is the cloud provider, which offers several storage services, each one representing a storage abstraction, across multiple regions that vary on cost and performance.

```

<xsd:element name="Object">
<xsd:complexType>
  <xsd:sequence>
    <xsd:element name="AccessControl" type="tns:AccessControlType"
      minOccurs="0"/>
    <xsd:element name="Interface" type="tns:InterfaceType" maxOccurs="1"/>
    <xsd:element name="Metadata" type="tns:MetadataType" minOccurs="0"
      maxOccurs="1"/>
    <xsd:element name="Data" minOccurs="0" maxOccurs="1">
      <xsd:complexType> <xsd:choice>
        <xsd:element name="AttributeValue" > <xsd:complexType>
          <xsd:attribute name="AttributeNameLength" type="xsd:integer"/>
          <xsd:attribute name="AttributeValueLength" type="xsd:integer"/>
        </xsd:complexType></xsd:element>
        <xsd:element name="Stream" type="xsd:string"/>
        <xsd:element name="RandomAccess" type="xsd:string"/> </xsd:choice>
        <xsd:attribute name="Formats" type="xsd:string"/>
        <xsd:attribute name="DaysToExpiration" type="xsd:float"/>
        <xsd:attribute name="ReadOnly" type="xsd:boolean"/>
      </xsd:complexType></xsd:element></xsd:sequence>
    <xsd:attribute name="ID" type="xsd:string"/>
    <xsd:attribute name="Name" type="xsd:string"/>
    <xsd:attribute name="Description" type="xsd:string"/>
    <xsd:attribute name="NamingRegularExpression" type="xsd:string"/>
    <xsd:attribute name="CreationDate" type="xsd:boolean"/>
    <xsd:attribute name="ModificationDate" type="xsd:boolean"/>
    <xsd:attribute name="MaxSizeNumber" type="xsd:integer"/>
    <xsd:attribute name="MaxSizeKB" type="xsd:float"/>
  </xsd:complexType></xsd:element>

```

Figure 4.2: The type definition of the Object element. An object can represent different entities such as a blob, a file or a table item (set of attribute/value pairs).

```

<Object ID='AZURE_BLOB_PAGE' Name='Windows Azure Page Blob'
  Description='The Blob ...'
  NamingRegularExpression='^(?![0-9]+$)(?!-)[a-zA-Z0-9-]{,63}(&lt;1t!-)&lt;1$'
  ModificationDate='true' CreationDate='false' MaxSizeKB='1073741824'>
  <Interface>
    <CustomInterface RandomAccess='true'>
      <Delete>Delete Blob</Delete>
      <Download>Get Blob</Download>
      <Upload>Put Blob</Upload>
      <CreateSnapshot>Snapshot Blob</CreateSnapshot>
      <ListParts>Get Page Regions</ListParts>
      <UploadPart>Put Page</UploadPart>
      <Lease Duration='60' API='Lease Blob'/>
      <Copy>Copy Blob</Copy>
    </CustomInterface>
  </Interface>
  <Metadata>
    <MetadataInterface>
      <CustomInterface>
        <Download>GetBlobMetadata; GetBlobProperties</Download>
        <Upload>SetBlobMetadata; SetBlobProperties</Upload>
      </CustomInterface>
    </MetadataInterface>
    <MetadataSet type='SystemMetadata' abstraction='ValuePair'/>
    <MetadataSet type='UserMetadata' abstraction='ValuePair'/>
  </Metadata>
  <Data DaysToExpiration='0' Formats='binary;text' ReadOnly='false'>
    <RandomAccess/>
  </Data>
</Object>

```

Figure 4.3: The Object element that describes the paged Blob storage service in the Windows Azure Platform.

StorageService element represents a certain Storage Abstraction that is offered in several Regions. An example of a Storage Abstraction is the Windows Azure Table. In general, we find that each storage abstraction can be thought of as a set of containers which store objects. For example, a container may be a table (Windows Azure Table), a bucket (Amazon S3) or a directory structure (NFS). The respective contained objects are items with attribute/value pairs, S3 objects and files. This part of the schema focuses on the functional description of the storage system, that is, the characteristics and features that appear on the service documentation.

The second child of the StorageService element is the Regions element. Inside this element we find the datacenters where this service is being offered, each one with its cost and its performance. Non-functional characteristics like performance vary from region to region; it is common to have

different costs depending on the location of the datacenter because of the variation in electricity prices, labor costs, regulation and taxation, etc. We provide the user with multiple complex types to express the different cost models of clouds: `StorageCost` (GB per month), `DataTransferCost` (in and out), `RequestCost` (measured in number and type of request), `QueryProcessingCost` (measured in compute time to process a query), `OffNetworkDataTransferCost` (usually a flat fee for processing a hard drive), `HourlyCost` (usage cost for a certain resource, such a database server) and `ReservationCost` (usage prepayment for reservation of a resource during a set period of time).

The `Performance` element allows us to express performance characteristics for every datacenter. The two that we have used are latency and throughput. In general, we can declare a `Measurement` of a variable (latency in ms) for an operation (read) and specify the details in several ways: as a simple scalar number, as a polynomial approximation, as a histogram or as a set of sample measurements. Polynomials, histograms and sample sets are based on one or more variables; in our research we have found the number of concurrent clients as the most useful one because of the performance variability. As the number of concurrent clients increases many storage services' performance diminishes; users need to take this into account during the design of cloud applications. Other variables that we could use are, for example, size in KB of the request, size of the object and number of items in the container. The focus on this section of the schema is to provide enough information so our matching algorithm can make a good estimate of the performance of the user's application on this datacenter. The information in this section may originate from our experiments (Chapter 3) or from websites like Azure Scope [68], which provided up-to-date benchmark results for the Windows Azure platform.

Figure 4.2 shows the type definition details for the `Object` element. The attributes of each object, aside from the common ones (ID, Name, Description), describe some features, such as support for creation and modification dates; and limitations of the service, such as the maximum number of sub-elements (for attribute/value pairs) or the maximum size. The possible child elements are: `AccessControl`, `Interface`, `Metadata` and `Data`. The `AccessControl` element can be used to describe the multiple systems supported by the storage service: from simple UNIX type permission bits to more elaborated systems such as access control lists and custom access policies languages (both

supported in Amazon S3). The Interface element enables us to list every operation supported: from simple create/delete and upload/download to creating snapshots or acquiring a lease; additionally we can include the consistency options and transaction support. The Metadata element contains information about the supported metadata formats (if any), most commonly a set of attributes with string values. The Data element could be a simple stream of bytes with possibly random access support (blobs) or a set of attributes (SimpleDB or Windows Azure Table items).

Finally, we acknowledge that cloud computing is a rapidly evolving field that can make our schema fall out of sync with the storage services. Even though we have included all the features and characteristics of the Amazon and Windows Azure clouds anytime a new feature could be announced that can not be expressed with our schema. Our main response to this challenge is to plan for extensibility: features should be able to be easily and fully included in our system. Regarding the XML schema, this requirement translates in the addition of *xsd:anyAttribute* and *xsd:any* so that new elements and attributes can be included in XML files describing storage systems while conforming to our schema. Correctly processing these new elements is more difficult; further sections in this chapter will address this issue.

4.3 Descriptions of storage systems

In this section we discuss the actual XML representation of some of the most popular storage systems from the Amazon and Windows Azure clouds. We also give a high level description of how we process these input files. Figure 4.3 is a snippet of the XML file describing the Windows Azure platform that belongs to the paged Blob service. (Figure 4.2 is the corresponding section of the schema). This piece of code lists all the valid operations on blobs, as well as the support for attribute/value metadata and the type of data supported (both binary and text that can be randomly accessed). The AccessControl element is absent in this case, since Windows Azure does not allow access control lists at the blob level, but rather at the container level. The rest of the XML file for the Windows Azure platform, as well as the Amazon platform, can be downloaded from our website².

²<http://www.cs.virginia.edu/~ar5je/SCPaper.html>

Table 4.1: Average number of lines of XML code needed to describe a storage system, divided by schema elements. Performance and Cost are child elements of Region. Region and StorageAbstraction are children of StorageService.

Cloud Provider	StorageAbstraction	Region	Cost	Performance	StorageService	Sum all services
Windows Azure	80	70	6	62	731	3673
Amazon	94	36	17	13	246	1498
Local Cluster	72	23	4	15	100	315

We present a high level view of the files in Table 4.1, where we present some statistics about the different parts of our schema. The description of the StorageAbstraction is a manual and involved process: it requires reading the full documentation of each storage service. The output in terms of lines of code is between 72 and 94, which leads us to believe that any future modifications to this part would be easy to implement. The description for each region (or datacenter) is not larger than the descriptions of the abstraction. However, the number of regions in which a storage service is offered is a significant factor in the length of the file: we count 9 different region options for Windows Azure and 4 for Amazon. Here we are not concerned with the length of the file, because after all it is supposed to be part of program input. What we are trying to highlight is the expected effort to create and maintain these files with accurate and up-to-date information about the cloud providers.

The description of the storage abstraction of the schema is a one-time effort only, plus the corresponding updates whenever a cloud provider changes the storage service to modify some features or add new ones (the latter one being much more likely). More prone to change is the information regarding each region. Cost do change over time, albeit slowly. More commonly there are offers and deals that expire such as temporary free data input to a certain cloud for a couple of months. Cloud providers could publish this cost information in a machine accessible way or perhaps there is the possibility of automatically parsing the HTML documentation. The performance measurements are the most variable part of the schema and we believe that automation is very much possible. We have already mentioned Azure Scope [68], which used to provide up-to-date information of the performance of the storage services in the Windows Azure cloud. Using this website (now unavailable) we were able to provide more accurate information about the expected latency and throughput of the different services, under different conditions (size of the data, number of batch requests, number of

objects already in the container, etc.). A similar website, such as CloudHarmony [71], or a benchmark running on behalf of the user could provide importable performance information for other target cloud systems so there is no need for human intervention; in this chapter we used the performance information from the Azure Scope website for Windows Azure and from our own micro-benchmarks for the Amazon cloud and the local cluster.

4.4 Use cases

In this section we present several use cases that, in our opinion, reflect common situations that scientists and other cloud users face. The selection of an storage system has many implications for cloud applications; we provide valuable information about performance and cost expectations evaluating different cloud providers and recommend for each dataset a certain storage service. Currently our storage selection application presents a standard Windows Forms interface to the user, where each of the following use cases is a separate section (tab). The tables included in this section represent what the user sees on the screen.

We would also like to note that cloud computing is an evolving field and therefore any decisions from our experiments, which are presented in the following sections, could change in the future if the cost and performance of cloud services change. Information on cloud services is essential and we believe that our XML descriptions of the storage services should be automatically updated, via cloud benchmarks run by the user or via a service that provides this information (we have already mentioned Azure Scope [68] and CloudHarmony[71]).

4.4.1 Design of an application

The first of our use cases focuses on the choices that cloud users make during the application design. At this stage we assume that we have a good estimation of the data requirements of the application: for each dataset we have the size, the required access latency and throughput and the number of concurrent clients. Our application takes this information and together with our description of the

storage capabilities of the different clouds produces an assignment of datasets to storage services. For example, let's consider the following list of datasets and characteristics for a climate simulator:

- Climate measurements come from a satellite feed. It is very important to not lose the data. The size of the data is in the order of several GBs, it is read-only and concurrent access is limited to up to 10 concurrent clients.
- The application follows a bag-of-tasks model where intermediate results, in the range of 10s of MBs, are continuously being produced and consumed for up to 100 concurrent clients. High access latency or low throughput could affect the overall application's performance.
- The final result for each run is a single file, in the order of MBs, which is read only and is globally accessible for any scientist around the globe to download.

The storage service choices made by our application can be summarized in Table 4.2. For the Windows Azure cloud, the paged Blob service may store the first dataset, where the expected throughput is almost 14 MB/sec and the latency 225 ms. For the second dataset there is not a single storage service that would meet all the requirements. The paged Blob service does not offer very low access latency (205 ms expected). The Table service does have low latency (15 ms) but cannot store items greater than 1MB. Thus, the user would have to judge whether the application may tolerate higher access latency or whether the data could be partitioned. In addition to the user choosing from several storage services that do not meet all the requirements there could be cases where there is missing information. For example, latency benchmarks may not adequately cover the current use case: we have measurements with 10 concurrent clients but not with 100. In this case we choose to display a warning message; the final decision is the user's. Finally, both the paged Blob and the block Blob can meet the requirements for the third dataset.

Similar results are presented for the Amazon cloud. Even in this example with simple user requirements, we believe that our application can provide value to the cloud user by highlighting the differences between latency and bandwidth for each cloud, as well as estimating the storage cost in order to guide the application design process.

Table 4.2: Storage services recommendations for the datasets in our first use case by cloud platform. An * indicates storage systems that do not meet at least one user requirement.

Dataset	Amazon	Windows Azure	Local Cluster
Satellite Data	S3	Page Blob	Hadoop*, NFS*
Intermediate Results	S3 RRS*, SimpleDB*	Page Blob*, Table*	NFS*
Experimental Results	S3	Page Blob, Block Blob	NFS

Table 4.3: Storage services recommendations for our second use case, including monthly savings and tradeoffs.

Current Service	Amazon Service Region	Service Recommendation Cloud	Service Recommendation Service	Service Recommendation Region	Savings	Pros	Cons
S3	US CA	Azure	Page Blob	US	\$11	2.09x better latency	
S3	US CA	Azure	Block Blob	US	\$11	2.07x better latency	
S3	US CA	Amazon	S3	US	\$36		
S3	US CA	Amazon	S3 RRS	US	\$153.5		0.0099999% less durability
S3	US CA	Amazon	S3 RRS	US CA	\$127.5		0.0099999% less durability
S3	US CA	Local	NFS	US	\$407.5	117.6x better latency	0.499999% less durability
SimpleDB	US CA	Amazon	SimpleDB	US	\$.2		
RDS	US CA	Amazon	RDS	US	\$92		
RDS	US CA	Azure	SQL	US	\$130	1.31x better latency	

4.4.2 Cost savings analysis

In this use case we analyze an existing cloud application that is currently running in a cloud provider. As an example we will use the Amazon cloud. The developers and users of the application would like to find out if they could lower the monthly bill by switching storage services, within the same cloud or from another cloud provider. For this example, let's consider the following usage of the Amazon storage services:

- S3 holds a total of 2.5 TB of data, which are located in one bucket in the US Northern California Region. The number of blobs stored is 10000.

- SimpleDB hosts data one data domain which stores around 250,000 elements (8 GB).
- Relational Database Service stores a 4 GB database. The average number of requests is 5,000 per day.

Table 4.3 shows the results. For each storage service we present several alternatives that are cheaper. The storage service with more alternatives is S3 and some of them within Amazon. We can cut costs by changing to a cheaper region in the US or by changing to the Reduced Redundancy Storage. While in a cheaper region S3 continues to offer essentially the same level of service, RRS has a tradeoff because of the reduced durability. Also, the Windows Azure cloud provides a service slightly cheaper and with better access latency. Storing data in a local cluster is “free” and has very good latency (local LAN), but is less durable. For the SimpleDB service there are essentially no cheaper options (although there exists equivalent alternatives such as Windows Azure Table or SimpleDB in other regions with a similar cost). Finally, SQL Azure is a cheaper alternative to RDS with better latency. The user, given this information, can evaluate whether the tradeoffs are possible and the savings are worth it.

4.4.3 Cost and performance estimation

There are many applications in which a change of the backend storage services is difficult: cost of re-programming, no access to source code, scientists have credits for use with a certain cloud provider, etc. But even when the storage services will continue to be the same ones, we believe that there is value in giving the application user estimations for future costs and performance. Given the resource usage detailed in the previous Section 4.4.2, we shall consider the following trends, this time using the Windows Azure cloud:

- The storage and number of requests in both Blob, Table and SQL Azure will increase at three following rates: 2%, 5% and 15% each month.
- The cloud user would like to consider three different scenarios for concurrent access to the storage services: 10, 25 and 60 clients accessing the Blob and Table data.

In our application we present a table of monthly costs for a certain period of time -a year by default- and for every storage service. Users can see the evolution of the storage costs under different scenarios by changing the assumptions (data growth rates and number of clients). For some cloud applications growth could follow a predictable pattern, for example an application that is only used within a closed community (research lab, university). Other cloud applications, which are open to global users, could scale up suddenly if they become popular. In the first case, we can offer the user with the value of the future bill from the cloud provider. For the second case, our application can provide the user with cost estimations under several scenarios. Aside from cost, we can also generate estimations for the performance metrics included in our XML schema, if they are available. For example, using the Windows Azure Table service we estimate that with 10 concurrent clients the per client throughput will be 31 items/sec. This number does not vary if we increase the number of concurrent clients up to 60. For the Blob service, on the other hand, expected throughput is 28 MB/sec with 10 clients, 25 MB/sec with 25 clients and 17 MB/sec with 60 clients. In summary, given the information collected from these different cloud storage services we can give the users good cost and performance estimates for several scenarios.

4.4.4 Amazon EC2 to Eucalyptus

Cloud computing software such as Eucalyptus [5] and OpenNebula [6] can be used to deploy on-premise (private) Infrastructure as a Service clouds. In this use case, we consider the following scenario: an application is being moved from the Amazon EC2 cloud to a local Eucalyptus deployment. What does the application user do with the current data? One option would be to continue using the Amazon cloud, but data transfer fees and increased latency can make this approach inviable. Another option is to deploy a local storage system: given the latency and throughput requirements we can choose a NFS installation, a Hadoop deployment or other local data storage systems. Since researchers have already developed private cloud systems that mirror the functionality of public clouds (Eucalyptus and Amazon; Hadoop and Google's MapReduce and Big Table), it is not unreasonable to expect that, in the near future, an open implementation of storage services such as Blob and Table

Table 4.4: Storage recommendations for our fourth use case. Each current storage service used is compared with local storage services and the current storage service; computation is moved to the local cluster from Amazon EC2. Reflected costs include one time costs (transfer data to new service) as well as monthly costs.

Current	New	Latency	Throughput	One-time Cost	Monthly Cost
S3	NFS	>1ms	39.02 MB/sec	\$250	\$0
S3	Hadoop DFS	N/A	N/A	\$250	\$0
S3	GPFS	N/A	N/A	\$250	\$0
S3	S3 (no change)	205 ms	3.17 MB/sec	\$0	\$362.5
SimpleDB	MySQL	3.45 ms	288.8 items/sec	\$0.8	\$0
SimpleDB	SimpleDB (no change)	35.46 ms	28 items/sec	\$0	\$5.88
RDS	MySQL	>1 ms	14359 items/sec	\$0.7	\$0
RDS	RDS (no change)	13 ms	14172 items/sec	\$0	\$328.2

Current	New	Comments
S3	NFS	S3 offers 1% more durability (99.99999999%) NFS container capacity is 10 GB (2500 GB req.)
S3	Hadoop DFS	S3 offers 0.00099999% more durability (99.99999999%) Hadoop container capacity is 1024 GB (2500 GB req.) Hadoop does not support random access
S3	GPFS	S3 offers 0.00099999% more durability (99.99999999%)
S3	S3 (no change)	Data transfer fees incurred by each data access
SimpleDB	MySQL	SimpleDB offers 1% more durability (99.99999999%) Interface differences: SQLInterface and AttributeValue
SimpleDB	SimpleDB (no change)	Data transfer fees incurred by each data access
RDS	MySQL	RDS offers .5% more durability (99.5%)
RDS	RDS (no change)	Data transfer fees incurred by each data access

(SimpleDB) that would expand the options for local storage systems further. In this use case we continue with the usage example in Section 4.4.2 and recommend the best storage system to select in the cloud or deploy locally.

The output of our application is synthesized in Table 4.4. We compare each current storage service against local options and itself. In the latter case the client is moved from the Amazon cloud to the local cluster and our application considers the data transfer costs and the performance of the remote accesses. Given this information the user can decide where to move the data. For example, data currently being stored in S3 can be moved to a local NFS if durability is not a requirement. The available space is also very limited (10 GB), so a local deployment needs to expand the current storage capacity. Other options, such as Hadoop and GPFS will be specially suitable if the number of concurrent clients increases (here is assumed to be at most 10 for each file/blob). Hadoop is closely related to the MapReduce application model; this special access interface differs from a randomly

accessible file in UNIX and it must be taken into account. As our last option, part of the data could be archival data that may continue in S3 since the increase in access latency is less important than the durability guarantees. The user would have to continue paying storage fees and data transfer fees (if the data needs to be retrieved).

SimpleDB is a more complex storage service that currently has no local counterpart: the most similar storage system would be a traditional relational database. The migration from SimpleDB to a relational database would imply the modification of the application access to the tables (or domains in SimpleDB). Keeping the data in SimpleDB may be the best course of action, especially if the latency increase does not impact global application performance and the data transfer cost is bearable. Since there are good and free solutions to relational databases RDS can be easily substituted by a local MySQL deployment. In summary, in a migration from a public cloud to a private cloud deployment there are many possible choices for data movement; our application can take the information about the capabilities of each storage system to guide the migration.

In this use case we have assumed “free” local storage services. However, there is no such thing as a free lunch; someone has to pay for it. It is a common case that scientists get financial support from grants and companies to buy their computational resources; the university provides the electricity and real estate. In this situation the scientists’ final bill is what we present in Table 4.4. This does not imply that we believe this to be true always: by modifying the storage cost values in our XML descriptions of the local storage we are able to express different cost models (the same ones from the public cloud providers) and accurately represent each local cluster.

4.5 Evaluation

4.5.1 Extensibility of schema and algorithms

We have already mentioned the extensibility of the XML schema; in this section we present an overview of the cost of extending our algorithms so they are able to process new elements and attributes in the XML descriptions of the cloud provider. We will consider the following example:

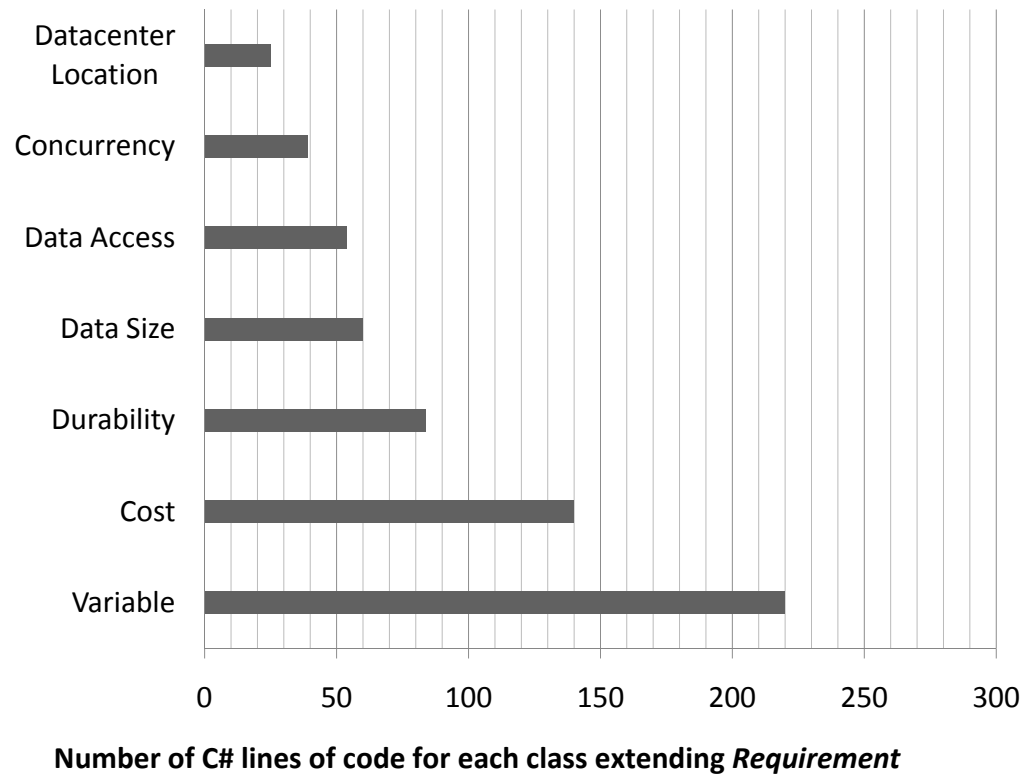


Figure 4.4: Number of C# lines of code for some of the data requirements' handling code in our application.

data durability. Durability is a common requirement and some cloud providers, such as Amazon, provides the actual number (99.99999999% for S3) that each storage service of the Amazon platform was designed to provide. In the simple case, in order for a cloud to meet the user's durability requirement we just have to find this attribute in the `StorageAbstraction` element and compare it to the user's input. However, some of the cloud providers, such as Windows Azure, do not provide this measurement and it is difficult to compare storage services or to evaluate them. Therefore, we have found the necessity of extending this simple durability requirement evaluation to a more complex one.

In order to better evaluate the durability of two storage services we have chosen to compare the number of replicas and its location (in the same datacenter or multi-region replication). If both storage services have equivalent levels of replication we consider them to have essentially the same durability (while this is not exactly true, we consider it to be a good estimate when we lack more

information about the storage service implementation). Every data requirement in our prototype application inherits from the *Requirement* class, whose method *match* is called when evaluating a certain storage service and region combination from a cloud provider. Within this method the programmer has access to the XML document (*CloudProvider* type). The return of this method (*MatchResult* type) tells us if the storage service matches the given requirement and any warnings or errors.

```
class Durability : Requirement
...
    override public Matchresult match(
        CloudProvider c, Matchresult r,
        string storageServiceID, string regionID)
```

We give the numbers of C# lines of code in Figure 4.4 to give the user a general overview of the programming cost of extending our application. Requirements that involve checking for simple elements and attributes, such as support for concurrency or a certain type of access (random, stream, attribute/value), can be coded in under 50 lines. More complex ones, such as calculating the cost of a certain variable (latency, throughput) require between 140 and 220 lines of code. Our implementation of the *Durability* class was done in 84 lines; we believe that the cost of extensibility is quite low since the size of the code is manageable and it is a one-time effort.

4.5.2 Use cases

Here we present the performance results of our application for each of the use cases introduced in Section 4.4. We run each use case 10 times, and Figure 4.5 shows the average wall clock time. For every use case our application finishes in less than 70 ms. We detail the time it takes to process the XML descriptions of each cloud provider in the same graph. Windows Azure takes longer than Amazon and Local because we were able to include more detailed performance information, thanks to the Azure Scope website. In our second use case (cost savings) we first calculate the cost for each storage service: if it is greater than the current service we skip to the next service on the list. Thus,

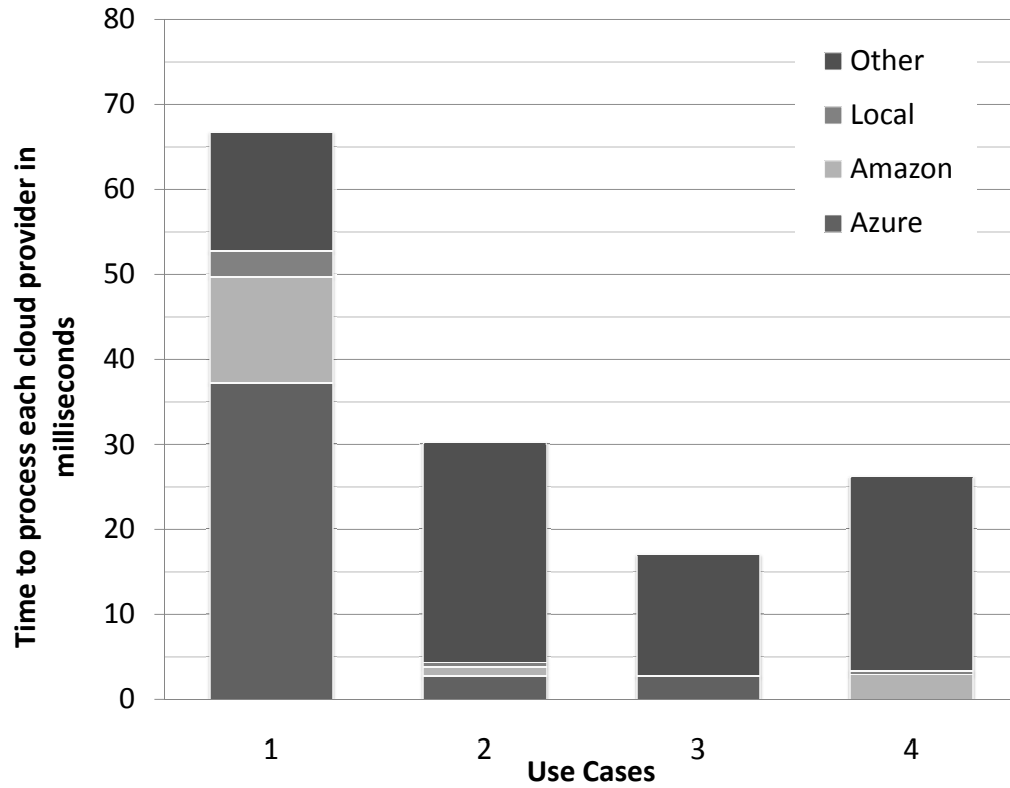


Figure 4.5: Wallclock application time for each of the use cases presented in Section 4.4 in milliseconds, divided into time spent processing each cloud provider and input/output processing (*Other*).

we do not need to process most of the services. Use case number 3 (performance and cost estimates) focuses exclusively on the Windows Azure cloud, as use case number 4 (Amazon to Eucalyptus) does with the Amazon cloud and the local cluster. The XML files are located in the hard drive; the application time incurred to load these files is included under the *Other* label together with input and output processing.

In the future, running times could become larger due to the following factors: a more detailed description of each service, an increase in the number of requirements processed, and a greater number of cloud providers and regions. An increase of details for the performance section is certainly possible and advisable: we can estimate better variables such as latency and throughput with more data samples from benchmarks. For example, our performance description for Windows Azure Table includes latency measurements that vary based on the number of concurrent clients. Additional data can reflect the performance impact of other factors such as entity size, number of entities per

table, etc. For our use cases we analyze requirements such as data size, durability, calculate cost and latency, etc. For any given community there will be additional data requirements to process; these requirements, however, would be essentially limited by the storage capabilities described and the number of requirements expressed by users. Finally, every region added will mean more processing for cost and performance calculations; every new cloud provider will have an impact similar to the ones detailed in Figure 4.5. Given the current performance results, we feel confident that our approach can accommodate these changes without impacting the user experience. For future work we are planning to include these software as a library that can be used by a higher level data management software to further automate the processing of data by cloud applications.

4.6 Conclusion

We have presented in this chapter an automated approach to the selection of cloud storage services that can meet the user's requirements. First, we have defined an XML schema that guides the description of the different capabilities of cloud storage systems. This schema is based on the documentation of different storage services and our experiences with cloud computing. We use this XML schema to provide descriptions for the storage services of Amazon, Windows Azure and some software that is commonly deployed in local clusters (NFS, Hadoop, MySQL, etc.).

Second, we have developed an application that processes these XML descriptions and attempts to match common data requirements from users to them. Our application is also able to provide cost and performance estimates. Both our XML schema and our application can be easily extended to describe new features and process new requirements. We have shown that, in less than 70 ms, we are able to recommend a list of storage services for a cloud application, estimate possible cost savings and tradeoffs by switching storage services, estimate storage costs and performance under different growth scenarios, and provide information to assist in the migration of cloud applications to private cloud deployments. This work is not limited to the use cases presented in this chapter, as we will see in Chapter 5. At this stage we can generate a list of storage systems per dataset in the application that meet the user's requirements and provide cost and performance estimates. In the

next chapter we will take these lists of compatible storage services and generate a globally optimal storage solution for all the datasets in a given application.

Chapter 5

Cloud data management

To decide which cloud platform is best for a new cloud application, we have found in our experience that typically a designer focuses on compute capabilities as the deciding factor. That is, after it is decided if it will be public-, private-, or hybrid-cloud, a high-level decision is made regarding PaaS vs. IaaS, and then a subsequent decision selects the particular platform within the class (e.g., IaaS and then Amazon EC2). However, we believe that data capabilities should be a first-class consideration when selecting a cloud platform, at the same level of importance as computation. Arguably computation is more flexible: a Windows application can run on a native Windows OS (local Windows HPC cluster or Windows Azure) or within a virtual machine (local Eucalyptus cluster or Amazon EC2). Storage services, on the other hand, present many different options whose capabilities are sometimes exclusive to a cloud provider; choices range from traditional files (NFS) and SQL databases in local clusters to a variety of cloud services (for Amazon there are S3, EBS, SimpleDB, RDS and ElastiCache).

In previous chapters we presented the first components of a system designed to help with the cloud storage service selection decision. The output of the previous components is, for each dataset, a list of storage services that meet the user's requirements, along with cost and performance estimates. The user could then choose a data allocation from these options, mainly by selecting the cloud provider with the best (based on cost or performance) storage options for all the datasets. This

choice meets all the user’s data, cost and performance requirements. There are, however, three important limitations: we rely on the user to make a choice from the list of storage options for each dataset; each dataset is analyzed in isolation so it is not obvious what the best global solution is; and the computational side (number of application runs, cost per hour, machine speed, etc.) is not taken into account. In this chapter (focused on the data side) and the following chapter (focused on the computational side) we address these limitations so we can produce a global data allocation solution that balances cost and performance of both storage and computation. If the best storage service for a single dataset resides in a cloud that does not have good choices for the rest of the application data, then we may arrive to a sub-optimal data allocation. As we will show, trying to find an optimal solution increases the complexity of the problem to NP hard. We provide a model for this data allocation problem and a software implementation that is both fast and scalable.

We will first present our problem model, whose solution represents the data allocation decision. Factors such as storage cost, compute cost, latency, and bandwidth are combined into an objective function that has to be minimized. The combination of this function and additional restrictions (linear constraints) forms an integer linear programming problem, which is NP hard. In order to solve this computationally hard problem we use a modern ILP solver (lpsolve); we show that, for simple use cases, a couple hundred milliseconds are enough to solve the problem instance. For a more complex application with 3 datasets and where each dataset could be matched to 48 possible storage systems the solver takes 1 second to solve the ILP problem. We show that even with an order of magnitude increase in the number of possible storage and compute services our approach is able to come up with an optimal data allocation within seconds.

We present two use cases for our system: BLAST [72] and MODIS Azure [4]. For BLAST we take our standard model and include one additional restriction: a monthly budget. Thus, the user may ask for the best data and compute allocation which fits her budget. Latency and bandwidth are not the only performance metrics that we have considered. For MODIS Azure we add job turnaround time as another factor in our objective function. In this case we try to make the allocation decisions that will minimize job turnaround time the most, while still meeting the data requirements and the

budget. For example, our storage and compute allocation for a budget of \$1,000 has an average turnaround time of 1.64 hours per job. Had the user selected a pure cloud allocation (Windows Azure) the monthly cost would have been over \$2,000. Conversely, a purely local solution would have been cheaper, but the turnaround time would have been 52% higher.

5.1 Integer linear programming

In linear programming there is a mathematical function to be optimized, subject to a set of linear constraints. The objective function to be optimized (either maximized or minimized) is a linear function that uses a set of unknown variables. If these variables can only take integer values then it is an integer linear programming problem. In mixed integer linear programming problems there is a mix of integer and real variables. The canonical form for a linear programs is:

$$\begin{aligned} &\mathbf{maximize} \quad c^T x \\ &\text{subject to} \quad Ax \leq b \quad (5.1) \\ &\text{and } x \geq 0 \end{aligned}$$

In practice, the ILP solvers allow users to express the linear program in more flexible terms. Internally the solver can convert some formulations into an equivalent problem in standard form. Objective function are defined as:

$$\mathbf{minimize} \quad c^T x \text{ or } \mathbf{maximize} \quad c^T x \quad (5.2)$$

The objective function in our problem formulations could be as simple as **minimize** *cost* or be expressed in more complex terms, as in equation 5.4. ILP solvers also allow users to add constraints of both type $Ax \leq b$ and type $Ax \geq b$. When in our work we introduce a constraint of type $Ax = b$ we are implying two different constraints $Ax \leq b$ and $Ax \geq b$. We introduce three different types of variables in our problem formulations, real, integer and binary (0 or 1). For each integer and real variable the ILP solver gives us three choices: without constraints, positive and ranged. Variables

without constraints are uncommon. A positive real variable maybe for example *cost*, since it can take any real positive value or zero. We make extensive use of ranged variables since we know the boundaries for many variables in advance. For example, if the variable $x_{0,0}$ represents the number of local cores running at job start time and there are only 16 cores available we implicitly include the constraints $x_{0,0} \geq 0$ and $x_{0,0} \leq 16$ in the problem formulation.

Mixed integer linear programming is an NP hard problem. That is, as the problem size increases a solution can not be reached in a reasonable amount of time. However, one of the central points of this dissertation is that the sizes of cloud resource allocation problems are small enough to be solved by the current generations of solvers. New heuristics have been proved to be useful to solve ILP problems and other NP hard ones such as boolean satisfiability. In our work we have picked lpsolve [63], an open source project that solves mixed and pure linear programming with support for integer, binary and semi-continuous variables. We have chosen this solver because it is free to use and available in our platform of choice (Windows with plug-ins for Visual Studio). The core algorithm is Branch-and-bound; however we do not interfere with the default settings. For us, lpsolve is a black box that provides us the solution. The main advantage of using an ILP solver is that the output is guaranteed to be optimal, that is, no other algorithm can do better (for this problem formulation).

5.2 Resource allocation problem model

In this section we describe our mathematical model used to express the data allocation problem in cloud computing. Our goal is the following one: to select the best storage systems which meet the user's data requirements and optimize cost and/or access latency/bandwidth. Recall that in Chapter 4 we developed a matching process whose inputs are a list of user's requirements and the storage services' capabilities. The output of this first stage is a list of compatible storage services for each dataset in the application; these lists constitute the input for our data allocation problem. We use integer linear programming to model this problem. The general idea is to include the cost, latency, and bandwidth as parameters in the objective function that needs to be minimized. Of the variables that we introduce, 0-1 integer variables tell us which storage systems will store which

datasets $(x_{i,j})$; the solution to the problem will be an optimal assignment of datasets to storage systems. We also introduce integer variables that represent the amount of computation required per month; the solution also yields an assignment of computation to cloud sites ($computation_k$). We use additional linear constraints to enforce different restrictions; for example, that each dataset is stored in at least one storage system and that each site can support the computations that access each dataset. A glossary of all the terms used in the equations in this section is shown in Table 5.1. This table gives the type, description and source (user input, part of the storage capabilities information and part of the solution) for each variable. The objective function is the following one, where each ω_i is the combination of a weight assigned by the user and a normalizing factor:

$$\begin{aligned} MIN(\omega_1 \times AverageStorageCost + \omega_2 \times AverageComputeCost \\ + \omega_3 \times AverageLatency + \omega_4 \times AverageBandwidth) \end{aligned} \quad (5.3)$$

We need to combine every term in a meaningful way. In order to evaluate cost (in dollars), latency (milliseconds), and bandwidth (MB/s), we consider the average over all the application datasets and normalize it. We normalize each parameter to the average calculated from all the cloud storage systems (optionally the user may provide their own). The user will give us the α_i , which represents the weight (between 0.0 and 1.0, totaling 1.0) for each term:

$$\begin{aligned} MIN(\frac{\alpha_1}{\text{average cost GB / month}} \times AverageStorageCost + \frac{\alpha_2}{\text{average cost per hour}} \times AverageComputeCost + \\ \frac{\alpha_3}{\text{average latency ms}} \times AverageLatency + \frac{\alpha_4}{\text{average bandwidth MB/s}} \times AverageBandwidth) \end{aligned} \quad (5.4)$$

For example a solution could have a normalized storage cost of 1.15 and a normalized latency of 0.95, meaning that the storage cost is on average 15% more expensive but latency is 5% better. The best solution is then determined by the α_i . It is possible to easily give the user more control

Table 5.1: Data management model variables

Name	Description	Source
$x_{i,j}$	Binary, allocation of dataset i in storage j	Solution
$y_{i,j,k}$	Integer, number of data transfers for dataset i from storage j to site k	Solution
$computation_k$	Integer, number of application runs at site k	Solution
$DatasetSize_i$	Float, size of dataset i in GB	User input
$DatasetRequest_i$	Float, number of monthly storage requests for dataset i	User input
$DatasetUsage_i$	Float, percentage of dataset i accessed by the average application run	User input
$ComputationLength$	Float, number of hours per application run	User input
$SiteCapacity_k$	Float, number of computational hours available per month at site k	User input
$CostTransferIN/OUT_{j,k}$	Float, cost in dollars for GB of data transfer IN/OUT storage j to site k	Storage Capabilities
$CostRequests_j$	Float, cost per request on storage j	Storage Capabilities
$CostHour_k$	Float, cost per compute hour in site k	Storage Capabilities
$CostStorage_j$	Float, cost per GB per month in storage j	Storage Capabilities
$Latency_{j,k}$	Float, latency in ms when accessing data in storage j from site k	Storage Capabilities
$Bandwidth_{j,k}$	Float, bandwidth in MB/sec when accessing data in storage j from site k	Storage Capabilities

by expanding the formula and introducing $\alpha_{i,j}$ (weights that depend on each parameter and the dataset). This way it would be possible to fine tuned each dataset in case there is one that is critical to the application flow, which could have, for example, a low latency requirement.

We will first expand the Average Storage Cost term, which represents the average monthly cost per GB of data stored:

$$AverageStorageCost = \frac{StorageCost + TransferCost + RequestCost}{Total\ Gigabytes\ Stored} \quad (5.5)$$

$$StorageCost = \sum_{i,j} x_{i,j} \times datasetSize_i \times costStorage_j \quad (5.6)$$

$$RequestCost = \sum_{i,j} x_{i,j} \times datasetRequests_i \times costRequest_j \quad (5.7)$$

$$TransferCost = \sum_{i,j,k} y_{i,j,k} \times datasetSize_i \times datasetUsage_i \times (costTransferIN_{j,k} + costTransferOUT_{j,k}) \quad (5.8)$$

In our actual implementation, these equations are more complex since the pricing structure for some cloud providers is not flat: there is layering pricing, monthly plans, special offers, etc. However, these equations reflect the most important parameters involved and we'll use them in our problem model description for simplicity. The calculation of transfers cost is straightforward if the dataset is not replicated across different storage systems. If we want to allow datasets to be stored in several storage systems then each computation is going to select one of the replicas based on the transfer cost and access latency/bandwidth; depending on the actual values for the α_i . In order to express this in the objective function we introduce the variable $y_{i,j,k}$ which represents the number of transfers of data (per month) of dataset i from storage j to site k (where presumably there are some computational resources that process the data). Together with this part of the objective function we need additional constraints for each variable $y_{i,j,k}$:

$$\forall i \in 1 \dots I \left\{ \begin{array}{l} \sum_j y_{1,j,1} = \text{computation}_1 \dots K \text{times} \dots \sum_j y_{1,j,K} = \text{computation}_K \\ \sum_j y_{I,j,1} = \text{computation}_1 \dots K \text{times} \dots \sum_j y_{I,j,K} = \text{computation}_K \end{array} \right. \quad (5.9)$$

These constraints establish that, for every dataset i and for every site k , the total number of dataset i transfers needed (from any storage j , thus $\sum_j y_{1,j,1}$) equals the number of computations at that site k . An additional restriction is that if we transfer data from storage j to site k , the data must be there (if $x_{i,j}$ is zero, then so it is $y_{i,j,k}$):

$$\forall y_{i,j,k} : y_{i,j,k} \leq x_{i,j} \times \text{sitecapacity}_k \quad (5.10)$$

The second term, the compute cost equation, can be expressed as:

$$\text{AverageComputeCost} = \frac{\sum_k \text{computation}_k \times \text{costHour}_k}{\text{Number of compute hours per month}} \quad (5.11)$$

The third main term, *Average Latency*, can be expressed as:

$$\text{AverageLatency} = \frac{\sum_{i,j,k} y_{i,j,k} \times \text{latency}_{j,k} \times \text{datasetUsage}_i}{\text{Number of datasets} \times \text{Number of application runs}} \quad (5.12)$$

This term leverages the variables $y_{i,j,k}$ introduced for the cost calculation. Here we find the latency for each data transfer, multiply by the weight of that data transfer and divide it by the number of data transfers so we can obtain the average access latency.

The expression for fourth term, *Average Bandwidth*, mirror *Average Latency*:

$$AverageBandwidth = \frac{\sum_{i,j,k} y_{i,j,k} \times bandwidth_{j,k} \times datasetUsage_i}{\text{Number of datasets} \times \text{Number of application runs}} \quad (5.13)$$

Aside from the objective function, we must provide our integer linear programming solver with these additional linear constraints (data has to be stored somewhere, $computation_k$ does not exceed site capacity and all $computation_k$ add up to the application needs):

$$\text{For each } dataset_i : \sum_j x_{i,j} \geq 1 \quad (5.14)$$

$$\text{For each } site_k : computation_k \times computationLength < sitecapacity_k \quad (5.15)$$

$$\text{Computation} : \sum_k computation_k \geq \text{Number of application runs per month} \quad (5.16)$$

In this section we have considered the following four factors: storage cost, compute cost, latency, and bandwidth. We believe that these are important metrics for the user; others are certainly possible. Some, such as availability or durability, come into play in our previous stage, where datasets are matched to possible storage systems based on capabilities and requirements. Thus, these metrics come into play as a filter, where the decision is binary and do not participate in the objective function. We present an example in Section 5.4.2 where a new metric (job turnaround time) should be included into the objective function so the solution strives to minimize this metric.

5.3 Resource allocation problem solver

In this section we describe our software implementation that solves the problem model introduced in the last section. As we have mentioned in the introduction, this software does not exist in isolation or as a purely theoretical approach; rather we see the data allocation problem solver as a component

of a larger project, shown in Figure 1.3. The first component is the storage capability matcher, which takes two inputs: a machine readable description of the cloud storage systems (storage capabilities), and a set of requirements from the end user for each dataset in the application. The output is a filtered set of possible storage systems for each dataset, along with the information that our problem model requires. Like the storage capability matcher that it interacts with, our solver is implemented as a C# prototype that uses the Microsoft Solver Foundation ¹ library. Thus, we have the ability to use the default solver or plug in another that is compatible with the Solver Foundation interface; for the experiments presented in this chapter `lpsolve` [63] was selected as the ILP solver. At this stage, the output of our GUI is a textual representation of the problem instance and the solution, along with additional details/statistics that MSF provides. The experiments have been run on our desktop machine, an AMD Athlon II X4 2.90GHz with 6 GB of RAM running Windows 7.

5.3.1 Basic examples

Our first example is taken from Section 4.5.2. In this case, we present an application with three different sets of data: a) satellite data (10 GB), b) intermediate storage shared by workers (1 GB), and c) output files (2 GB). The solver presents us with the following optimal solution: satellite data goes to both Amazon S3 in Virginia and the local NFS cluster, both intermediate results and output go to the S3 Reduced Redundancy Storage, 30 of the application runs happen in the local cluster (this maxes out the allocated capacity for this application) and the rest of them go to Amazon EC2 in Virginia. This solution takes into account many issues: S3 is selected to comply with the user requirements of high durability for satellite data, a local copy of the input dataset is created to reduce transfer costs, local computational resources are used to minimize cost, and additional cloud resources are chosen based on cost and access latency. The problem model for this example has a total of 149 variables and it takes 88 ms to solve. The problem input and the output from the solver for this example and the next one is available on our website².

¹<http://msdn.microsoft.com/en-us/devlabs/hh145003>

²<http://www.cs.virginia.edu/~ar5je/ILP.html>

Our second example is a MapReduce application which has an input dataset of 1 TB and generates 10 GB output. Local resources can support a normal daily run of this application, but a few times per month a more complex analysis is required. For this use case we get the following solution: store the input data in both the Amazon cloud (S3 RRS in Virginia) and the local NFS; the output is stored locally only. In this case the cost of data transfer exceeds the cost of storage for additional replicas; again the Amazon cluster in the region comes out as the best option for cost/latency. Solving time for this example is 148 ms; the ILP problem contains 94 variables.

5.3.2 Scalability of the solver

In this subsection we show the scalability of our approach when the number of variables starts increasing. This aspect is very important since we are dealing with an NP hard problem. More complex examples than in the previous subsection are certainly possible: the number of cloud providers could increase in the future, cloud providers will launch new storage services, new datacenters will be built and applications may include more datasets. The potential increase for each of these factors is also limited, though: the space for potential new cloud providers is limited (it requires capital to build datacenters and the software infrastructure); cloud providers cannot develop and offer support for a large number of storage abstractions; there are constraints in the placement of new, big datacenters (e.g. availability of cheap electricity); and users may have a limited ability to manage multiple sets of data instead of consolidating multiple data with similar characteristics into a dataset to be managed as a unit.

We present our results on Figure 5.3.2. For this graph we generate different storage systems with random costs (normally distributed against around averages such as 10 cents per GB per month for storage cost) and feed the problem model to the solver. In this scenario we generate 4 different cloud providers, each one with a number of datacenters within the United States (from 1 to 6) a several matching storage systems (again, from 1 to 6). We consider an application with 3 different datasets. For example, if we choose 3 datacenters and 4 storage systems, the possible number of storage systems for a datasets is: $4 \text{ clouds} * 3 \text{ datacenters/cloud} * 4 \text{ storage systems/datacenter} =$

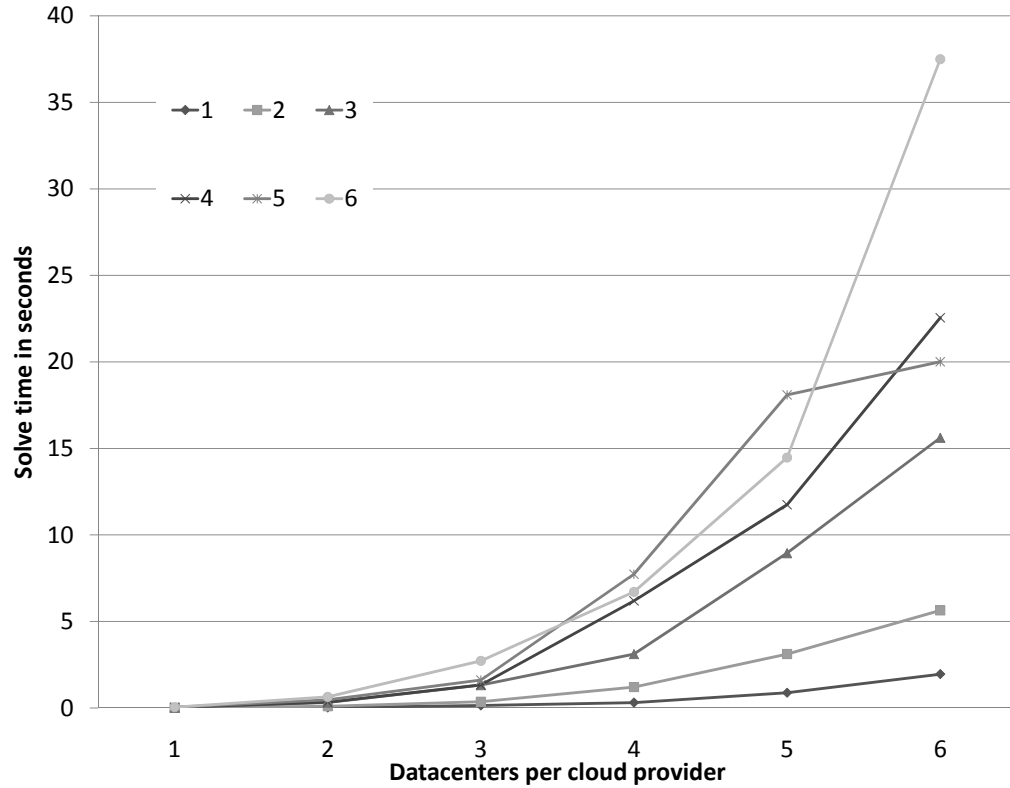


Figure 5.1: Solve time in seconds as a function of the number of datacenters per cloud provider (X axis) and the number of storage systems available per cloud provider and per datacenter (graph lines). Each data point represents the average over 20 runs.

48 possible storage systems. In this case the ILP solver comes up with a solution in 1.08 seconds, on average. In our worst case scenario there are 144 possible storage systems for each dataset and the average time it takes to solve the allocation problem is 37.49 seconds; right now we believe that for each dataset there may be an order of 10 possible storage systems (0.106 seconds solving time) and that, for the reasons mentioned above, an increase of several orders of magnitude is unlikely. And even if this increase were to take place there are still a number of ways to reduce solving time. One of the most obvious ways is to perform better filtering based on storage capabilities matching since it can greatly reduce the size of the problem. Another option is tuning the ILP solver to the characteristics of our objective function and constraints. Right now we use the lpsolve solver with the default settings with one customization: we add a pre-solve stage that deletes variables dominated by other variables. Further customizations may be possible or even using a complete different solver. However, we consider that these results meet our requirements and no further optimizations are

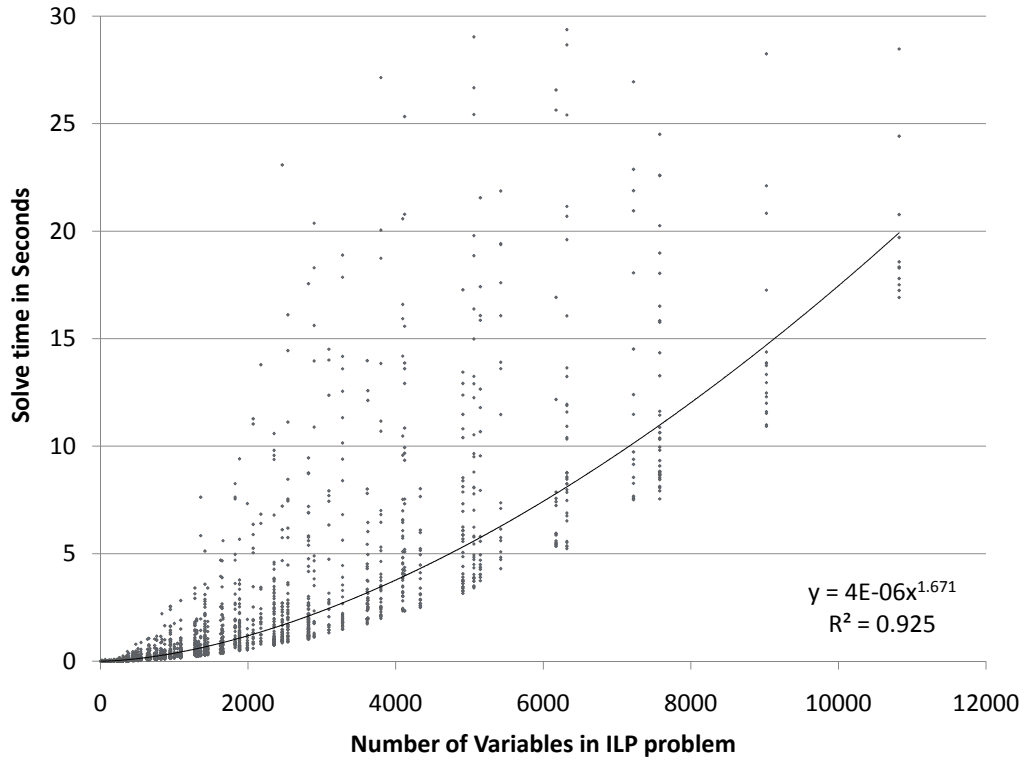


Figure 5.2: Solve time in seconds as a function of the number of variables in the data allocation problem formulation. One data point represents a single solver execution that returns a 2-tuple (# variables, solve time). A fitted power equation is also included.

needed.

In Figure 5.3.2 we chose to set a constant number of cloud providers and application datasets; we can generate an n-dimensional graph in which we vary these two parameters too. However, we think that a more clear representation will be to plot the solving time against the number of variables in the problem model, like in Figure 5.3.2. The parameters number of cloud providers and number of datacenters per cloud provider affect the number of possible storage systems. The size of the ILP problem is based on this number:

$$\text{N.of variables} = \sum_i datasets_i \times (storagesystems_i \times (computesites + storagesystems_i)) \quad (5.17)$$

The data in Figure 5.3.2 comes from running different scenarios, which include variations in the number of cloud providers, datacenters and storage systems. This figure shows a relationship

between the number of variables and solving time that can be approximated by a power function, which fits the data very well (R^2 is greater than 0.92). Since the exponent of this power function is small ($x^{1.67}$), problem sizes with a few thousand variables can be solved fast. The same figure also shows that as the problem size increases the variability of the results does too.

In summary, we believe that, given problem sizes based on current cloud offerings, the data allocation problem in cloud computing can be solved in under a second. Future growth of cloud providers and interfaces may push this threshold to half a minute if there is an order of magnitude increase; these results were generated with a standard desktop machine and make no assumptions regarding performance improvements of future ILP solvers or the development of new heuristics.

5.3.3 Sensitivity of the solution

Previously we have described how we arrive to an optimal solution based on the inputs from the user and the current state of the cloud providers. In this subsection we discuss how this solution is affected by the inputs. We have considered three factors that affect the user's confidence on a given solution: the variability of cloud providers cost and performance; the user-provided weights in our objective function; and the accuracy of the user's estimation of data requirements. Over the long term the performance and cost of different cloud providers will vary; however we consider this to be a factor that is too difficult (or impossible for the user) to predict and that its short term variability is small. Price changes are infrequent and our experience seems to show that performance over the short term (weeks) is, on average, mostly stable [7].

The user-provided weights for our objective function will have a much greater impact. For example, a user may select the weights for the storage costs, compute costs, latency and bandwidth to be 0.30, 0.30, 0.20, and 0.20, respectively. How does the user select these quantities and not 0.25, 0.30, 0.25 and 0.20? Would that lead our system to arrive to a completely different solution? Essentially we have here a 4-dimensional space in which each point represents the data allocation solution. Since our solver is fast, we can choose to re-run the solver with the different parameters and compare the new solution with the given one; if they are the same we consider these two points to be

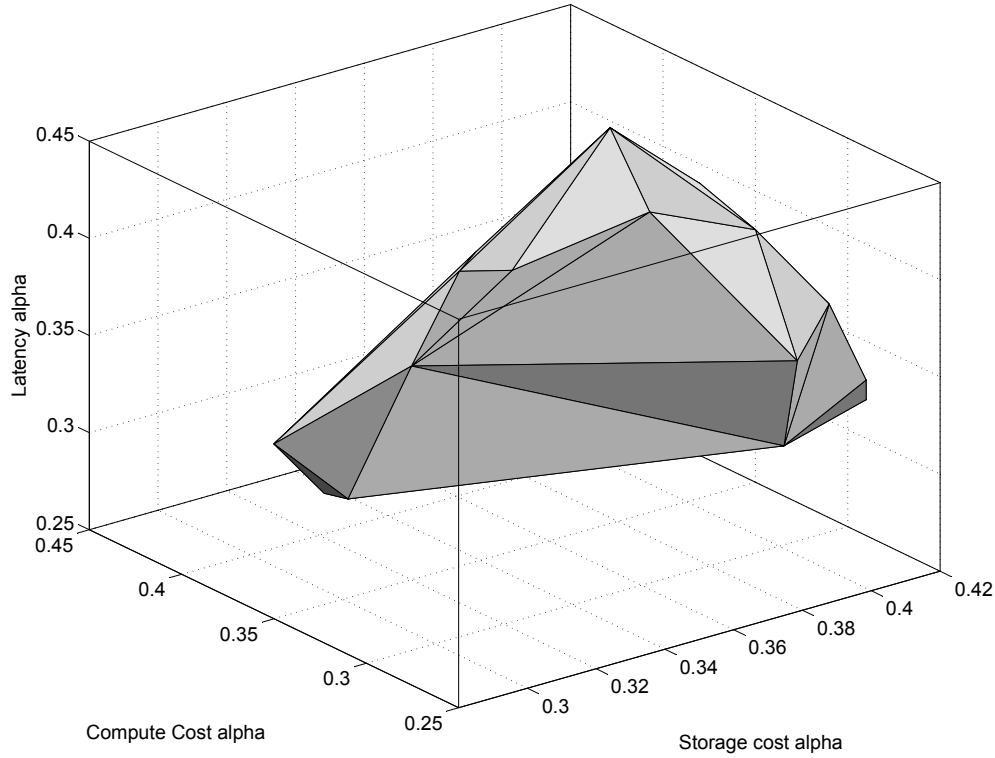


Figure 5.3: Volume that represents the alpha values for which the same optimal solution exists. The starting values are 0.33 for all storage, compute and latency alphas.

in the same volume (which represents a data allocation). In order to provide a visual representation of this, we have run our first example in Section 5.3.1. with different weights for storage, compute and latency. We start with data point (0.33, 0.33, 0.33) and process its neighbors; if they are the same solution we add the point to the output and recurse; the 3D convex hull of these points is shown in Figure 5.3.3. From this data set we can also find out the limit values for each alpha: all other alphas being equal, what is the range for each alpha that maintains the same solution? These ranges are: for the storage cost [0.26, 0.42], for the compute cost [0.275, 0.375] and for latency [0.29, 0.37]. In this example we do have a medium range of values whose solution is shared; in other problem instances we may have a much smaller (or larger) range. We want to emphasize that a short range is not necessarily a bad option; if the user is confident about the weight values then we give a correct and optimal solution. However, in cases where these weights are a ballpark estimate the graphs and numerical ranges shown should be useful.

We end this section with a discussion of the user-provided data requirements. In many instances it is difficult to estimate the output size of an application, or the number of application runs (and their length). Similarly to our discussion of the weights (alphas) we can re-run our solver varying different input parameters; we continue using the same example. In this case we have chosen the sizes of the first and second datasets. We assign each dataset a range of possible sizes: for dataset 1 from 10 to 30 GB (in increments of 512 MB) and for dataset 2 from 1 to 4 GB (in increments of 128 MB); each point in this 2D space represents a data allocation solution. As it turns out, there are two possible optimal solutions in this 2D space. One lies approximately in the rectangle delimited by sizes for dataset 1 10GB to 15 GB and for dataset 2 1.5 GB to 4 GB. The solution for the rest of this space is the same as the original one (dataset 1 is 10 GB and dataset 2 is 1 GB). Here we are comparing only storage allocation decisions; compute (job scheduling) is not considered. Thus, the user knows the limits for which the current storage allocation solution is optimal and the different solutions outside this range.

In summary, the solution that our system finds for a concrete data allocation problem is an optimal one, but we recognize that the user's inputs to this problem maybe approximate. In order to address this issue we re-run the solver with several variations of the input parameters and compare the output solutions. In this section we have explored a couple of data representations that can provide the user with information on how stable the solution is. This analysis is possible because of the fast solving stage; we can analyze hundreds of data points and generate the graphs shown in a few minutes.

5.4 Use cases

In this section we present two possible use cases with two scientific applications, BLAST [72] and MODIS Azure [4]. In the first use case (BLAST) we modify our formula to add a budget constraint. Thus, a user can ask our system to give the best solution given a budget of \$250 per month (or any other figure). In the second use case (MODIS Azure) we show how we can modify our formula beyond the cost (storage and compute), latency, and bandwidth terms. Here we add computational

length as a term so the user can ask for the solution with the shortest job completion time, given a set budget (and in addition to the usual data requirements).

5.4.1 BLAST

The Basic Local Alignment Search Tools [72] is a very popular algorithm in bioinformatics which enables the search of genetic sequences. We use the following parameters for the datasets and the computation requirements: 20 GB input dataset (approximately the size of the publicly available human databases), 30 seconds query time, and 30 KB of output in table format per query. We want to find out, with a limited budget, what is the best solution for a set number of queries per month. In order to do so we will modify the problem formulation by moving the storage and compute costs from the objective function (2) to a new linear constraint:

$$Budget \geq StorageCost + TransferCost + RequestCost + ComputeCost \quad (5.18)$$

The first three costs are defined in equations 5.6, 5.7 and 5.8; the Compute Cost term is equal to:

$$ComputeCost = \sum_k computation_k \times costHour_k \quad (5.19)$$

We run different scenarios that are represented in Table 5.4.1. Each run of our prototype is given a budget and a number of queries and returns the data allocation for both the input and output datasets. We iteratively increment the number of queries per month (5,000 more each step) till the system is not solvable; each row of the table shows the scenario with the maximum number of queries.

All the solutions share the same data allocations: the input dataset is replicated in the local cluster (NFS) and in the Amazon datacenter in Northern Virginia (S3 Reduced Redundancy Storage); the output dataset is stored in a local MySQL database. For query processing the local machine is maxed out in every case at 60,000 queries per month; additional compute power is allocated in Amazon. The different budget levels give us the maximum number of queries; this takes into account the cost of the data replication, the transfer of output data for the computations carried out in

Table 5.2: BLAST on a budget

Monthly Budget	Input Dataset	Output Dataset	Number of Queries
\$100	AWS S3 RRS, VA Local NFS, VA	Local SQL, VA	60,000 (local) 15,000 (EC2) 75,000 (total)
\$250	AWS S3 RRS, VA Local NFS, VA	Local SQL, VA	60,000 (local) 50,000 (EC2) 110,000 (total)
\$500	AWS S3 RRS, VA Local NFS, VA	Local SQL, VA	60,000 (local) 110,000 (EC2) 170,000 (total)
\$1000	AWS S3 RRS, VA Local NFS, VA	Local SQL, VA	60,000 (local) 225,000 (EC2) 285,000 (total)

Amazon, and the cost of the EC2 machines. In this example we consider a local machine to be equal in power to the Amazon EC2 medium instance; in the next section we show how to take into account the differences between instance types.

5.4.2 MODIS Azure cloud bursting

Our next example uses the MODIS Azure [4] scientific application. This application processes satellite data from several years to present analysis on certain processes, for example evapotranspiration on the Earth’s surface. A previous paper [65] has presented the performance results of the application running on Windows Azure, the local Windows HPC cluster, and a combination of both. The combination of local and cloud resources is labeled cloud bursting; the paper presents performance numbers for tasks that are allocated in the cloud and need to find the input data (stored locally in blobs or remotely in files). This paper on MODIS Azure [65] concludes the evaluation section with “We have found that in general the determining factor is data (where it is and how much is moved). In many situations the key to successful cloud bursting is to minimize data movement”. Hence, we believe that this application can benefit from our data allocation algorithms.

In this case we are not considering latency or bandwidth as important metrics; we use average computation length (or turnaround time) instead. Thus, our first step is appending the following term in our general formula 5.3:

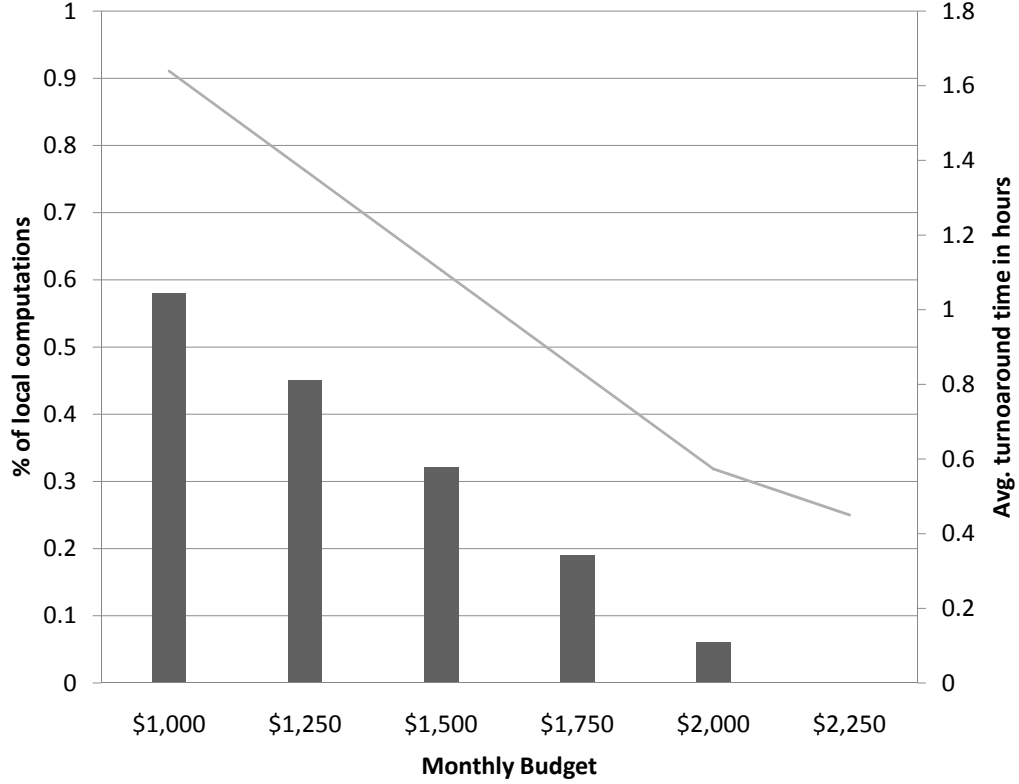


Figure 5.4: Relationship between the average turnaround time of jobs processing one year’s data and the monthly budget. We also show the percentage of local computations: as the budget increases better machines are rented from the Windows Azure cloud and turnaround time improves.

$$\omega_5 \times AverageComputationLength = \frac{\alpha_5}{avg.compute\ time} \times \frac{\sum_{k,h} compute_{k,h} \times speed_{k,h} \times computelength}{\text{Number of compute hours per month}} \quad (5.20)$$

We changed the variable $compute_k$ to $compute_{k,h}$; this variable now means “number of monthly computations on $site_k$ using $profile_{k,h}$ ”. We think of a profile as a different running configuration; for example one profile could be 8 extra-large workers in Windows Azure, and another one 32 medium Windows Azure workers. The computation length is the time it takes to complete in the standard local profile; the $speed_{k,h}$ modifiers come from benchmarking. Given this modification, and the one presented in the previous subsection, we can ask our system to give us the best data and compute allocation that give us the fastest turnaround time for jobs for a given budget.

The input parameters are the following ones: each year’s data is separated into day files; on

average each day has 2.96 GB of input data and its process generates 5.70 MB of output after using 416 MB of temporary storage; we store the data for years 2000 to 2010. Each computation access a complete year, that is, 1/10th of each dataset and its length depends on the machine being run on. The actual numbers used for these input parameters are taken from the referenced paper. The solution allocates the input and output data in both the Windows Azure Blob (US North Central datacenter) and the local HPC Cluster. The computation is done by the local cluster nodes and medium size nodes in Windows Azure.

Figure 5.4.2 presents two measurements regarding the computation (at different budget levels): the percentage of application runs done in the local cluster and the average turnaround time for each application run (which processes and reduces one year of satellite data). The lower the budget the more computations we do locally and the slower these computations are. The job turnaround for a local only solution is 2.5 hours; this baseline job turnaround time is 52% higher than our solution with a monthly budget of \$1,000 (1.64 hours). Given this information about this trade-off (cost vs. speed) the user can make sound allocation decisions based on her preferences or requirements.

5.5 Limitations

As we have seen in the previous sections, our approach relies on having accurate information on the capabilities of the cloud providers. Currently there are multiple websites that continuously benchmark cloud providers like Amazon or Windows Azure; we believe that the community will greatly benefit of having a more thorough approach with more metrics and making the data machine consumable (as opposed to web graphs). Another limitation for developed applications is that interfaces are different across clouds; it is difficult to modify an application to make it possible to run on different clouds. The solution to this issue will probably come by having cloud-agnostic APIs for data access (such as CSAL [73]) and by introducing more compatibility at the execution level: running arbitrary applications for Platform as a Service providers (Windows binaries on Windows Azure), having compatible APIs (Eucalyptus and Amazon) or other ports (Google App Engine on Amazon EC2).

One final limitation is related to our allocation of computation. In this chapter we have introduced a planning phase that gives us a data allocation solution and a coarse-grained approach to computation: we do not take into account factors such as the hourly billing of cloud providers, the VM startup time and the shape of the computation (single-threaded, workflow, etc.). We believe that all these factors are better accounted for with an online approach. Thus, the next step in our work is the scheduling algorithms, which would be described next, in Chapter 6.

5.6 Conclusion

In this chapter we have presented our approach to data allocation in cloud computing. We build upon our previous work (presented in Chapter 4), where we match each application dataset with a set of possible storage services based on storage capabilities and data requirements. We present a problem model for data allocation that takes into account the unique characteristics of cloud computing and balances cost and performance. Our software implementation uses an ILP solver to find an optimal data allocation solution in one second or less; we have also shown that our approach is scalable as the number of cloud providers, datacenters or storage services increase. These short running times also allow us to gather more information about the sensitivity of this solution and present it to the user. Finally we have presented two use cases with the BLAST and MODIS Azure applications. Small changes in our problem formulation allows us to add a monthly budget restriction and to minimize job turnaround time in our optimal solutions; by combining local and cloud resources we can halve the cost compared to a cloud-only approach or be 52% faster than a local-only approach.

Chapter 6

Cloud data-aware scheduling

In Chapter 3 we presented our comprehensive performance evaluation of cloud services. This evaluation was used in Chapter 4 as part of our description of storage capabilities; Chapter 4 also introduced our definition of data requirements and a matching process between datasets and storage services. The results of this matching process are used in Chapter 5, where we provide an optimal data allocation solution that takes into account all the application’s datasets. Scheduling algorithms are the last piece in our end-to-end solution; they are the focus of this chapter.

New cloud applications have the potential to alter the traditional role of the scheduler, which has been to control access to certain shared resources (e.g. a single CPU or a Beowulf cluster). The scheduler assigns these resources to processes, threads or parallel applications in such a way that optimizes throughput, latency, fairness and/or other metrics. However, in a cloud environment, applications within a reasonable range of computing requirements operate under the abstraction that computational resources are unlimited and almost immediately attainable. For example, in a local cluster (managed by PBS [74] or other scheduler) or in a compute grid [75] an application user could generate a request of 25 machines for 40 minutes for the scheduler and then the user will wait for the application to run. In cloud computing the user may request the application to immediately boot 25 machines in Amazon EC2 and run the application for as long as it needs to. A scheduler which simply replies to requests composed by a number of machines and the job’s execution time seems

trivial to implement in cloud computing since we do not have to worry about resource sharing. In this chapter we address the following questions: Is this interface the correct one for cloud computing applications? Is there a need for the scheduler to optimize some metric instead of blindly processing a request?

We believe that there are new requirements for scheduler algorithms for certain cloud applications. Thus, we are not looking for a general scheduler, but for a scheduler that takes advantage of the characteristics of a specific class of applications in order to produce a scheduling plan that is more cost-effective and/or faster. The first class of applications that we consider in this paper is MapReduce [76] jobs. The MapReduce framework is introduced in Section 6.2. The second one is simulations whose running time can be estimated (similar also to Monte Carlo algorithms). For each of these two classes we generate an Integer Linear Programming (ILP) problem based on the information from the cloud providers, the local cluster and the application. The solution to this ILP problem will be the optimal assignment of compute resources based on cost or execution time.

In order to make an optimal scheduling decision for a MapReduce job we need to know: the amount of work to be done, the processing capacity of each instance (and its cost) and a constraint on the job's execution time (provided by the user at submission time). The output of our algorithm will be the amount and type of instances to use during the map, the same during the reduce phase, and an estimation of the cost and duration. For example, the output could be "Start 2 High CPU Medium instances and 17 High CPU Extra large instances and run for 3 hours at a cost of \$6.3". As we will show in Section 6.2, the scheduling decisions made by this strategy form a Pareto efficient frontier that are faster or cheaper than any alternatives (and often both by a margin as high as 2.6 times cheaper and 5.46 times faster for naive strategies).

Our second use case is the watershed model calibration application, which attempts to calibrate a watershed model by comparing the simulation results to actual observations. Each job starts 1,000 different tasks that can be assigned to a local Windows HPC cluster or to instances in Windows Azure. Using the same information than we gather for the MapReduce we present our ILP solver with a similar problem; the solutions are also optimal and form a Pareto frontier. We compare our

approach with both naive and incremental strategies. Our approach avoids the potential pitfalls of naive scheduling where the cost can be as high as 91% more. The incremental strategy uses the same information as our ILP solver but it still can be up to 11% slower or 38% more expensive.

In Section 6.3.4 we introduce a new variable: the location of the input datasets. Here we present a variation of the watershed model calibration where we have 5 different models (each with different running times) that access different datasets that are located in the local cluster, in Windows Azure or in both. Therefore we need to generalize the problem model from our previous use case to support different tasks and data location. We compare our strategy with a data co-location algorithm: in this algorithm we have local and cloud compute resources available and we assign the tasks where the input data is. We find that our strategy (which can move input data around) is superior to data co-location even for the latter's best case scenario. Depending on the amount of datasets stored in the cloud the best data co-location case can be from 5.4% to 18.8% more expensive than our solver.

6.1 Integer linear programming solver

The core of our approach is to generate an integer linear programming problem based on the input data and whose solution is the scheduling decision (number and type of instances at job start time and over time). We need to use an integer linear programming model since the number of instances that are used at any given time needs to be an integer number. The main advantage to this approach is that the ILP solver will return an optimal solution; thus no other approach can be better (at most is equal), given the problem model. The two potential issues with this approach are the validity of the problem model and the running time. In the next sections we will present the problem model for both MapReduce and the watershed model calibration.

Based on our previous results with data management in Chapter 5 we do not consider the running time of the ILP solver to be an issue for the scheduler. Even though the problem is NP hard, recent solvers are remarkably efficient for problems that are small enough. In the previous chapter we found that a problem can be solved under a second if we have less than 2,000 variables; in our model that would be equivalent to managing storage data for an application with 3 different datasets where there

are 48 different valid storage systems for each dataset. As we will present in the following sections, problem sizes for our scheduler are smaller than this: the number of variables for the watershed model calibration are $O(n*m)$, where n is the number of machine types (5 for each Windows Azure datacenter that we consider) and m is the number of time intervals (12 for the examples presented in this paper). The constant factors behind the O notation are also small (lesser than 2). The average time the scheduler spends in the solver phase for our watershed model calibration use case is 9 milliseconds. Thus, we believe that this approach will remain viable even if we scale up the number of datacenters or machine types.

The implementation of the scheduler was done in C# with the solver interface being the Microsoft Solver Foundation and the actual solver used is lpsolve [63]. The results based on simulation were run in a desktop machine (Dual Core 2.40 GHz, 2 GB of RAM). The experimental results with the watershed model calibration are based on application runs in Windows Azure in the Chicago datacenter and in our local Windows HPC cluster with 16 cores.

6.2 Applicability to MapReduce

MapReduce is a popular framework for data parallel applications with an open source implementation, Hadoop [77]. The Amazon cloud offers running Hadoop jobs as a service, which is called Elastic Map Reduce (EMR). A user is able to go to the website, select the type and number of instances to run, select the input and output data, and start the job execution. Amazon also offers a programmatic interface that on top of the job configuration and starting commands includes others such as resizing the number of instances being used.

In order to provide an optimal decision to the user, we need an estimation of the time it takes to process the map and reduce phases on different machines. This estimation comes from benchmarking the application on different instance types. Since the MapReduce jobs are data parallel it should be possible to obtain a good estimation of the running time of an application by executing a subset of it. Aside from this benchmarking information we need a deadline from the user; our solution will find the cheapest way to run the job and meet the deadline.

6.2.1 Problem formulation

We model a MapReduce job as a set of instances of possibly different types during the map phase and another set of instances for the reduce phase. We present a glossary of all the variables used in the following equations and their type in Table 6.1. In order to find the cheapest solution for the user we generate an ILP problem in which the objective functions is the cost of the job:

$$\text{MIN} : \sum_{i,j} cost_{i,j} \quad (6.1)$$

Here, each variable $cost_{i,j}$ has two possible values: 0 or the cost of running the job using i hours for the map phase and j hours for the reduce phase. Therefore, all the $cost_{i,j}$ will have a value of 0, except the one with the minimum cost, which corresponds to the solution of our problem. Each $cost_{i,j}$ is related to the variables $xm_i x r_j$, which are 0-1 integer variables. Only one variable will be equal to one, and that variable corresponds to the number of hours for the map (i) and reduce (j) phases in the solution. For example, if the duration is 5 billable hours for the map time and 1 billable hour for the reduce time then $xm_5 x r_1$ will be one. Thus, we would like to define each $cost_{i,j}$ as the actual cost multiplied by the corresponding $xm_i x r_j$. This, however, will lead to a non-integer constraint, so we convert the problem by introducing three different constraints instead of one non linear:

$$cost_{i,j} = (i \times costperhourmap + j \times costperhourreduce) \times xm_i x r_j. \quad (6.2)$$

We accomplish this by using a MAX_MONEY constant (an amount of money that cannot be spent by the job) and the following constraints:

$$cost_{i,j} \leq MAX_MONEY \times xm_i x r_j \quad (6.3)$$

$$cost_{i,j} - i \times costperhourmap - j \times costperhourreduce - MAX_MONEY \times xm_i x r_j \geq -MAX_MONEY \quad (6.4)$$

$$cost_{i,j} - i \times cost_{perhourmap} - j \times cost_{perhourreduce} + MAX_MONEY \times xm_i x r_j \leq MAX_MONEY \quad (6.5)$$

In our example, every $cost_{i,j}$ will be zero via equation 6.3, except $cost_{5,1}$ which will have the correct value (via equations 6.4 and 6.5). Equations 6.6 and 6.7 define the variables cost per hour map and cost per hour reduce and 6.8 enforces the restriction that only one $xm_i x r_j$ should not be zero:

$$cost_{perhourmap} = \sum_k cost_k \times coremapinstances_k \quad (6.6)$$

$$cost_{perhourreduce} = \sum_k cost_k \times corereduceinstances_k \quad (6.7)$$

$$\sum_{i,j} xm_i x r_j = 1 \quad (6.8)$$

We are only using core instances. If the user wants to take advantage of the lower cost of spot instances we can trivially add the variables $coremap(reduce)instances_k$. The only restriction is that spot instances must have a fixed amount of work assigned to them (for example, 20%) to keep the constraints linear. Also, in this case the user has to acknowledge the risk of having instances shut down and the job finishing later than expected. We need also to enforce the following constraints: if we choose to run the job for a certain amount of time ($xm_i x r_j$) the capacity must be there to finish it before the deadline. Thus, we introduce total work per hour map and total work per hour reduce, which represent the compute capacity for each phase.

$$totalworkperhourmap = \sum_k capacity_k \times coremapinstances_k \quad (6.9)$$

$$totalworkperhourreduce = \sum_k capacity_k \times corereduceinstances_k \quad (6.10)$$

If we choose to run the map phase for five hours that means that $5 \times totalworkperhourmap$ must have enough capacity to finish the required work (MAP WORK and REDUCE WORK is the input data from the user). Thus we add the following $i + j$ constraints:

$$i \times totalworkperhourmap \geq MAPWORK \times xm_i \quad (6.11)$$

Table 6.1: Problem model variables for MapReduce.

Name	Description	Source
$cost_{i,j}$	Float, cost of running the job for i map hours and j reduce hours	Solution
cost per hour map	Floats, hourly job cost during the map and reduce phases	Solution
cost per hour reduce		
$cost_k$	Float, hourly cost for instance type k	Input
core map instances $_k$,	Integer, number instances of type k to run during the map	Solution
core reduce instances $_k$	(reduce) phase	
$xm_i xr_j$	0-1 variable, 1 if job runs for i billable map hours and j reduce hours	Solution
total work per hour map	Floats, work done per hour of map (reduce) computation	Solution
total work per hour reduce		
$capacity_k$	Float, work done by instance type k per hour	Input
$xm_i(xr_j)$	0-1 variable, 1 if job runs for i (j) billable hours during the map (reduce)	Solution
$xm_l(xr_m)$	0-1 variable, 1 if job runs for l (m) hours during the map (reduce) where l (m) belongs to 0, 0.25, 0.5, , i (j)	Solution
MAP WORK	Float, amount of work to be done in each phase	Input
REDUCE WORK		
JOB TIME LIMIT	Float, time in hours by which the job should be finished	Input

$$j \times totalworkperhourreduce \geq REDUCEWORK \times xr_j \quad (6.12)$$

Here we have introduced the 0-1 integer variables xm_i and xr_j . If $xm_i xr_j$ equals 1, then xm_i and xr_j are 1; otherwise they equal 0. These equations will give us the minimum cost for the job; however we cannot be sure about the job duration because these variables represent billable hours of computation. If a computation takes 2.5 hours for map and 0.25 hours for reduce there are 3 billable hours for map and 1 for reduce. The job duration is not 4 hours though, but 2.75. In order to increase the accuracy up to the quarter of hour we introduce new 0-1 variables xm_l and xr_m similar to the ones defined before:

$$l \times totalworkperhourmap \geq MAP_WORK \times xm_l \text{ where } l \in \{0.25, 0.5, , i, i.25, i.50, i.75\} \quad (6.13)$$

$$m \times totalworkperhourreduce \geq REDUCE_WORK \times xr_l \text{ where } m \in \{0.25, 0.5, , j, j.25, j.50, j.75\} \quad (6.14)$$

These new variables allow us to introduce the deadline constraint:

$$\sum_l l \times xm_l + \sum_m m \times xr_m \leq JOB_TIME_LIMIT \quad (6.15)$$

Finally we must take into account that of the x variables in each category to be 1 (and the rest 0), the relationship between $xm_i xr_j$, xm_i and xr_j ; and the maximum amount of instances running at the same time (Amazon's restriction):

$$\sum_i xm_i = 1, \sum_j xr_j = 1, \sum_l xm_l = 1, \sum_m xr_m = 1 \quad (6.16)$$

$$0.5 \times xm_i + 0.5 \times xr_j \geq xm_i xr_j \quad (6.17)$$

$$\sum_i coremapinstances_i \leq 20, \sum_j corereduceinstances_j \leq 20 \quad (6.18)$$

6.2.2 Use case

Here we present an example of the results produced by our algorithm. The example application to schedule is a MapReduce job whose map phase requires 1,000 hours of work (running on Amazon's Small instance, single core). The reduce phase requires 50 hours of work (taking also the Small instance as the reference). One of the requirements is that we have data on how long it takes to run on different instance types, for example an Extra large with 8 compute units will be able to process 8 times more work than a Small instance. We compare our approach (ILP Solver) with the naive strategy of selecting a number and type of instances and running them till job completion. Our results are shown in Figure 6.1.

Each data point represents a scheduling plan that results in a certain amount of money spent and an estimated job execution time. The naive strategies here start a fixed number of instances of different types. For example, the data points in the naive 10 instances strategy will be: use 10 Small instances, use 10 Large Instances, use 10 Extra Large instance, etc. Using 10 High CPU Extra Large instances results in a running time of 5.25 hours and a cost of \$7.2, this is the point (7.2, 5.25) in the

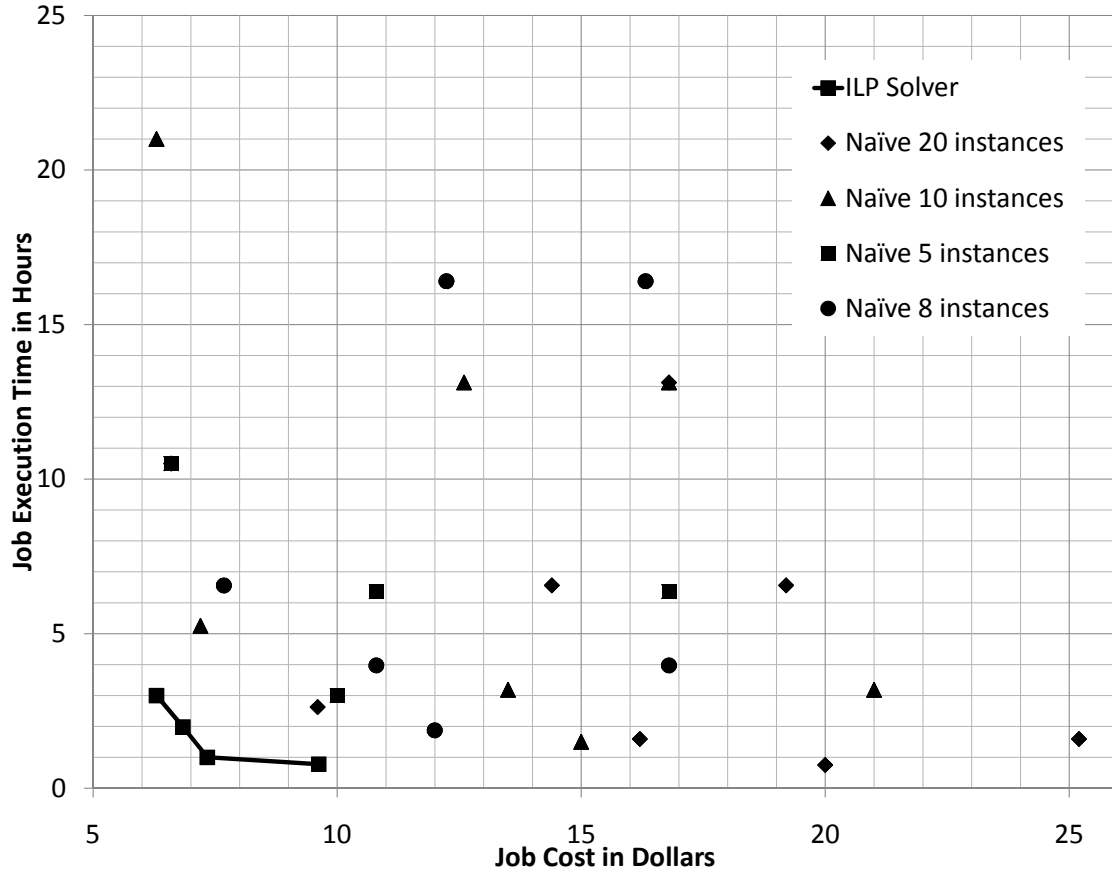


Figure 6.1: Job cost and execution time for different MapReduce schedulings. Naïve strategies start n instances of a type (different points in the graph represent a different instance type choice). ILP Solver selects a variable number of instances (and types) for the map and reduce phases.

graph. For the ILP Solver strategy we give the algorithm a deadline, for example 3 hours, and it returns a scheduling plan (2 High CPU Medium instances and 17 High CPU Extra large instances for 3 hours for the map and reduce phases) and a cost (\$6.3). If the deadline is not possible to meet then the ILP solver will detect this situation and notify the user. As we can see in this graph, the ability of stopping instances when they are not needed and sizing them properly for the amount of work to be done can greatly reduce the cost and/or the execution time. ILP Solver can complete the job in 45 minutes at a cost of \$9.62; the naïve strategies solutions go up to 21x slower and up to 2.6x more expensive and often they are both. The ILP solver data points in the graph show the shape of a Pareto frontier.

Table 6.2: Problem model variables for the watershed model calibration.

Name	Description	Source
$cost$	Float, cost of running the job	Solution
$x_{i,j}$	Integer, number of instances of type j running concurrently during interval i	Solution
$work_{i,j}$	Float, amount of done work by all instances of type j during interval i	Solution
$billing_{h,j}$	Integer, number of billable instances of type j running during hour h	Solution
$hourlyCost_j$	Float, cost in dollars of running an instance of type j for an hour	Input
$minuteCost_j$	Float, cost in dollars of running an instance of type j for a minute	Input
$capacity_{i,j}$	Float, work done by instance type j during interval i	Input
NUMBER_OF_TASKS	Integer, work to be done (1,000 by default in watershed)	Input
AVAILABILITY $_j$	Integer, maximum number of concurrent instances of type j	Input

6.3 Applicability to watershed model calibration

In this section we present our scheduling algorithm for the watershed model calibration application. In essence, the user uploads a watershed model through the web interface and introduces the parameters for the search. The goal is to calibrate the model by comparing its simulation-based outputs to the actual measurements. Once the job is submitted, the node in charge of the execution distributes the data to the worker nodes and starts 1,000 tasks. Each task analyzes a subset of the search space, which is partitioned in such a way that we can predict each task's duration. Once all the tasks are finished the best result (that is, the one that is closest to reality) is returned to the user. The worker nodes can be located in the local Windows HPC Cluster or in Windows Azure (Chicago datacenter).

6.3.1 Problem formulation

Similar to the problem model in MapReduce, we aim to minimize the cost of the watershed model calibration while trying to finish the job execution before a deadline (a reference of all variables and constants used can be found in Table 6.2). Our objective function will be simply:

$$\text{MIN: } cost \tag{6.19}$$

In order to formulate the problem we need to define a series of time intervals. For example, we can use 10 minutes as the duration of our time intervals; if a job needs to finish in less than 1 hour

and a half then we will consider 9 time intervals. We should highlight that we are using intervals of equal duration for simplicity; nothing in our model precludes us from using an arbitrary division of time. For each time interval we can then calculate the amount of work that can be done by the different available machines. Thus, we introduce the following variables, $x_{i,j}$ and $work_{i,j}$:

$$capacity_{i,j} \times x_{i,j} \geq work_{i,j} \quad (6.20)$$

$$\text{For each instance type } j, \text{ interval } i : x_{i,j} \leq AVAILABILITY_j \quad (6.21)$$

$x_{i,j}$ is a positive integer that represents the number of instances of type j that are active in interval i . $work_{i,j}$ is the amount of tasks that can be processed by all the instances of type j in the same interval. The first intervals will be an special case since we need to take into account the time it takes to boot a virtual machine on Windows Azure (or the waiting time for a local core) and the time it takes to transfer the model data. Thus, the first (or firsts if one interval is not enough to initialize a worker) $capacity_{i,j}$ can be either zero or a lower number than the regular capacity of each instance. Equation 6.20 adds an upper bound for every variable $x_{i,j}$: this upper bound maybe equal to the number of cores available (for the local cluster) or the maximum number of concurrent instances allowed (for Windows Azure). With all the $work_{i,j}$ defined, we can add the restriction which enforces that all tasks can be processed:

$$\sum_{i,j} work_{i,j} \geq NUMBER_OF_TASKS \quad (6.22)$$

The next step for us is to add a restriction that makes the series over time of the number of instances of a certain time monotonically decreasing. Basically, if at some point during the computation we are going to need 20 small instances in Windows Azure, start them at the beginning. Thus, if the user sets a deadline of 2 hours but the job could be done in an hour and a half (at the same cost), then these linear constraints will make sure that our ILP solver finds the fastest solution:

$$\text{For each instance type } j, \text{ interval } i : x_{i,j} \geq x_{i+1,j} \quad (6.23)$$

It is time now for us to define the cost variable. First we will introduce the formulation that takes into account the current hourly billing model in Windows Azure. In order to do so, we introduce new variables $billing_{h,j}$. For every hour of the computation we get billed the maximum number of active instances at any time during the hour. So, we add the following constraints for every machine type j :

$$\text{For every interval } i \text{ in hour } h : billing_{h,j} \geq x_{i,j} \quad (6.24)$$

Then, the cost is simply the weighted sum of all billing variables:

$$\sum_{h,j} hourlyCost_j \times billing_{h,j} = cost \quad (6.25)$$

An alternative formulation may be useful for certain contexts in which billing by the minute is more appropriate. For example, multiple scientists share the same application and residual capacity from other job runs can be used for the current job submission. In this case, the staff in charge of managing the application may choose to bill users by the minute (and maybe include the amortized cost of the wasted capacity). Or it can be the case that each application run lasts for several hours or even days and the wasted cost of the last hour of computation is negligible. In this case, we can skip equations 6.24 and 6.25 from our problem formulation and directly define cost using the variables $x_{i,j}$:

$$\sum_{i,j} intervalLength_i \times minuteCost_j \times x_{i,j} = cost \quad (6.26)$$

In the presentation of our problem formulation we have introduced the definition of intervals instead of allowing an arbitrary amount of time. This is because in order to calculate the cost of running the application we need to multiply the number of instances by the time they are active. If both the number of instances and running time are unknowns this will produce a non linear constraint that cannot be processed by our solver. So, if we divide the time into intervals (it does not matter if they are equal in size) we can calculate the interval cost and capacity before generating the problem. Therefore, interval cost and capacity are constants and the model can be expressed linearly. There

are a couple of disadvantages, though. The first one is that if a user asks for a 73 minute deadline our application will process it as a 70 minute deadline. We feel that in our application this is not an issue, and we can always increase the number of intervals to get a 5 minute resolution (or better). The second disadvantage is that the size of the problem increases linearly with the application's running time. We are not concerned about this since it is always possible to define intervals to be one hour or more, get the solution and finally refine the result. For example, an application that has a deadline of 34 hours could be processed with 2 hours intervals. If the resulting scheduling plan takes 30 hours we can a) use this result if it's accurate enough b) rerun the problem by introducing a big 24 hour interval at the beginning and then 30 minute (or less) increment intervals. For our watershed model calibration application, however, we have found that 10 minute intervals work well.

The solution to our problem will give us the number of active instances of each type that will be active at each interval ($x_{i,j}$), the number of tasks that can be processed during each interval by each set of machine types ($work_{i,j}$), the cost ($cost$) and execution time (max interval for which any $x_{i,j}$ is greater than zero).

6.3.2 NP hardness

Here we present a proof of the NP hardness of the problem model just introduced. Thus, we can show that an optimal solution can only be obtained by using an exponential algorithm; although, as we have argued in Section 6.1, the typical problem size that we encounter can be solved successfully within milliseconds by lpsolve. Other strategies, such as the naive and incremental ones that will be introduced with our simulation and experimental results, could approximate or even equal the optimal solutions in some cases, but not in all of them (since they are polynomial time algorithms).

The basic idea is to perform an efficient reduction from the bounded knapsack problem to our problem formulation (the minute billing version). Thus, the input to the reduction algorithm is the n items, the maximum weight W , the value of each item v_j , the availability of each item c_j , and the weight of each item w_j . The transformation is simple, the number (and length) of intervals is always 1, the NUMBER OF TASKS is W , the $AVAILABILITY_{0,j}$ are the c_j , the $minuteCost_j$ are

v_j , and the $capacity_j$ are the w_j . This reduction is trivially done in polynomial time.

The solution to our billing problem is the $x_{0,j}$; that is the number of instances of each type running during the only interval. Each $x_{0,j}$ corresponds to the number of items of each type that will be in the knapsack, x_j . Each $x_{0,j}$ will obey the bounded knapsack problems restrictions since $x_{0,j} \leq AVAILABILITY_{0,j}$ implies $x_j \leq c_j$. The capacity restriction is also taken into account since if we combine equations 6.20 and 6.22 we have:

$$\sum_j capacity_j \times x_{0,j} \geq NUMBER_OF_TASKS \implies \sum_j w_j \times x_j \leq W \quad (6.27)$$

since in the reduction we have effectively multiplied by -1 the equation by assigning W and w_j to $NUMBER_OF_TASKS$ and $capacity_j$. Finally, the solution to the scheduling problem (combining equations 6.19 and 6.26) has to correspond to the solution to the knapsack problem; we can arrive to this conclusion simply by substituting the variables assigned during the reduction:

$$\text{MIN: } \sum_{i,j} intervallength_i \times minuteCost_j \times x_{i,j} \implies \text{MIN: } \sum_j 1 \times (-v_j) \times x_{0,j} \implies \text{MAX: } \sum_j v_j \times x_j \quad (6.28)$$

Thus, it holds that in order a solver for our scheduling problem formulation (minute billing) will also solve the bounded knapsack problem (Bounded Knapsack \leq_P Scheduling), proving its NP hardness. From this point it is easy to see a further reduction to the hourly billing version from the minute billing version of the problem in which we divide the capacity and cost arrays by 60 to convert hours into minutes and then adjust the number of intervals accordingly. We do not want to imply that there are no good polynomial-time algorithms for schedulers to implement. Indeed, as we will see in the next sections algorithms that are not naive and take into account the same information available to our ILP solver can do well under certain circumstances. Our argument is that, given that an optimal algorithm which is $O(2^n)$ in theory but it is fast in practice for the size of problems we are interested in solving, this algorithm should be the preferred choice.

6.3.3 Use case with hourly billing

We present the results of our approach with the hourly billing model in Figure 6.2. For this use case the watershed model calibration requires running 1,000 tasks, where each task takes 90 seconds to run on a single core and the input data size is 1 GB. We compare our approach with two different strategies, naive and incremental. The naive approach is simple: the scientist selects the number of instances to run for the complete duration of the job, for example, 20 instances. The scheduler selects all the available local nodes and if that's not enough it starts several instances in Windows Azure to reach that number. In our example we always use 16 local cores and start anywhere from 0 to 44 small instances in Windows Azure. The dashed line in the graph plots the duration of each job against its cost (local instances are considered free). This dashed line makes a Z type curve around the 1 hour limit; even if we select other instance types we observe the same phenomena. The cause of this shape is the billing model in Windows Azure, where the user pays by the hour, regardless of where she uses 1 or 59 minutes of time. A job that with the help of 20 small instances lasts for 63 minutes costs \$4.8 since we are paying for 40 compute hours. However, had the user selected 24 small instances the job would have finished in just under one hour and the bill would have been \$2.88 (24 billable compute hours). Thus, we can see here the two main problems with the naive approach: the user interface and the suboptimal choices. The user interface asks the user for the number of instances but that is not a metric important to her; the metrics the scientist cares about are cost and execution time. There is no way for the user to relate cost and time to number of instances unless she has previous experiences with the system. The other problem is that we can clearly see the suboptimal choices that are made because of the hourly billing model in cloud computing. The worst case scenario for the naive strategy is at the 61 minutes mark, where it costs \$5.28 to run the job, compared to the \$2.76 cost using our strategy.

We introduce another approach, Incremental, and compare both of them to our ILP solver. The incremental approach uses benchmarking information to predict the cost and execution time of each job given a set of instances of different types. If we have information about how long each task usually takes on each type of instances, the time it takes to initialize a node and stage in the data then we

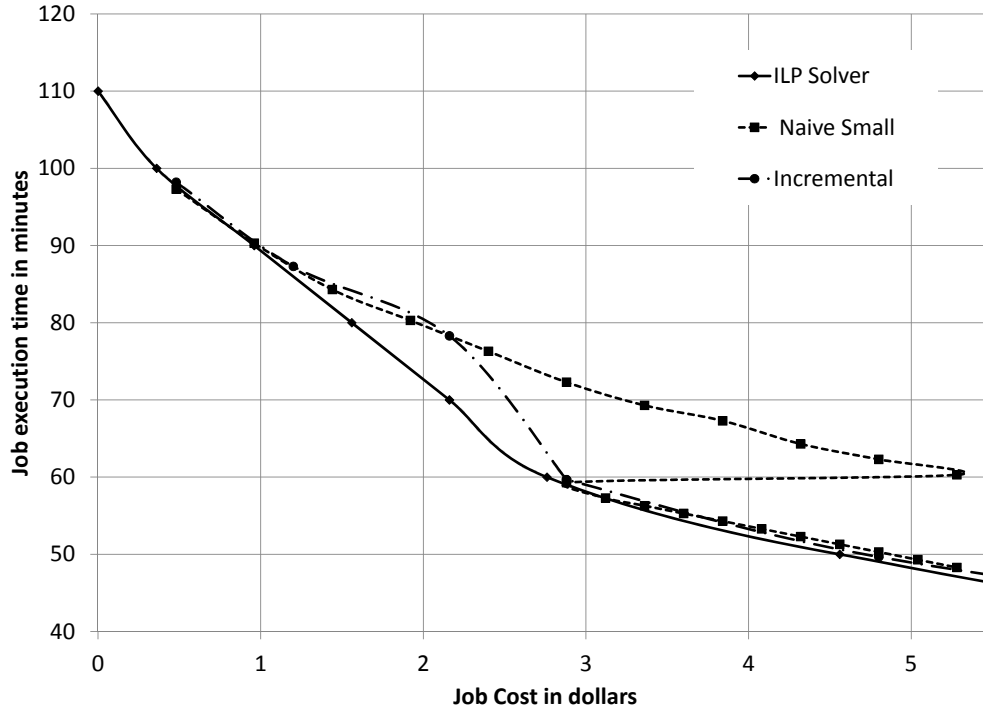


Figure 6.2: Job cost and execution time for different watershed model calibration scheduling strategies. Naive strategies use 16 local cores and additional instances of a given type on Windows Azure. ILP Solver selects a variable number of instances (and types) over time.

can calculate the cost and time of each job before it is submitted with a certain configuration. This is the same type of information available to our ILP solver. For this approach the input from the user is the expected execution time. The incremental algorithm starts with the local machines as the starting configuration and calculates cost/time. Then it incrementally adds instances from Windows Azure (mixing different types) up to a limit, and returns the best solution. The best solution is the cheapest one whose completion time is below the user's limit. We can see this approach as simulating every possible naive strategy and selecting the best one. This way we can avoid the two main problems of the naive strategy: the interface is now presented in terms of what the user cares about, cost and time; and there are no choices which are clearly wrong (more expensive and slower than other possible configurations). In the graph the dot-dashed line represents this strategy.

Even though the choices for the incremental algorithm are good, we can do better. Remember that in both naive and incremental we select some instances and run them till job completion time. It is possible to stop some instances before the job completion time to avoid being billed an additional

hour. For example, we can start 18 small instances in Windows Azure for one hour and use the 16 local cores for the whole job. This way the job can be completed in 70 minutes and it will cost \$2.16. A comparable data point for the incremental algorithm is 78 minutes and \$2.16; that is, 11.4% slower for the same amount of money. Our strategy (ILP solver) is represented by the continuous line.

As we can see in the graph, both Incremental and ILP lines get much closer when the job duration is less than one hour. This is expected, since the main advantage for the ILP strategy (stopping instances at hour intervals to avoid incurring charges because of partial hours) is lost. However, we still consider it to be the best mechanism since it gives the best solution in all instances. Incremental may get close to the best solution for jobs under one hour, but these jobs may not be that common in practice since they are twice or thrice more expensive to run in exchange for a 25% or 33% (15 or 20 minutes) decrease in job execution time.

6.3.4 Use case with multiple watershed models and data-aware scheduling

In our last use case the computational job is divided into 1,000 tasks, where the only different between tasks is some input parameters (each task runs the same watershed model). In this section we present the modifications to our model in order to support a more complex scenario:

- The job is composed by k different models where each model is composed by $NUMBER_OF_TASKS_k$ tasks. The execution time varies for each model; all tasks of one model last the same amount of time.
- The input data may be stored locally, in Windows Azure or in both.

The objective function is the same one as in the last use case (equation 6.19). In order to account now for the different watershed models we have to modify existing variables. Basically we add need the k suffix in order to account for the k different models. The new variables $work_{i,j,k}$ and $x_{i,j,k}$ are defined in:

$$capacity_{i,j,k} \times x_{i,j,k} \geq work_{i,j,k} \quad (6.29)$$

$$\text{For each instance type } j, \text{ interval } i : \sum_k x_{i,j,k} \leq AVAILABILITY_j \quad (6.30)$$

The following constraint enforces that the number of machines (local or in Windows Azure) used for this job decreases monotonically. Thus, if we need to ask for 16 local cores at some point during the job execution, do it at the beginning:

$$\text{For each instance type } j, \text{ interval } i : \sum_k x_{i,j,k} \geq \sum_k x_{i,j,k} \quad (6.31)$$

In our last use case we knew that at the start of every job every machine that was participating will be staging in the necessary data. Here we do not have that advantage since a machine could be processing job 1 for the first half an hour and then switch to job 2. Thus, the initialization can happen at any time and we can not zero out (or decrease) the $capacity_{i,j,K}$ constants, which is what we did before. Instead we add the initialization time as negative work done; this is used in every machine so we have to multiply this quantity by the maximum number of machines that participate processing watershed model k :

$$\text{For each instance type } j, \text{ watershed model } k : \sum_i work_{i,j,k} - InitPenalty \times max_machines_{j,k} \geq total_work_{j,k} \quad (6.32)$$

$$\text{For each interval } i, \text{ instance type } j, \text{ watershed model } k : max_machines_{j,k} \geq x_{i,j,k} \quad (6.33)$$

We also need to add a constraint to make sure that all tasks for each watershed model are completed:

$$\text{For each watershed model } k : \sum_j total_work_{j,k} = NUMBER_OF_TASKS_k \quad (6.34)$$

Finally, we define the new *cost* variable. In this case it will be composed by the compute costs and possibly data transfer costs. In order to define the transfer costs we need to introduce new binary variables, $transfer_{k,l,m}$, which are one in case we run a watershed model in a location where the input data needs to be transferred. For example, running a model locally but the input data is

located only on Windows Azure. The definition of these new variables is:

$$\text{For each instance type } j, \text{ watershed model } k : \sum_i x_{i,j,k} \leq MAX_TASKS \times t_{k,l,m} \quad (6.35)$$

$$transfer_costs = \sum_{k,l,m} input_data_size_k \times (costGBOut_l + costGBIn_m) \times transfer_{k,l,m} \quad (6.36)$$

$$compute_costs = \sum_{i,j,k} interval_length_i \times minuteCost_j \times x_{i,j,k} \quad (6.37)$$

$$cost = transfer_costs + compute_costs \quad (6.38)$$

We compare our approach with a data co-location strategy. Since the input data may be stored in only one cloud, the data co-location strategy starts the computation where the data is available. In this case we are submitting for execution 5 different watershed models, each one composed of 200 tasks (1,000 tasks total for calibration). The execution time varies from 1 minute to 7.5 minutes (all tasks of one model last the same amount of time). We have run two scenarios:

- In scenario A the inputs for models 1, 3 and 4 are stored locally and the ones for 2, 3 and 5 are stored in Windows Azure.
- In scenario B the inputs for models 1, 2, 3 and 4 are stored locally and the ones for 3 and 5 are stored in Windows Azure.

For scenario A models 2, 3 and 5 run on Windows Azure and in scenario B models 3 and 5 do. We present our experimental results on Figure 6.3. In the graph each data point for the data co-location strategy represents the run with a fix number of cloud instances of the medium size (from 1 to 32). For the ILP solver, on the other hand, it represents a configuration (the solution to the ILP problem) where the maximum number of intervals has been set (in the figure, every 10 minutes from 290 to 90). The data co-location strategy works well when you have the right amount of instances on Windows Azure so that both the local and cloud instances finish around the same time. In the graph this is represented by the data point at the elbow. For all the points before this data point, local resources finish first and they are idle. Idle local resources waste no money but they can not help

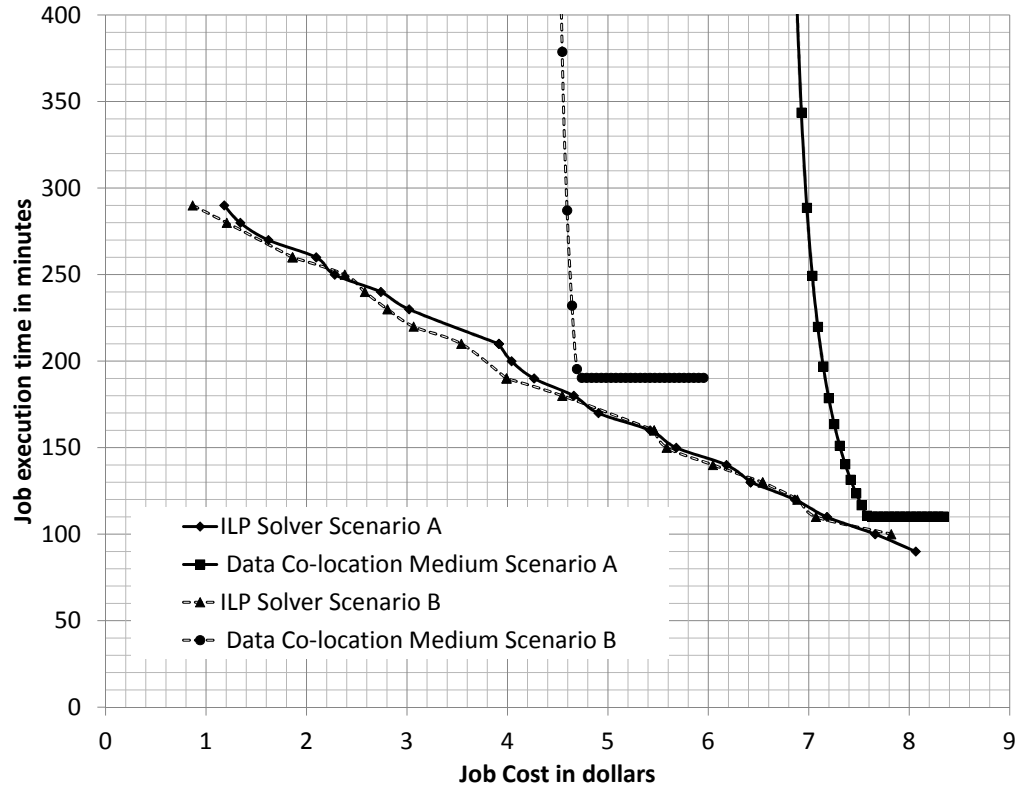


Figure 6.3: Scheduling cost and turnaround time for multiple watershed models.

accelerating the job since they do not have the input data locally available. After this data point the reverse happens, cloud resources finish first while local resources are still processing tasks and some money is wasted. Of course, if we have this information before hand there is only one reasonable configuration for job execution (the data point at the elbow).

The advantages of the ILP strategy are two-fold. First, the ILP solver's ability to move the data and tasks around gives the user a lot more options for balancing the cost and execution time. Second, even in the best case data co-location for scenario A is 5.4% more expensive than ILP solver and data co-locations for scenario B is 18.8% more expensive (than ILP solver). The ILP solver's ability to select the best matching instance sizes gives us the ability to outperform the data co-location strategy in the best case.

6.4 Conclusion

In this chapter we have presented new requirements for schedulers in cloud computing. The traditional view of the scheduler managing a shared resource on behalf of multiple applications and users is no longer valid in a cloud environment where computational resources are elastic. Traditionally users request the scheduler a number of compute resources during a time period; we show that this strategy can easily lead to overpayment for cloud resources (because of the hourly billing model) and slower applications. We focus on two types of applications: MapReduce and simulations whose running time is deterministic (our use case in this category is the watershed model calibration). Since we can make a reasonable estimation of the running time of each task we can make a scheduling plan that fits compute resources to the work to be done. The user interface can be summarized as “give us the maximum amount of time allowed for the job execution and we will present you with the cheapest solution”.

For both MapReduce and the watershed model calibration we have proposed two problem formulations based on integer linear programming whose solutions represent the scheduling decisions to be made. As we have shown, this problem is NP hard and no efficient solutions exist in general. We have also shown, however, that the size of the problems we are interested in solving are small enough for a modern ILP solver (lpsolve) to process them in a few milliseconds. The main advantage of using the ILP solver is that the solutions are proven to be optimal; thus any polynomial time algorithm (such as the Incremental or Data Co-location strategies presented for the watershed model calibration) can only match the optimal solution in the best case, which is not common.

The output of our algorithm for MapReduce will be the amount and type of instances to use during the map, the same during the reduce phase, and an estimation of the cost and duration. For example, the output could be “Start 2 High CPU Medium instances and 17 High CPU Extra large instances and run for 3 hours at a cost of \$6.3”. A similar approach is taken with the watershed model calibration, although the solutions to this problem can potentially stop a subset of the instances at any time during the job execution to prevent overpayment. The scheduling decisions made by our strategy form a Pareto efficient frontier that are faster or cheaper than any alternatives; in

MapReduce naive alternatives can often be worse than our approach by a margin as high as 2.6x more expensive and 5.46x slower. For the watershed model calibration we find similar results for naive strategies, although more advanced algorithms can still be up to 11% slower and up to 38% more expensive. If we introduce data location in our problem model our strategy (which can move input data around) is superior to data co-location even for the latter's best case scenario. Depending on the amount of datasets stored in the cloud the best data co-location case can be from 5.4% to 18.8% more expensive than our solver.

Chapter 7

Conclusions and future work

In this chapter we highlight the most important results and summarize the main points of our research. We also discuss possible directions to extend our research.

7.1 Summary of results

The key experimental results in our research are:

- There is a wide range of variability in the performance of cloud storage services, as we have seen in Chapter 3. Cloud users should carefully consider performance when designing new cloud applications, where some examples are:
 - Blob data transfer rate can go down as much as 50% with 32 concurrent clients compared to a single client access. Single client access is limited by the bandwidth available to the instance, the cumulative throughput among all clients can not exceed 400MB/s (that is, 3 replicas serving data over a 1Gb/s link).
 - Data stored in Table should be accessed using partition keys and row keys under heavy concurrency; half of 32 concurrent clients will get a time-out error if querying a Table with property filters.

- Users should be aware that instance boot time in Windows Azure takes 10 minutes or more. This long boot times can affect the strategies used for dynamic scalability.
- SQL Azure shows better scalability than local SQL servers and consistent performance; however, we have also detected a slowdown of up to 2x compared to a local SQL server.
- In Chapter 4 we present our approach to describing cloud storage services and data requirements, and generating a list of compatible storage services per application dataset:
 - We have introduced a novel and extensible XML schema capable of fully describing the storage systems at Amazon, Windows Azure and local clusters. We use general C# code to express the user requirements such as datacenter location, type of data access, durability, cost estimates. Most of these requirements should only take 50-150 lines of code to express.
 - For every use case our storage matching algorithm takes less than 70 ms to complete. This algorithm attempts to match every dataset in the application with every storage service by evaluating every data requirement associated with the dataset.
- We present a novel algorithm in Chapter 5 for resource allocation in cloud applications. Our unique approach relies on an ILP solver:
 - By using an ILP mathematical formulation for resource allocation we can optimize cost, latency, bandwidth and/or job turnaround time. The ILP solver can find a solution to this problem in under a second for current use cases. We have analyzed the scalability of our solver and found that even if the problem size increases by an order of magnitude (new storage services, new datacenters, new cloud providers) solving times are still under a minute.
 - By using our resource allocation (mixing cloud and local resources) for the MODIS Azure application users should expect a 50% decrease in cost (compared to a cloud-only approach) or a 52% decrease in job execution time (compared to a local-only approach).

- In Chapter 6 we have created new scheduling algorithms for cloud computing applications. The goal is to utilize application specific information to come up with optimal scheduling decisions that balance cost and execution time:
 - For MapReduce the output of our algorithm will be the amount and type of instances to use for each phase. In a graph that represents cost vs job execution time the decision points made by our algorithm form a Pareto optimal shape.
 - For the watershed model calibration application our algorithm avoids the worst possible scenario in a naive strategy (which costs almost twice as much), and also improves other strategies (which are 11% slower for the same amount of money).

7.2 Conclusions

Two trends in scientific computing, the use of simulation and the *data deluge*, will lead to an ever increasing need for computational resources. We believe that cloud computing has the capacity of providing these resources in a cost efficient way. There are, however, many open questions on how users can benefit from the current and future offer of cloud services. The success of cloud applications depends on how we answer the following questions: how we can evaluate different cloud services, how we can choose the best storage services for a given application, and how we can balance the cost and performance when running cloud applications.

In Chapter 3 we have presented the results from experiments we have conducted on Windows Azure. We have shown an exhaustive performance evaluation of each of the integral parts of the platform: virtual machines, storage services (Table, Blob and Queue) and SQL services. Based on these experiments, we also provide our performance-related recommendations for users of the Windows Azure platform. These cloud services are the building blocks for cloud applications, and are usually presented to the user as a black box, with no performance guarantees. Our main focus is to provide the community with performance information and concrete recommendations that help the design and development of scalable cloud applications.

The main focus of Chapter 4 is storage. We presented an automated approach to the selection of cloud storage services that can meet the user's requirements. First, we have defined an XML schema that guides the description of the different capabilities of cloud storage systems. We use this XML schema to provide descriptions for the storage services of Amazon, Windows Azure and some software that is commonly deployed in local clusters (NFS, Hadoop, MySQL, etc.). Second, we have developed an application that processes these XML descriptions and attempts to match common data requirements from users. With accurate information on cloud services and a matching algorithm we can provide users with a list of storage options, for each dataset, that are guaranteed to meet the application requirements.

The next chapter, Chapter 5, provides a global optimal solution for resource allocation. We take the list of storage options from the last chapter and generate an Integer Linear Programming problem. The solution to this problem represents an optimal solution that optimizes (following the user's preferences) any of the following metrics: latency, bandwidth (for data access), cost and job execution time. Our software implementation uses an ILP solver to find a data allocation solution in one second or less and prove its optimality; we have also shown that our approach is scalable as the number of cloud providers, datacenters or storage services increase. At the end of this chapter we presented two use cases with the BLAST and MODIS Azure applications.

Finally, in Chapter 6 we present new scheduling algorithms for cloud applications. In the traditional scheduler interface the user asks for a number of compute resources during a time period; we show that this strategy can easily lead to overpayment for cloud resources (because of the hourly billing model) and slower applications (because of idle cloud and local resources). We focus on applications whose running time is deterministic (MapReduce and Watershed) and generate a scheduling plan that fits compute resources to the work to be done. The user interface can be summarized as "give us the maximum amount of time allowed for the job execution and we will present you with the cheapest solution". We continue using the ILP mathematical model since this way we can prove the optimality of the scheduling decisions. The scheduling decisions made by our strategy form a Pareto efficient frontier that are faster or cheaper than any alternatives.

7.3 Future work

As a final point in this dissertation, we identify and describe several possible ways to extend our research:

- We start this dissertation by analyzing the performance of Windows Azure. The main issue regarding this analysis is that it will be obsolete eventually: the hardware in datacenters gets replaced continuously with newer components (and maybe expanded) and the software that manages these datacenters evolves. Therefore, it will be worthwhile to start monitoring these services continuously and offering performance data in a machine readable way. Right now there are some websites with scattered performance information about cloud platforms (Amazon mainly); these performance numbers are commonly presented for human consumption (e.g. graphs). We believe that an organized effort to provide an accurate and fresh picture of cloud platforms will be very valuable to the community.
- In our storage capabilities chapter we introduced several extensible definitions of cloud services for Amazon, Windows Azure and local software (NFS, etc.). Since this work was published new datacenters and new storage services have been announced. Similarly to the performance of cloud platforms, an update of this information (in XML format) is needed. Also, we implemented several data requirements in this chapter and presented a framework for defining more. We think that it will be valuable to ask users what are the most common data requirements found in scientific applications and implement those. Since there exist a finite number of requirements, having a library with the most common ones ready to be used would be very valuable and it will free users from having to implement them.
- Our work has extensively used ILP solvers and lpsolve has been the one chosen in our experiments. Our point of view regarding the ILP solver has been the user's: the solver is a black box with a well defined interface that we follow. It will be interesting to take an in-depth look at the workings of the branch-and-bound algorithm that is at the core of these solvers. By knowing how the algorithm works it may be possible to customize it to run faster or improve

its scalability. This project will entail coming up with heuristics for the branch-and-bound algorithm that are tailored to our data management and scheduling models. For example, we may favor more logical paths like selecting the storage system for the largest dataset first so the ILP solver can find an optimal solution faster.

- We have dedicated an entire chapter to new scheduling algorithms. These new algorithms focus on MapReduce and Monte Carlo type applications, that is, we need some characteristics (deterministic execution times) to provide optimal scheduling. We believe that exploring other type of applications will be worthwhile. For example, there are many cloud applications that are continuously running and servicing requests. The metrics important for these applications are cost, latency and throughput, as opposed to the ones considered in our work (cost and job execution time). A new mathematical model for these applications should be developed. It is also possible that our methodology (using ILP modeling and solvers) will work for this application type, or other ones.
- In this dissertation we lay out a global vision for the problem of data management in cloud applications. We have divided this problem in four different parts: analyzing the performance of cloud services, matching services and requirements, making globally optimal allocation decisions and proposing new scheduling algorithms. It would be interesting to include all these parts into a distributed data management system; it is also a project that will take a considerable amount of software engineering effort. Maybe an existing system like iRODS [25] could be used as the starting point, so starting a new software system from scratch can be avoided. We have mentioned iRODS because of its extensibility; the rule engine at iRODS's core interprets different rules that define how the system behaves and responds to some events and conditions. Thus, it may be possible to adapt the different parts of this dissertation to work with this system or similar ones.

Bibliography

- [1] Tony Hey and Anne Trefethen. The Data Deluge: An e-Science Perspective. In *Grid Computing - Making the Global Infrastructure a Reality*, pages 809–824. Wiley and Sons, July 2003.
- [2] Amazon Inc. Amazon Elastic Compute Cloud (EC2). <http://aws.amazon.com/ec2/>, 2008.
- [3] David Chappell. Introducing the Windows Azure Platform. <http://go.microsoft.com/fwlink/?LinkId=158011>, 2010.
- [4] Jie Li, Marty Humphrey, Catharine van Ingen, Deb Agarwal, Keith Jackson, and Youngryel Ryu. eScience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform. In *Proceedings of the 26th IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*, pages 1–10, Shanghai, China, 2010. IEEE.
- [5] Daniel Nurmi, Rich Wolski, Chris Grzegorzczak, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'09)*, pages 124–131, 2009.
- [6] Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13(5):14–22, September 2009.
- [7] Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphrey. Early observations on the performance of Windows Azure. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*, page 367, Chicago, USA, June 2010. ACM Press.
- [8] Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphrey. Early observations on the performance of windows azure. *Scientific Programming*, 19(2-3):121–132, April 2011.
- [9] Arkaitz Ruiz-Alvarez and Marty Humphrey. An automated approach to cloud storage service selection. In *Proceedings of the 2nd international workshop on Scientific cloud computing (ScienceCloud '11)*, page 39, San Jose, USA, June 2011. ACM Press.
- [10] Arkaitz Ruiz-Alvarez and Marty Humphrey. A Model and Decision Procedure for Data Storage in Cloud Computing. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '12)*, Ottawa, Canada, 2012.
- [11] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for science grids: a viable solution? *High Performance Distributed Computing*, pages 55–64, 2008.
- [12] Beth Plale, Dennis Gannon, Dan Reed, Sara Graves, Kelvin Droegemeier, Bob Wilhelmson, and Mohan Ramamurthy. Towards Dynamically Adaptive Weather Analysis and Forecasting in LEAD. In Vaidy S. Sunderam, Geert Dick Van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Proceedings of the The International Conference on Computational Science 2005 (ICCS 2005)*, volume 3515 of *Lecture Notes in Computer Science*, pages 624–631, Berlin, 2005. Springer Berlin Heidelberg.

- [13] Yogesh Simmhan, Catharine Van Ingen, Girish Subramanian, and Jie Li. Bridging the Gap between the Cloud and an eScience Application Platform. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, pages 474–481, 2010.
- [14] Girish Subramanian and Yogesh Simmhan. Tools for Genome Haplotyping in the Windows Azure Cloud. In *Microsoft Research eScience Workshop*, Microsoft Research, 2009.
- [15] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. BEOWULF: A Parallel Workstation For Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14, 1995.
- [16] Narasimha R Adiga et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the ACM/IEEE Supercomputing Conference*, page 60, 2002.
- [17] Luiz André Barroso and Urs Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, January 2009.
- [18] Ian Foster, Carl Kesselman, Jeffrey M Nick, and Steven Tuecke. The Physiology of the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, chapter 8. John Wiley & Sons, Ltd, Chichester, UK, 2003.
- [19] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. In *Technical Report No. UCB/EECS-2009-2*, University of California at Berkeley, 2009. Berkeley.
- [20] Charlie Catlett. The Philosophy of TeraGrid: Building an Open, Extensible, Distributed TeraScale Facility. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, page 8, Berlin, Germany, May 2002. IEEE.
- [21] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187–200, July 2000.
- [22] Sergio Andreatto, Stephen Burke, Laurence Field, Steve Fisher, Balazs Konya, Marco Mambelli, Jennifer Schopf, Matt Viljoen, and Antony Wilson. GLUE Schema Specification Version 1.2. <http://infnforge.cnaif.infn.it/glueinfomodel/index.php/Spec/V12>, 2005.
- [23] Sudharshan Vazhkudai, Steven Tuecke, and Ian Foster. Replica selection in the Globus Data Grid. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 106–113. IEEE Comput. Soc, May 2001.
- [24] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, pages 28–31, 1998.
- [25] Mark Hedges, Adil Hasan, and Tobias Blanke. Management and preservation of research data with iRODS. *Conference on Information and Knowledge Management*, pages 17–22, 2007.
- [26] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC storage resource broker. *IBM Centre for Advanced Studies Conference*, 1998.
- [27] Ron Oldfield and David Kotz. Armada: A Parallel File System for Computational Grids. In *Proceedings of the 1st IEEE International Symposium on Cluster Computing and the Grid (CCGRID'01)*, Brisbane, Australia, May 2001. Published by the IEEE Computer Society.
- [28] Brian S White, Michael Walker, Marty Humphrey, and Andrew S Grimshaw. LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications. In *Proceedings of the ACM/IEEE Supercomputing Conference*, page 19, 2001.

- [29] T. Kosar and M. Livny. Stork:making data placement a first class citizen in the grid. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 342–349. IEEE, 2004.
- [30] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Elastic management of cluster-based services in the cloud. *International Conference on Autonomic Computing*, pages 19–24, 2009.
- [31] Eugene Ciurana. *Developing with Google App Engine*. Apress, Berkely, CA, USA, 2009.
- [32] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (VEE '05)*, page 13, New York, New York, USA, June 2005. ACM Press.
- [33] Adit Ranadive, Mukil Kesavan, Ada Gavrilovska, and Karsten Schwan. Performance implications of virtualizing multicore cluster machines. In *Proceedings of the 2nd workshop on System-level virtualization for high performance computing (HPCVirt '08)*, pages 1–8, New York, New York, USA, March 2008. ACM Press.
- [34] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164, December 2003.
- [35] Simson L. Garfinkel. An Evaluation of Amazons Grid Computing Services: EC2, S3, and SQS. *Center for Research on Computation and Society, School of Engineering and Applied Sciences, Harvard University*, 2007.
- [36] Zach Hill and Marty Humphrey. A quantitative analysis of high performance computing with Amazon's EC2 infrastructure: The death of the local cluster? In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (GRID'09)*, pages 26–33. IEEE, October 2009.
- [37] Edward Walker. Benchmarking Amazon EC2 for High-Performance Scientific Computing. *;Login: The Usenix Magazine*, 33(5), 2008.
- [38] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: The Montage example. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, pages 1–12. IEEE, November 2008.
- [39] Christina Hoffa, Gaurang Mehta, Tim Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the Use of Cloud Computing for Scientific Workflows. In *Proceedings of the 4th IEEE International Conference on eScience*, pages 640–645. IEEE, December 2008.
- [40] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications. *SERVICE-ORIENTED COMPUTING. ICSOC/SERVICEWAVE 2009 WORKSHOPS. Lecture Notes in Computer Science*, 6275/2010:197–207, 2010.
- [41] Constantinos Evangelinos and Chris Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *Cloud Computing and its Applications (CCA 2008)*, Chicago, IL, 2008.
- [42] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS:an elastic transactional data store in the cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, page 7, San Diego, 2009. USENIX Association.

- [43] Rajagopal Ananthanarayanan, Karan Gupta, Prashant Pandey, Himabindu Pucha, Prasenjit Sarkar, Mansi Shah, and Renu Tewari. Cloud analytics:do we really need to reinvent the storage stack? In *Proceedings of the 2009 conference on Hot topics in cloud computing*, page 15, San Diego, 2009. USENIX Association.
- [44] Swapnil Patil, Garth A. Gibson, Gregory R. Ganger, Julio Lopez, Milo Polte, Wittawat Tantisiroj, and Lin Xiao. In search of an API for scalable file systems:under the table or above it? In *Proceedings of the 2009 conference on Hot topics in cloud computing*, page 13, San Diego, 2009. USENIX Association.
- [45] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Cappello, and David P. Anderson. Cost-benefit analysis of Cloud Computing versus desktop grids. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, May 2009.
- [46] Ian F. Adams, Darrell D. E. Long, Ethan L. Miller, Shankar Pasupathy, and Mark W. Storer. Maximizing efficiency by trading storage for computation. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, page 17, San Diego, 2009. USENIX Association.
- [47] Jon Weissman and Siddharth Ramakrishnan. Using proxies to accelerate cloud applications. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, page 20, San Diego, 2009. USENIX Association.
- [48] Hashim Mohamed and Dick HJ Epema. An Evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusters. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 287–298. IEEE Society Press, 2004.
- [49] Hashim Mohamed and Dick HJ Epema. Experiences with the koala co-allocating scheduler in multiclusters. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGRID '05)*, pages 784–791. IEEE Computer Society, 2005.
- [50] Ruay-Shiung Chang, Jih-Sheng Chang, and Shin-Yi Lin. Job scheduling and data replication on data grids. *Future Generation Computer Systems*, 23(7):846–860, 2007.
- [51] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid. In *Grid computing: making the global infrastructure a reality*. John Wiley & Sons, Ltd, 2003.
- [52] W. Chu. Optimal File Allocation in a Multiple Computer System. *IEEE Transactions on Computers*, C-18(10):885–889, October 1969.
- [53] R. G. Casey. Allocation of copies of a file in an information network. In *AFIPS Joint Computer Conferences*, pages 617–625, Atlantic City, New Jersey, 1972. ACM.
- [54] A Kumar. Genetic algorithm based approach for file allocation on distributed systems. *Computers & Operations Research*, 22(1):41–54, January 1995.
- [55] Enrique Grapa and Geneva G. Belford. Some theorems to aid in solving the file allocation problem. *Communications of the ACM*, 20(11):878, 1977.
- [56] K. Lam and C. T. Yu. An approximation algorithm for a file-allocation problem in a hierarchical distributed system. In *International Conference on Management of Data*, pages 125 – 132, Santa Monica, California, 1980.
- [57] Samy Mahmoud and J. S. Riordon. Optimal allocation of resources in distributed information networks. *ACM Transactions on Database Systems (TODS)*, 1(1):66, 1976.
- [58] Lawrence W. Dowdy and Derrell V. Foster. Comparative Models of the File Assignment Problem. *ACM Computing Surveys (CSUR)*, 14(2):287, 1982.
- [59] Bezalel Gavish and Olivia R. Liu Sheng. Dynamic file migration in distributed computer systems. *Communications of the ACM*, 33(2):177, 1990.

- [60] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Competitive distributed file allocation. In *Annual ACM Symposium on Theory of Computing*, pages 164–173, San Diego, California, 1993. ACM.
- [61] Howard L. Morgan and K. Dan Levin. Optimal program and data locations in computer networks. *Communications of the ACM*, 20(5):315, 1977.
- [62] Lintao Zhang and Sharad Malik. The Quest for Efficient Boolean Satisfiability Solvers . *Computer Aided Verification*, 2404:641–653, September 2002.
- [63] Michael Berkelaar, Kjell Eikland, and Peter Notebaert. lpsolve: Open source (mixed-integer) linear programming system. <http://lpsolve.sourceforge.net/>, 2011.
- [64] Gabriel Mateescu, Wolfgang Gentzsch, and Calvin J. Ribbens. Hybrid Computing: Where HPC meets grid and Cloud Computing. *Future Generation Computer Systems*, 27(5):440–453, May 2011.
- [65] Marty Humphrey, Zach Hill, Keith Jackson, Catherine van Ingen, and Youngryel Ryu. Assessing the Value of Cloudbursting: A Case Study of Satellite Image Processing on Windows Azure. In *Proceedings of the 7th IEEE International Conference on e-Science (e-SCIENCE 2011)*, Stockholm, Sweden, 2011. IEEE Society Press.
- [66] Sriram Kailasam, Nathan Gnanasambandam, Janakiram Dharanipragada, and Naveen Sharma. Optimizing Service Level Agreements for Autonomic Cloud Bursting Schedulers. In *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW'10)*, pages 285–294. IEEE, September 2010.
- [67] Tekin Bicer, David Chiu, and Gagan Agrawal. A Framework for Data-Intensive Computing with Cloud Bursting. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, pages 169–177. IEEE, September 2011.
- [68] Microsoft Extreme Computing Group. Azure Scope. <http://azurescope.cloudapp.net>, 2010.
- [69] Frank Schmuck and Roger Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, 2002.
- [70] Brad Calder, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Ju Wang, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, Leonidas Rigas, Aaron Ogus, Niranjana Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, and Jiesheng Wu. Windows Azure Storage. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, page 143, Cascais, Portugal, October 2011. ACM Press.
- [71] Jason Read, Timo Brimhall, and Jason Martin. Cloud Harmony, benchmarking the cloud, 2012.
- [72] S. Altschul. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, September 1997.
- [73] Zach Hill and Marty Humphrey. CSAL: A Cloud Storage Abstraction Layer to Enable Portable Cloud Applications. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 504–511. IEEE, November 2010.
- [74] Robert Henderson. Job scheduling under the Portable Batch System. In Dror Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer Berlin / Heidelberg, 1995.

- [75] Kavitha Ranganathan and Ian Foster. Identifying dynamic replication strategies for a high-performance data grid. In Craig A. Lee, editor, *Grid Computing GRID 2001*, volume 2242 of *Lecture Notes in Computer Science*, pages 75–86. Springer Berlin Heidelberg, Berlin, Heidelberg, March 2001.
- [76] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107, January 2008.
- [77] K Shvachko, Hairong Kuang, S Radia, and R Chansler. The Hadoop Distributed File System. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, pages 1–10, May 2010.