The Ginga Approach to Adaptive Query Processing in Large Distributed Systems

A Thesis
Presented to
The Academic Faculty

by

Henrique Wiermann Paques

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
November 2003

The Ginga Approach to Adaptive Query Processing in Large Distributed Systems

Approved by:

Dr. Calton Pu, Advisor

Dr. Karsten Schwan

Dr. Ling Liu, Co-Advisor

Dr. Tamer Özsu (University of Waterloo)

Dr. Shamkant B. Navathe

Date Approved: November 21st, 2003

To my wife Sonia,

whose love and support made this possible.

ACKNOWLEDGEMENTS

This dissertation contains the results of my Ph.D. research carried out at the Georgia Institute of Technology. I am grateful to the many people who offered their advice and criticism to this dissertation in its early stages. Especially, I wish to thank my thesis advisor, Professor Calton Pu, and thesis co-advisor, Dr. Ling Liu, for giving me the opportunity to do this work, for the stimulating discussions which forced me to better explicate my ideas, for their constant support and consideration during my completion of the kind of research reported here, and for the freedom I experienced while working on this research project. I am greatly indebted to them for the guidance, encouragement and confidence which I felt privileged to receive during the past years.

I wish to thank professor Shamkant B. Navathe, Karsten Schwan, and M. Tamer Özsu for taking part in my Ph.D. defense committee. Their critical reading and penetrating remarks make the final revisions both necessary and useful.

I have also benefited by the comments on various portions of the work reported here from David Buttler, Wei Han, Wei Tang, James Caverlee, Lakshmish Ramaswamy, and from the participants in the conferences in which I have participated

I wish to take this opportunity to acknowledge the financial subsidies I received from CAPES (Brasília, Brazil), the US Department of Energy, and the Georgia Institute of Technology for their support of my research work and for the assistance in travel to various conferences across the United States.

Last, but not least, I wish to thank my family, especially my wife, for their understanding and love. I thank all my friends and colleagues who helped me in one way or another and who made my stay in Atlanta a pleasant one.

Since the manuscript went through a number of revisions, none of the people acknowledged should take responsibility for any remaining errors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Processing and optimizing ad-hoc and continual queries in an open environment with distributed, autonomous, and heterogeneous data servers (e.g., the Internet) pose several technical challenges. First, it is well known that optimized query execution plans constructed at compile time make some assumptions about the environment (e.g., network speed, data sources' availability). When such assumptions no longer hold at runtime, how can I guarantee the optimized execution of the query? Second, it is widely recognized that runtime adaptation is a complex and difficult task in terms of cost and benefit. How to develop an adaptation methodology that makes the runtime adaptation beneficial at an affordable cost? Last, but not the least, are there any viable performance metrics and performance evaluation techniques for measuring the cost and validating the benefits of runtime adaptation methods?

To address the new challenges posed by Internet query and search systems, several areas of computer science (e.g., database and operating systems) are exploring the design of systems that are adaptive to their environment. However, despite the large number of adaptive systems proposed in the literature up to now, most of them present a solution for adapting the system to a specific change to the runtime environment. Typically, these solutions are not easily "extendable" to allow the system to adapt to other runtime changes not predicted in their approach.

In this dissertation, I study the problem of how to construct a framework where I can catalog the known solutions to *query processing* adaptation and how to develop an application that makes use of this framework. I call the solution to these two problems the Ginga approach.

I provide in this dissertation three main contributions: The first contribution is the adoption of the Adaptation Space concept combined with feedback-based control mechanisms for coordinating and integrating different kinds of query adaptations to different

runtime changes. The second contribution is the development of a systematic approach, called Ginga, to integrate the adaptation space with feedback control that allows me to combine the generation of predefined query plans (at compile-time) with reactive adaptive query processing (at runtime), including policies and mechanisms for determining when to adapt, what to adapt, and how to adapt. The third contribution is a detailed study on how to adapt to two important runtime changes, and their combination, encountered during the execution of distributed queries: memory constraints and end-to-end delays.

CHAPTER I

INTRODUCTION

1.1 Dissertation Context

The problem of query optimization has been studied extensively in centralized, parallel, and distributed local area networked environments. The cost-driven query optimization technology is known as a unique characteristic that differentiates DBMS from other computer systems. Conventional approaches to distributed query optimization [40] are designed for database environments characterized by three fundamental assumptions:

1. Data sources typically consist of a small fixed number of databases.
2. Databases are described by a global schema and managed centrally by database administrators.
3. The runtime environments of such distributed database systems (e.g., communication links and response time from different data servers) are relatively stable and predictable.

However, optimization and execution of queries over wide-area distributed data sources impose significant new challenges. First, the number of data sources accessible via the Internet is huge and continuously growing. Second, there are semantic and performance problems that arise due to the heterogeneity and rapid evolutionary nature of the data sources. Such heterogeneity and rapid evolution happen at both the content structure level and the access interface and connectivity level. Third, the runtime environment involves a large number of remote data sources, intermediate sites, and communication links, all of which are vulnerable to congestion and failure. Such vulnerability leads to the high response-time variability[1] experienced by many Internet users. Finally, the optimizer has

[1] The time required for obtaining data from remote sources can vary significantly, depending on the specific data sources accessed and the current state of the network at the time that such access is attempted [2].

little or no reliable statistics about the data (e.g., cardinalities, histograms) collected from the remote sites. Without such estimates, it is almost impossible for the optimizer to guarantee that the generated plan will have optimized execution.

To address the new challenges posed by Internet query and search systems, several areas of computer science (e.g., database and operating systems) [49, 54, 24] are exploring the design of systems that are adaptive to their environment. A system is considered to be *adaptive* if its behavior can adapt gracefully and transparently to structural and behavioral changes in the environment. In the context of wide-area information systems, the adaptive query processing addresses query optimization and execution issues encountered by large-scale query engines that operate in an unpredictable and rapidly changing environment.

However, despite the large number of adaptive systems proposed in the literature to date, most of them present a solution for adapting the system to a specific change in the runtime environment. Typically, these solutions are not easily "extendable" to allow the system to adapt to other runtime changes not predicted in their approach. If I could have a framework that allows me to integrate and combine almost any adaptation approach, I could have a system that is intelligent enough to cope with most runtime changes.

In this dissertation, I study the problem of how to construct a framework where the known solutions to *query processing* adaptation can be cataloged and how to develop an application that makes use of this framework. I call the solution to these two problems the *Ginga* approach. The discussion of the adaptation framework proposed in this dissertation is not necessarily restricted to adaptation query processing; it could be applied to any other system where adaptation is advantageous.

The core of the Ginga Approach is the concept of *adaptation spaces* (see Chapter 3). An adaptation space is a powerful abstraction and framework that describes a collection of possible adaptations of a software component or system and provides a uniform way of viewing a group of alternative software adaptation processes. In addition, adaptation spaces describe the different means of monitoring the conditions that different adaptation processes depend on and the particular configurations through which adaptive systems navigate for coping with runtime changes.

The remainder of this chapter is organized as follows: To make the discussion of query adaptation more concrete, I describe in Section 1.2 a real-world application drawn from the bioinformatics domain, where the deployment of Ginga can be advantageous. Section 1.3 sets the scope of the Ginga approach as compared with the current query adaptation approaches. Section 1.4 lists my research contributions, and Section 1.5 lists the key assumptions that I made for the Ginga approach. I conclude the chapter with the dissertation organization in Section 1.6.

1.2 Motivation Application Scenario

There are more than 500 well known Bioinformatics data sources accessible on the Internet, which are used by many biologists around the world. For example, these data sources are used in the discovery of new drugs that may effectively treat serious diseases, such as AIDS, which is caused by Human Immunodeficiency Virus (HIV). There are several distinct features of such applications. First, the application needs to access and integrate information contained in different bioinformatic-specialized web services. Second, many sources experience unstable network connections and suffer from unpredictable latency fluctuations. Last, but not least, typical information collection processes in the bioinformatics domain are complex, requiring multiple steps with partial execution dependencies. In addition, the amount of data that needs to be processed can be larger than expected, causing variation in memory requirements.

Specifically, I use a next-generation-drugs discovery application as the running example scenario throughout the dissertation to illustrate the Ginga approach to distributed query processing adaptation.

Example 1 (Next-Generation-Drugs application) Consider a company specialized in developing Next Generation Drugs. In the rest of the dissertation, this company is referred to as NGD for presentation convenience. For each new drug that NGD starts to develop, the first step is to gather all the available information about the virus that the new drug will attack. In general, this information describes the nucleotides, proteins, and protein structures associated with the virus, along with assay results on how chemical compounds

can affect the virus and its sub-structures. From the nucleotides, the NGD pharmacologist can generate the proteins that the virus produces, and from the protein structures he identifies the protein compounds that can be manipulated using drugs. The drug discovery process is then guided mainly by the assay results of using different drugs against the virus proteins. The pharmacologist can also obtain further information about the virus by performing BLAST (Basic Alignment Search Tool) searches, which finds protein sequences similar to those found in the virus. This information may be relevant, for example, when determining what the possible side effects of the new drug over other similar proteins would be.

Typically, collecting the information described above requires contacting several web services, such as NCBI Entrez[2] for data on nucleotides, SWISS-PROT[3] for proteins, PDB[4] for protein structure, and NCBI BLAST[5] for similar protein sequences.[6] Figure 1 shows the latency fluctuations experienced with NCBI, SWISS-PROT, and PDB. Both graphs are plotted with the average response times measured every 15 minutes for 32 days (from April 11, 2002, to May 13, 2002) between 8:00 and 22:00 EST. Figure 1(a) shows the average response times from PDB, SWISS-PROT, and NCBI Entrez to a keyword query on "HIV". Figure 1(b) shows the average response times from NCBI BLAST searches on nucleotide and protein sequences that are similar to a nucleotide and a protein sequence found in HIV, respectively.

From Figure 1, it is observed that the response time of these popular web services can be very unpredictable. Concretely, I observe three interesting facts. First, NCBI BLAST service not only has relatively slower responses from 10:00 to 22:00 but also has frequent fluctuations throughout the day (see Figure 1(b)). A similar situation is experienced with PDB (see Figure 1(a)). In comparison, SWISS-PROT and NCBI Entrez are relatively stable. Second, even though NCBI Entrez (see Figure 1(a)) offers relatively stable performance, it experiences visible delays from 8:00 to 9:00 and from 16:00 to 17:00 daily between

[2] `http://www.ncbi.nlm.nih.gov/Entrez`
[3] `http://ca.expasy.org/sprot`
[4] `http://www.rcsb.org/pdb`
[5] `http://www.ncbi.nlm.nih.gov/BLAST/`
[6] For illustration purpose, in this example I omit the collection of assay results.

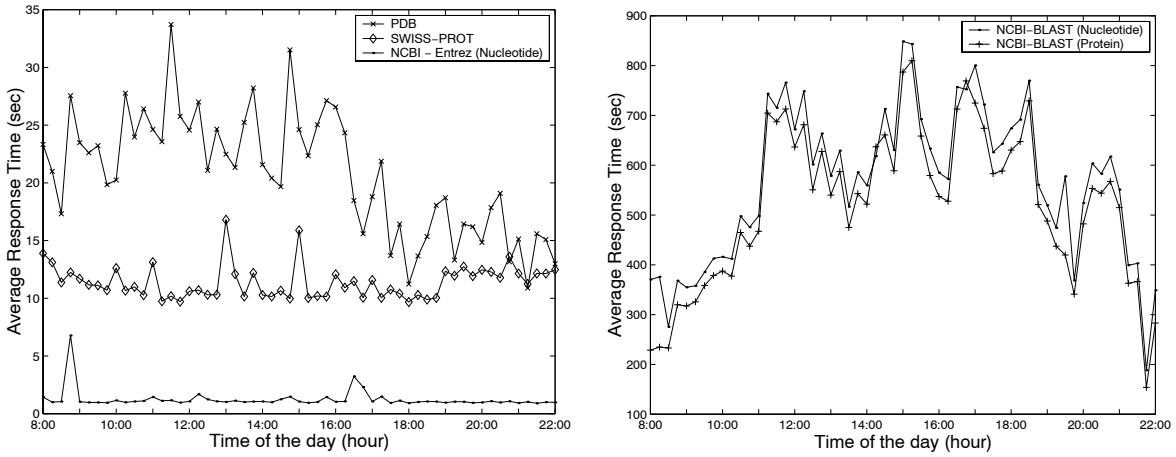


Figure 1: Average response times measured every 15 minutes for 32 days (from 4/11/2002 to 5/13/2002) between 8:00 and 22:00 EST. (a) A keyword query on "HIV" using PDB, SWISS-PROT, and NCBI Entrez; (b) NCBI BLAST searches on nucleotide or protein sequences, matching a nucleotide or a protein sequence found in HIV.

4/11/2002 and 5/13/2002. Furthermore, SWISS-PROT, in general, has up to a 12-second latency compared with NCBI; however, the visible delays at SWISS-PROT often occur from 13:00 to 14:00 and from 15:00 to 16:00. Third, although NCBI Entrez has a fast response time (on the order of a few seconds), waiting for PDB results can take up to 34 seconds, depending on what time of day the query request is posted to the site. Therefore, considering that the data collected from different web services have to be combined or integrated before delivering to the users, such differences in response time could become costly. The situation may become aggravated when NGD needs to execute this type of query repeatedly (practically, one for each new chemical compound that can potentially affect the virus). The cumulative effect of these delays can significantly affect the company's productivity.

One possible solution that NGD may use is the Ginga approach (see Chapter 4). Ginga uses a two-phase distributed query adaptation mechanism to overcome not only the unpredictability of the response time from web services due to end-to-end delays and latency fluctuations, but also the unexpected memory requirement changes when the amount of data that needs to be processed is larger than predicted.

Example 2 (NGD Web Service) To illustrate how NGD could use Ginga, let us assume

that this company has an internal web service NGD-WS that is responsible for gathering the data needed to begin the development of a new drug. The pharmacologist enters the virus name, and NGD-WS executes query Q expressed in an SQL statement[7] as shown in Figure 2(a), and it returns the result to the pharmacologist.

NGD-WS implements a distributed query scheduler (DQS), a query scheduling algorithm that consists of three main steps: *query routing*, *query decomposition*, and *query post-processing generation* [32, 31]. DQS takes as input the user query statement expressed in SQL and generates an initial query tree T annotated with the potential data sources that can be used to answer the query (query routing [30]).[8] Then, T is broken into a collection of subqueries, where each query is targeted at a given data source (query decomposition). Finally, an optimized query plan is constructed for combining the results returned from the subqueries (query post-processing generation) [32].

Let us assume that during query routing, NGD-WS has identified and selected the following bionformatics sources:[9] $NCBI$ for nucleotides, $SWISS\text{-}PROT$ for proteins, and PDB for protein structures. Then assume that after query decomposition and query post-processing generation, plan P depicted in Figure 2(b) is produced. In this plan, nodes represent atomic physical operators (e.g., select, project, join), and edges represent dataflow. NGD-WS executes the physical operators of plan P according to the iterator-based model [21], i.e., operators are scheduled in a left-deep recursive way (see Section 4.3.2 for details on the query execution model).

Now suppose that during the execution of P, $NCBI$ takes too long to respond or the data collected from this same source is larger than anticipated, thus affecting the memory requirements. If no adaptation is fired, the whole query execution will be delayed either due to unexpected delays or to an increase of paging. In this dissertation, I describe how the Ginga approach attempts to adapt the query execution plan to overcome these two

[7] Any other query language, such as XQuery, could very well be used instead.

[8] The initial query tree T is generated using the available global schema and completely disregarding the distribution of the data. Thus, T can be constructed using any optimization method proposed for single database management systems that produces a binary query operator tree [18].

[9] Although there are many sources that could be used for answering query Q, for simplicity, I assume that only these three sources were selected.

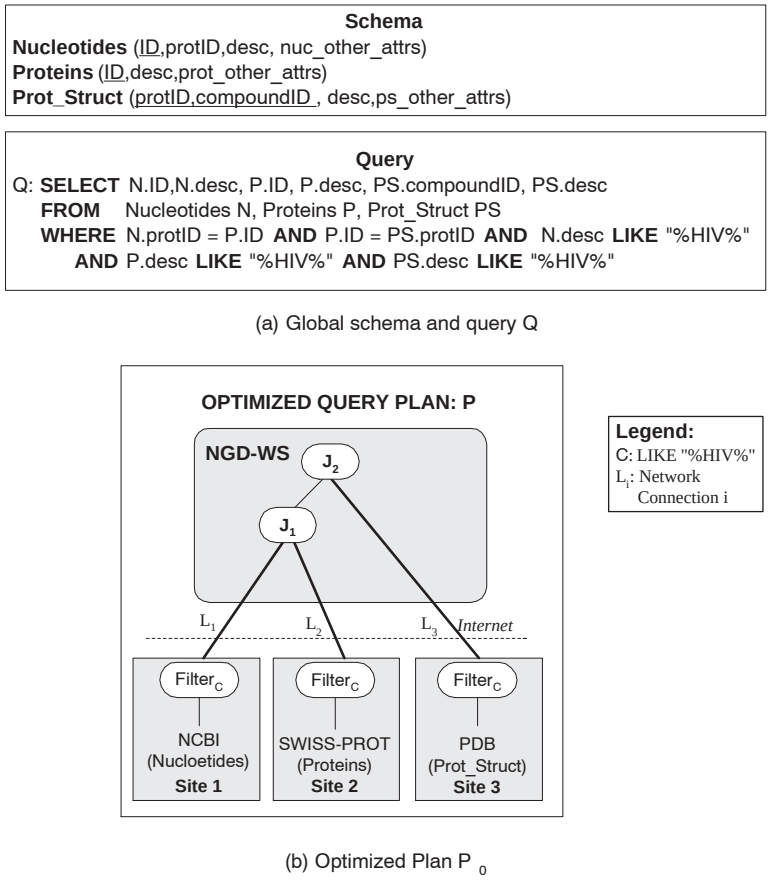


Figure 2: Query plan for collecting specific information on HIV.

problems, namely, changes in memory requirements and unexpected end-to-end delays. □

So far, I have discussed how the delays of bioinformatics web services can affect the productivity of my fictitious NGD company and have demonstrated how Ginga can improve NGD's productivity. In the next paragraphs, I describe the current cross-database search service provided by NCBI-Entrez and argue that Ginga could also improve this service.

The NCBI-Entrez Case: NCBI-Entrez offers the end-user a service for cross-database search over a number of different databases with bioinformatic-related information through the interface shown in Figure 3. Some of these databases are PubMed (citations and abstracts of biomedical literature), GenBank (nucleotide sequence database), and PopSet (population study data sets). However, this search service is limited to running the same keyword query in all of the databases maintained by NCBI-Entrez without combining any

information. If a user is looking for all related information on "HIV," as in Example 1, he has to combine the information retrieved by NCBI-Entrez manually.

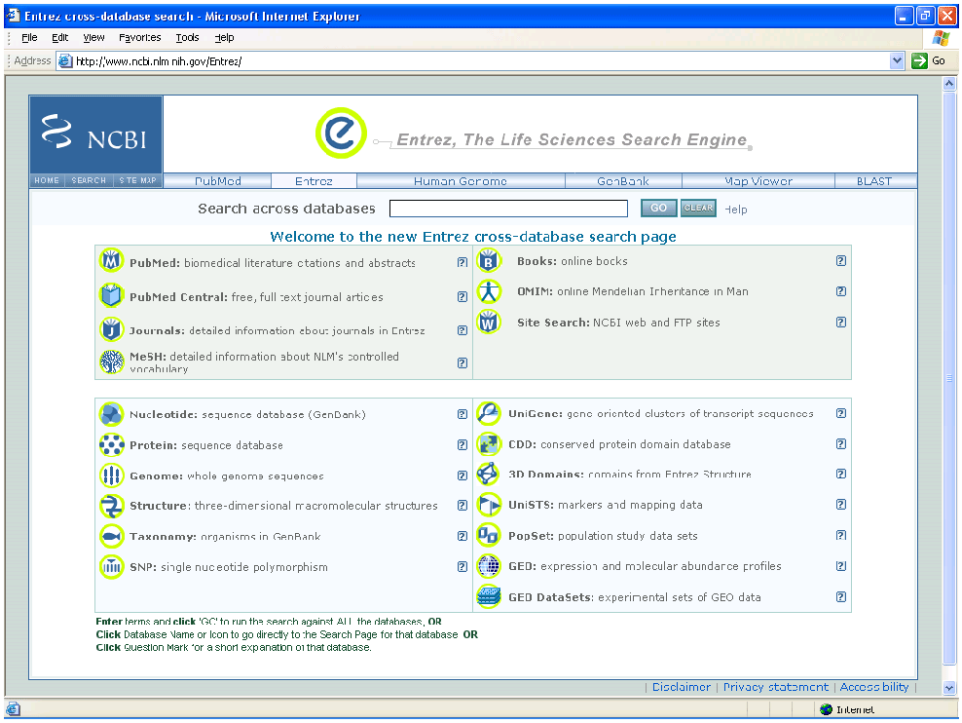


Figure 3: Snapshot of NCBI-Entrez cross-database search service (Source: `http://www.ncbi.nlm.nih.gov/Entrez/`).

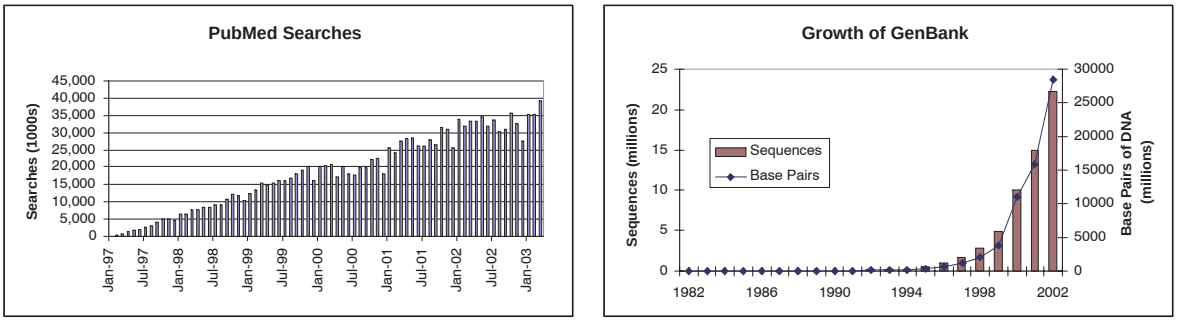
NCBI-Entrez could certainly benefit if an "information consolidation" (IC) service were provided through its portal. For instance, companies like NGD would be potential heavy users of this service. However, considering that NCBI-Entrez databases are geographically distributed, end-to-end delays are likely to occur during the execution of consolidation queries. In addition, given the high number of user accesses to their site, it is also possible that memory constraints will be observed during the execution of these consolidation queries due to server overload. Consequently, the response time of each request submitted to the NCBI-IC service would be quite unpredictable. To support these observations, I provide the following concrete statistical evidence obtained from the NCBI web site (`http://www.ncbi.nlm.nih.gov/`).

- The NCBI-Entrez databases are quite large, which translates into the transmission of

a large amount of data over network connections, when requests are submitted to the IC service. For example, consider the GenBank database. As of January 2003, this database stores approximately 28,507,990,166 bases in 22,318,883 sequence records (see Figure 4(b)), where the average size of each record is 20Kbytes. This means that currently (year 2003) the size of the GenBank database is about 440 Gigabytes.

If I submit a keyword search on "HIV" to GenBank, I get a result with around 111,500 records, a total of 200MBytes. Considering that these data will be transmitted back to the IC service through network connections that are not necessarily stable, the chances of experiencing end-to-end delays are high. As a matter of fact, end-to-end delays do occur, as reported in the graph in Figure 1(a).

- To give an idea of the chances of experiencing memory constraints due to server overload, consider the graph shown in Figure 4(a) delineating the growth of total of searches submitted to PubMed alone in the last six years. In January, 1997, the total number of searches was 163,000, and in March, 2003, this number had jumped to 39,238,000: an increase of about 240%! Assuming that this trend toward an increasing number of searches is similar to that of the other databases under NCBI-Entrez, it is very likely that the execution of IC requests will experience memory constraints due to server overload.



(a) (b)

Figure 4: (a) PubMed searches (Source: `http://www.ncbi.nlm.nih.gov/About/tools/restable_stat_pubmed.html`); (b) Growth of GenBank (Source: `http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html`).

Based on this evidence, it is clear that Ginga could certainly be a good approach for the NCBI-IC service to handle the unexpected response time fluctuations.

1.3 Adaptive Query Processing and the Ginga Approach

Adaptive query processing has been recognized as an important area with significant research activities in both Internet data management and quality-of-service communities. Most of the previously proposed approaches to adaptive query processing can be classified as either *predefined* or *reactive* [4].

Predefined approaches generate, prior to runtime, a set of *predefined* plans where each plan is optimized for one possible combination of values that resource parameters (e.g., memory size, network transfer rate) may assume at runtime. At start-up time (i.e., just before starting the query execution), the query engine reads the current values of the resource parameters and selects the appropriate plan that fits these values. Figure 5 shows the control flow of predefined approaches.

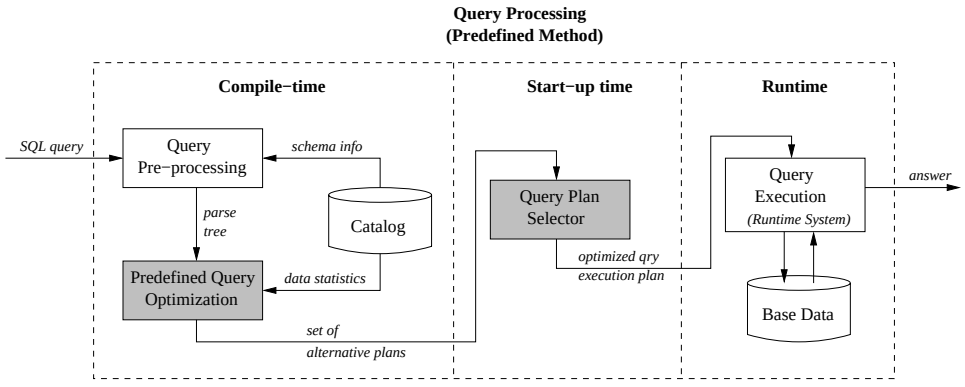


Figure 5: Control flow of *predefined* approaches to query adaptation.

According to the size of the set of alternative plans, predefined solutions are further classified as *parametric query optimizer* and *selective query optimizer*. The former generates one optimized plan for each possible combination of resource parameter values, while the latter generates plans for only those combinations of values that represent interesting situations where adaptation is beneficial. An *interesting situation* is represented by a combination of resource parameter values that are likely to occur during query execution and

may represent a fluctuation in the runtime system.

Most of the current predefined solutions are based on the parametric query optimizer approach [27, 12]. The advantage of these solutions is the guarantee that the query engine will start executing a query plan that is optimized for the current runtime resources availability. However, if not designed properly, predefined approaches can incur considerable overhead at compile-time to generate a large number of alternative query plans where only a few are likely to be used. In addition, these approaches have the following limitation: once the optimized plan is selected at start-up time, the query engine does not re-optimize the plan in case of unexpected changes to the runtime resources availability (e.g., unexpected end-to-end delay).

At compile-time, reactive approaches generate a query plan that, at runtime, monitors the resource parameters that are likely to change (e.g., transfer rate of unstable network connections). Upon detecting changes in these parameters, the query reacts by generating and switching to an alternative query plan that takes these changes into consideration. Figure 6 depicts the control flow of reactive approaches.

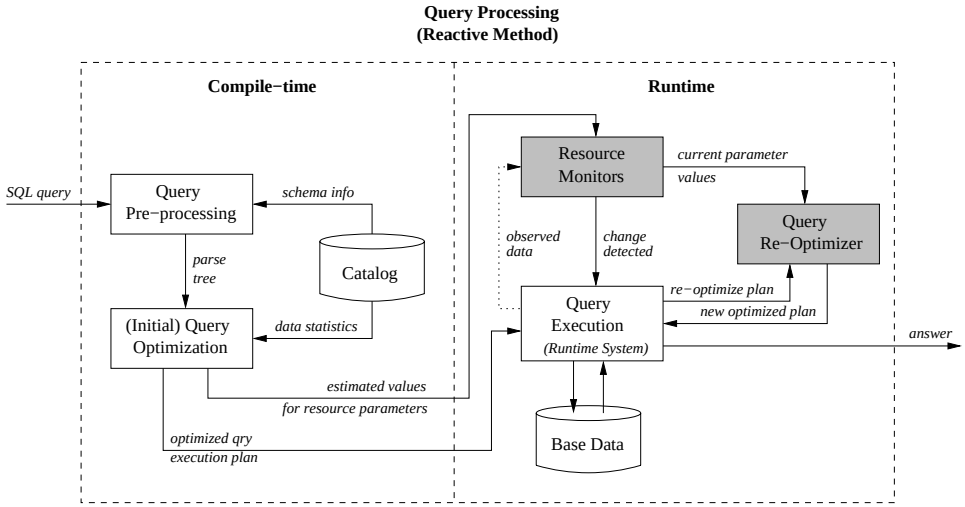


Figure 6: Control flow of *reactive* approaches to query adaptation.

When compared to the predefined solutions, the advantage of reactive approaches is runtime adaptation to dynamic changes to the runtime environment. But reactive solutions have two problems: The first problem is the overhead incurred during query execution to

generate the alternative plan. The second problem is that they may react to any change in the runtime resources, even if this change represents a small system fluctuation.

In general, existing solutions to predefined and reactive methods suffer from one major weakness: they are targeted to adapt the query execution plan to a single type of runtime change (e.g., for distributed queries, the change in data transfer from remote data servers). In addition, the combination of these solutions to address different runtime changes is not a trivial task.

To address the problems and limitations of predefined and reactive approaches, in this dissertation I present *Ginga* [41, 43], a hybrid query adaptation approach, which is a combination of both predefined and reactive methods. The Ginga approach combines predefined (compile-time) alternative query plan generation with reactive (run-time) monitoring of runtime changes and consequent switching to a better-suited plan. Figure 7 shows the control flow for the hybrid approach to query adaptation.

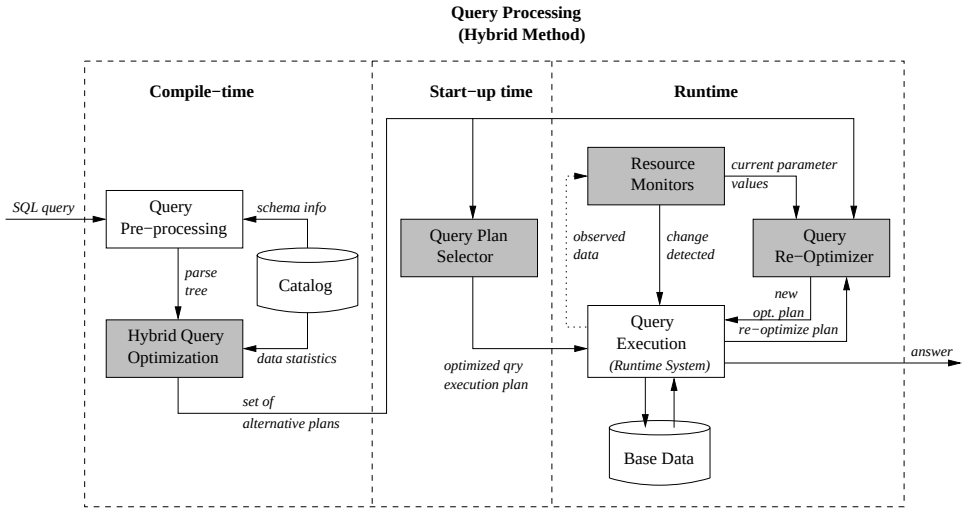


Figure 7: Control flow of *hybrid* approaches to query adaptation.

Ginga improves existing predefined approaches in two ways: First, Ginga generates optimized plans for only the combination of resource parameter values that are likely to occur at runtime (i.e., interesting situations) instead of generating a plan for each possible combination (selective query optimizer), thus reducing the overhead at compile-time. Second, Ginga supports query adaptation not only at start-up time, but also at runtime. In

contrast, predefined solutions adapt only at start-up time.

Ginga also improves the reactive approaches in two different aspects: First, Ginga bypasses the overhead required to generate the alternative plans at runtime. Second, unlike reactive approaches, Ginga does not react to every single change to the runtime resource availability. Based on the set of interesting combinations of resource parameter values, Ginga is able to determine whether a change in the runtime resources represents a small fluctuation or a situation in which it is worth firing the adaptation.

To address the problem of enabling query adaptation to more than one type of runtime change, Ginga has at its core an adaptation space model that allows the systematic organization and combination of different solutions to predefined and reactive methods.

In this dissertation, I assume that I can efficiently generate, at compile-time, the set of alternative query plans. My main focus is on how one can organize and combine these predefined plans into an adaptation space used for query adaptation so that I can efficiently cope with various types of runtime changes. Investigating the costs and the mechanics involved in the generation of alternative plans is part of my future research work.

I also assume that the transition from one query plan to another at runtime can be done "smoothly", i.e., the consistency of the query execution state is maintained (e.g., data will neither be missed nor will duplicated data be generated). When this transition cannot be done smoothly, I may need to have a *dynamic* query re-optimizer which generates an optimized plan for executing the remainder of the query. The work described in [29] proposes a re-optimization approach when operator selectivities are detected to be different from those assumed during plan generation. [59] studies the re-optimization approach for the case when end-to-end delays are detected during the execution of a distributed query. Incorporating dynamic query re-optimization into Ginga approach is part of my future research work.

1.4 Dissertation Contribution

In this dissertation, I provide the following contributions:

- The adoption of the Adaptation Space concept combined with feedback-based control

mechanisms for coordinating and integrating different kinds of query adaptations to different runtime failure types, namely, memory constraints and end-to-end delays.

- The development of a systematic approach, called *Ginga*, to integrate the adaptation space with feedback control that allows me to combine predefined and reactive adaptive query processing, including policies and mechanisms for determining when to adapt, what to adapt, and how to adapt.

- A detailed study of how to adapt to two important failure types, and their combination, encountered during the execution of distributed queries: memory constraints and end-to-end delays.

1.5 Dissertation Main Assumptions

For work that I describe in this dissertation, I make the following assumptions:

1. Queries submitted to Ginga are processed over *read-only* data (e.g., data warehouse).

2. Network connections between client and servers are independent of each other, in the sense that the failure of a connection does not affect the others.

3. Remote data servers deliver the requested data as a sequence of small fixed-size pages. I specify the exact size of each page in my experiments.

4. Queries are executed in a multi-user environment, where memory blocks are shared.

5. Queries submitted to Ginga involve the processing of Gigabytes of data.

1.6 Dissertation Organization

This dissertation is organized as follows: Chapter 2 revises the related work. Chapter 3 describes the adaptation space model. Chapter 4 presents the Ginga approach which has at its core the adaptation space model. In Chapter 5 and Chapter 6, I use Ginga to investigate the benefits and trade-offs of query adaptation when coping with memory constraints and end-to-end delays. Chapter 7 presents a study on Ginga performance characteristics when both end-to-end delays and memory constraints are detected at runtime. I conclude the

dissertation in Chapter 8 with a summarization of the work presented and a discussion of future research work.

CHAPTER II

RELATED WORK

2.1 Research Context

There are three main research streams that are related to the Ginga approach: program generalization and specialization, adaptation space generation, and adaptive query processing.

2.2 Program Generalization and Program Specialization

The adaptation space based approach for program adaptation described Chapter 3 can be seen as a seamless generalization and evolution of program specialization concept [13, 57, 60, 48, 37]. Program adaptation is a generalization of program specialization in both the objectives of adaptation and the methods of adaptation. Similar to program specialization, program adaptation presents an approach for adapting a generic program component to a given use context. However, *program specialization* is commonly viewed as a means for performance optimization. More concretely, given a use context, program specialization aims at achieving performance gains by adapting programs to the specialized code where all the aspects, which do not directly concern the given context, are eliminated. Thus, the cost of conducting unnecessary consistency checking on the irrelevant aspects is avoided. *Program adaptation* aims at a wider spectrum of enhancement of system capabilities, including performance, quality of service, and survivability. Program adaptation provides a uniformed framework for adapting programs to different levels of code specialization or capability relaxation (degradation) according to the various quality of service requirements and resource availability. For instance, thrashing and crashes occur in system saturation for many statically adaptive resource management algorithms, including CPU, memory, and network congestion. Program adaptation supports alternative responses to saturation control, maintaining program and system stability and progress, instead of thrashing.

2.3 Adaptation Space Generation

Another stream of related work to the Ginga approach is on the generation of adaptation spaces, which can be created *statically* (i.e., at compile-time) and/or *dynamically* (i.e., at runtime). Two classical projects on to the generation of adaptation space *at compile-time* are Volcano Dynamic Plan [23, 12] and Parametric Query Optimization [27]. Volcano Dynamic Plan uses *choose-plan* operators to enable the optimizer to cope with the inability to precisely estimate all the resource parameter values at compile-time. For every set of incomparable plans for each operator, the optimizer inserts in the query tree a choose-plan operator, which will group these plans. For example, if the optimizer cannot determine the selectivity of a SELECT operator, all possible alternative ways of executing this operator (e.g., file scan or index access) will be grouped using a choose-plan operator. At start-up-time, when the exact values for the unknown resource parameters (e.g., operator selectivity) are available, the optimizer re-calculates the cost of each alternative plan under choose-plan operator and chooses the plan with the least cost. The query then runs to its completion. [12] has implemented an algorithm based on Volcano Query Optimizer Generator [22] that generates alternative plans only for the case when operator selectivities are unknown.

Parametric Query Optimization [27] attempts to generate one optimized query execution plan for each possible combination of values for the resource parameters that are unknown at compile-time. At start-up time, based on the current values for these parameters, the optimizer chooses the respective optimized query execution plan. [27] has implemented randomized algorithms for generating alternative plans for different sizes of memory (failure mode: memory constraints). These algorithms were based on Simulated Annealing, Iterative Improvement, and Two Phase Optimization [26].

One representative work on the *dynamic* generation of adaptation space is [50]. This work proposes an adaptive resource allocation (ARA) mechanism to be used in conjunction to high-performance real-time applications. These applications have very strict timing constraints and very dynamic resource requirements. Guaranteeing the proper resource allocation to these applications at all time so that they can meet their timing constraints is the challenge that ARA addresses. ARA dynamically constructs an adaptation space,

where each adaptation case represents the best resource allocation to the running applications at a given moment. Whenever this allocation is detected to be no longer appropriate, ARA will dynamically generate another resource allocation (i.e., adaptation case).

Another approach on dynamic generation of adaptation space is the work by Bodorik et al. [8]. According to [8], the execution of a distributed query proceeds through three phases: Phase one monitors the execution progress of the query. Phase two decides whether a new strategy for executing the query should be computed. Phase three is a corrective one where the current execution maybe aborted and a new execution is initiated. Similar solution was proposed in Rdb/VMS [5].

Antoshenkov [5] presents a solution for generating the most appropriate query execution plan by suggesting a competition method. For a given query, different execution plans are fired concurrently, and the most appropriated one is kept. More specifically, the approach addresses only the retrieval of a single table. Given that it is possible to have different indexes for that table, the question is to how to determine the best combination of indexes to be used for the query. After an initial stage (at compile time), which arranges the available useful indexes into single or combined scan strategies, two communicating processes are fired: foreground and background. They compete to "find" the best strategy for the table retrieval, where the best one runs to completion, having the other plan aborted.

In this dissertation, I do not describe, implement, or evaluate a detailed algorithm for generating the adaptation space for the execution of distributed queries. Developing and evaluating such algorithm is part of my future research work.

2.4 Adaptive Query Processing

Adaptive query processing has been an important area of research, starting from early 90's [27, 23, 12], and continuing to today [29, 2, 58, 59, 4, 25, 28, 6, 39, 1, 56, 17, 35]. In section 1.3, I have broadly classified adaptive approaches to query processing into predefined and reactive methods. Another way to group the existing work on adaptive query processing is based on which *runtime changes* they attempt to adapt the query execution plan. In that respect, I identify two major groups: (1) approaches that adapt to changes in memory

constraints, and (2) approaches that adapt to fluctuation of end-to-end delays. Changes in memory constraints occur when the memory requirements for the execution of a query plan varies at runtime.

2.4.1 Adaptation to Memory Constraints

There are three main problems related to memory management for query execution under memory constraints. The first problem focuses on how to allocate memory blocks to concurrent operators within a query in a way that will minimize the total query response time. Different approaches to memory allocation were proposed in the literature [38, 10] for addressing this problem. These approaches, which assume that query execution follows the Segmented Execution Model (see Section 4.3.2), can be broadly classified into *static* and *dynamic* memory allocation strategies. The former is executed at start-up time whereas the later is applied at runtime, before executing each segment. [38] has proposed and evaluated four static memory allocation strategies, which included Max2Min (also called SMALL). As reported in [38], in general Max2Min presented best performance. However, the main problem with static strategies is that they allocate memory based on *estimated* size of intermediate results. If there is a significant difference between actual and estimated sizes of these results, query plans using static memory allocation strategies may have sub-optimal performance. [10] addresses this problem with a dynamic strategy that I call in this dissertation AlwaysMax. As demonstrated in [10], in general AlwaysMax presents better performance than Max2Min. However, as shown in Section 5.4.2, there are situations where Max2Min can be a better choice than AlwaysMax. Ginga considers both strategies at runtime, before executing each segment, and selects the one that yields the best response time.

The second problem associated to managing memory constraints represents the situation where memory has been properly allocated to the query but the estimated sizes for the intermediate results were inaccurate, which may result in a significant increase of paging. Research work on query re-optimization [29, 6, 39, 46, 47] addresses this problem. Mid-query re-optimization [29] attempts to re-optimize the query execution whenever a

significant difference between estimated and observed values for operators' selectivity is detected at runtime. Eddies [6] suggests the re-ordering of operators in the presence of configuration fluctuations of the runtime environment during query execution. Similar solution is proposed by dQuob [46, 47], which re-order the select operators on the fly as their selectivity changes. [39] assumes that the estimated values for operators' selectivity and base relation sizes are not accurate and constructs the query plan at runtime as data become available. Ginga is similar to these approaches in the sense that it also provides query re-optimization. However, these approaches are limited to use only the adaptation actions that they propose. Extending these approaches to handle runtime changes other than inaccurate estimated values is a non-trivial task. In contrast, Ginga uses a unified simple model based on adaptation space that allows the incorporation of other dynamic adaptation methods to handle different runtime changes in memory constraints.

The last problem related to managing memory constraints is how to allocate memory blocks available in the system among concurrent queries, which is addressed by the research work on multi-query optimization [14, 61]. [14] proposes a global optimization where the idea is to optimize all query plans at once while determining how to distribute among the concurrent queries the memory blocks available in the system. [61] introduces the concept of *return on consumption* for studying the overall performance improvement of individual join queries due to additional memory allocation. The inclusion of multi-query optimization techniques into Ginga system is part of my future research work. In this dissertation, I assume that the Memory Manager will handle the problem of distributing the system memory blocks among the concurrent queries.

Methodologically, the Ginga system follows my previous research on program specialization [37]. In program specialization, I take advantage of quasi-invariants to eliminate unnecessary work. Similarly, Ginga uses the memory constraint predicates in adaptation conditions to optimize query execution. The monitoring of adaptation events is similar to the guarding of quasi-invariants, when an invalidation causes re-plugging of specialized code. Unlike program specialization, which relies primarily on program manipulation methods such as partial evaluation, Ginga uses many different techniques such as switching join

algorithms, operand ordering, and memory allocation strategies to optimize query processing.

2.4.2 Adaptation to End-to-End Delays

In this section, I review the adaptive approaches to distributed query processing that cope with fluctuation on end-to-end delays. During the execution of a distributed query, end-to-end delays occur possibly due to the following reasons: connections to the remote data sources experience unpredictable bandwidth and latency fluctuations, remote server is overloaded, or both network and remote server experience performance problems. Adapting to such variations is important because they directly affect the query response time as the remote data is delivered at unpredictable rate.

Two reactive approaches for adapting to end-to-end delays that are closely related to Ginga are *Query Scrambling* [2, 4, 59] and *Dynamic Query Scheduler* [9]. Query Scrambling uses materialization and operator synthesis to adapt the execution of distributed queries in the presence of initial delay and bursty arrival. Initial delay occurs when there is a delay from the remote data server on responding to the data request posted by the client. After that, no further delay is observed, and data transmission resumes to its expected rate. With bursty arrival, the data downloaded from the remote server is delivered at an unpredictable rate.

The reactive approach implemented by the Dynamic Query Scheduler (DQS) [9] also adapts the execution of distributed queries in the presence of slow delivery. The slow delivery state is detected when the data transmitted by the remote data source is being delivered at a slower rate than the one expected.

In comparison, Ginga uses a unified framework (adaptation space model), which is powerful enough for me to study initial delay, bursty arrival, and slow delivery, as well as other constraints such as memory shortage and sudden unavailability of remote data servers.

There are several other approaches to reactive adaptive methods that address adaptation to end-to-end delays. MOOD [39], a heterogeneous distributed database, optimizes the query execution at runtime as the results from the sub-queries become available. The

Tukwila project [28] proposes query (re-)optimization based on rules defined over possible end-to-end delays. XJoin [58], a modified version of hash join, provides query adaptation at the operator level by transparently absorbing the fluctuation of the data delivery rate from remote operands. While these projects propose more sophisticated operators and mechanisms to support adaptation, Ginga uses a unified simple model and feedback-based monitoring mechanism to support both predefined and reactive adaptation using the existing relational operators.

CHAPTER III

ADAPTATION SPACE MODEL

3.1 The Core of Ginga Approach

The core of Ginga approach is the adaptation space model [42], which can be applied not only for query processing adaptation, but also to any other system (or program, or function) that requires adaptation. Taking the scenario from my motivation application described in Section 1.2 where a web service needs to adapt its execution to unexpected runtime changes, I describe the adaptation space model assuming that I want to adapt a web service.

The core of this adaptation model is the notion of adaptation space and the mechanisms for coordinating and integrating different kinds of adaptations. An adaptation space is defined by a use context and a partial order of adaptation cases. Each adaptation case describes a specific adaptation of a program or component of a system. Given a program or component, say T, the adaptation process typically consists of three steps:

Step 1: what to adapt construct an adaptation space for T where preferred adaptation alternatives are specified by a use context and a partial order of adaptation cases (see Section 3.3 for concrete syntax);

Step 2: when to adapt monitor the use conditions of each specified adaptation alternative in the adaptation space of T, and notify the adaptation engine whenever a use condition becomes true;

Step 3: how to adapt turn on the appropriate adaptation case to prevent the running program or component from unnecessary lost of performance, thrashing, abnormal exit, or crash. The selected adaptation case fulfills this job by redirecting the ongoing program to run using specialized code that can accomplish the intended task through partial evaluation, such as using specialized code with either fewer resources or requiring fewer capabilities.

There are three main thrusts of adaptation space based approach: First, program adaptation presents a unified framework for representing and viewing of multi-dimensional adaptation context and adaptation opportunities for a given program or component of a web service. Second, program adaptation provides a uniformed way of capturing and coordinating different kinds of adaptation at different levels of a web service. Third, program adaptation promotes a declarative and incremental approach to define the multi-dimensional adaptation context and adaptation alternatives, allowing the seamless incorporation of new adaptation behavior of a web service into the existing adaptation framework.

The rest of this chapter proceeds as follows: Section 3.2 motivates the adaptation space based approach using the example describe in Section 1.2. Section 3.3 provides an informal introduction to the concept, the syntax, and the semantics of adaptation cases and adaptation spaces. Section 3.4 presents the formal semantics of the adaptation space model. I illustrate the adaptation space concept and semantics by walking through the application scenario introduced in Section 3.2. Section 3.5 discusses various relationships between two adaptation spaces that may have potential interest to the reasoning of adaptation behavior of a given program component. Section 3.6 examines the issues that should be addressed by the adaptation space model. I summarize the chapter in Section 3.7.

3.2 Walkthrough Example

Before presenting the details of the adaptation approach, let me briefly illustrate its application using the NDG-WS web service scenario described in Example 2 (Section 1.2). NGD-WS is responsible for gathering the needed data for starting the development of a new drug. The pharmacologist enters the virus name, and NGD-WS executes query Q expressed in a SQL statement shown in Figure 8(a), and returns the result back to the pharmacologist. The information on nucleotides, protein, and protein structures is typically provided by remote web services such as NCBI Entrez (nucleotides), SWISS-PROT (proteins), and PDB (protein structures). However, the response time from these web services can be very unpredictable as discussed in Section1.2. Considering that NGD-WS has to combine the collected data before delivering them to the pharmacologist, the difference

in response times could become costly.

One possible solution for NGD-WS is to adapt the execution of query Q by modifying the data collection and integration process so that the fluctuation on end-to-end delays can be masked. I illustrate next how to implement this solution using adaptation space.

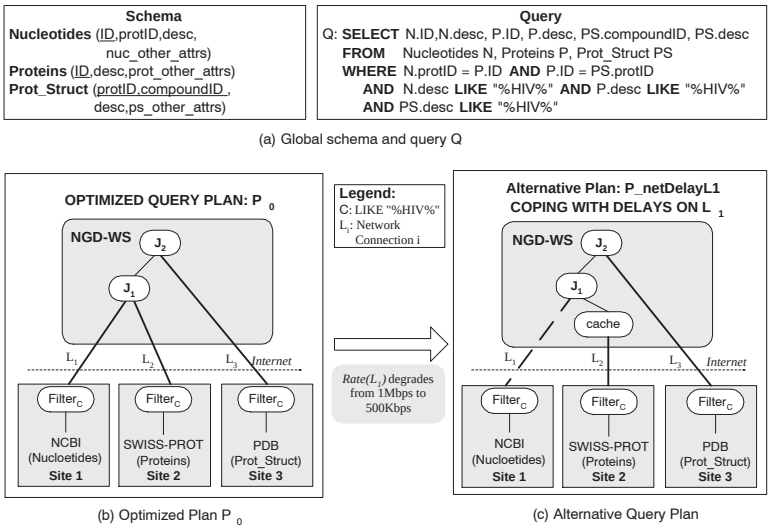


Figure 8: (a) Query Q and associated global schema; (b) Optimized query plan P_0; (c) Alternative query plan $P_{netDelayL1}$ for coping with delays on network connection L_1.

Example 3 (Adapting to End-to-End Delays) Let us assume that NGD-WS generates the optimized query plan P_0 (see Figure 8(b)) for executing Q, and that the expected transfer rate for the network connection[1] L_i is $Rate(L_i) = 1Mbps$, for $1 \leq i \leq 3$. Now, suppose that as NGD-WS starts the execution of P_0 the transfer rate of connection L_1 degrades from 1Mbps to 500Kbps causing the data from *NCBI* to be slowly delivered. Assuming that P_0 is executed according to the iterator-based model [21], i.e., query operators are scheduled in a left-deep recursive way, without adaptation, the execution of P_0 will be delayed. One way to solve this problem is to start collecting the data from *SWISS-PROT* and caching it locally *while* slowly receiving data from *NCBI*. Figure 9 presents a sketch of such an adaptation case, declaring an adaptation opportunity: mask the slow delivery

[1] I use the term *end-to-end delay* to refer to the delays experienced by the clients due to both bandwidth fluctuation in networks and contention at the servers. In this chapter, I concentrate my study on end-to-end delays caused by bandwidth fluctuation, which results in slow delivery, i.e., the data from the remote server arrives at a rate slower than expected.

of data from *NCBI* while executing other parts of the query. If *NCBI* is completed downloaded by the time the data from *SWISS-PROT* is fully cached, the query execution can then proceed without affecting considerably the final response time.

Based on this adaptation specification, the adaptation engine can infer that this adaptation can be done at compile time if the connection to remote source *NCBI* is known to be slow. Otherwise, the adaptation must be done at runtime, and query plan P_0 will be replaced by $P_{netDelayL1}$ shown in Fig. 8(c).

More concretely, the adaptation case ac_caseNetDelayL1 specifies that the initially optimized query plan P_0 will be transformed to plan $P_{netDelayL1}$ whenever data from *NCBI* is slowly delivered. The choice of switching P_0 to the specialized query plan $P_{netDelayL1}$ can be made at compile time only if the use-condition of adaptation case ac_netDelayL1 can be verified at compile-time. Otherwise, the decision should be made at runtime (see Section 3.6 for further discussion).

ADAPTATION CASE ac_caseNetDelayL1 **adapts execution of query** Q.
goal: query adaptation when Rate(L_1) degrades.
context
resource objects
Nucleotides = {NCBI}; *Proteins* = {SWISS-PROT}; *Prot_Struct* = {PDB};
NetworkLinks = $\{L_1, L_2, L_3\}$
memory_size(Q) `// Memory blocks allocated for executing` Q.
resource object parameters
must-provide
ID, desc **from** *Nucleotides*; ID, desc **from** *Proteins*;
compoundID, desc **from** *Prot_Struct*; bandwidth **from** *NetworkLinks*
require
Nucleotides.protID, *Proteins*.ID **where** *Nucleotides*.protID = *Proteins*.ID
Proteins.ID, *Prot_Struct*.protID **where** *Proteins*.ID = *Prot_Struct*.protID
Nucleotides.desc **where** *Nucleotides*.desc LIKE *virusName*
Proteins.desc **where** *Proteins*.desc LIKE *virusName*
Prot_Struct.desc **where** *Prot_Struct*.desc LIKE *virusName*
adapted query plan: $P_{netDelayL1}$ (see Fig. 8(c))
adaptation condition
$(Rate(L_1) < 1Mbps) \wedge (Rate(L_2) \geq 1Mbps) \wedge (Rate(L_3) \geq 1Mbps) \wedge$
available(*Nucleotides*) = `true` $\wedge$ available(*Proteins*) = `true` $\wedge$ available(*Prot_Struct*) = `true`
adaptation action
"collect and cache data from *Protein* while slowly receiving data from *Nucleotide*."
end ac_caseNetDelayL1

Figure 9: Adaptation case that adapts the execution of Q when $Rate(L_1) < 1Mbps$.

ADAPTATION CASE ac_initialP **adapts execution of query** Q.
goal: fast response time with survivability to network delays.
context
resource objects
// Same as **resource objects** described in ac_caseNetDelayL1 (Fig. 9).
resource object parameters
// Same as **resource object parameters** described in ac_caseNetDelayL1 (Fig. 9).
adapted query plan: P_0 (Fig. 8(b))
adaptation condition
available($Nucleotides$) = `true` $\wedge$ available($Proteins$) = `true` $\wedge$ available($Prot\_Struct$) = `true` $\wedge$
$\{\forall \ell \in NetworkLinks, Rate(\ell) \geq 1Mbps\} \wedge$
$\{J_i$ is hash-join and $|left\_Operand(J_i)| < |right\_operand(J_i)|$, where $i = 1, 2\} \wedge$
$\{memory\_size(Q)$ matches the number of blocks expected by the optimizer$\}$
adaptation action
none.
end ac_initialP

Figure 10: Initial adaptation case definition.

ADAPTATION CASE ac_netDelayL2 **adapts case** ac_initialP.
goal: query adaptation when Rate(L_2) degrades.
context
resource objects
// Same as **resource objects** described in ac_caseNetDelayL1 (Fig. 9).
resource object parameters
// Same as **resource object parameters** described in ac_caseNetDelayL1 (Fig. 9).
adapted query plan: P_{12} (see Fig. 14)
adaptation condition
$(Rate(L_2) < 1Mbps) \wedge (Rate(L_1) \geq 1Mbps) \wedge (Rate(L_3) \geq 1Mbps) \wedge$
available($Nucleotides$) = `true` $\wedge$ available($Proteins$) = `true` $\wedge$ available($Prot\_Struct$) = `true`
adaptation action
"collect and cache data from *Prot_Struct* while slowly receiving data from *Protein*."
end ac_netDelayL2

Figure 11: Adaptation case that adapts the execution of Q when $Rate(L_2) < 1Mbps$.

3.3 Adaptation Space Model: An Overview

In this section, I describe the concept of adaptation case and adaptation space, and present an overview of the syntax for specifying adaptation cases and adaptation spaces. I illustrate the core components for declaring adaptation cases and adaptation spaces by examples.

3.3.1 Adaptation Cases

Adaptation cases define the adaptation behavior of a web service program in anticipation of resource shortage or in the need for performance optimization. More concretely, an adaptation case defines a context for adaptation through program specialization or generalization.

ADAPTATION CASE ac_netDelayL12 **adapts case** ac_netDelayL1.
goal: query adaptation when Rate(L_1) and Rate(L_2) degrade.
context
resource objects
// Same as **resource objects** described in ac_caseNetDelayL1 (Fig. 9).
resource object parameters
// Same as **resource object parameters** described in ac_caseNetDelayL1 (Fig. 9).
adapted query plan: P_{21} (see Fig. 14)
adaptation condition
$(Rate(L_1) < 1Mbps) \wedge (Rate(L_2) < 1Mbps) \wedge (Rate(L_3) \geq 1Mbps) \wedge$
available(*Nucleotides*) = `true` $\wedge$ available(*Proteins*) = `true` $\wedge$ available(*Prot_Struct*) = `true`
adaptation action
"collect and cache data from *Prot_Struct* while slowly receiving data from *Nucleotides* and *Proteins*."
end ac_netDelayL12

Figure 12: Adaptation case that adapts the execution of Q when $Rate(L_2) < 1Mbps$ *and* $Rate(L_2) < 1Mbps$.

For instance, it identifies what components of a web service should be specialized for adaptation to resource availability or for performance optimization in the presence of certain quasi-invariants [15, 60, 48].

Generally, adaptation cases are concerned with data and control flow exchanges between a collection of program components, modules, and systems. Resources[2] are the currency of exchange among these system components. In object-oriented systems, all resources are ultimately provided by *classes*. However, it is not always the case in most of the legacy systems. Therefore, in the proposed adaptation space model, both classes and modules are units that provide a set of resources and require another set of resources. Without loss of generality, let us assume in the sequel that the granularity of an adaptable program component is a class described by a set of methods attached to it or a module described by its interfaces (input required) and outerfaces (outputs required). Each adaptation case is specific to a class or a module (a program) in the target system. I refer to such a class or a module as a target unit of the adaptation in the rest of this paper.

Specifically, given a target web service and a reference program unit, an adaptation case description includes

[2]A resource is any entity that can be named in a programming language (e.g., variables, contents, procedure, type definitions, etc.) and which can be actually be made available for reference by other program components, modules, or subsystems within a large software system.

- *case name*, described by a character string;
- *goal description* (optional), specified by the synopsis of the goal;
- *context* of the case, defined by a list of resource parameters internal to the reference program unit and critical to the specification of adaptation behavior of the reference program. In addition, a pointer to the adapted code for the reference program is provided.
- *use condition*, expressed by a Boolean expression that indicates when it is appropriate to use this adaptation case; Variables in use conditions are defined in the adaptation case context;
- *adaptation action* describes the action that is fired when the use condition is satisfied.

To enable on-demand generation of adaptation action, I need to capture **type**, **state**, and **alternation** of each resource parameter specified in the adaptation case context.

- The **type** of a resource parameter can be either primitive types such as *integer, char, float, Boolean*, or structured types as those defined in object-oriented systems.
- The **state** of a resource parameter includes *provide, must-provide, require, must-require*. Such state information distinguishes (1) the *required* resources from the *provided* resources, and (2) the *mandatory* provide-parameters or require-parameters from *optional* ones.
- The attribute **alternation** indicates the possible alternatives of the given resource and a list of degradation attributes such as extra-cost, freshness, extra-overhead, to name a few. I can use the predicates as a form of specification of alternation semantics to allow the list of possible alternatives to be added at runtime.

The complete syntax of the adaptation case description language is still under development. Below, I present a simple example to illustrate the general idea of adaptation cases. I provide a richer example in the next section.

Example 4 Recall the query plan depicted in Figure 8(b). Figure 10 presents the initial adaptation case for the execution of query Q, which describes the target program (in this case the query Q) in terms of the adaptation model syntax and the particular adaptation context defined in its adaptation space (see Section 3.3.2 for further detail). For instance, from the SQL statement given in Figure 8(a), Q provides four output data items from *Nucleotides* and *Protein_Structure* and three output data items from *Proteins*, and requires the input condition that the data collected is relevant to HIV. I list these resources along with the network links in the initial case ac_initialP as critical parameters for defining the adaptation behavior of Q.

I can define as many as needed adaptation cases for a given query execution plan. For instance, in addition to degradation of $Rate(L_1)$ (Figure 9), I can also define one adaptation case for Q that deals with degradation of only $Rate(L_2)$ (Figure 11) and another that deals with degradation of both $Rate(L_1)$ and $Rate(L_2)$ (Figure 12).

The automatic generation of program adaptation code from the original code may not always be feasible, especially when the target software is a legacy system with sophisticated procedures and state transitions. The potential for generating specialized code for adaptation cases primarily depends on the declarative nature of the program. I believe that programs written in declarative languages are more likely to be supported for automatic generation of specialized code. The development of mechanisms for semi-automatic or automatic generation of specialized adaptation code for given adaptation cases is part of my ongoing research work.

3.3.2 Adaptation Space

An adaptation space consists of a reference context which sets up the scope of adaptation behavior and a partial order of adaptation cases, each case representing a specific way of adapting the same program unit. In addition to the set of adaptation cases, an adaptation space description includes the following header information:

- *Name* of the adaptation space, served as a unique identifier;

- *Goal description*, consisting of a list of objectives and a set of quasi-invariants for this adaptation space;

- *Reference context* description, describing the common resources that all adaptation cases may share, and the nominal functionality and behavior of the program component that this adaptation space deals with. This description may distinguish "required aspects" from "desired aspects". A reference context can be an initial adaptation case which describes the original, unspecialized program component. The main components of the reference context include:

  - the initial case (also called the root case).
  - the set of common resource parameters that all adaptation cases may share.
  - the set of local parameters used in the goal description and use condition specification.

- *Use condition*, specifying the general condition to turn on this adaptation space.

Note that the local parameters and common resource parameters, once defined or declared in the adaptation space, can be used by all the adaptation cases within the given adaptation space. They can be referenced as global variables in individual adaptation cases without declaration.

Quite often, the use condition of an adaptation space (logically) implies the use condition of some adaptation case within the given space. It is also possible that the use condition used to turn on an adaptation space may lead to more than one adaptation cases eligible to be activated. In this case a selection policy is required to make a choice [36]. There is only one case active at any given point in time for any adaptation space.

Example 5 Recall the examples shown in Figure 9, Figure 10, Figure 11, and Figure 12. The initial adaptation case ac_initialP and the specialized adaptation cases for coping with network delays together form an adaptation space named as_spaceQ as shown in Figure 13.

ADAPTATION SPACE as_spaceQ **adapts the execution of query** Q.
goal: g_1: fast response time with survivability in the presence of network delays.
reference context
initial-case: ac_initialP;
resource-object: *Nucleotides*
parameters
ID, protID, desc, nuc_other_attrs; **instance:** *NCBI*
resource-object: *Proteins*
parameters
ID, desc, prot_other_attrs; **instance:** *SWISS-PROT*
resource-object: *Prot_Struct*
parameters
prot_ID, compound_ID, desc, ps_other_attrs; **instance:** *PDB*
resource-object: *NetworkLinks*
parameters
bandwidth; **instances:** L_1, L_2, L_3
adaptation condition
$\exists x \in NetworkLinks, Rate(x) < 1Mbps$
end as_spaceQ

Figure 13: Example of an adaptation space description for the execution of query Q.

The reference context of this space includes the resource objects and their descriptive attributes that are involved in Q, in addition to the initial adaptation case ac_initiaP. This adaptation space is equipped with a single goal: to cope with network delays with the price of increasing memory contention once that concurrent caching processes are fired.

Given a program component, there may be more than one adaptation space, each holding a different set of goals and a specific context for adaptation. Different adaptation spaces of the same program present different adaptation dimensions. For example, I may create and maintain an adaptation space for a given query plan (reference program) with performance optimization as my goal, by specializing the query plan to cope with the resource degradations. At the same time, I may also want to build an adaptation space that promotes query relaxation, which generalizes the query to require fewer (or different) data sources. The latter space could be used when some of the remote sources are completely unreachable.

3.4 Adaptation Space Model: Formal Semantics

In this section I formally describe the concept of adaptation case, adaptation space, and related notions (such as adaptation-of relationship) using graph-based notation, and illustrate these concepts by examples. The formal definitions introduced in this section will be served

as the basis for defining adaptive coordination mechanisms that guarantee the correct use of adaptation cases and adaptation spaces.

3.4.1 Basic Definitions

Given a program component T in a target web service, I can define the adaptation behavior of T through introduction of multiple adaptation cases. Each adaptation case is applicable when its use-conditions become true.

A use-condition is a Boolean expression involving comparisons between functions, and between functions and constants. Comparison operators include both the usual arithmetic operators $(==, =, \neq, <, \leq, >, \geq)$ and the set comparison operators $(==, =, \neq, \subset, \subseteq)$ and membership operators $(\in, \notin)$.

Definition 1 (Use Condition) Let T denote a program component in a target web service. Let $Param(T)$ denote the list of adaptation context parameters of T, and $dom(P_i)$ denote the domain of possible values of an adaptation parameter P_i ($P_i \in Param(T)$). A use-condition, denoted by F, can be defined recursively as follows:

(a) $F = \emptyset$ is a use-condition (empty condition);

(b) $F = P_i \Theta P_k$, $P_i \Theta v_{ij}$, or $v_{ij} \Theta P_i$ are use-conditions, where

  (i) P_i and P_k are comparable functions returning atomic values, $P_i, P_k \in Param(T)$ $v_{ij} \in dom(P_i)$, and $\Theta \in \{==, =, \neq, <, \leq, >, \geq\}$;

  (ii) P_i and P_k are comparable functions returning set values, $v_{ij} \subseteq dom(P_i)$, and $\Theta \in \{==, =, \neq, \subset, \subseteq\}$;

(c) If F, F_1, and F_2 are use-conditions, then the conjunction $F_1 \wedge F_2$, the disjunction $F_1 \vee F_2$, and the negation $\neg F$ are use-conditions;

(d) If F is a use-condition, and x is a free variable in F, then $(\exists x)F(x)$ and $(\forall x)F(x)$ are use-conditions;

(e) Nothing else is a use-condition. □

The use-conditions in the form (i) and (ii) are called atomic Boolean expressions (atoms). All variables referenced in use-condition are strongly typed.

Definition 2 (Adaptation Case) Let T denote a program component in a target web service. An adaptation case of T is described by a tuple $(N, TARG, GOAL, CTX, UC, AACTION)$ where

- N is the name of the adaptation case.
- $TARG$ specifies the identifier of the target adaptation unit. It can be a regular class, a module, a program, a procedure, or an existing adaptation case.
- $GOAL$ is a character string, presenting the goal description of the adaptation case.
- CTX is the context description of the case, defined by three components: a pointer to the adapted code, the *list of resource objects* used in the reference program, and the *list of resource object parameters*, where each parameter is further described by a quadruple $(Pr, State, Type, Rdom)$, where
  - Pr is a name for the resource parameter;
  - $State(Pr)$ specifies one of the following use states of the resource parameter Pr: *provide*, *must-provide*, *require*, *must-require*.
  - $Type(Pr)$ defines the type of the resource parameter Pr and optionally the domain constraint of the parameter Pr.
  - $Rdom(Pr)$ defines the set of permissible values of the resource parameter Pr.
- UC is a set of use-condition, expressed by a Boolean expression, indicating when it is appropriate to use this adaptation case. Variables in use conditions are defined in the adaptation case context.
- $AACTION$ is the adaptation action to be fired when UC is satisfied. □

Example 6 Recall adaptation case ac_netDelay2 (Figure 11). According to Definition 2, I can describe this adaptation case as follows:

N(ac_netDelayL2) = ac_netDelayL2.

$TARG$(ac_netDelayL2) = ac_initialP.

$GOAL$(ac_netDelayL2) = "query adaptation when Rate(L_2) degrades".

CTX(ac_netDelayL2) = { (P_{12}), (*NetworkLinks*, {(*bandwidth*, must-provide)}),
(*Proteins*, { (*ID, desc*, must-provide)}), (*Nucleotides*, {(*ID, desc*, must-provide)})
(*Prot_Struct*, { (*compoundID, desc*, must-provide)})}

UC(ac_netDelayL2) = {($Rate(L_2) < 1Mbps$)$\wedge$ ($Rate(L_1) \geq 1Mbps$) $\wedge$ ($Rate(L_3) \geq 1Mbps$)$\wedge$
available(*Nucleotides*) = `true` $\wedge$ available(*Proteins*) = `true` $\wedge$ available(*Prot_Struct*) = `true` }

$AACTION$(ac_netDelayL2) = "collect and cache data from *Prot_Struct* while slowly receiving data from *Protein*."

Definition 3 (Adaptation-of Relationship) Let T denote a program component in a target web service. Let α and β denote two adaptation cases of T, where each case is described by a tuple $(N, TARG, GOAL, CTX, UC, AACTION)$. The case β is said to have an *adaptation-of* relationship with the case α, denoted by $\alpha \Longrightarrow \beta$, if and only if the following conditions are verified:

1. $TARG(\beta)$ = case α.
2. $\forall x \in CTX(\beta) \exists y \in CTX(\alpha)$ s.t. $identical(x, y) \vee (specialized\_version\_of(x, y) \wedge N(x) = N(y))$.
3. $\forall p \in UC(\alpha) \exists q \in UC(\beta)$ s.t. $degradation\_of(q, p)$.
4. $\forall p \in UC(\alpha) \forall q \in UC(\beta)$, $(q = p) \vee degradation\_of(q, p)$.
5. $AACTION(\beta) = AACTION(\alpha) + new\_action$

The case α is said to be *adaptable* by case β. The relationship between α and β is referred to as *adaptation-of* relationship. I call case α *supercase* and case β *subcase*. □

This definition states that new adaptation cases can be defined incrementally in terms of existing adaptation cases, rather than being defined from scratch. However, the intuition

between supercase and subcase is quite different from the superclass and subclass relationship in object-oriented systems. For example, a subcase in the adaptation model may have more restricted use conditions but may consume fewer resources than its supercase. More concretely,

- Condition (1) specifies that each subcase takes its immediate supercase as the target unit of adaptation.
- Condition (2) amounts to saying two points: (i) not every resource of a supercase will be used in its subcases, and (ii) for any resource of a subcase, its implementation may be more specialized than the corresponding one in its supercase (e.g., an alternative remote data source, which has better response time, can be accessed instead).
- Condition (3) states that there must exist at least one condition in $UC(\beta)$ that represents a *degradation of* some resource parameter used by T.
- Condition (4) asserts that the state of all resource parameters listed in case α are either degraded or remain unchanged in case β.
- Condition (5) specifies that the adaptation action of a subcase consists of all the actions considered by its supercase plus a new one. The intuition is that besides all the actions that were taken to cope with the runtime changes up to this moment, an additional adaptation action is needed to consider the new degradation of the runtime environment configuration described in the subcase's use condition.

To illustrate the above definition more intuitively, let us look at the following example:

Example 7 Recall the adaptation cases depicted in Figure 9, Figure 10, Figure 11, and Figure 12. By Definition 3, I have:

$$ac\_initialP \Longrightarrow ac\_netDelayL1,$$
$$ac\_initialP \Longrightarrow ac\_netDelayL2,$$
$$ac\_netDelayL1 \Longrightarrow ac\_netDelayL12,$$
$$ac\_netDelayL2 \Longrightarrow ac\_netDelayL12.$$

In general, given an adaptation case, each of its immediate subcases describes an adaptation alternative. The subcases that are siblings with one another may have use-conditions that are either mutually exclusive or overlapping. Thus, it is possible that more than one sibling cases are eligible for adaptation at the same time. Selection policies [36] are therefore required to reconcile the situation and guarantee that there is only one adaptation case being activated at any specific point in time. I call such case the active case of the adaptation space.

Definition 4 (Adaptation Space) Let T denote a program component in a target web service. An adaptation space of T is defined as a labeled directed graph $G = (V, L, E)$ where V is a finite set of adaptation vertices of G; L is an ordered set of labels, each described by a set of use-conditions; E is a ternary relation on $V \times L \times V$, representing adaptation edges. For any α, $\beta \in V$ and $\ell \in L$, if $\alpha \Longrightarrow \beta$, then $(\alpha, \ell, \beta) \in E$. □

Given an adaptation space graph $G = (V, L, E)$, each vertex in V represents an adaptation case and each label in $(u, \ell, v) \in V \times L \times V$ represents a use condition describing when to switch off case u and turn on case v.

The reference case of the adaptation space of G is a special vertex in V, which cannot be adaptation reachable by any other vertices of V and there exists no other vertices from which this special case can be adaptation reachable.

Example 8 Recall the four examples of adaptation cases for Q given in Figure 9, Figure 10, Figure 11, and Figure 12. The root adaptation case (labeled as $ac\_initialP$) is the initial optimized query plan P_0 (unspecialized reference program), and the other nodes represent alternative query plans (adapted code) when network delays occur. I can describe this adaptation space for query Q as a graph $G = (V, L, E)$, where

$$V = \{ac\_initialP, ac\_netDelayL1, ac\_netDelayL2, ac\_netDelayL12\}$$
$$L = \{(Rate(L_1) < 1Mbps), (Rate(L_2) < 1Mbps)\}$$
$$E = \{(ac\_initialP, Rate(L_1) < 1Mbps, ac\_netDelayL1),$$
$$(ac\_initialP, Rate(L_2) < 1Mbps, ac\_netDelayL2),$$

$(ac\_netDelayL1, Rate(L_2) < 1Mbps, ac\_netDelayL12)$,

$(ac\_netDelayL2, Rate(L_1) < 1Mbps, ac\_netDelayL12)\}$

The initial case ac_initialP can be adapted (i) by ac_netDelayL1 when $Rate(L_1)$ degrades, (ii) by ac_netDelayL2 when $Rate(L_2)$ degrades, and (iii) by ac_netDelayL12 when both $Rate(L_1)$ and $Rate(L_2)$ degrade. If $Rate(L_1) < 1Mbps$, the adaptation case ac_netDelayL1 will be turned on to adapt P_0 to cope with the unexpected delay at network link L_1.

The four adaptation cases are related with one another through the adaptation-of relationships, and together they form an adaptation space for Q, as shown in Figure 14.

Interesting to note that not all transitions in the adaptation space are relevant. For instance, the transition from $ac\_netDelayL2$ to $ac\_netDelayL12$ is not relevant because by the time connection L_2 is detected to be slow, I may have already downloaded the data from A if I assume a hash-based join. In Figure 14, the irrelevant transition is represented by a dotted line.

3.4.2 Adaptation Reachability

Adaptation reachability refers to the capability of adapting a program to different kinds of adaptation at different levels of a web service. More concretely, for any two adaptation cases in a given adaptation space, say α and β, I say that the case β is adaptation reachable if and only if β has a direct or indirect adaptation-of relationship with α.

Definition 5 (Adaptation Reachable $\Longrightarrow^*$) Let T denote a program component in a target web service and $G = (V, L, E)$ be an adaptation space graph of T. A vertex $\gamma \in V$ is *adaptation reachable* from vertex α ($\alpha \in V$), denoted by $\alpha \Longrightarrow^* \gamma$, if and only if one of the following conditions is satisfied:

(i) $\alpha = \gamma$.

(ii) $(\alpha, \ell, \gamma) \in E$.

(iii) $\exists \beta \in V, \beta \neq \gamma$ and $\beta \neq \alpha$, s.t. $\alpha \Longrightarrow^* \beta$ and $\beta \Longrightarrow^* \gamma$. □

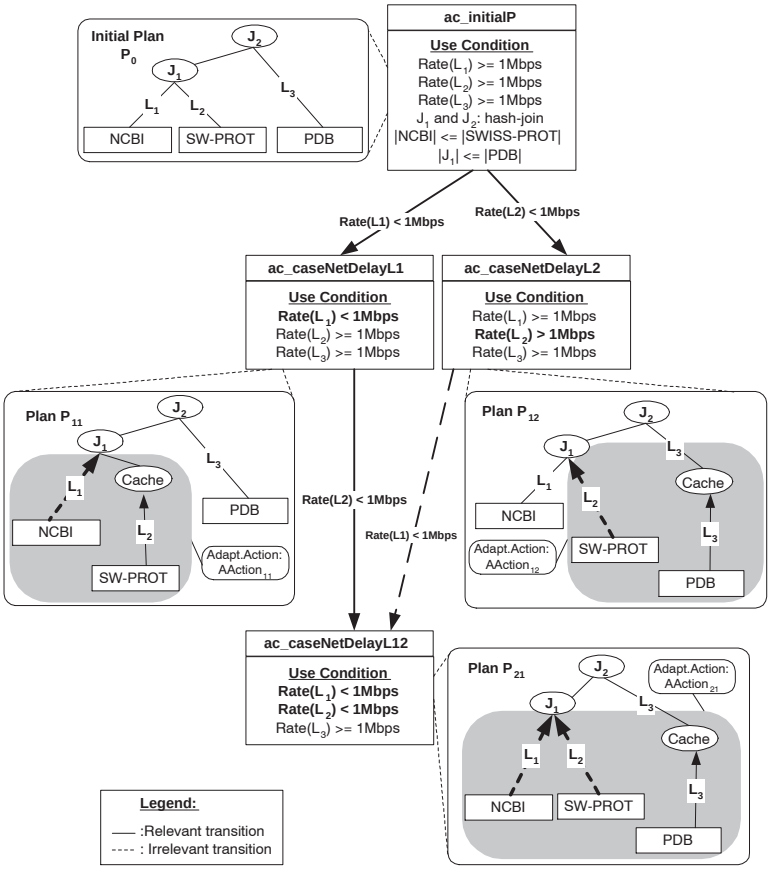


Figure 14: Example of an Adaptation Space.

This definition formally introduces the concept of adaptation reachability: for any two adaptation cases α and γ, if γ is adaptation reachable from case α, then either α and γ are identical case, or there is an adaptation-of relationship between γ and α, i.e., $\alpha \Longrightarrow \gamma$ holds, or there exists another adaptation case β satisfying that β is adaptation reachable from α, and γ is adaptation reachable from β.

This adaptation reachability concept is powerful in the sense that it provides a uniformed adaptation framework which is independent of the objectives, context, and methods of adaptation spaces. For instance, when applied to the kind of adaptations designed for capability relaxation scenario, the adaptation reachability concept means that a case with more capabilities can be adapted to a case with less or weaker capability. When applied to adaptations for performance gains (program specialization), the adaptation reachability means that a case with generic assumptions (such as the transfer rate of all network links

are at least $1Mbps$) can be adapted to a case with specific assumptions (such as some of the network links have transfer rate less than $1Mbps$, and this condition will hold over a certain period of time).

Example 9 Let us take a look at Figure 14. According to Definition 5(ii), the vertex of ac_net-DelayL12 is adaptation reachable from vertex ac_initialP, ac_netDelayL1, and ac_netDelayL2. That is:

- $ac\_initialP \Longrightarrow^* ac\_netDelayL1$
- $ac\_initialP \Longrightarrow^* ac\_netDelayL2$
- $ac\_initialP \Longrightarrow^* ac\_netDelayL12$
- $ac\_netDelayL1 \Longrightarrow^* ac\_netDelayL12$
- $ac\_netDelayL2 \Longrightarrow^* ac\_netDelayL12$

The following proposition proves that "$\Longrightarrow^*$" is the transitive closure of E, since it is reflexive, antisymmetric and transitive. No cyclic adaptation path is allowed. An adaptation space graph is considered **legal** if there are no cyclic adaptation paths:

$$\{\alpha \in V | \exists \beta \in V, \text{ s.t. } \beta \neq \alpha \text{ and } \alpha \Longrightarrow^* \beta \Longrightarrow^* \alpha\} = \emptyset$$

This condition is served as one of the guidelines to check the consistency of adaptation case specifications. In the sequel, we assume that adaptation space graphs are legal.

*** Proposition: Partial order for the adpatation space.

Proposition 1 Let T denote a program component in a target web service and $G = (V, L, E)$ be an adaptation space graph of T. The adaptation reachable relation $\Longrightarrow^*$ is a *partial order* over $V \times V$. □

Proof:

Case (a) By Definition 5(i), $\forall \alpha \in V$, $\alpha \Longrightarrow^* \alpha$ holds. Thus $\Longrightarrow^*$ is reflexive.

Case (b) To prove $\Longrightarrow^*$ is antisymmetric, we need to prove that, $\forall \alpha, \beta \in V$, if $\alpha \Longrightarrow^* \beta$, then

$\beta \not\Longrightarrow^* \alpha$. By $\alpha \Longrightarrow^* \beta$ and the antisymmetric property of the subset inclusion operator $\subseteq$ and logical implication operator $\rightarrow$, $\beta \not\Longrightarrow^* \alpha$ holds.

Case (c) Similarly, we can prove that $\Longrightarrow^*$ is transitive, i.e., $\forall \alpha, \beta, \gamma \in V$, if $\alpha \Longrightarrow^* \beta$ and $\beta \Longrightarrow^* \gamma$, then $\alpha \Longrightarrow^* \gamma$. □

This proposition ensures that the incremental adaptation specification yields a partial order of adaptation cases within any given adaptation space.

3.5 Advanced Properties of Adaptation Spaces

3.5.1 Basic Relationship of Adaptation Spaces

Given two adaptation spaces S_1 and S_2, if S_1 and S_2 share the same initial case (root adaptation case), then an immediate question one would ask is "are these two adaptation spaces comparable?" If S_1 and S_2 are comparable, are they equivalent? Does one cover, overlap with, or disjoint with, another? In this section, I formally define the notion of comparable adaptation spaces and the concept of equivalence, coverage (inclusion), overlap and disjoint of two adaptation spaces.

Definition 6 (Equivalent Adaptation Cases) Two adaptation cases α and β are considered to be *equivalent*, denoted by $\alpha \simeq \beta$, if and only if the following conditions are satisfied:

(i) $TARG(\alpha) = TARG(\beta)$.

(ii) $CTX(\alpha) = CTX(\beta)$.

(iii) $UC(\alpha) = UC(\beta)$.

(iv) $AACTION(\alpha) = AACTION(\beta)$. □

This definition says that two adaptation cases can have different case names and goal description syntax, but still be functionally equivalent.

Definition 7 (Comparable Adaptation Spaces) Let S_1 and S_2 denote two adaptation spaces. Let $initial(S)$ denote the initial case of adaptation space S, $SCTX(S)$ denote the

set of parameters referenced in the reference context of S, and $SUC(S)$ denote the union of the set of use-conditions of S and of its adaptation cases. Two adaptation spaces are considered *comparable* if and only if the following conditions are satisfied:

(i) $initial(S_1) \simeq initial(S_2)$.

(ii) $SCTX(S_1) = SCTX(S_2)$. □

Note that two adaptation spaces S_1 and S_2 are comparable even if they have completely different space names, and different sets of adaptation-conditions. Let us look at an example:

Example 10 Recall the query plan P_0 shown in Figure 8(b). Assume the root of the adaptation space for query Q is the initial case ac_initialP (Figure 10). Besides adapting to changes in transfer rates, one could also adapt the execution of query Q to changes in memory constraints (see Chapter 5). Suppose that at runtime the data to be delivered by NCBI is larger than expected. Consequently, the execution of join J_1 in plan P_0 will require more memory blocks than originally assumed by the optimizer. If there are more memory available in the system, the query engine can request the extra memory blocks and the performance of P_0 is not affected. However, if extra memory is not available, then the execution of J_1 will thrash given the increase of I/Os due to extra paging. Assuming that J_1 is a hash-join, one way to address this problem is by switching the operands of J_1 if I can detect that the data to be delivered by *NCBI* is larger than the data to be delivered by *SWISS-PROT*. Similar analysis applies to J_2 if $|J_1| > |PDB|$.

The adaptation space graph for this example is described as follows:

V = {ac_initialP_MC, ac_switchOprnd_1, ac_switchOprnd_2}

L = {$|NCBI| > |SWISS\text{-}PROT|$, $|J_1| > |PDB|$}

E = {(ac_initialP_MC, $|NCBI| > |SWISS\text{-}PROT|$, ac_switchOprnd_1),
(ac_initialP_MC, $|J_1| > |PDB|$, ac_switchOprnd_2)}

Let QS_1 the adaptation space given in Figure 14 and QS_2 the adaptation space in

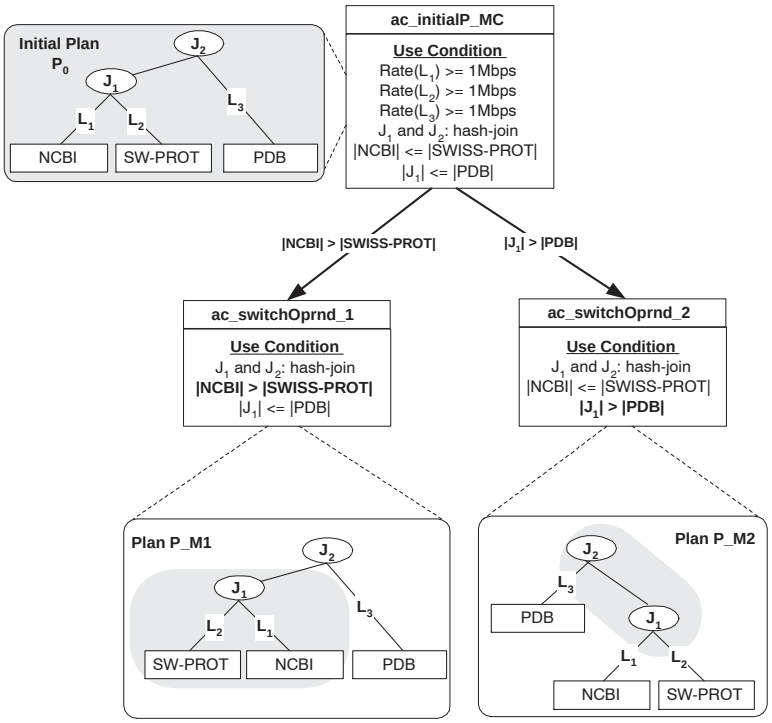


Figure 15: Example of an Adaptation Space for coping with end-to-end delays.

Figure 15. Comparing the initial case of QS_1, namely ac_initialP, with the initial case of QS_2, namely ac_initialMemP, I observe the following facts:

(1) ac_initialP and ac_initialP_MC are equivalent (by Definition 6)

(2) $SCTX(QS_1) = SCTX(QS_2) = \{Nucleotides, Proteins, Prot\_Struct, memorySize, NetworkLinks\}$

(3) $SUC(QS_1) = \{Rate(L_1) < 1Mbps, Rate(L_2) < 1Mbps\}$

$SUC(QS_2) = \{|NCBI| > |SWISS\text{-}PROT|, |J_1| > |PDB|\}$

Thus, $SUC(QS_1) \neq SUC(QS_2)$

I can conclude that QS_1 and QS_2 are comparable adaptation spaces, but they are not equivalent.

Definition 8 (Adaptation Scope) Let α denote any adaptation case in a given adaptation space $G = (V, L, E)$. An *adaptation scope* of α, denoted as $Ascope(\alpha)$ is the set of adaptation cases that are adaptation-reachable from α.

$$Ascope(\alpha) = \{\gamma | \alpha, \gamma \in V, \alpha \Longrightarrow^* \gamma\}.$$

□

Example 11 Recall Example 8 and Figure 14. I have

```
Ascope(ac_initialP) = { ac_initialP, ac_netDelayL1, ac_netDelayL2,
                        ac_netDelayL12}
Ascope(ac_netDelayL1) = {ac_netDelayL1, ac_netDelayL12}
Ascope(ac_netDelayL2) = {ac_netDelayL2, ac_netDelayL12}
Ascope(ac_netDelayL12) = {ac_netDelayL12}
```

Given an adaptation space case α, the adaptation scope of α identifies the adaptation context of α. It helps the system to keep track of the adaptation behavior of α, and facilitates the monitoring of use-conditions and quasi-invariants of the goals involved in the adaptation scope of α (See Section 3.6 for further detail).

Definition 9 (Equivalence Relationship) Let S_1 and S_2 denote two comparable adaptation spaces. Let $SUC(S_1)$ denote the set of use-conditions of S_1 and its adaptation cases. I say that S_1 is *equivalent to* S_2, denoted by $S_1 \simeq S_2$, if and only if the following conditions are satisfied:

(i) $SUC(S_1) = SUC(S_2)$

(ii) $\forall \alpha \in S_1$, $\exists \beta \in S_2$ such that $\alpha \simeq \beta$ (i.e., α and β are equivalent cases), and vice-versa.

□

Definition 10 (Inclusion Relationship) Let S_1 and S_2 denote two comparable adaptation spaces. Let $SUC(S_1)$ denote the set of use-conditions of S_1 and its adaptation cases. I say that S_1 *covers* (or includes) S_2, denoted by $S_1 \sqsupset S_2$, if and only if the following conditions are satisfied:

(i) $SUC(S_1) \supseteq SUC(S_2)$.

(ii) For any adaptation case $\alpha \in S_2$, there is an adaptation case $\beta \in S_1$, such that $\alpha \simeq \beta$.

□

Definition 11 (Overlap Relationship) Let S_1 and S_2 denote two comparable adaptation spaces. Let $SUC(S_1)$ denote the set of use-conditions of S_1 and its adaptation cases. I say that S_1 *overlaps* with S_2, if and only if the following conditions are satisfied:

(i) $SUC(S_1) \cap SUC(S_2) \neq \emptyset$.

(ii) $\exists \alpha \in S_1$ and $\exists \beta \in S_2$ such that $\alpha \simeq \beta$. □

Recall Example 10. The two adaptation spaces, QS_1 and QS_2, do not overlap because $SUC(QS_1) \cap SUC(QS_2) = \emptyset$.

Definition 12 (Disjoint Relationship) Let S_1 and S_2 denote two comparable adaptation spaces. Let $SUC(S_1)$ denote the set of use-conditions of S_1 and its adaptation cases. I say that S_1 *disjoints* with S_2, if and only if the following condition is satisfied: $SUC(S_1) \cap SUC(S_2) = \emptyset$. □

3.5.2 Composition of Adaptation Spaces

It is interesting to study the composition of adaptation spaces, especially the composition of comparable adaptation spaces.

The more sophisticated composition of different adaptation spaces requires more thought and careful analysis of the benefits and usage of the adaptation space. I discuss composition of adaptation spaces in Chapter 7.

3.5.3 Other Complications

One complication is the question of what kind of adaptation derivation I would like to have. For instance, an adaptation case may also be derived from more than one existing adaptation cases. Do I need such kind of complication? What are the benefits and applications I can obtain? Studying these issues is part of my future research agenda.

3.6 Issues of Using Adaptation Space Model

I have formally described the adaptation space model and the inherent properties. In this section, I summarize my discussion by classifying the concepts introduced into the following three categories: (1) what to adapt, (2) how to adapt, and (3) when to adapt.

What to adapt The first task I need to deal with in adaptation specification is how to identify the program components of a web service that should adapted. I have addressed this issue by identifying what should be included in an adaptation context and what are the target units that I consider in my adaptation model.

- **Adaptation context:** The key elements of the adaptation context include the goal of adapting a given component, the reference case (e.g., the source code of a component to be adapted), the set of use conditions, and the resources required. In this paper, I have concentrated my discussion on performance gain-driven adaptation, namely specialization. Another representative type of query adaptation is relaxation-based adaptation, where a more generic SQL statement is generated, and the response returned to the user is a super-set of the result expected from the initially submitted query.

- **Adaptation target units:** In general, the adaptation target units may range from a software system (a set of programs), a complete program (a set of modules), a module (a class, a set of functions), a function (procedure), to a block of code (within a function). The choice of a particular entity depends on the language structure, the adaptation context, and the power of the given program adapter.

When to adapt I have defined the use condition and the incremental approach for adaptation case specification in Section 3.3 and Section 3.4. The key points are summarized as follows:

- **Use conditions:** A use condition is a set of predicates over some parts of the program state or adaptation space state, which hold over a period of time. It identifies when

the corresponding adaptation case should be activated. In contrast to program specializers which usually consider equality predicates, and perform specialization when some specific variables having some specific values, program adaptation allows the use condition to be predicates consisting of inequalities, and possibly involving external events.

- **Incremental adaptation:** Usually, a set of use conditions may not become true all at once. Some use conditions may even be in conflict with each other. Therefore is useful to be able to describe a partial order of adaptation contexts for a given query execution so that it can be adapted further as more use conditions become satisfied. In the proposed adaptation model, the notion of adaptation space is introduced to capture the partial order of adaptation cases (contexts) for each given program component.

How to Adapt The question of how a program component should be adapted involves the operational aspects of adaptation, which include issues such as how to monitor the use conditions, when to produce the adaptation code, how to apply (trigger) an adaptation, and how to integrate the adaptation code into the program.

- **Compile-time vs. runtime adaptation:** In general, given an adaptation case, if its use condition can be evaluated at compilation stage, then I refer to the use condition as compile-time use condition and the adaptation case as compile-time adaptation. Otherwise, it is called runtime adaptation and its use condition is called runtime use condition. Note that for those adaptation contexts depending only on compile-time use conditions, adaptation can be applied either at compile-time or at runtime. For the other contexts, adaptation must be applied at runtime. In this chapter, I only considered runtime use conditions.

- **Monitoring use condition:** In the runtime adaptation case, I need some runtime support which detects when all the predicates in the use condition become valid, and when the adaptation is triggered to replace the target component. There are a number of ways to conduct situation monitoring. For example, I can use periodic monitoring

with regular or irregular time interval [34]. This approach allows us to check the use condition periodically with reasonable frequency and trigger the adaptation whenever the use condition is evaluated to be true. Another alternative approach for monitoring use conditions is to use the content-based update monitoring [34], which checks the use condition whenever there is a new update to some context variables involved in the use condition.

- **Triggering adaptation:** Once the use condition becomes true, an immediate question to ask is when to apply adaptation. Both eager approach and lazy approach can be used here. When the use condition is evaluated to be true, the eager approach applies adaptation and replaces the component immediately, whereas the lazy approach only replaces the component at the time when the component is used.

- **Preserving adaptation validity:** For runtime adaptation case, the runtime support must preserve the validity of the adaptation. This implies two basic tasks: (1) detecting when the predicates in the use condition are invalidated, and (2) replacing the adaptation component by an appropriate adaptation case whose use condition is satisfied. A sophistication here is the situation where there may exist more than one adaptation cases whose use conditions are true during the given period of time, and thus the runtime support has to decide which one to pick. The decision is particularly difficult when the candidate adaptation cases have potentially conflicting goals and interests [36].

3.7 Summary of Adaptation Space Model

In this chapter, I have described the notation and the formal semantics of the adaptation space model used for designing adaptive web services. The core of this adaptation model is the notion of adaptation space and the mechanisms for coordinating and integrating different kinds of adaptations. An adaptation space is defined by a use context and a partial order of adaptation cases. Each adaptation case describes a specific adaptation of a program or component of a web service. There are three main thrusts of the adaptation

space approach. First, it defines a multi-dimensional adaptation context for capturing and coordinating different kinds of adaptation at different levels of a web service. Second, it provides a uniformed way for representing and viewing a collection of alternative adaptations for a given query program or component of a web service. Third, it promotes a declarative and incremental approach to adaptation specification, allowing the incorporation of new adaptation behavior of a web service in terms of existing adaptation cases.

In the next chapter, I describe the Ginga approach, which has in its core the adaptation space model describe in this chapter. For the remaining of the dissertation, I use Ginga to investigate the trade-offs and benefits of using proposed the adaptation space model.

CHAPTER IV

GINGA: AN ADAPTIVE APPROACH TO QUERY PROCESSING

4.1 Overview

In contrast to the closed world assumption taken by most of conventional database applications, advanced Internet applications use an open world assumption. The runtime environment may change dynamically. Data sources available on-line are constantly changing in number, volume, content, and capability. The assigned resources for executing a query constantly change in their configuration and their availability. There are several mechanisms that help users to query these massive data. However, these mechanisms are either limited to local search indexes (e.g., search engines) or do not have means to adapt to the sudden changes of the runtime environment.

In this chapter, I describe *Ginga* [41, 43], an adaptive approach to query processing based on feedback control. My approach aims at providing continuously efficient query execution, taking into account the important changes to the runtime environment. I am particularly interested in distributed queries that are long running or need to be executed more than once (e.g., continual query [33]).

Ginga has three interesting features. First, it utilizes the Adaptation Space concept to manage the predefined generation of query plans. These parameterized plans will service as alternatives to react to unexpected shortages of runtime resources. Second, it provides feedback-based control mechanism that allows the query engine to switch to alternative plans upon detecting runtime environment variations. Third, it describes a systematic approach to integrate the adaptation space with feedback control that allows me to combine predefined and reactive adaptive query processing, including policies and mechanisms for determining when to adapt, what to adapt, and how to adapt.

4.2 System Architecture

The Ginga query adaptation engine is a distributed software system that supports adaptive query processing. When executing a query in a highly unstable runtime environment, the query processor needs to be highly adaptive in order to cope with unpredictable runtime situations (e.g., unexpected end-to-end delays, latency fluctuations, memory shortage). *Ginga* is a Brazilian word typically used to describe a quality that a person needs to have when dancing *samba*. Like the quick rhythm and movements of samba, the query processing system equipped with Ginga query adaptation engine can quickly and efficiently change the execution of a query plan in order to keep up with the rhythm imposed by the runtime variations in the environment.

Figure 16 shows a sketch of the Ginga system architecture. A query submitted to the Ginga query system is initially processed by the *query manager*. The major tasks of the query manager are to coordinate clients' query sessions and to invoke query routing process [30], which performs source selection by pruning those data sources that cannot directly contribute to the answer of the query. After query routing, each end-user query is transformed into a set of subqueries associated with some execution dependencies. Each of such subqueries is targeted to one of the chosen data sources.

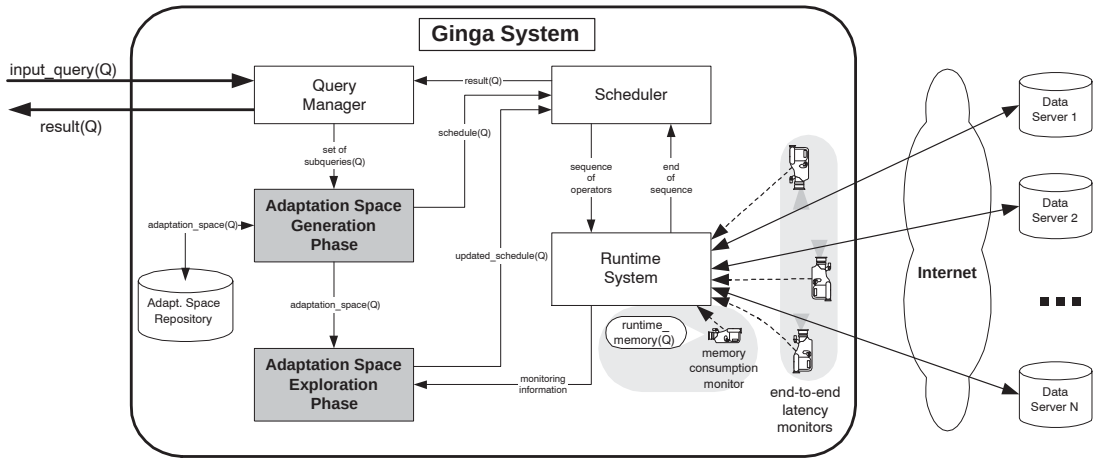


Figure 16: Ginga System Architecture.

Ginga takes the resulting queries from the query routing process, and builds an initial

optimized plan and some alternative execution plans that may be used when the runtime environment state changes significantly. The query adaptation in Ginga combines an Adaptation Space *Generation* phase (AdaptS-GP), before query execution, with an Adaptation Space *Exploration* phase (AdaptS-EP) during the execution. At AdaptS-GP, an adaptation space is built with the initial optimized plan as the root and the alternative plans as the non-root nodes. During query execution, Ginga monitors the runtime environment and resource availability continuously, and determines *when* to change the query plan and *how* to adapt by choosing an alternative plan from the corresponding adaptation space in the AdaptS-EP.

4.3 Adaptation Space Generation Phase

In the Adaptation Space Generation phase (AdaptS-GP), Ginga builds the initial query plan P_0 and establishes a selection of alternative plans for adaptation. The first task is to generate the initial optimized plan. This can be done using any existing query optimization algorithm for distributed databases [55, 19, 7, 32, 51, 40]. The second task is to generate query adaptation alternatives ($\{P_i, i = 1, \ldots\ n\}$). Since there are many potential alternative plans, Ginga organizes them into an adaptation space.

4.3.1 Ginga's Adaptation Space

In Chapter 3, I discussed and formalized the notion of adaptation space for the general situation of program adaptation. In this section, I specialize the adaptation space for the Ginga approach.

In Ginga, I organize the opportunities for adaptation and alternative query plans into an adaptation space. An adaptation space is a directed graph where the nodes are called *adaptation cases* and the arcs are called *adaptation triggers*. An adaptation_case(P_i) is a pair (P_i, adaptation_condition(P_i)), where P_i is a query plan and adaptation_condition(P_i) is the set of predicates over runtime parameters under which P_i was optimized (e.g., how many memory blocks have been allocated and the assumptions made about join operand sizes and network transfer rates).

An adaptation_trigger(P_i) is a quadruple (adaptation_event(P_i), adaptation_action(P_i, P_j),

adaptation_condition(P_j), *wait_time*) where P_i is the current plan executing under adaptation_condition(P_i). When a change in system parameters invalidates a predicate in adaptation_condition(P_i) (e.g., unexpected latency fluctuation while receiving data from remote servers), adaptation_event(P_i) is fired. Then, adaptation_action(P_i, P_j) finds adaptation_case(P_j) such that adaptation_condition(P_j) has become valid by the adaptation_event(P_i). Since P_j is the plan optimized for the new set of predicates over runtime parameters, the adaptation_action(P_i, P_j) switches from P_i to P_j by replacing code or runtime flags in all the nodes involved in processing this query.

The *wait_time* component of an adaptation trigger indicates for how long the invalidation of the predicate in adaptation_condition(P_i) must hold before the described transition takes place. This *wait_time* is used to prevent oscillations in feedback-based adaptation when, for example, temporary network latency and bandwidth fluctuations (i.e., end-to-end delays) cause repeated transitions back and forth between two query plans. As in all feedback-based adaptation, a short *wait_time* results in fast adaptation soon after the onset of end-to-end delay. A long *wait_time* slows down the adaptation process, but it decreases the probability of an oscillation. A precise setup for *wait_time* is a hard problem yet to be solved as it implies being able to predict the future state of a runtime environment.

In the experimental studies reported in this dissertation, I take a simple assumption that once the adaptation process has taken place, Ginga does not backtrack when the invalidated predicates are validated again. Incorporating various types of oscillations in a runtime environment into the query adaptation decision-making process is an important and interesting subject, which I defer to future research work.

During the AdaptS-GP, Ginga builds an adaptation space for executing query Q through three main steps. First, Ginga generates the initial query plan P_0 under the initial assumptions made about the runtime environment (e.g., expected memory size) in adaptation_condition(P_0). Second, Ginga creates the set of monitors that trigger an adaptation_event when one of the predicates in adaptation_condition(P_0) becomes invalid. Third, for each such event, Ginga creates a new set of predicates over runtime parameters and an alternative query plan optimized for it (adaptation_condition(P_j)) and adaptation_trigger(P_0).

This process is repeated for each one of P_j until all the runtime parameter predicates have been covered.

The construction of an adaptation space can be eager, as outlined above, or lazy, when the alternative query plans are generated only when needed. In general, the trade-offs in adaptation space generation are a subject for future research and beyond the scope of this dissertation (see Section 4.5). For a relatively small adaptation space, it is reasonable to generate it at initialization time. This is the assumption made in this dissertation.

In this dissertation, I do not describe, implement, or evaluate a detailed algorithm for generating the adaptation space. Developing and evaluating of such algorithm is part of my future research work. Thus, when I study Ginga's performance characteristics in Chapter 5, Chapter 6, and Chapter 7, I assume that the associated adaptation space has been previously generated.

I now illustrate the adaptation space construction process using a simple example.

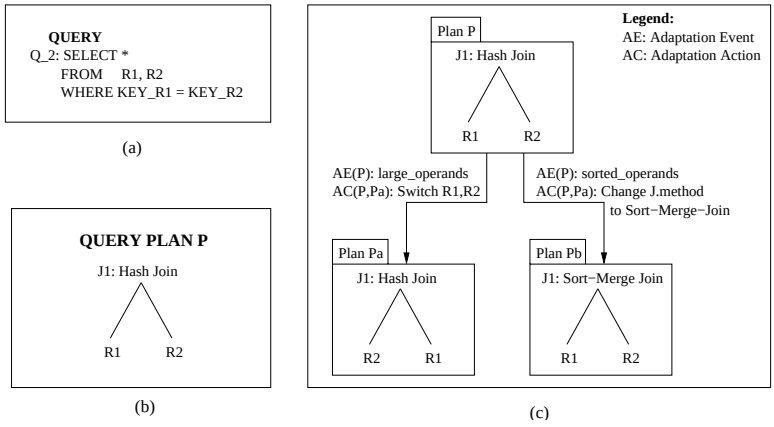


Figure 17: (a) Query Q_2 and (b) associated optimized query plan P; (c) An adaptation space for Q_2.

Example 12 Consider query Q_2 shown in Figure 17(a) where base relations are local. Also, let assume that I want to adapt the execution of Q_2 to changes in memory constraints. In the first step of adaptation space construction for Q_2, Ginga generates the optimized query plan P shown in Figure 17(b). This plan is created based on the set of resource parameter predicates in adaptation_condition(P) summarized in Table 1. On Step 2, Ginga generates

the monitors that will trigger the adaptation events listed in Table 2. In this table, I also describe the associated adaptation action to be executed. In Step 3, the alternative plans are generated for each new adaptation condition created. In this example, Ginga creates two new adaptation conditions as described in Table 3. Observe that it is possible for both adaptation triggers to be fired at the same time. In this case, Ginga will select the adaptation action that yields a query plan with the least cost. Figure 17(c) depicts the adaptation space for P. □

Table 1: Resource Parameter Predicates in adaptation_condition(P).

Predicate Name	**Description**
OperandOrganization	`isSorted`(R_1) == FALSE **and** `isSorted`(R_2) == FALSE
OperandSize	$\|R_1\| \leq \|R_2\|$

Table 2: Adaptation events and adaptation actions.

adaptation_event(P)	**Invalidated Predicate**	**adaptation_action**(P, P_j)
sorted_operands	OperandOrganization	change J_1.*method* from Hash Join to Sort-Merge Join.
large_operands	OperandSize	switch the operands of J_1.

Table 3: Generated Adaptation Conditions.

adaptation_condition(P_a)	
Predicate Name	**Description**
OperandOrganizationP_a	`isSorted`(R_1) == FALSE **and** `isSorted`(R_2) == FALSE
OperandSizeP_a	$\|R_1\| > \|R_2\|$
adaptation_condition(P_b)	
Predicate Name	**Description**
OperandOrganizationP_b	`isSorted`(R_1) == TRUE **and/or** `isSorted`(R_2) == TRUE
OperandSizeP_b	$\|R_1\| \leq \|R_2\|$

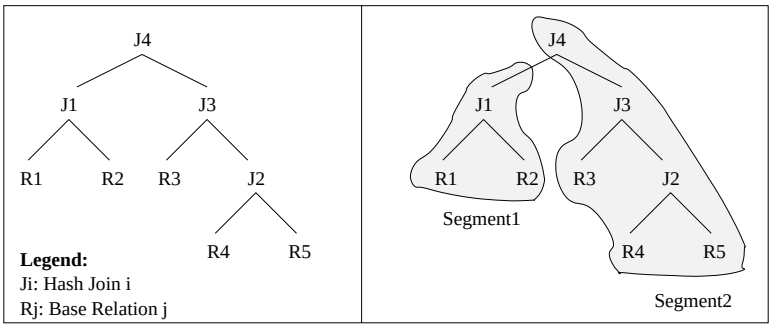
Discussion on the construction of an adaptation space: Constructing an adaptation space with adaptation cases for every single change to the runtime environment is unrealistic due to the following reasons. First, it is very unlikely that all possible changes will indeed occur. Second, the overhead cost for constructing such adaptation space is prohibitive. A more realistic approach for constructing an adaptation space is to generate adaptation cases

for only those runtime environment changes that are known to occur with a frequency above a realistic threshold.

In Ginga approach, for the changes to the runtime environment that are not covered by the adaptation space, no adaptation action is fired. However, new adaptation cases can then be inserted to the adaptation space as I learn more about the runtime environment behavior over multiple runs of the same query. I gather information of the runtime environment by monitoring the usage of the resources allocated during the query execution.

4.3.2 Query Execution Model

In Ginga, query plans are processed according to the Segmented Execution Model (SEM) as described in [53]. In this model, a query plan P_i is first decomposed into a set of right-deep segments. Then, a segment execution schedule, $sch(P_i)$, is generated by assigning numbers to each segment in a left-deep recursive manner. Segments are executed one at a time. For example, suppose that P_i is the query plan shown in Figure 18(a). According to SEM, P_i will be divided into two segments, $Segment_1$ and $Segment_2$ (Figure 18(b)), where $sch(P_i) =< Segment_1, Segment_2 >$. Segments are further represented as a sequence of operators. In this example, $Segment_1 =< J_1 >$ and $Segment_2 =< J_4, J_3, J_2 >$.



(a) Query Plan P_i (b) Segmented Query Plan P_i

Figure 18: Query plan decomposed into right-deep segments.

Segments are right-deep because right-deep trees allow me to pipeline the execution of hash joins and achieve inter-operator parallelism. If enough memory is available for constructing the hash tables (or at least part of them, if partitioning is needed) for the join operators involved in a right-deep (sub) tree, then I can execute these operators concurrently

in a pipeline fashion. The advantage is that I can eliminate the need for caching intermediate results into disk (as needed) and consequently improve the query total response time. SEM assumes that join operators are hash-based.

Considering that at any given moment there will be only one segment $s_k \in sch(P_i)$ being executed, in the rest of this dissertation, I study how to adapt the execution of s_k when there is a violation of predicates in adaptation_condition(P_i) that affects the execution of s_k. Observe that by adapting s_k, Ginga generates a new query plan P_j.

When generating the initial plan P, Ginga's initial query optimizer can consider different tree shapes for P: left-deep, right-deep, and bushy. As demonstrated in [53], bushy trees are the most general and the most interesting because they offer the best opportunities to minimize the size of intermediate results. Thus, I assume that Ginga's initial optimizer will always favor for bushy trees. However, for most of my experiments reported in this dissertation, I use right-deep trees because I want to investigate the efficiency of Ginga's adapting each individual segment.

In this dissertation, I concentrate on adaptation actions are applied to a single segment. Consequently, actions that consider join re-ordering across multiple segments are not covered. The implementation of such adaptation actions translates into the construction of a dynamic query re-optimizer, which is part of my future research work.

4.4 Adaptation Space Exploration Phase

So far, I have discussed the Adaptation Space Generation phase, where the initial query plan P_0 is chosen for the input query Q and its adaptation space is constructed to cover those degradation situations that I am interested in providing adaptation support. Once the adaptation process is initiated, the schedule for the initial query plan is sent to the scheduler for execution and the Adaptation Space Exploration phase (AdaptS-EP) receives the associated adaptation space. In addition, the newly created adaptation space for Q is stored in the Adaptation Space Repository.

Before executing segment s, the scheduler checks with AdaptS-EP whether adaptation is needed for s. For example, AdaptS-EP can adapt segment s by changing the memory

allocation strategy and/or changing the method of some of the operators. Once the adaptation is done (if any), the revised segment is returned to the scheduler, which in turn will submit it to the runtime system. When the runtime system finishes executing the segment, it sends an *end-of-sequence* message to the scheduler.

During the execution of a segment, the runtime system keeps AdaptS-EP informed of the observed values of the resource parameters and AdaptS-EP monitors the the validity of the resource parameters predicates in adaptation_condition(P_i). If one (or more) predicate in adaptation_condition(P_i) becomes invalid, an adaptation_event(P_i) is generated and adaptation_trigger(P_i) is fired. Ginga makes the transition in the adaptation space to the next query plan P_j by matching the new set of runtime parameter predicates with adaptation_condition(P_j). The query execution continues with P_j. Observe that changing from one query plan to another may change the sequence of segments that a scheduler needs to submit for execution. Consequently, when the scheduler receives the *end-of-sequence* from the runtime system, the scheduler needs to contact the AdaptS-EP to get the next segment to be executed (if any).

At the end of the query execution, the scheduler sends the result R of query Q to the Query Manager, which in turn delivers R to the caller process.

For the subsequent executions of Q, the AdaptS-GP will do one of the following actions: If no new adaptation techniques for adapting the query execution are available, then AdaptS-GP will load Q's adaptation space from the Adaptation Space Repository. Otherwise, AdaptS-GP will not only load Q's adaptation space, but also update it with the new adaptation techniques. For example, consider that at first Ginga had only adaptation techniques available for coping with end-to-end delays. Then, as the server where Ginga is running becomes overloaded, the system administrator decides to also incorporate adaptation techniques for memory constraints. Now, when Ginga executes the new (and old) queries, it will also incorporate the memory constraint adaptation techniques.

4.5 Discussion: Combining Adaptation Space Generation and Exploration Phases

To gain a better understanding of the cost and effectiveness of adaptation space generation and exploration, I conducted a series of experiments comparing the performance of (1) no adaptation, (2) static generation of adaptation plans, and (3) dynamic generation of adaptation plans for the case where the distributed query experiences end-to-end delays (i.e., the remote data is delivered at a slower rate than expected). Figure 19 compares the total query execution time of the three cases, taking into account the execution of successive query plans as well as the adaptation overhead. For generating this graph, I used the experimental setup described in Section 6.3.1 (Chapter 6: Adaptation to End-to-End Delays).

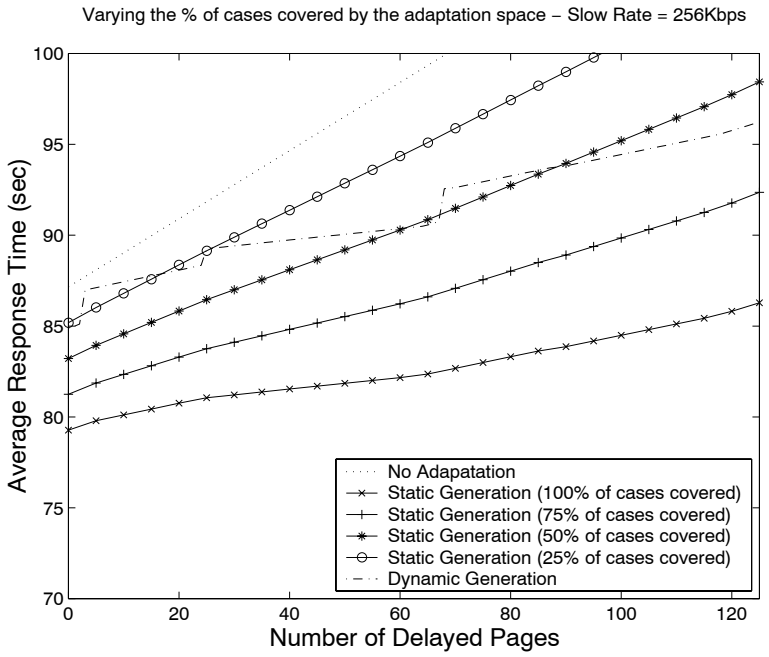


Figure 19: Static and Dynamic Generation of Alternative Query Plans

The graph shows the total query response time (seconds) as a function of network delay (number of pages delayed) in a representative scenario (with detailed explanations in Section 6.3.1). The graph shows (not surprisingly) that the query response time is longest without adaptation (the top line). Then, as the number of alternative query plans becomes

larger, covering an increasingly larger number of cases (from 25% to 100% of the cases), the response time improves significantly (about 20% faster for the 100% coverage in the middle of the graph). Dynamic generation (the "staircase" curve) fits between the 25% and 50% coverage in most cases, improving more when delays become longer.

This study (Figure 19 and other graphs omitted due to similarity) shows that static generation of alternative query plans is advantageous in many cases of interest. Even when only 25% of the adaptation cases have been generated, for a small number of delayed pages static generation can still be more efficient since dynamic generation has to compensate for the generation cost. The vertical jumps in the dynamic generation "staircase" line are due to the overhead of dynamically generating new query plans to cope with the increasing number of delayed pages. In this dissertation, I adopt the static generation approach.

4.6 Summary of Ginga Approach

Adaptive query processing is an active research area that has received considerable attention (see Section 2.4) due to the increasing openness of information systems such as Internet, where sudden changes such as network delays are common. Most of the previously proposed approaches can be classified into predefined (compile-time or start-up time generation of foreseeable query plans) or reactive (runtime adaptation according to actually observed environmental changes). In this chapter, I described Ginga, an approach to adaptive query processing that combines generation of predefined query plans with run-time monitoring and reactive switching of query plans.

Ginga has two main phases: Adaptation Space (AdaptS) Generation and Adaptation Space Exploration. At AdaptS Generation phase, prior to runtime, Ginga generates an adaptation space for the input query. At runtime, AdaptS Exploration phase monitors the use conditions. Whenever one or more use conditions become true, Ginga then navigates the adaptation space, selects the most appropriate adaptation case, and fires the associated adaptation action which replaces the current query plan with an alternative one optimized for the changes to the runtime environment configuration.

In the next chapters, I analyze the benefits and trade-offs of using Ginga for adapting to

two different failure types, namely, memory constraints (Chapter 5) and end-to-end delays (Chapter 6). In Chapter 7, I investigate Ginga's performance characteristics when both memory constraints and end-to-end delays are detected at runtime.

CHAPTER V

ADAPTATION TO MEMORY CONSTRAINTS

5.1 Coping with Memory Allocation Mismatches

Despite the continued evolution of computer systems under Moore's Law, real world database systems almost always need more main memory than is available. This is the case for complex decision support queries involving large relations and also in multi-user environments where a number of concurrently executing queries compete for a finite amount of main memory. Careful memory management for query execution under memory constraints has been studied by several researchers, as summarized in Section 2.4.1.

During the execution of a complex query, the relationship between memory resource availability and query processing requirements may change for a number of reasons. On the query processing requirements side, for example, if intermediate join result sizes turn out to be significantly different from what is estimated at query plan generation time, the execution may thrash due to insufficient memory. Similarly, a large number of concurrent queries in the system may result in a query execution receiving fewer memory blocks than the requirements established at the query plan generation time.

In this chapter, I investigate several dynamic adaptation techniques that overcome the variable memory constraint bottlenecks when available memory no longer matches query processing requirements at runtime [45]. These adaptation techniques are organized into an adaptation space, which provides a uniformed way for representing and viewing a collection of alternative adaptations to memory constraints for a given query execution plan.

The rest of the chapter is organized as follows. Section 5.2 presents preliminary background related to the adaptation techniques described in Section 5.3. Also in Section 5.3, I discuss how to organize these techniques into an adaptation space G, and how Ginga navigates G to get around the changes in memory constraints. Section 5.4 studies the performance of the three techniques for representative scenarios. Section 5.5 summarizes the

chapter.

5.2 Preliminaries

In this section, I first revisit three classical join algorithms and discuss their memory requirements. Then, I describe three alternative memory allocation strategies for query execution plans. Both join algorithms and memory allocation strategies are used for defining the adaptation techniques described in Section 5.3.

5.2.1 Join Methods

In a relational database environment, three join algorithms have been commonly used: nested-loop join, hash join, and sort-merge join [18, 52]. Each of these operators can operate in a range of memory allocations between their minimum (M^{op}_{min}) and maximum (M^{op}_{max}) requirements with different performance. If the memory allocate to an operator is smaller than M^{op}_{min}, the operator cannot be executed [16]. When the maximum memory required is allocated to the operator, it has optimal performance. The operator performance degrades if memory allocation falls between M^{op}_{min} and M^{op}_{max}.

In the remainder of this section, for each join method, I present their minimum and maximum memory requirements, and briefly discuss how the memory blocks allocate to each operator is used.[1] I assume that I am joining relations R and S using M memory blocks, where $|R| < |S|$, for $|R|$ representing the size of R in pages.

Hash Join: For Hash Join method, $M^{hj}_{min} = \sqrt{\delta * |R|}$ and $M^{hj}_{max} = \delta * |R|$, where δ is an overhead factor for the hash table. When $|M| \geq M^{hj}_{max}$, hash join can be execute in two steps. In the first step, the tuples from R are hashed and a hash table is built in main memory. In the second step, tuples from S are sequentially read and used to probe R's hash table in main memory. If a match is found, an output is generated by concatenating the corresponding tuples from R and S and adding it to the result relation.

When $M^{hj}_{min} \leq |M| < M^{hj}_{max}$, I use a hybrid hash join algorithm [52] that executes the

[1] An in-depth discussion on how minimum and maximum memory requirements are defined for each operator can be found in [52]. Presenting this discussion in this dissertation is out of the scope.

join in three main steps.

Step 1: Hash tuples from R into multiple partitions, where the first partition is kept in main memory, and the others are flushed to disk.

Step 2: Hash tuples from S into multiple partitions using the same hash function, where the tuples that are hashed into the first partition are joined with those tuples from R kept in main memory in Step 1. The other partitions of S are flushed to disk.

Step 3: Join the remaining partitions. For each pair of corresponding partitions, construct in main memory the hash table with the tuples from R and use the tuples from S to probe the hash table to find a match.

The number of partitions in Step 1 is chosen in a way that the hash table for each partition alone can fit in main memory.

Sort-Merge Join: For Sort-Merge Join method, $M_{min}^{smj} = \sqrt{|S|}$ and $M_{max}^{smj} = |S|$. SMJ has three phases. In the first phase, each operand is scanned and sorted runs are produced. In the second phase, the sorted runs from each operand are merged to generate a single sorted relation. In the third phase, R and S are joined by merging the matching tuples on the join attribute. When $|M| \geq M_{max}^{smj}$, phases one and two can be combined together. If one or both operands are sorted, phases one and two are bypassed and M_{min}^{smj} and M_{max}^{smj} are equal to two memory blocks: one for each operand.

Nested-Loop Join: For Nested-Loop Join method, $M_{min}^{nlj} = 2$ and $M_{max}^{nlj} = |R|$. The algorithm for NLJ is a simple one: scan the outer relation R once, and for each tuples of R scan S, the outer relation, to find a possible match. The minimum memory amount is two memory blocks: on for the outer relation and another for the outer relation. If $M_{min}^{nlj} \leq |M| < M_{max}^{nlj}$, the inner relation will be read $\frac{|R|}{|M|}$ times. For $|M| \geq M_{max}^{nlj}$ both inner and outer relation need to be read only once.

Table 4 summarizes the memory requirements for the joins discussed above. Note that I also include the requirements for a simple variation of the HJ methods, that I call Hash

Table 4: Memory requirements for join methods.

Method	Minimum	Maximum
Nested-Loop Join	2	$\|R\|$
Hash Join	$\sqrt{\delta * \|R\|}$	$\delta * \|R\|$
Hash Join Improved	$\sqrt{\delta * min(\|R\|, \|S\|)}$	$\delta * min(\|R\|, \|S\|)$
Sert-Merge Join	$\sqrt{\|S\|}$	$\|R\| + \|S\|$

Join Improved (HJ-Improved). Before starting its execution, HJ-Improved checks whether the left operand is indeed smaller than the right operand. If this is not the case, then HJ-Improved switches the operands. As I demonstrate in Section 5.4.1, this simple modification can maintain the efficiency of the hash join method when the left operand becomes larger than expected at runtime.

5.2.2 Memory Allocation Strategies

Memory allocation strategies address the problem of how to allocate memory to concurrently running operators of a query in a way that will minimize the total query response time. There are at least three memory allocation strategies proposed in the literature: Equal [38, 10], Max2Min [38, 10], and the strategy proposed in [10], which I refer in this chapter as AlwaysMax.

The *Equal* strategy evenly distributes the memory blocks among the operators in a segment. It first assigns to each operator the minimum memory required and then evenly distribute the remaining memory blocks (if any) among the operators. This strategy can hardly be optimal as it does not take into consideration the real memory requirements of each operator. In fact, as demonstrated in [10], Equal does present poor performance in many different scenario, and I do not further consider this allocation strategy in this dissertation.

The *Max2Min* strategy follows the heuristic described in [61]: in order to obtain better return on memory consumption, allocate the maximum amount of memory to operators with small maximum memory requirements. For each segment, Max2Min first assigns each operator with the minimum memory requirement. Then, the remainder memory blocks are distributed as follows. First, operators are sorted in reverse order with respect to

their maximum memory requirements. Then, distribute the remaining memory blocks by following the sort order and trying to match the maximum memory requirements of the first operators.

The *AlwaysMax* strategy attempts to always provide the operators with their maximum memory requirements. When there is not enough memory to match these maximum requirements, the segment is divided in two sub-segments, which will be executed sequentially. The idea is to free up as much memory as possible for executing the joins in each of the segments. However, the main disadvantage of this approach is the need to write to the disk the intermediate result generated by the first segment. Nevertheless, as shown in [10], the extra I/O cost can be amortized when using bushy query trees.

5.3 Adaptation Space Generation and Exploration: Memory Constraints

In this section, I first discuss how the adaptation cases for memory constraints are constructed. Then, I discuss how the adaptation space is generated, by Adaptation Space Generation phase, prior to runtime, and navigated (as needed) by Adaptation Space Exploration phase, at runtime.

5.3.1 Adaptation Cases

Given an $adaptation\_case(P_i)$, when one of its memory constraint predicates described in $adaptation\_condition(P_i)$ becomes invalid, Ginga needs to provide an alternative adaptation case. This new adaptation case has a new set S of memory constraint predicates and an alternative query plan P_j optimized for S ($adaptation\_condition(P_j)$) along with the respective $adaptation\_trigger(P_i)$. The new adaptation case is represented as $adaptation\_case(P_j)$ and the $adaptation\_event$ that fires the transition from $adaptation\_case(P_i)$ to $adaptation\_case(P_j)$ is described in $adaptation\_trigger(P_i)$.

A key component of $adaptation\_trigger(P_i)$ is $adaptation\_action(P_i, P_j)$ (see Section 4.3.1), which Ginga uses for generating P_j. In the next subsections, I describe three adaptation actions that Ginga can use for generating alternative query plans (and the associated adaptation case) for coping with variable memory constraints. These adaptation actions

are (1) changing join algorithms, (2) switching operands of a hash join if left operand is larger than right operand, and (3) choosing a memory allocation strategy appropriate for the query plan and memory available.

I observe that these three adaptation actions are based on the decisions that a query optimizer makes while constructing the optimized query plan. Ginga uses these actions to re-adjust the optimizer's decisions in case some of the estimated runtime environment parameters used at plan generation are found to be different from their actual (observed) values during query execution.

5.3.1.1 Adaptation Action: Changing Join Algorithm

The first adaptation action, "Changing Join Algorithm", consists of selecting the join method JM that yields the least cost based on the file organization of the input operands and the actual memory allocated to JM. When I compare the three methods described in Section 5.2.1, nested-loop join (NLJ) has the worst performance. In general, hash join (HJ) offers superior performance [52]. However, sort-merge join (SMJ) can outperform HJ if at least one of the operands is sorted (see Section 5.4.1). Based on this observation, I define "Changing Join Algorithm" adaptation action (Figure 20) as follows: Given s_k, the current segment to be executed from input plan P_i, for all join operator $op_j \in s_k$, if op_j is a HJ and at least one of its operands is sorted, then change op_j to SMJ.

ADAPTATION ACTION `adapt_action_chgJoinAlgo`
Input: P_i: current plan.
Output: P_j: plan P_i adapted.
1: **Let** $s_k =< op_1, \cdots, op_n > \in sch(P_i)$ be the current segment to be executed.
2: P_j = copyOf(P_i); // `Make a copy of` P_i `to generate the adapted plan` P_j.
3:
// `For each join operator in` s_k, `check whether I should change its method.`
4: **for** $\forall op_i \in s_k$ **do**
5: **if** ($op_i.method \neq$ `sort_merge_join`) **and** (`isSorted`($op_i.leftOperand$) **or** `isSorted`($op_i.rightOperand$)) **then**
6: $op_i.method =$ `sort_merge_join`;
7: **return** P_j;

Figure 20: Adaptation Action "Changing Join Algorithms."

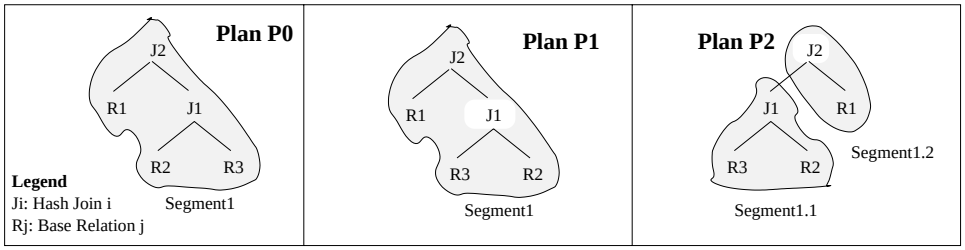
5.3.1.2 Adaptation Action: Switching Operands of Hash Joins

The adaptation action "switching operands of hash joins" is a simple one: if the actual size[2] of the left operand of a hash join is larger than the size of the right operand, then switch the operands. However, despite its simplicity, this action has interesting properties, as I illustrate next. I assume that operands are switched *before* starting the execution of the hash join. Figure 21 depicts the adaptation action for switching operands.

ADAPTATION ACTION `adapt_action_switchJoinOprnd`
Input: P_i: current plan.
Output: P_j: plan P_i adapted.
1: **Let** $s_k =< op_1, \cdots, op_n > \in sch(P_i)$ be the current segment to be executed.
2: P_j = copyOf(P_i);
// `For each hash join operator in` s_k`, check whether I should switch its operands.`
3: **for** $i = n, \cdots, i$ **do**
4: **Let** $op_i \in s_k$ // `Iterate thorugh the operators of` s_k `in reverse order.`
5: **if** ($op_i.method ==$ `hash_join`) **and** (`sizeOf`($op_i.leftOperand$) > `sizeOf`($op_i.rightOperand$)) **then**
6: $aux = op_i.leftOperand$;
7: $op_i.leftOperand = op_i.rightOperand$; // `Switching operands of` op_i.
8: $op_i.rightOperand = aux$;
9: $op_i.method =$ `hash_join_improved`; // `Flag that this join was improved.`
// `Check whether the switching of operands will result in changing the shape of the query tree.`
10: **if** op_i is **not** the last operator in s_k **then**
11: P_j.`reshapeQryTree`(op_i); // `Divide` s_k `at` op_i `and change the shape of the tree.`
12: **break**; // `Interrupt the` *for*-`loop.`
13: **return** P_j;

Figure 21: Adaptation Action "Switching Operands of Hash Joins."

Example 13 Consider the query plan P_0 shown in Figure 22(a) where $sch(P_0) =< Segment_1 >$. Also, assume that `sizeOf`$(R_2) >$ `sizeOf`(R_3) and `sizeOf`$(R_1) >$ `sizeOf`(J_1), where `sizeOf`(X) returns the size of X in *bytes* and `sizeOf`(J_1) is the result size of J_1.



(a) Initial Segment (b) Switching operands of J1 (c) Switching operands of J2

Figure 22: Effects of switching operands.

[2]The observed size at runtime of a base relation or intermediate result.

When Ginga executes `adapt_action_switchJoinOprnd`, the current segment of P_0 to be executed is $Segment_1$ and J_1 is the first operator considered in the for-loop of Line 3. J_1 satisfies the condition in Line 4 and its operands are switched generating a new plan, P_1 (Figure 22(b)). A similar process is executed for the next operator J_2, resulting in query plan P_2 shown in Figure 22(c).

When Ginga switches the operands of J_1, the original tree shape of P_0 is not affected. However, by switching the operands of J_2, the original right-deep tree is transformed into a left-deep tree. In this case, Ginga needs to change $Segment_1$ by dividing it in two sub-segments, namely, $Segment_{1.1}$ and $Segment_{1.2}$, and updating the segment schedule of P_2 to $sch(P_2) =< Segment_{1.1}, Segment_{1.2} >$. The division of $Segment_1$ is necessary because I am using SEM, where each segment is a right-deep (sub) tree. By switching the operands of J_2, I violated this property. The process of detecting the change in shape of a query plan tree and updating the respective segment schedule is captured in Lines 9 and 10 (Figure 21).

The process of re-shaping the query tree by simply switching the operands of a hash join has two interesting properties. First, I can perform a simple re-optimization of the current query plan by improving the performance of hash joins. Second, if the size of the left operand is larger than expected, it is likely that the result of the associated hash join operator will also not match its estimated size. By changing the shape of the query tree, I am able to (1) stop the propagation of inaccurate estimated size of intermediate results up into the tree and (2) handle the estimation error when I schedule the next segment generated in the re-shaping process. This is possible because Ginga adapts each segment of a plan P_i as they are scheduled (see Section 4.3.2).

5.3.1.3 Adaptation Action: Choosing Memory Allocation Strategy

Our third adaptation action for coping with variable memory constraints chooses the appropriate memory allocation strategy for distributing the available memory among the operators of a segment so that the total query response time is minimized. Figure 23 depicts the adaptation action for choosing the appropriate memory allocation strategy.

Given s_k, the current segment to be executed from input plan P_i, Ginga first estimates

the costs of executing s_k using Max2Min and AlwaysMax. Then, Ginga chooses the strategy that yields the least response time. An important parameter for estimating the execution cost of s_k is *memory_size*. Considering that each segment is executed one at a time, *memory_size* is equal to the memory allocated by the Memory Manager for executing P_i. Ginga estimates the cost of executing s_k using the function `segmentExecCost` described in Figure 25. When a segment has a single operator, there is no need to determine which allocation strategy to choose. All the memory blocks allocated for the query will be assigned to the only operator.

ADAPTATION ACTION `adapt_action_chooseMemAlloc`
Input: P_i: current plan.
Output: P_j: plan P_i adapted.
1: **Let** $s_k =< op_1, \cdots, op_n > \in sch(P_i)$ be the current segment to be executed.
2: P_j = copyOf(P_i);
   // `Calculate the cost of executing` s_k `using Max2Min.`
3: $minCost$ = `segmentExecCost(`s_k, `max2min,` $memory\_size$`)`;
4: $allocAlgo$ = `max2min`;
   // `Calculate the cost of executing` s_k `using AlwaysMax.`
5: $alwaysMaxCost$ = `segmentExecCost(`s_k, `alwaysMax,` $memory\_size$`)`;
   // `Choose the allocation strategy that yields the least cost.`
6: **if** ($minCost \geq alwaysMaxCost$) **then**
7: $allocAlgo$ = `alwaysMax`;
8: s_k.memAllocAlgo = $allocAlgo$; // `Records the selected memory allocation strategy.`
9: **return** P_j; // $s_k \in sch(P_j)$.

Figure 23: Adaptation Action "Choosing Memory Allocation Strategy."

FUNCTION `estimatedExeCost`
Input: (1) P_i: current plan; (2) *memory_size*: memory allocated to execute P_i.
Output: *relativeCost*: cost of executing P_i starting at s_k, the current segment to be executed.
1: **Let** $sch(P_i) =< s_1, \cdots, s_n >$
2: **Let** $s_k \in sch(P_i)$ be the current segment to be executed and $k \leq n$;
3: $relativeCost = 0$;
4: **for** $j = k, \cdots, n$ **do**
5: $relativeCost = relativeCost +$ `segmentExecCost(`$s_j, s_j.memAllocAlgo, memory\_size$`)`;
6: **return** $relativeCost$;

Figure 24: Segment cost estimation.

5.3.2 Adaptation Space

So far, I have described three adaptation actions for generating adaptation cases to get around changes to memory constraints during the execution of query plan P_i. In this section, I briefly describe how to organize these actions into an adaptation space G, and

```
FUNCTION segmentExecCost
Input: (1) s_k: segment to be executed; (2) allocStrat: memory allocation strategy for s_k;
    (3) memory_size: memory allocated to execute s_k.
Output: segCost: cost of executing s_k.
 1: Let s_k =< op_1, ..., op_n >;
 2: allocate_memory(s_k, allocStrat, memory_size);   // Allocate memory to operators in s_k using allocStrat
    strategy.
 3: segCost = 0;
 4: for i = 1, ..., n do
 5:    segCost = segCost + op_i.cost(); // op_i.cost() is the cost function of HJ or SMJ.
 6: return segCost;
```

Figure 25: Cost estimation of a query plan, starting from the current segment to be executed.

how Ginga navigates G for coping with variable memory constraints.

Ginga will adapt the execution of P_i when at least one of the memory constraint predicates listed in Table 5 is invalidated. The associated adaptation events and actions to be fired are summarized in Table 6. For example, if the predicate Operand_Predicate is invalidated, then event *large_operands* is fired and `adapt_action_switchJoinOprnd` is executed. Based on these events and actions, Ginga generates the adaptation space for P_i shown in Figure 26. I now describe how Ginga navigates this adaptation space.

Table 5: Memory Constraint Predicates for plan P_i, where $s_k \in sch(P_i)$ is the current segment to be executed.

Predicate Name	**Description**
Algo_Predicate	$\forall op_i \in s_k$, **if** $op_i.method == SMJ$, **then** `isSorted`$(op_i.leftOperand)$ == TRUE **or** `isSorted`$(op_i.rightOperand)$ == TRUE;
Operand_Predicate	$\forall op_i \in s_k$, **if** $op_i.method == HJ$, **then** `sizeOf`$(op_i.leftOperand) \leq$ `sizeOf`$(op_i.rightOperand)$.
MemorySize_Predicate	memory allocate to P_i is greater or equal to the expected memory size (i.e., the memory size assumed during the optimization process).

Table 6: Adaptation events and adaptation actions.

adaptation_event(P_i)	**Invalidated Predicate**	**adaptation_action**(P_i, P_j)
sorted_operands	Algo_Predicate	adapt_action_chgJoinAlgo
large_operands	Operand_Predicate	adapt_action_switchJoinOprnd
small_memory	MemorySize_Predicate	adapt_action_chooseMemAlloc

Let $evnts(P_i)$ be the set of adaptation events of P_i that were fired, and let $comb\_evnts(P_i)$ be the set of all possible combination of these events. I call each combination $ap_k \in$

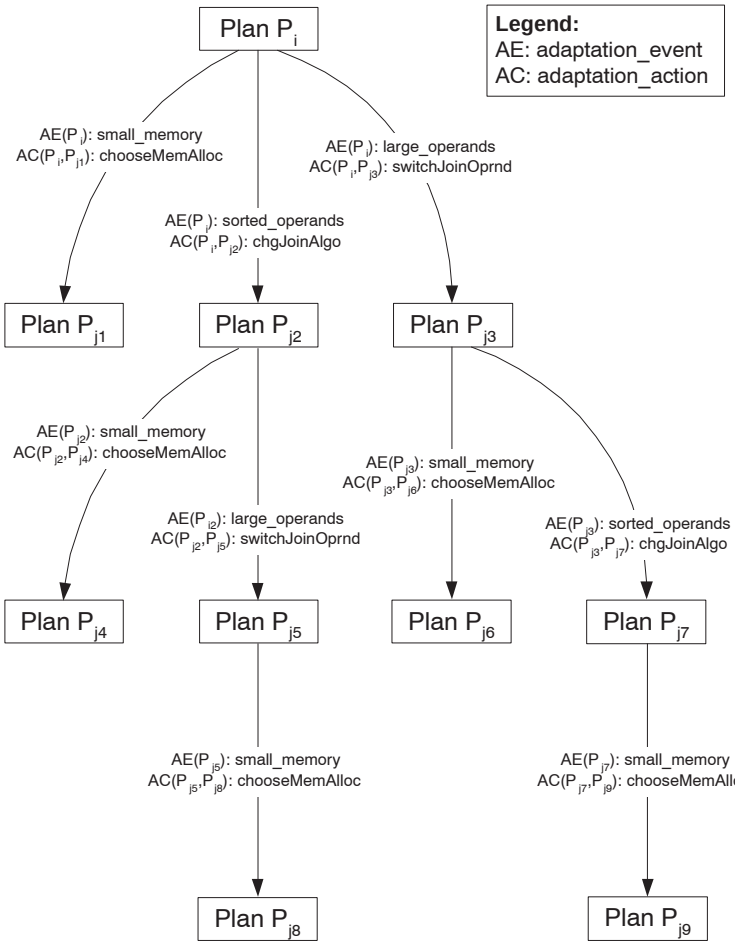


Figure 26: Adaptation space for coping with memory constraints.

$comb\_evnts(P_i)$ an *adaptation path*, which describes a possible path in the adaptation space that Ginga can navigate to find the appropriate alternative plan for P_i.

Example 14 Consider the adaptation space shown in Figure 26. Now, suppose that both Algo_Predicate and Operand_Predicate of plan P_i were invalidated, firing events *sorted_operands* and *large_operands*, respectively. In this case, I have $comb\_evnts(P_i) = \{$ (*sorted_operands, large_operands*); (*large_operands, sorted_operands*)$\}$. If Ginga follows the first adaptation path in $comb\_evnts(P_i)$, plan P_{j5} will be selected (see Figure 26). On the other hand, following the second adaptation path, Ginga will choose plan P_{j7}. The decision on which plan to use is made by comparing the cost of each plan. Ginga selects the plan with the least cost. The algorithm for navigating the adaptation space from Figure 26 is described

in Figure 27.

It is important to observe that not all paths in $comb\_evnts(P_i)$ are relevant. For example, if both Algo_Predicate and MemorySize_Predicate are invalidated, the alternative plan generated by executing first adapt_action_chooseMemAlloc followed by adapt_action_chgJoinAlgo is not a good solution. The selection of the appropriate memory allocation strategy is based on the memory requirements of the operator algorithms in the segment. If after the selection is done I change the operator algorithms, this will invalidate the decision made adapt_action_chooseMemAlloc. The adaptation space depicted in Figure 26 shows only the relevant adaptation paths.

When Algo_Predicate or Operand_Predicate are invalidated, MemorySize_Predicate may also become invalid. This occurs because as Ginga changes the configuration of join operators, the expected memory size for executing the new plan may also change. Consequently, *small_memory* may be fired if the modified join operators require more memory than what was originally allocated to P_i.

5.4 Performance Analysis

For all experiments reported in this chapter, I use a simulator (based on the CSIM toolkit) that models a server machine running Ginga. Table 7 lists the classical parameters [10, 52, 61] used in configuring the simulator. All base relations involved in the queries submitted to Ginga are assumed to be available on the server's local disk. I use different sizes of base relations, where each tuple in a relation has 200 bytes. In particular, the size of the relations reflect the size of the answers provided by web services such as NCBI-Entrez discussed in Section 1.2.

The experiments are divided into three groups: The first group (Section 5.4.1) studies the effects of changing join algorithms or operand ordering. The second group (Section 5.4.2) studies the effects of changing memory allocation strategies. The third group (Section 5.4.3) studies the combination of different algorithms and strategies.

ADAPTATION SPACE `MgtMemoryConstraints`
Input: adapt_space(P_i): adaptation space for P_i.
Output: P_j: plan P_i adapted.
Require: at least one of memory constraint predicates in adaptation_condition(P_i) is invalidate.

```
 1: minCost = ∞;
    // Adaptation Path: (small_memory)
 2: if (MemorySize_Predicate(P_i) == FALSE) then
 3:   adapt_action_chooseMemAlloc(P_i, P_j1);
 4:   minCost = estimatedExeCost(P_j1, memory_size);
 5:   P_j = P_j1;
 6:
    // Adaptation Paths: (sorted_operands); (sorted_operands, small_memory);
    // (sorted_operands, large_operands); and (sorted_operands, large_operands, small_memory).
 7: if (Algo_Predicate(P_i) == FALSE) then
 8:   adapt_action_chgJoinAlgo(P_i, P_j2);
 9:   if (MemorySize_Predicate(P_j2) == FALSE) then
10:     adapt_action_chooseMemAlloc(P_j2, P_j4);
11:     cost_j4 =estimatedExeCost(P_j4, memory_size);
12:     if (minCost > cost_j4) then
13:       minCost = cost_j4;
14:       P_j = P_j4; // Adaptation Path: (sorted_operands, small_memory)
15:   else if (Operand_Predicate(P_j2) == FALSE) then
16:     adapt_action_switchJoinOprnd(P_j2, P_j5);
17:     if (MemorySize_Predicate(P_j5) == FALSE) then
18:       adapt_action_chooseMemAlloc(P_j5, P_j8);
19:       cost_j8 =estimatedExeCost(P_j8, memory_size);
20:       if (minCost > cost_j8) then
21:         minCost = cost_j8;
22:         P_j = P_j8; // Adaptation Path: (sorted_operands, large_operands, small_memory)
23:     else
24:       cost_j5 =estimatedExeCost(P_j5, memory_size);
25:       if (minCost > cost_j5) then
26:         minCost = cost_j5; P_j = P_j5;// Adaptation Path: (sorted_operands, large_operands)
27:   else
28:     cost_j2 =estimatedExeCost(P_j2, memory_size);
29:     if (minCost > cost_j2) then
30:       minCost = cost_j2; P_j = P_j2;// Adaptation Path: (sorted_operands)
31:
    // Adaptation Paths: (large_operands); (large_operands, small_memory);
    // (large_operands, sorted_operands); and (large_operands, sorted_operands, small_memory).
32: if (Operand_Predicate(P_i) == FALSE) then
33:   adapt_action_switchJoinOprnd(P_i, P_j3);
34:   if (MemorySize_Predicate(P_j3) == FALSE) then
35:     adapt_action_chooseMemAlloc(P_j3, P_j6);
36:     cost_j6 =estimatedExeCost(P_j6, memory_size);
37:     if (minCost > cost_j6) then
38:       minCost = cost_j6; P_j = P_j6;// Adaptation Path: (large_operands, small_memory)
39:   else if (Algo_Predicate(P_j3) == FALSE) then
40:     adapt_action_chgJoinAlgo(P_j3, P_j7);
41:     if (MemorySize_Predicate(P_j7) == FALSE) then
42:       adapt_action_chooseMemAlloc(P_j7, P_j9);
43:       cost_j9 =estimatedExeCost(P_j9, memory_size);
44:       if (minCost > cost_j9) then
45:         minCost = cost_j9;
46:         P_j = P_j9;// Adaptation Path: (large_operands, sorted_operands, small_memory)
47:     else
48:       cost_j7 =estimatedExeCost(P_j7, memory_size);
49:       if (minCost > cost_j7) then
50:         minCost = cost_j7; P_j = P_j7;// Adaptation Path: (large_operands, sorted_operands)
51:   else
52:     cost_j3 =estimatedExeCost(P_j3, memory_size);
53:     if (minCost > cost_j3) then
54:       minCost = cost_j3; P_j = P_j3;// Adaptation Path: (large_operands)
55: return P_j;
```

Figure 27: Adaptation Space for Managing Memory Constraints.

Table 7: Simulation Parameters

Parameter	Value	Description
Speed	100	CPU speed (MIPS)
AvgSeekTime	8.9	average disk seek time (msecs)
AvgRotLatency	5.5	average disk rotational latency (msecs)
TranferRate	100	disk transfer rate (MBytes/sec)
DskPageSize	8192	disk page size (bytes)
MemorySize	$10 \cdots 100$	memory size (MBytes)
DiskIO	5000	instructions to start a disk I/O
Move	2	instructions to move 4 bytes
Comp	4	instructions to compare keys
Hash	25	instructions to hash a key
Swap	100	instructions to swap two tuples
F	1.2	incremental factor for hash join

5.4.1 Join Algorithms

To study the effects of changing join algorithms and operand ordering, I use a simple workload: a single join $J = R_1 \bowtie R_2$, where I fix $|R_2| = 50MBytes$ and vary the size of R_1 from $5MBytes$ to $200MBytes$ using two different memory sizes, namely, $25MBytes$ and $50MBytes$. Arguably, more sophisticated joins will contain this elementary configuration as a subset and the results of this simple experiment will apply to them as well. Although the join algorithms are well known, I was surprised by the non-trivial differences among them. I ran experiments with a variety of operand and memory sizes to test the sensitivity of the parameter settings, and found similar results.

The graphs in Figure 28 show the response time of executing join J as a function of R_1 size using four different methods, namely, nested-loop join (NLJ), sort-merge join (SMJ), hash join (HJ), and hash-join improved (HJ-Improved). HJ-Improved switches the operands when the left is larger than the right operand. For sort-merge join, I study four subcases: with R_1 sorted, R_2 sorted, both sorted, and neither sorted.

When there is enough memory to fully execute J in main memory, all four join methods have similar performance. The difference between their response time stems from the different number of CPU instructions. However, as the size of the left operand (R_1) increases, requiring I/O operations, significant differences arise. First, as expected, NLJ has the worse performance. The vertical lines at 25MB in Figure 28 (a) and at 50MB in Figure 28 (b) show the well understood scalability problems of NLJ and I do not discuss it further. However,

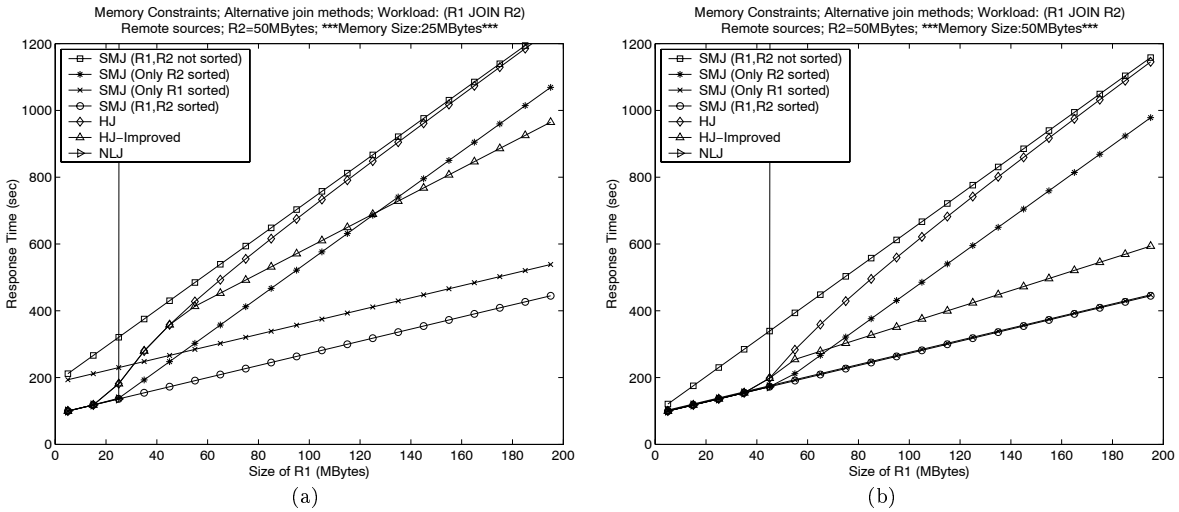


Figure 28: Response time of alternative join algorithms using two memory sizes: (a) 25MBytes and (b) 50MBytes.

NLJ can be improved when an index on the join attribute is available for at least one of the join operands [20]. The reason is because for each tuple on the left (outer) operand, I can use the index to quickly determine whether there is a match on the right (inner) operand.

For the main memory size of 50MB, Figure 28 (b) shows that SMJ is the fastest join method when R_1 or both R_1 and R_2 are sorted. It also shows that when only R_2 is sorted, SMJ is still better than HJ, but HJ-Improved now performs better than SMJ. This is an interesting observation, since SMJ is slower than HJ in the general case, when both R_1 and R_2 are unsorted. The figure shows that when one of the operands is sorted, it is worth switching to SMJ in most situations. Similarly, HJ-Improved is a significant improvement over static HJ, since HJ-Improved is able to cope with an increasingly larger size of left operand by always switching the smaller operand to the left.

For a smaller main memory size of 25MB, Figure 28 (a) shows that the relative performance of HJ, HJ-Improved, and SMJ becomes more intricate when the memory constraints are tighter. While the shape of the curves remained the same, vertical displacements changed the trade-offs among them. First, since R_2 does not fit in main memory anymore, when only R_1 is sorted, SMJ is significantly slower than when both operands are

sorted. Second, the advantage of HJ-Improved starts later, losing to SMJ R_2-sorted for a significant range of smaller R_1 sizes.

5.4.2 Memory Allocation Strategies

To study the effects of different memory allocation strategies, I chose the simple example query plan P_0 (right-deep tree) depicted in Figure 29 with the following configuration: $|R_1| = 25MBytes$, $|R_2| = 50MBytes$, $|R_3| = 100MBytes$, and $|J_2| = 25MBytes$. I ran experiments with a variety of tree shapes, and operand and memory sizes to test the sensitivity of the parameter settings, and found similar results.

The graphs in Figure 30 show the response time of executing P_0 as a function of the memory size allocated to this plan using two different memory allocation strategies, namely, Max2Min and AlwaysMax. For a smaller intermediate result (50MB), Figure 30 (a) shows that AlwaysMax loses to Max2Min when main memory size is smaller (less than 20MB). The reason for that is because the intermediate result will not fit in main memory, and it will have to be completely cached into disk and read back again. For intermediate memory sizes (between 20MB and 75MB), AlwaysMax wins. This occurs because the extra I/O cost is amortized by the gain that I get by giving the maximum memory blocks to each operator. But, as I increase the size of the result of the J_2, even by amortizing the I/O cost, AlwaysMax will be no better than Max2Min (Figure 30(b)).

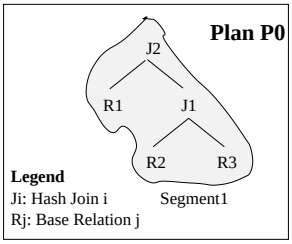
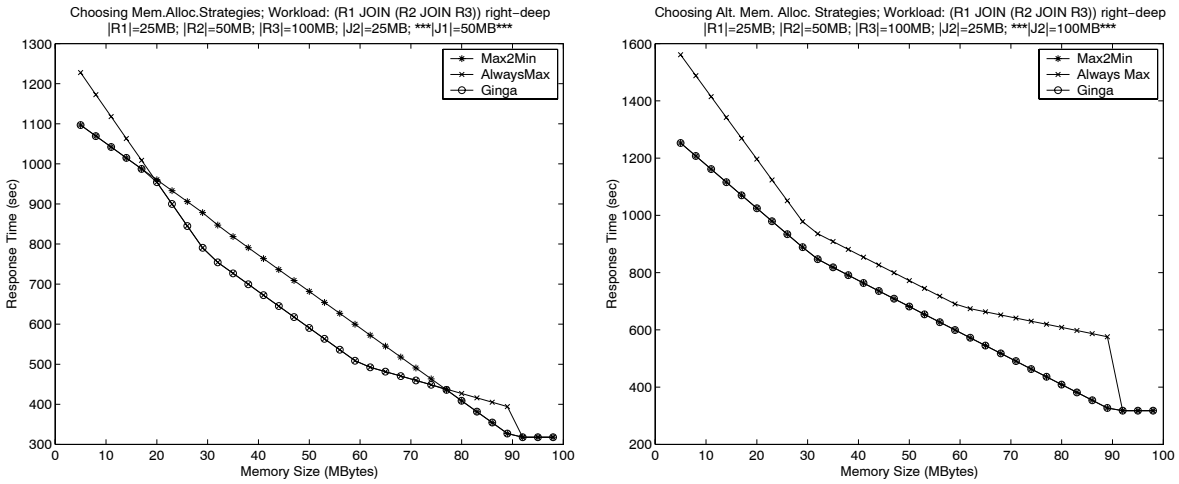


Figure 29: Query Plan P_0 used in the experiments described in Section 5.4.2 and Section 5.4.3.

In Figure 30(a) and (b), I can see 3 inflection points in the AlwaysMax curve, dividing it into 4 segments. In the first segment (up to about 30MB main memory), the response time rapidly decreases because each join operator is receiving all memory blocks that are



(a) $|J_1| = 50MBytes$ (b) $|J_1| = 100MBytes$

Figure 30: Performance of alternative memory allocation under two scenarios as I vary the memory size: (a) $|J_1| = 50MBytes$; (b) $|J_1| = 100MBytes$.

available to execute P_0. The second segment (between 30MB and 60MB) the response time does not decrease as fast as before because now J_2 (the smaller of the two join operators with respect to memory requirements) receives all the memory that it needs to execute in its best performance (i.e., without incurring any I/O costs), but J_1 still does not receive its maximum required memory to also execute without I/O cost. The third segment shows intermediate results still present. Finally, when both joins can be executed in main memory, no more disk costs are necessary for caching the intermediate result. That explain why around 92MBytes I have a sudden drop in the response time for AlwaysMax. From this point on response time becomes dependent only on CPU cost for performing the actual joins and I/O costs for reading the base relation. Similar observations are applied to the graph shown in Figure 30(b).

In contrast, the Max2Min curve has one inflection point in Figure 30(a) and two in Figure 30(b). Up to 30MB of main memory, both operators increasingly receive more memory blocks, resulting in a fast decrease of the response time. For the smaller intermediate result, Figure 30(a), the same processing rate continues. However, for a larger intermediate result, Figure 30(b), the smaller join receives its maximum required memory, and the decreasing

rate of the response time is given by the improvement in performance of the execution of J_1, which receives more memory blocks. Finally, at around 92MBytes, the response time becomes dependent only on CPU costs.

Figure 30(a) shows a non-trivial relationship between the two memory allocation strategies. It is non-obvious when it is better to use one or the other. So I can conclude that indeed it is advantageous to choose the most appropriate memory allocation strategy for the current memory constraints. As it is clearly shown in this graph, there are moments where it is beneficial to use AlwaysMax and moments where Max2Min is a better alternative. A third line (the bottom most) illustrates the adaptive strategy, where it will always provide the response time using the memory allocation strategy that yields the fastest response time.

5.4.3 Combination of Adaptation Actions

One of the main results of using adaptation space is Ginga's ability to combine different adaptation strategies in a systematic way. My next experiments combine the join method adaptations described in Section 5.4.1 with the memory allocation strategy adaptations described in Section 5.4.2. I again use query plan P_0 from Figure 29, but this time with the following configuration: $|R_1| = 25MBytes$, $|R_2| = 50MBytes$, $|R_3| = 100MBytes$, $|J_1| = 10MBytes$, and $|J_2| = 25MBytes$. Observe that with this configuration two adaptation events will be fired during the execution of P_0: *sorted_operands* and *large_operands*. I also assume that event *small_memory* is fired. I ran experiments with a variety of tree shapes, and operand and memory sizes to test the sensitivity of the parameter settings, and found similar results.

Table 8: Adaptation Paths for the Experiments

Adapt. Path	Events	Sequence of actions
1	(*memory_size*)	Choosing Memory Allocation Strategies
2	(*sorted_operands, memory_size*)	Changing Join Algorithm → Choosing Memory Allocation Strategies
3	(*large_operands, memory_size*)	Switching Hash Join Operands → Choosing Memory Allocation Strategies

The graphs in Figure 31 and Figure 32 show the response time of executing P_0 as a

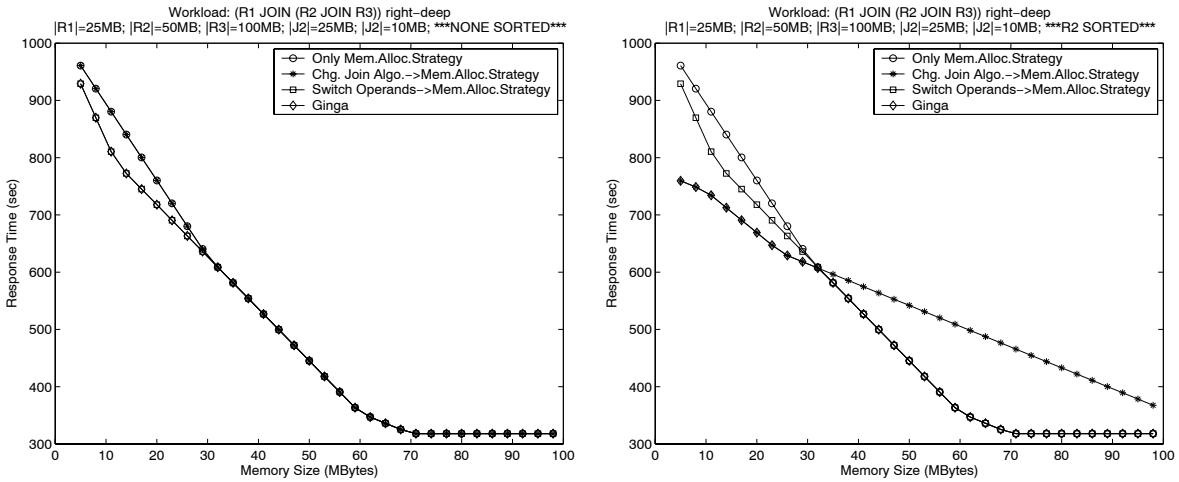


Figure 31: Performance of combining adaptation actions: (a) No operand is sorted; (b) Only R_2 is sorted.

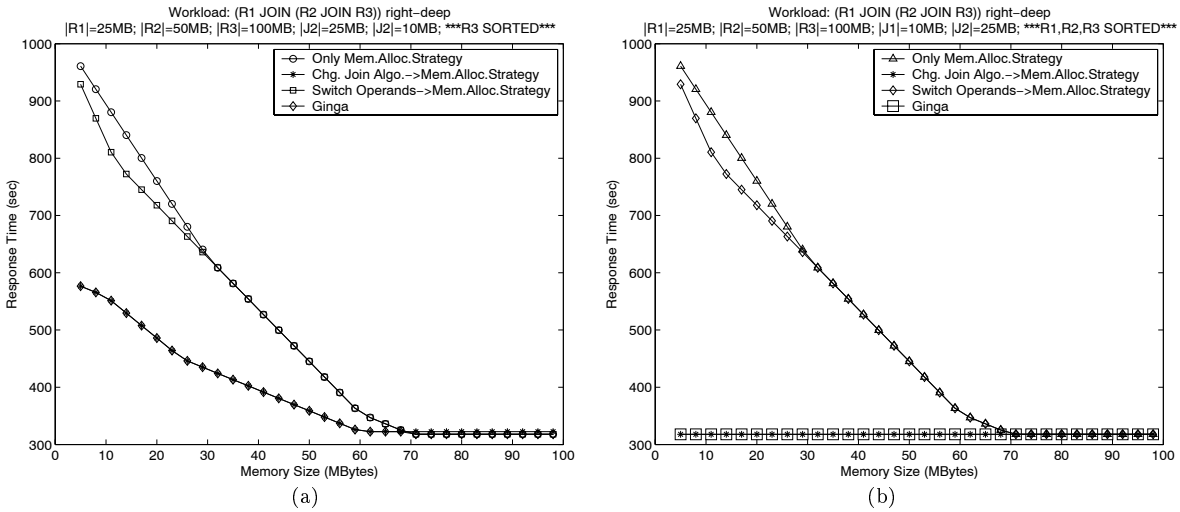


Figure 32: Performance of combining adaptation actions: (a) Only R_3 is sorted; (b) All operands are sorted.

function of the memory size allocated to this plan when Ginga navigates the adaptation paths listed in Table 8. I omit the analysis of other adaptation paths such as (*large_operands, sorted_operands, memory_size*) and (*sorted_operands, large_operands, memory_size*) since they are subsumed by the cases in Table 8.

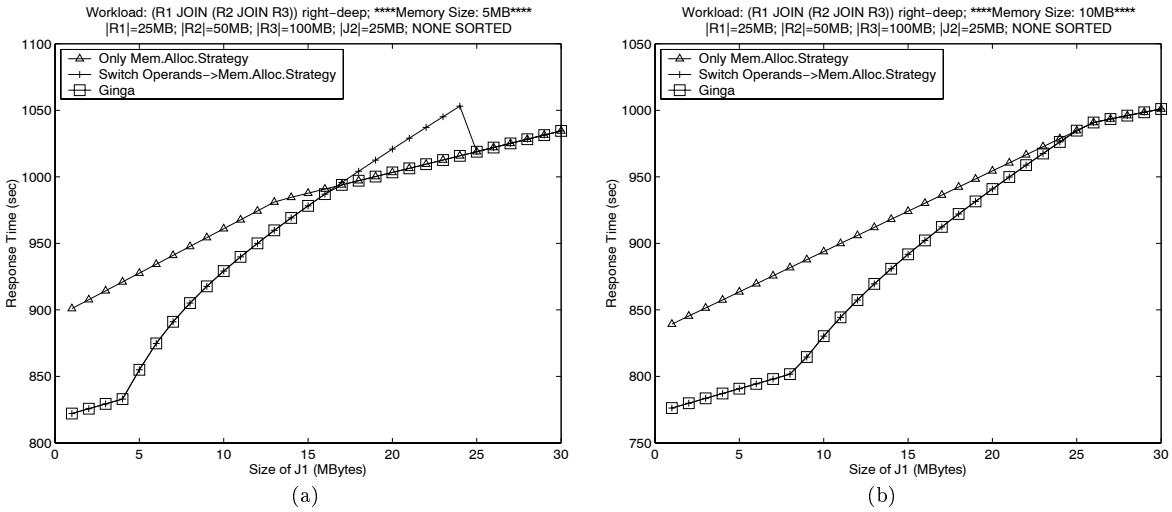


Figure 33: Performance of adaptation action Switching Operands, when no operands are sorted and memory size equal to (a) 5MBytes and (b) 10 MBytes.

Figure 31(a) shows the benefits of switching the operands (Section 5.3.1.2) and called the HJ-Improved case in the figures in Section 5.4.1. (I include this case here to simplify the explanation of figure 31(b).) By switching the operands alone, Ginga can improve the response time modestly up to memory size of 30MBytes, when both operands fit in main memory and the response time of switching versus not switching the operands becomes the same. Figure 31(b) shows the effects of combining adaptive memory allocation with operand switching and changing to sort-merge join when one (or both) of the operands is sorted. As expected, if the sorted relation has a small size (e.g., R_1), the benefits are also small (not shown). When a larger relation (R_2) is sorted, Figure 31(b) shows an improvement of up to 20% compared to the reference line (adaptive memory allocation only). With an even larger relation (R_3) sorted, Figure 32(a) shows an improvement of up to 42% over the response time of the reference line. When all operands are sorted, Ginga shows an improvement in

the response time of up to 70% for P_0 (Figure 32(b)), since the main memory requirements are minimized. The benefits of adaptation action "Changing Join Algorithms" (Adaptation Path 2) are maximized when the last operator (op_n) of a segment s is switched to SMJ. By changing op_n to SMJ, all join operators in s that depend on the result of op_n (i.e., all remaining operators $op_i \in s$, $1 \leq i < n$) will also change to SMJ. Consequently, all SMJ join operators in s will have at least one operand sorted. As demonstrated in Section 5.4.1, with this configuration, SMJ outperforms HJ in most situations.

I now further analyze the benefits and trade-offs of adaptation action "Switching Hash Join Operands" (Adaptation Path 3) using the experiment results reported in Figure 33. The graphs in this figure show the response time of executing P_0 as a function of result size of J_1 using two different memory sizes, namely, 5MBytes and 10MBytes. The graph shows an improvement of up to 10%, when $|J_1| < |R_1|$. As $|J_1|$ becomes larger than $|R_1|$ (=25MB), Ginga will no longer switch the operands, and the response time becomes the same as the reference line (memory allocation only).

In Figure 33(a), the reference line (higher curve) has one inflection point, dividing it into two segments. In the first segment (up to $|J_1|$ =13MB), Ginga chooses AlwaysMax as the memory allocation strategy. In this strategy, if each operator in the segment cannot receive their maximum required memory, the segment is cut, and the intermediate result between the two sub-segments must be cached into disk. AlwaysMax yields better performance than Max2Min for small intermediate result size. However, as the size of the intermediate result grows large, Max2Min becomes a better choice. In the second segment of the reference curve, Ginga uses Max2Min memory allocation strategy. Similar observations are applied to the graph shown in Figure 33(b).

In contrast, the lower curve for Adaptation Path 3 in Figure 33(a) has two inflection points, dividing it into 3 segments. In the first segment (up to 5MB), the hash table for the left operand of J_2, which now is J_1, can fit in main memory. In the second segment (between 5MB and 25MB), hash table for J_1 no longer fits in main memory, requiring increasingly I/O operations for execution J_2. In the third segment, because $|J_1| \geq |R_1|$, Ginga no longer switches the operands, and Adaptation Path 3 and reference line become the same. Similar

observations apply to Figure 33(b).

Figure 33(a) shows an interesting situation when $17MB \leq |J_1| < 25MB$, where switching operands becomes worse than only selecting memory allocation strategy. The reason for that is the following. When Ginga switches the operands of J_2, query P_0 is transformed in the left-deep tree, which results in two segments, with a single operator each. Consequently, each operator will receive all memory available for its execution, and if the intermediate result does not fit in main memory, it will have to be cached into disk. For $5MB \leq |J_1| < 17MB$, the extra I/O due to caching $|J_1|$ is amortized by the improvement of switching the operands of J_2. However, when $|J_1| \geq 17MB$, the benefits of switching operands no longer pays off if I compare it with the reference line. At this point, the caching cost can no longer be amortized by the improvements for J_2, and the response time for Adaptation Path 3 becomes worse than the reference line. In Figure 33(b) this situation is not observed due to two reasons. First, with larger memory size (=10MB), J_1 and J_2 have better performance. Second, given the improved performance of the hash joins, the cost of caching $|J_1|$ can always be amortized by the improvement of switching the operands of J_2.

5.5 Summary of Adaptation to Memory Constraints

With the availability of ever increasing data, more sophisticated queries will take longer time, and more likely the statically-generated initial query processing plan will become suboptimal during execution. For example, inaccurate selectivity estimates may cause the left operand of a hash join to become larger than the right operand, causing the algorithm to require more main memory than necessary or additional I/O overhead. Similarly, as main memory availability changes due to system load, memory allocation strategies may slow down query processing at different degrees due to thrashing. While individual memory adaptation methods have been studied in the past, combining them has been a non-obvious task due to the non-trivial interactions among the methods and system components.

In this chapter, I used Ginga to study the trade-offs of different adaptation strategies when query execution becomes suboptimal due to memory constraint changes (Section 5.3.1). While a number of unrelated adaptation techniques have been known to be effective in isolation, Ginga uses the concept of adaptation space to organize and combine them. In the case of memory constraints, I apply and combine several dynamic adaptation methods such as choosing sort-merge join when operands are sorted, switching the operands for hash joins when the left operand is found to be larger, and selecting memory allocation strategies, e.g., between Max2Min and AlwaysMax as appropriate.

I used Ginga's query processing simulation system to evaluate the effectiveness of these three methods in isolation and in combination (Section 5.4). My experimental results confirm that each method improves query response time by reducing memory bottlenecks. More significantly, combined query adaptation can achieve significant performance improvements (up to 70% of response time gain for representative system and workload configurations) when compared to individual solutions. These results show that it is a good idea to combine adaptive methods to address memory constraints at runtime, and Ginga offers a systematic approach to organize and combine these methods in an effective way. The Ginga approach also offers promise for the incorporation of other dynamic adaptation methods to handle runtime changes in memory constraints.

In the next chapter, I use Ginga to investigate the benefits and trade-offs of different adaptation strategies for coping with end-to-end delays during the execution of distributed queries.

CHAPTER VI

ADAPTATION TO END-TO-END DELAYS

6.1 Surviving to Unstable Connections Between Clients and Servers

During the execution of a distributed query, end-to-end delays occur possibly due to the following reasons: connections to the remote data sources experience unpredictable bandwidth and latency fluctuations, remote server is overloaded, or both network and remote server experience performance problems.

In this chapter, I investigate the trade-offs and benefits of using the following two adaptation actions for coping with end-to-end delays: *concurrent caching* and *keeping download alive*. Concurrent caching suggests that Ginga executes other parts of the query while waiting for the slow data to be delivered. With keeping download alive, for the sources known to be problematic with respect to the download process, Ginga contacts their mirror sites in parallel. These adaptation techniques are organized into an adaptation space, which provides a uniformed way for representing and viewing a collection of alternative adaptations to end-to-end delays for a given query execution plan.

The rest of the chapter is organized as follows: Section 6.2 first discusses how Ginga constructs the adaptation cases using the two adaptation techniques for coping with end-to-end delays. Then, also in this section, I describe how to organize these techniques into an adaptation space G, and how Ginga navigates G to get around unpredictable end-to-end delays. Section 6.3 studies the performance of these techniques for representative scenarios [43], and Section 6.4 summarizes the chapter.

6.2 Adaptation Space Generation and Exploration: End-to-End Delays

In this section, I first discuss how the adaptation cases for end-to-end delays are constructed. Then, I discuss how the adaptation space is generated by Adaptation Space Generation phase, prior to runtime, and navigated (as needed) by Adaptation Space Exploration phase, at runtime.

6.2.1 Adaptation Cases

When coping with end-to-end delays during the execution of a distributed query Q, the adaptation condition predicates of an $adaptation\_condition(P_i)$ in Q's adaptation space are defined over the transfer rate of the remote data servers involved in Q. When one of these predicates becomes invalid, Ginga needs to provide an alternative adaptation case. This new adaptation case has a new set S of data transfer rate predicates and an alternative query plan P_j optimized for S ($adaptation\_condition(P_j)$) along with the respective $adaptation\_trigger(P_i)$. The new adaptation case is represented as $adaptation\_case(P_j)$ and the *adaptation_event* that fires the transition from $adaptation\_-case(P_i)$ to $adaptation\_case(P_j)$ is described in $adaptation\_trigger(P_i)$.

A key component of $adaptation\_trigger(P_i)$ is $adaptation\_action(P_i, P_j)$ (see Section 4.3.1), which Ginga uses for generating P_j. In the next subsections, I describe two adaptation actions that Ginga can use for generating alternative query plans (and the associated adaptation case) for coping with end-to-end delays. These adaptation actions are (1) concurrent caching and (2) keeping download alive.

6.2.1.1 Adaptation Action: Concurrent Caching

During the execution of distributed query Q, if one of the remote data server involved in Q starts delivering data at a transfer rate slower than expected, Ginga can use adaptation action "Concurrent Caching" (Figure 34), which has two main steps:

Step 1: Cache independent segments from the original schedule *while* continuing to process the data retrieval through the slow network connection. I call *independent segment* a

segment from the original plan that does not depend on the input from the delayed source. When no more independent segments can be cached and the problematic connection is still slow, move to Step 2.

Step 2: Create and cache the result of new joins between the relations that were cached in Step 1.

I assume that when a caching operating is initiated, it will completely cache the respective operand or join result. The term *cache* in this scenario is similar to the concept of *materialization* [18], where the intermediate result of a query is materialized either in disk or in main memory.

ADAPTATION ACTION `adapt_action_concurrentCaching`
Input: P_i: current plan for the input query Q.
Output: P_j: plan P_i adapted.

1: **Let** $s_k =< op_1, \cdots, op_n > \in sch(P_i)$ be the segment currently been executed.
2: $P_j = \text{copyOf}(P_i)$; // `Make a copy of` P_i `to generate the adapted plan` P_j.
3:
// **Step 1:** `If there is an independent segment to be executed.`
4: **if** $(\exists s_\ell \in sch(P_j)$ **such that** $s_\ell \neq s_k \wedge s_\ell$ *is-independent-of* $s_k)$ **then**
5: $s_\ell =< cacheOperator > + s_\ell$;
6: $CachedSegments = CachedSegments \cup \{s_\ell\}$;
7: `segmentExec`(s_ℓ); // `Execute` s_ℓ `concurrently with` s_k
8:
// **Step 2:** `Check whether I can create new joins.`
9: **else if** (`sizeOf`$(CachedSegments) \geq 2$) **then**
10: **Let** $G(V, E)$ be the query graph of Q.
11: **Let** `resultOf`(s) be a function that returns the result relation of segment s, where $s \in sch(P_j)$.
12:
// `If there are two cached relations that can be joined.`
13: **if** $(\exists s_{\ell 1}, s_{\ell 2} \in CachedSegments$ **such that** (`resultOf`$(s_{\ell 1})$,`resultOf`$(s_{\ell 2})) \in E)$ **then**
14: $s_{\ell 3} =<$ *cacheOperator* $> + <$(`resultOf`$(s_{\ell 1}) \bowtie$ `resultOf`$(s_{\ell 2})>$;
15: $CachedSegments = CachedSegments - \{s_{\ell 1}, s_{\ell 2}\}$;
16: $CachedSegments = CachedSegments \cup \{s_\ell\}$;
17: `update`$(sch(P_j), s_{\ell 1}, s_{\ell 2}, s_{\ell 3})$; // `Replace` $s_{\ell 1}, s_{\ell 2}$ `with` $s_{\ell 3}$ `in` $sch(P_j)$.
18: `segmentExec`$(s_{\ell 3})$; // `Execute` $s_{\ell 3}$ `concurrently with` s_k
19: **return** P_j;

Figure 34: Adaptation Action "Concurrent Caching."

The goal of concurrent caching adaptation action is to generate an alternative query plan that *minimizes* the effects of a slow connection upon the final query response time while *maximizing* the use of the resources originally allocated to query execution. For example, while waiting for the slow data to arrive, the CPU can be allocated to process other parts of the query. I discuss the details of this adaptation action with the following example.

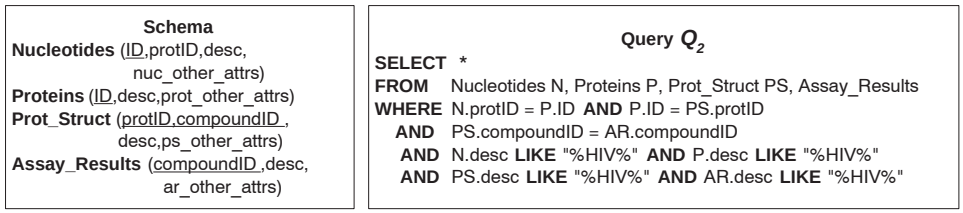
Example 15 Recall the next generation drugs discovery scenario described in Section 1.2. Assume that NGD uses Ginga to add query adaptation into their query processing system. In addition, consider that now the company has grown larger, and the assay results on how chemical compounds can affect the virus proteins are now available at remote labs. Let still assume that a pharmacologist wants to investigate new drugs to treat HIV. So now, NGD-WS will produce following SQL statement for collecting the necessary data to start the investigation:

```
Q2:  SELECT *
     FROM   Nucleotides N, Proteins P, Prot_Structure PS, Assay_Results AR
     WHERE  N.generateProtID = P.ID
     AND    P.ID = PS.ProtID
     AND    PS.compoundID = AR.compoundID
     AND    N.description LIKE "%HIV%"
     AND    P.description LIKE "%HIV%"
     AND    PS.description LIKE "%HIV%"
     AND    AR.description LIKE "%HIV%"
```

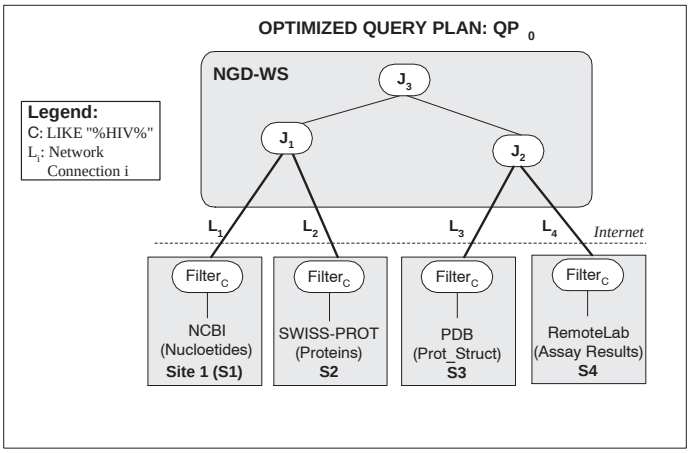
This query is first processed by the Ginga query router, which identifies the bioinformatics sources that can provide the requested information. Although there are many sources that could be used for answering this query, for simplicity, I assume that only the following sources were selected: *NCBI* for nucleotides, *SWISS-PROT* for proteins, *PDB* for protein structures, and *RemoteLab* for the results on how chemical compounds can affect the proteins identified.

Let assume that the expected transfer rate for the network connection L_i is $Rate(L_i) = 1Mbps$, for $1 \leq i \leq 4$. Ginga will adapt the execution of Q whenever $Rate(L_i)$ drops below $w \times 1Mbps$, $0 < w \leq 1$, which will cause slow delivery.

Now, suppose that as Ginga initiates the execution of query plan QP_0, slow delivery is detected on connection L_1. In order to cope with this end-to-end delay, Ginga fires the concurrent caching adaptation action (*cc_adaptAction*(QP_0)), which returns plan QP_{11}, which concurrently caches the the search results from *SWISS-PROT* (*cache(SWISS-PROT)*). If Ginga finishes processing *NCBI* search results while concurrently running *cache(SWISS-PROT)*, then no further adaptation is necessary. However, if the slow delivery on L_1 delays the processing of *NCBI* results beyond the completion of *cache(SWISS-PROT)*, then Ginga fires *cc_adaptAction*(QP_{11}), which returns QP_{12}. Plan QP_{12} (Figure 36) starts



(a) Global schema and query Q_2



(b) Optimized Plan QP 0

Figure 35: Query plan for collecting specific information on HIV.

concurrently the caching of join between the search results from *PDB* and *RemoteLab* ($cache(join(PDB, RemoteLab))$). The goal of both QP_{11} and QP_{12} is to change the retrieval and processing order of sources, so the fast sources are processed concurrently with the delayed source. At any point during the query execution, there is only one plan that is being used. When Ginga changes from plan QP_0 to plan QP_{11} or from QP_{11} to QP_{12}, the operator schedule is changed accordingly, as indicated in Figure 34.

When there are no more *independent* segments from the original plan to be executed and cached and Ginga still has not finished processing *NCBI* results, *cc_adaptAction* will start creating and caching new joins (Step 2). In this example, the new query plan QP_{13} will create and cache the result of a new join with the relation and join results that QP_{12} cached. Ginga creates new joins by following the query graph associated with QP_0. Only those relations that can be joined (i.e., there is an edge in the query graph connecting them) are considered during the creation of new joins. This way, Ginga avoids the generation of new joins that result in Cartesian products.

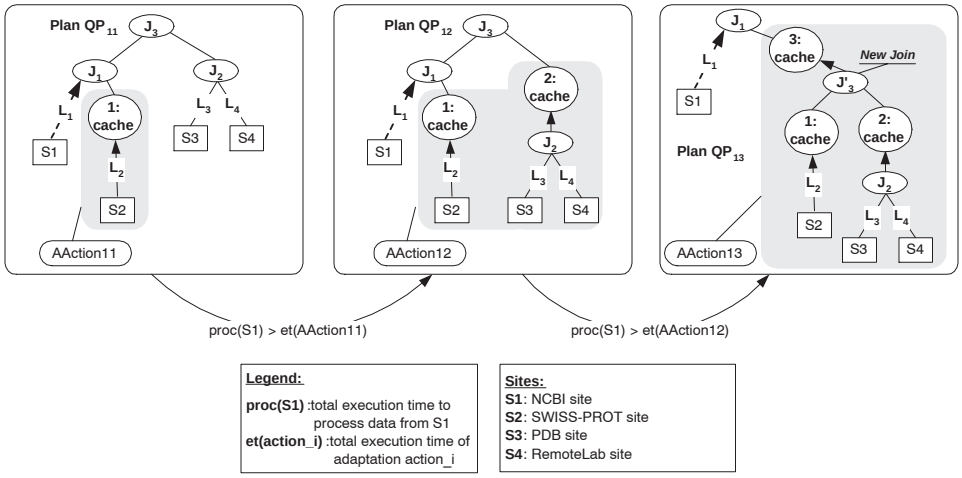


Figure 36: Alternative query plans for coping with end-to-end delays on connection L_1.

The eager execution of these expensive new join operators is a departure from the initial optimized plan QP_0. This means that QP_{13} was considered sub-optimal when compared to QP_0 originally. Therefore, QP_{13} may or may not finish before $QP_{1i}, i < 3$, depending on the completion time of processing *NCBI* results. An alternative solution is to use a detailed dynamic adaptation query plan generation to take these factors into account. This is similar to query re-optimization [29, 59].

The scheduling of QP_{13} results in changing the right operand of the originally scheduled join between *NCBI* and *SWISS-PROT* results. In my implementation of Ginga, this scenario is possible because I assume that all joins operators are hash-based [52] and the execution of a query plan follows the model presented in 4.3.2.

Figure 36 summarizes the adaptation actions used for coping with end-to-end delays detected on connection L_1. For each action, the caching operations are numbered in the order they should the executed. Observe that these new caching operations are scheduled in a way that allows a smooth transition from one query plan to another at runtime.

6.2.1.2 Adaptation Action: Keeping Download Alive

In the previous section, I described *concurrent caching* adaptation action, which assumes that the data will be constantly delivered, but at a slower rate. In this section, I describe *keeping download alive* (KDA) adaptation action, which assumes that the data server is

likely to unexpectedly stop sending data either due to a local software or hardware failure or due to a broken network connection.

There are at least two approaches for solving this problem. The first approach is to close the connection to the failed server and contact a mirror site (if any) to download the missing data. The second approach is to use query relaxation. The input query is re-written so that it returns partial answer to the user [3].

The implementation of both approaches involves solving the following important and difficult problem: how to determine when a remote server is indeed not responding. If I mistakenly close the connection to the server, which was momentarily out, using the first approach, I will be unnecessarily returning an imprecise answer to the input query. By using the second approach, the final response time may be considerably affected because I will have to pay an extra connection cost to the mirror site. Furthermore, the second approach incurs another problem, which is how to avoid (or minimize) the download of redundant data.

In this dissertation, I focus on second the approach, contacting alternative servers. Incorporating the first approach, query relaxation, to Ginga is part of my future research agenda.

The kernel of my approach to contacting alternative servers is an operator that I call Data Collector (DC). DC has two operands, where each operand represents a remote data server. Briefly, DC collects and combines the data from both data servers and deliver the collected data to the calling process. When DC detects that at least one of the servers is not responding, or not delivering data at the rate that it was expected, DC generates the adaptation event *DC_frequent_timeout* and adaptation action "keeping download alive" (KDA) is executed.

Adaptation action KDA is based on the assumption that for each remote base relation R involved in a distributed query Q there will be at least two remote data servers that can provide R. When the data collection slows down, KDA will close the connection to the slowest (or not responding) server, and establish another connection to an alternative site.

In the next paragraphs, I describe in more details the data collector operator and the

adaptation action KDA.

Data Collector Operator The data collector has two operands: the left data stream (LDS) and the right data stream (RDS). DC receives the data from these two operands, union the data, and the result to the caller. To avoid duplicates, LDS delivers the data in ascending order and RDS in descending order. This way, I can minimize the amount of duplicated data.

Figure 37 depicts a possible execution of DC where remote relation A is downloaded from two data servers. I assume that both servers are able to sort A accordingly. DC will consume the data from the page buffer. When the buffer becomes empty, DC waits for a *timeout* interval. After n failed trials of consuming data from the buffer (i.e., *timeout* expires n times), DC checks on the Mean Time Between Timeouts (MTBT). If MTBT is equal to timeout, it means that timeouts has become too frequent, and I should adapt the execution of the query. Ginga will then execute adaptation action KDA. MTBT is refreshed each time a timeout is detected.

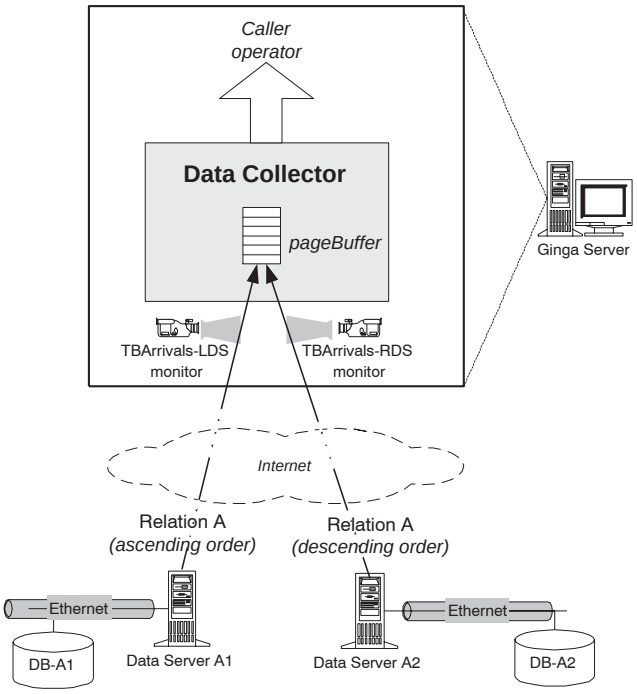


Figure 37: Data Collector operator.

```
CLASS DEFINITION DataCollector

   // Attributes
   networkConnection leftServer;
   networkConnection rightServer;
   boolean leftServer_alive; // TRUE after receiving the first page from LEFT server.
   boolean rightServer_alive; // TRUE after receiving the first page from RIGHT server.
   int totalLeftPages; // Total number of pages delivered by the LEFT server.
   int totalRightPages; // Total number of pages delivered by the RIGHT server.
   queueOfPages pageBuffer; // Pages received from the servers are stored in this queue.
   int numOfExpectedPages; // Total number of expected pages to be downloaded.
   int currentNumOfPagesRec; // Current number of pages received so far.
   double timeOut; // Timeout value.
   double MTBT; // Mean Time Between Timeouts parameter.
   double denMTBT; // Used for refreshing MTBT.
   double forgetFactor; // Forgetting factor [0..1].
   double t_lastTimout; // Time when occurred the last timeout.

   // Methods
   void open();
   page next();
   void close();
```

Figure 38: Data Collector Operator class definition.

As described in Section 4.3.2, Ginga processes queries according to the Segmented Execution Model, which in turn in based on the iterator model [21]. In the iterator model, each operator is considered as an *iterator* that supports three methods: `open`, `next`, and `close`. In general, `open` prepares the operator for producing an item, `next` produces an item, and `close` performs final clean up. In the next paragraphs, I describe these three methods for the data collector operator. The class definition of the this operator is depicted in Figure 38.

The `open` method for DC is depicted in Figure 39. In this method, DC first connects to both operands and establishes the *timeout* value as the maximum expected value for the *time between arrivals* (TBArrivals) of each page delivered by the two servers. Then, DC starts the monitor for the TBArrival for each of the servers. Each monitor will maintain the average of TBArrival for the associated servers. The information maintained by these monitors is used by the KDA adaptation action when deciding which server should be replaced. Finally, DC initializes the *mean time between timeouts* (MTBT) parameter, used for deciding when to generate the adaptation event *DC_frequent_timeouts*.

The `next` method of DC is depicted in Figure 40. The pages received from the operands are stored in the local *pageBuffer*. When DC receives a `next` call, if there are pages in

```
METHOD DataCollector::open
Input: ff: forget factor that this DC should use; r: data request to be sent to the operands;
Output: void;
    // Contact left and rignt servers.
 1: connect(leftServer);
 2: leftRequest = r;
 3: sendRequest(leftRequest,leftServer,ascending_order);
 4: connect(rightServer);
 5: rightRequest = r;
 6: sendRequest(rightRequest,rightServer,descending_order);
 7:
    // Set timeout to 10% higher than the maximum expected Time Between Arrival (TBA)
    // of pages from each server.
 8: timeOut = (1.10) * max(expectedTBA(leftServer),expectedTBA(rightServer));
 9:
    // Start the monitors for the TBA from each page.
    // Each monitor mantains the mean TBA for the associated server.
10: startMonitorTBArrivals(leftServer);
11: startMonitorTBArrivals(rightServer);
12: leftServer_alive = FALSE;
13: rightServer_alive = FALSE;
14: numOfExpectedPages = leftServer.expectedSize;
15: currentNumOfPagesRec = 0;
16: totalLeftPages = 0;
17: totalRightPages = 0;
18:
    // Set the forgetting factor.
19: forgetFactor = ff;
20:
    // Initialize Mean Time Between Timeouts (MTBT) parameter.
21: MTBT = 0;
22: denMTBT = 0;
23: t_lastTimout = 0;
24: return;
```

Figure 39: Data collector OPEN method.

pageBuffer, then DC returns one page of this buffer, which in turn is updated accordingly. However, if there are no pages available, DC waits for a `timeOut`. After n failed trials of consuming data from the buffer, DC checks on the MTBT. If MTBT is equal to `timeOut`, it means that timeouts have become too frequent, and I should adapt the execution of the query. Ginga will then execute adaptation action KDA.

When DC receives a `next` call and all pages have been received and returned to the caller, the `NULL` value is returned. In turn, the caller invokes DC's `close` method, which closes the connections to both operands.

MTBT is a critical and important parameter because it determines *when* adaptation should take place. Every time a timeout is detected, the MTBT is refreshed using the method `refreshMTBT` depicted in Figure 41. If the time interval `t` between the last timeout `t_lastTimeout` and the current one `t_timeout` is close to `timeOut`, it means that two

```
METHOD DataCollector::next
Input: void;
Output: page: one page received from one of the remote servers;
    // If we have received all the expected pages, return NULL.
 1: if (numOfExpectedPages == currentNumOfPagesRec) then
 2:   return NULL;
 3:
    // If there are pages in pageBuffer, then return the page at the head of the queue.
 4: if (sizeOf(pageBuffer) > 0) then
 5:   currentNumOfPagesRec = currentNumOfPagesRec + 1;
 6:   page = pageBuffer.head();
 7:
      // Record which server has delivered the page being returned.
 8:   if (belongsTo(page) == LEFT_SERVER) then
 9:     totalLeftPages = totalLeftPages + 1;
10:   else
11:     totalRightPages = totalRightPages + 1;
12:   return page;
13: else
14:   // There is no page available and Ginga needs to wait for the next page to arrive.
15:   numOfTrials = 0;
16:   while (1) do
17:     wait(timeOut); // Wait for a page to arrive from one of the servers.
18:     numOfTrials = numOfTrials + 1;
        // If no pages has been received after timeout...
19:     if (sizeOf(pageBuffer) == 0) then
20:       // If we have tried 10 times or if both servers are alive but not responding
21:       if (numOfTrials > 10 or (leftServer_alive and rightServer_alive)) then
22:         // Initialize or refresh the MTBT parameter.
23:         if (t_lastTimout == 0) then
24:           t_lastTimout = clock; // Current clock;
25:           MTBT = 0;
26:         else
27:           refreshMTBT(clock);
28:
            // If we are timing out too frequently, then we should adapt this operator.
29:         if (MTBT > 0 and MTBT == timeOut) then
30:           ADAPT(*this); // Request adaptation for this operator.
              // Re-initialize the MTBT.
31:           MTBT = 0;
32:           denMTBT = 0;
33:           t_lastTimout = 0;
34:           numOfTrials = 0;
35:
36:       else
37:         // Return the page that we have received within timeOut.
38:         page = pageBuffer.head();
39:         currentNumOfPagesRec = currentNumOfPagesRec + 1;
40:
            // Record which server has delivered the page being returned.
41:         if (belongsTo(page) == LEFT_SERVER) then
42:           totalLeftPages = totalLeftPages + 1;
43:           if (leftServer_alive == FALSE) then
44:             leftServer_alive == TRUE; // LEFT server is ALIVE.
45:         else
46:           totalRightPages = totalRightPages + 1;
47:           if (rightServer_alive == FALSE) then
48:             rightServer_alive == TRUE; // RIGHT server is ALIVE.
49:
50:         return page;
      endWhile // For receiving one page from one of the servers.
    endIf
51:
```

Figure 40: Data collector NEXT method.

consecutive timeouts have occurred within a short period of time. In this case, I assume that the runtime environment is becoming unstable, and I should forget some of the "timeout history" embedded in MTBT. Without forgetting some of MTBT history, consecutives timeouts detected after a long period without timeouts may never be detected by the DC. Consequently, Ginga will miss potential adaptation opportunities for improving the query execution.

I use the parameter `forgetFactor` to determine the percentage of MTBT that I need to forget when refreshing the MTBT in case consecutive timeouts are detected. For `forgetFactor = 0`, I always remember everything, whereas for `forgetFactor = 1`, I forget everything. As I demonstrate in my experiments (Section 6.3.2), depending on how unstable is the runtime environment, a particular `forgetFactor` will be the best solution.

METHOD `DataCollector::refreshMTBT`
Input: `t_timeout`: time when the timput occurred.
Output: `void`;

```
    // If the difference between t_timeout and t_lastTimout is close to
    // timeOut, then we should forget some of the MTBT history, using the forgetting factor.
 1: t = t_timeout - t_lastTimout;
 2: if (timeOut ≤ t ≤ timeOut * 1.10) then
 3:   MTBT = (1 - forgetFactor) * MTBT;
 4:
    // Refresh MTBT
 5: denMTBT = denMTBT + 1;
 6: MTBT = [(denMTBT − 1)/denMTBT * MTBT] + [(t_timeout − t_lastTimout)/denMTBT];
 7: t_lastTimout = t_timeout;
 8: return;
```

Figure 41: Function for refreshing the MTBT.

Keeping Download Alive The adaptation action keeping download alive (KDA) is depicted in Figure 42. KDA first checks whether there are alternative servers to be contacted for data collector DC_ℓ (Line 5). If no alternative server is available, then no adaptation takes place. However, if there is at least one alternative server srv, KDA will close the connection to the server with the largest mean time between page arrivals (`AVG_TBArrivals`) and establish the connection with srv.

An important method call in KDA is the `constructNewRequest(requestSrv,srv_slow)`. This method, invoked in Line 8 and Line 15 (see Figure 42) takes as input the request

```
ADAPTATION ACTION adapt_action_KDA
Input: P_i: current plan for the input query Q.
Output: P_j: plan P_i adapted.
 1: Let s_k ∈ sch(P_i) be the segment currently being executed.
 2: Let DC_ℓ ∈ s_k be data collector currently being executed and have generated a DC_frequent_timeout adaptation
    event.
 3: P_j = copyOf(P_i); // Make a copy of P_i to generate the adapted plan P_j.
 4:
    // If there are alternative server to contact
    // (Obs.:  If no alternative server is available, then no adaptation takes place.
 5: if (∃ server srv to be contacted for DC_ℓ) then
 6:
      // If the average time between arrivals (TBArrivals) for the left server
      // is larger than the TBArrivals for the right server, then...
 7:   if (AVG_TBArrivals(DC_ℓ.leftServer) ≥ AVG_TBArrivals(DC_ℓ.rightServer)) then
 8:     r = constructNewRequest(leftRequest,leftServer);
 9:     leftRequest = r; // Store the current request for the left server.
10:     DC_ℓ.leftServer = srv;
11:     reconnectServer(DC_ℓ.leftServer); // Contact an alternative server for DC_ℓ.leftServer
12:     sendRequest(leftRequest,leftServer,ascending_order);
13:   else
14:     // Otherwise, contact another server for DC_ℓ.rightServer
15:     r = constructNewRequest(rightRequest,rightServer);
16:     rightRequest = r; // Store the current request for the right server.
17:     DC_ℓ.rightServer = srv;
18:     reconnectServer(DC_ℓ.rightServer);
19:     sendRequest(rightRequest,rightServer,descending_order);
20: return P_j;
```

Figure 42: Adaptation Action "Keeping Download Alive."

`requestSrv` sent to the server `srv_slow`, determines how much `srv_slow` has already delivered, and constructs a new request for returning only the missing data. In my implementation of `constructNewRequest`, I assume that as DC consumes pages from the `pageBuffer` in Line 6 and Line 40 of Figure 40, I am able to determine from which server the page was originated. I assume that each server attaches this information to each page it delivers to DC.

Once KDA has completed the adaptation process, the control is returned to data collector DC_ℓ, which then proceeds with the download process.

6.2.2 Adaptation Space

So far, I have described two adaptation actions for generating adaptation cases to get around end-to-end delays during the execution of a distributed query plan P_i. In this section, I describe how to organize the cases generated by these actions into an adaptation space G, and how Ginga navigates G for coping with unexpected end-to-end delays.

Ginga will adapt the execution of P_i when at least one of the end-to-end delay predicates

Table 9: End-to-End Delay Predicates for plan P_i, where $s_k \in sch(P_i)$ is the current segment being executed.

Predicate Name	Description
join_operand_Predicate	$\forall$ join operator $j_i \in s_k$, j_i.`numOfTrials` < `trialsThreshold`;
dc_mtbt_Predicate	$\forall$ data collector operator $dc_i \in s_k$, dc_i.`MTBT` > dc_i.`timeOut` **or** dc_i.`MTBT` == 0;

Table 10: Adaptation events and adaptation actions for coping with end-to-end delays.

adaptation_event(P_i)	Invalidated Predicate	adaptation_action(P_i, P_j)
JOIN_frequent_timeout	join_operand_Predicate	adapt_action_concurrentCaching
DC_frequent_timeout	dc_mtbt_Predicate	adapt_action_KDA

listed in Table 9 is invalidated. The associated adaptation events and actions to be fired are summarized in Table 10. For example, if the predicate join_operand_Predicate is invalidated, then event *JOIN_frequent_timeout* is fired and `adapt_action_concurrentCaching` is executed. Based on these events and actions, Ginga generates the adaptation space for P_i shown in Figure 43.

The algorithm for navigating the adaptation for coping with end-to-end delays is depicted in Figure 44. The order in which I process the events is not relevant, and both adaptation events have the same "priority." The adaptation decisions made by adapt_action_concurrentCaching does not invalidate any of the changes that adapt_action_KDA makes to the query plan.

It is important to observe that the decision for firing the adaptation process using KDA adaptation action is different from the decision on when to adapt using the concurrent caching adaptation action. The former is based on MTBT while the latter is based on number of timeouts observed.

6.3 Performance Analysis

For all experiments reported in this chapter, I use a simulator (based on the CSIM toolkit) that models a client-server system with one client running Ginga and n remote data servers. All base relations involved in the queries submitted to Ginga are assumed to be available

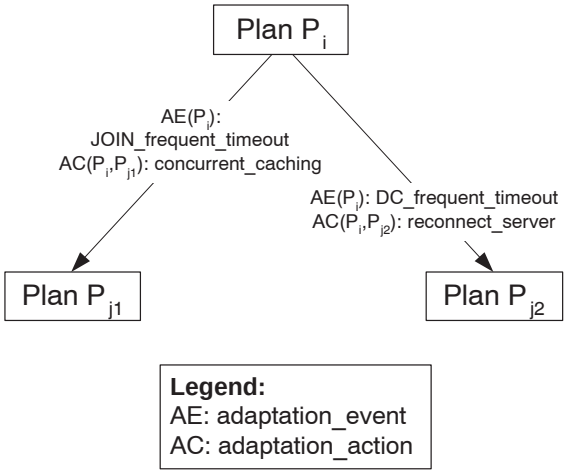


Figure 43: Adaptation Space for coping with end to end delays.

ADAPTATION SPACE `MgtEndToEndDelay`
Input: adapt_space(P_i): adaptation space for P_i.
Output: P_j: plan P_i adapted.
Require: at least one of memory constraint predicates in adaptation_condition(P_i) is invalidate.

1: P_j = copyOf(P_i); // `Make a copy of` P_i `to generate the adapted plan` P_j.
2:
// **Adaptation Path:** (*JOIN_frequent_timeout*)
3: **if** (join_operand_Predicate(P_j) == FALSE) **then**
4: `adapt_action_concurrentCaching`(P_j, P_{j1});
5: $P_j = P_{j1}$;
6:
// **Adaptation Path:** (*DC_frequent_timeout*)
7: **if** (dc_mtbt_Predicate(P_j) == FALSE) **then**
8: `adapt_action_concurrentCaching`(P_j, P_{j2});
9: $P_j = P_{j2}$;
10:
11: **return** P_j;

Figure 44: Adaptation Space for Managing End-to-End Delays.

at the remote servers. The size of the relations reflect the size of the answers provided by web services such as NCBI-Entrez discussed in Section 1.2.

Table 11 lists the classical parameters [10, 52, 61, 4] used in configuring the simulator. Client and server machines have the same hardware configuration. Disks are modeled as FIFO queues. I model end-to-end delays in terms of degradations to network bandwidth. Network connections are independent of each other, in the sense that the failure of a connection does not affect the others. The network transfer rates and their respective degradations considered in my experiments represent those typically observed in the Internet environment.

Table 11: Simulation Parameters

Parameter	Value	Description
Speed	100	CPU speed (MIPS)
AvgSeekTime	8.9	average disk seek time (msecs)
AvgRotLatency	5.5	average disk rotational latency (msecs)
TranferRate	100	disk transfer rate (MBytes/sec)
DskPageSize	8192	disk page size (bytes)
MemorySize	$10 \cdots 100$	memory size (MBytes)
DiskIO	5000	instructions to start a disk I/O
Move	2	instructions to move 4 bytes
Comp	4	instructions to compare keys
Hash	25	instructions to hash a key
Swap	100	instructions to swap two tuples
F	1.2	incremental factor for hash join

The experiments are divided into three groups: The first group (Section 6.3.1) studies the benefits and trade-offs of concurrent caching (CC) adaptation action. The second group (Section 5.4.2) investigates the performance characteristics of keeping download alive (KDA) adaptation action. The third group (Section 5.4.3) studies the combination of CC and KDA.

6.3.1 Concurrent Caching

For the experiments described in this section, it is assumed that during Adaptation Space Generation phase, Ginga builds the initial query plan P_0 depicted in Figure 45, where each remote source is expected to return 10,000 objects of 200 bytes each. I use a bushy tree because it allows me to fully investigate all aspects of Ginga adaptation engine. Nevertheless, similar results were obtained for left-deep and right-deep query trees as reported in Section 6.3.1.1.

Joins are assumed to be hash-based and relations can be equijoined in any order, always on the same attribute. Two memory sizes were considered for executing the query at the Ginga site: *limited* and *unlimited*. With *limited* memory, all operators are executed with the minimum memory requirement, while with *unlimited* memory the operators are executed at their optimal performance. For example, in a hash-join between relations R and S the minimum memory requirement is $\sqrt[2]{|R| \times F}$, where $|R| < |S|$ and F represents the overhead factor due to the hash structure [52]. With minimum memory allocation, executing a hash-join implies partitioning the relations, which results on I/O costs. With *unlimited* memory, hash-join has optimal performance because $|R| \times F$ fits in main memory, and no I/O cost

is incurred during the execution.

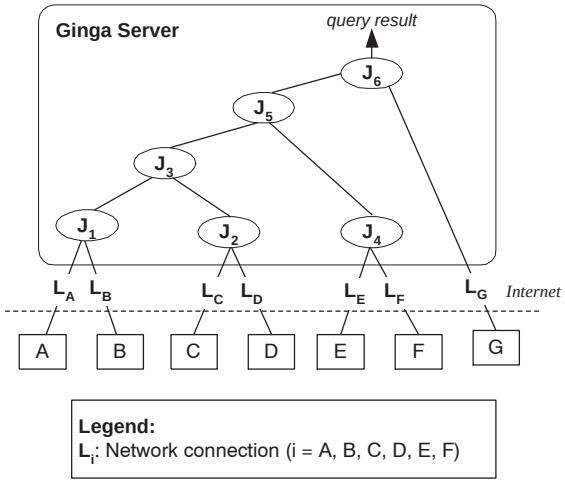


Figure 45: Initial optimized plan P_0 used for the experiments on concurrent caching.

The remainder of this subsection is organized as follows. I first investigate the effectiveness of Ginga while coping with end-to-end delays when using concurrent caching. Then, I introduce errors to the selectivity of the newly created joins (Step 2 in Figure 34) to explore the performance boundaries of Ginga approach.

6.3.1.1 Effectiveness Analysis of Ginga Adaptation

I first present the adaptation performance results for the commonly explored case when delays are observed for the network connection linking the left-most remote source of plan P_0 (i.e., connection L_A from Figure 45) to the query site. Then, I show that similar results are observed when delays are detected for the other connections. Finally, I present performance numbers when end-to-end delays are detected in multiple network connections.

Delay in a Single Connection The graph in Figure 46(a) shows the response time of Ginga's adapted plans when slow delivery is *first* detected in three different situations while retrieving data from Source A: (1) at the beginning, (2) after 50%, and (3) after 75% of pages were received without delay. Once the delay is detected, I assume that the data is slowly delivered under a degraded rate $Rate(L_A) = 128Kbps$. The duration of the slow delivery is represented by the x-axis. The response time of the original plan (without adaptation) is

represented by the dotted line. For the experimental results reported in this section, it is assumed that all intermediate results have 5,000 objects and the query is executed under *limited* memory size.

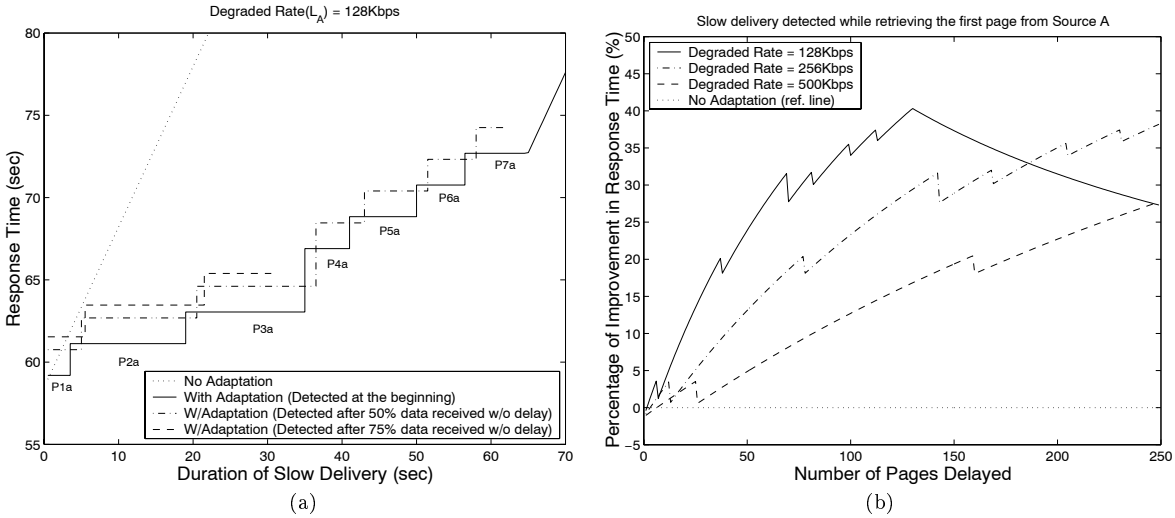


Figure 46: (a) Limited Memory ; Normal Rate = 5Mbps ; Degraded $Rate(L_A) = 128Kbps$. (b) Percentage of improvement in response time; Limited Memory; Degraded connection: L_A ; Slow delivery detected while retrieving the *first* page from remote Source A.

As it can be observed from the graph from Figure 46(a), the benefits of adapting the query execution in the presence of slow delivery are significant. This fact is illustrated by the "staircase" lines for the case when the adaptation process is used. The horizontal length of each step in each adaptation line represents the amount of slow delivery that each alternative query plan can absorb after Ginga schedules it. The height of each step shows the response time of the adapted query execution if $Rate(L_A)$ resumes to its expected data transfer rate of $5Mbps$ after a period of observed slow delivery. The label in each step indicates the query plan being used. A description of all alternative plans are depicted in Figure 47.

For short duration of slow delivery (less than 4 seconds), Ginga can practically maintain the same query response time as the initial plan when executed without delay. This can be observed from the solid line in Figure 46(a). However, this scenario does not occur for the

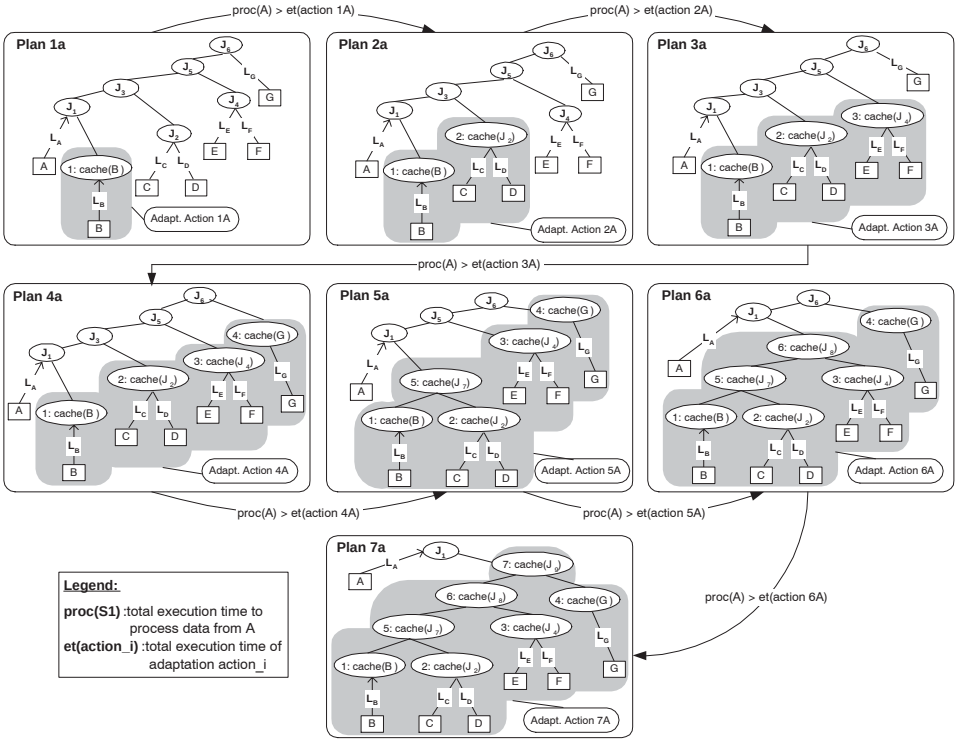


Figure 47: Adaptation actions for coping with delays on network connection L_A.

'50%'- and '75%'-lines. In fact, in these two cases, for slow delivery with very short duration the response time with adaptation is worse than the response time with no adaptation. The investigation of mechanisms to avoid this anomaly is part of my future research agenda.

For slow delivery duration longer than 4 seconds, Ginga's query adaptation is beneficial even when end-to-end delays are detected after 50% and 75% of pages were downloaded without delay. The longer the duration of slow delivery, the more processing can be done concurrent while downloading the slow data. However, I observe that there are a few moments when using adaptation can result in a query response time that is *almost* as bad as no adaptation. One of these moments can be observed at 5 seconds with the '50%'-line.

The improvements provided by Ginga adaptation process are limited by the number of alternative query plans. When it is no longer possible to fully process the slow data while executing the sequence of caching operations from query plan P_{7a}, there is nothing left to be done other than process the slow data as it arrives. This explains why after 65 seconds of slow delivery the solid-adaptation line becomes parallel to the no adaptation line. However,

Ginga adaptation is still beneficial even though it can no longer absorb all the slow delivery.

The number of needed alternative query plans is reduced according to when the slow delivery is first detected and how seriously the $Rate(L_A)$ is degraded. From the '75%'-line in the graph of Figure 46(a), it can clearly be observed that Ginga adaptation process uses only up to query plan P_{3a}. This occurs because when the end-to-end delay is detected there are only a few pages left from A that need to be processed concurrently with the sequence of scheduled caching operations. In other words, less critically the $Rate(L_A)$ is degraded and later the slow delivery is detected, fewer the alternative query plans that are needed.

Figure 46(b) shows the comparative benefits of using Ginga versus no query adaptation when slow delivery is first detected at the beginning under three different degraded rates for connection L_A: $128Kbps$, $256Kbps$, and $500Kbps$. The x-axis represents the number of pages delayed, and the y-axis represents percentage of improvement in response time obtained by using Ginga. As it can be observed from the graph, in this experiment Ginga can provide up to 40% of improvement on the query response time when compared with the no adaptation case.

The spikes in each curve in Figure 46(b) represent the moments at which Ginga switches plan. When there are no more alternative plans to schedule and the number of pages delayed increases, the response time improvement due to adaptation starts to gradually decrease. This occurs because the last scheduled plan can no longer fully absorb the processing of the slow data. Line '128Kbps' represents this scenario. In contrast, for the '256Kbps' and '500Kbps' lines, Ginga never experiences a situation where the alternative query plans cannot fully absorb the slow delivery.

The results from the experiments with slow delivery detected for the other network connections of query plan P_0 (Figure 45(a)) are similar to the results described so far in this subsection.

Delay in Left-Deep and Right-Deep Trees I also ran experiments with left-deep and right-deep trees using the same number of remote sources. As shown in Figure 48, similar results were obtained. The main difference is with respect to the alternative query plan

used for these trees. Different from the bushy tree considered so far, most of alternative query plans initially scheduled for left- and right-deep trees involve the caching of single remote relation. As a consequence, the query response time improvement due to adaptation is not significant until 34 seconds of slow delivery duration for left-deep and 22 seconds for right-deep. After this moment, the caching of joins created using previously cached data is then scheduled. It also important to observe that with small duration of slow delivery, there are moments where the response time with adaptation is worse than no adaptation.

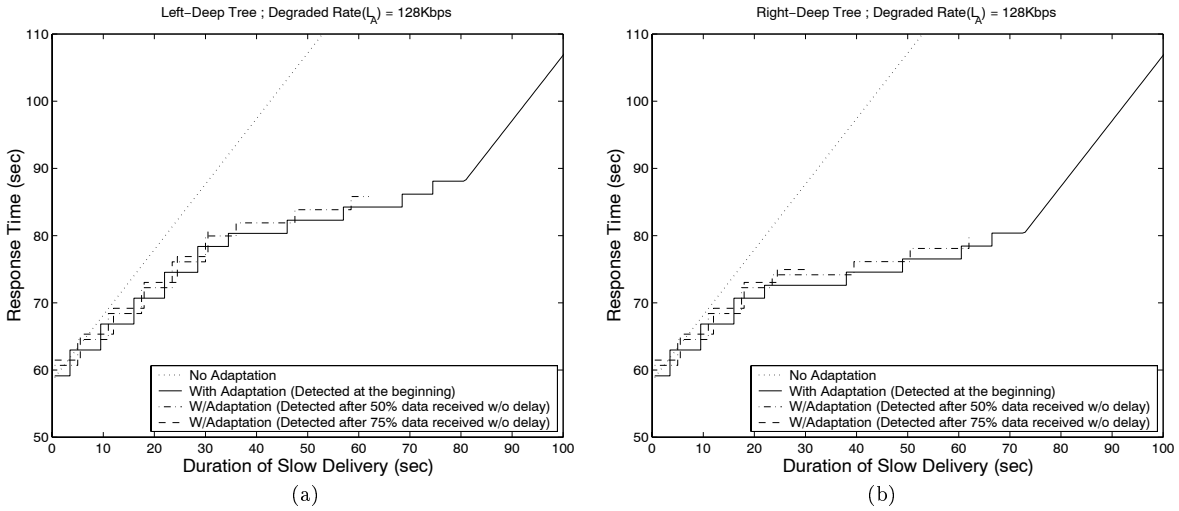


Figure 48: Performance results for (a) Left-deep and (b) Right-deep trees when the left-most remote relation is delayed ; Limited Memory; Normal Rate = 5Mbps ; Degraded $Rate = 128Kbps$.

Delay on Multiple Connections According to the adaptation process described in this chapter, upon detecting slow delivery on one network connection, Ginga schedules alternative query plans. In this case, having slow delivery over multiple remote data sources means that new end-to-end delays are detected while switching plans.

I present here a performance study for the case when slow delivery is detected for two network connections. I find this case to be representative for the generic case of multiple delayed connections.

Figure 49 summarizes the alternative query plans for coping with end-to-end delays on connections L_A and L_B. Upon detecting slow delivery on L_A, Ginga schedules the caching of Source B ($cache(B)$). However, delays on connection L_B are also detected. At this moment, Ginga suspends $cache(B)$ and switches to alternative query plan P_{1ab} which will be used to absorb the slow delivery from connections L_A and L_B. As the total duration of slow delivery increases and P_{1ab} can no longer complete the caching operation $cache(J_2)$ concurrently, Ginga switches to P_{2ab}. Similarly, as the slow delivery duration grows larger, Ginga will switch to the other alternative query plans.

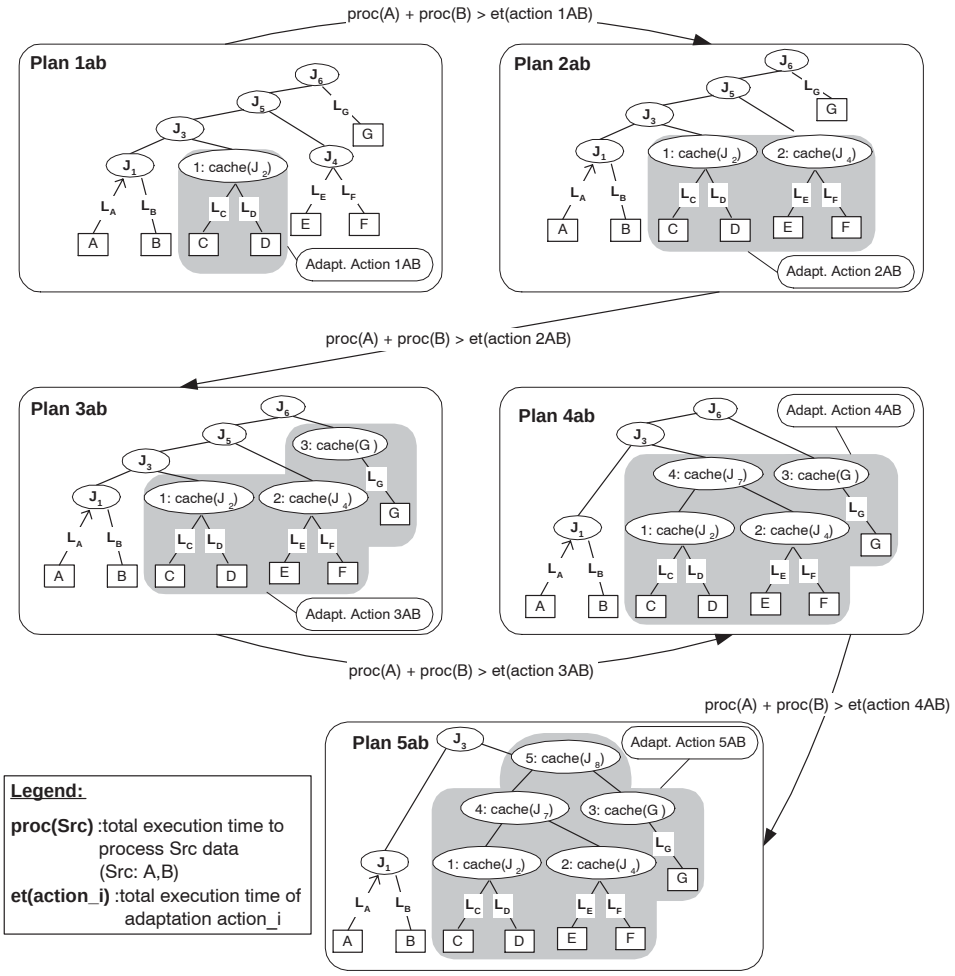


Figure 49: Summary of adaptation actions when slow delivery is detected on network connections L_A and L_B.

Figure 50(a) shows a 3-D graph when $Rate(L_A)$ and $Rate(L_B)$ are degraded to $500Kbps$

and slow delivery is detected while downloading the first page from Source A and Source B. The x-axis represents the slow delivery duration (seconds) observed for L_A, the y-axis the slow delivery for L_B (seconds), and the z-axis represents the response time (seconds).

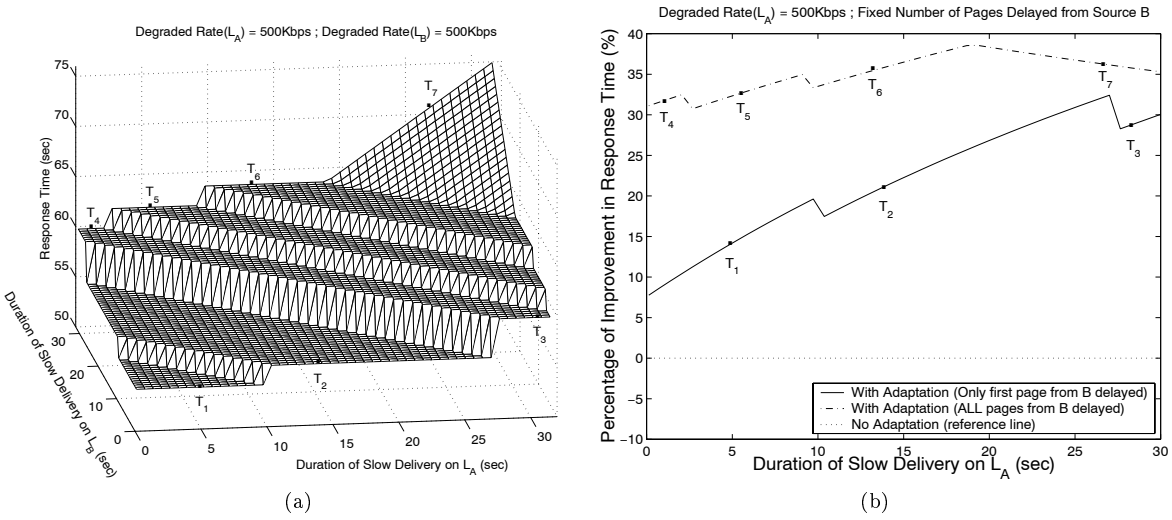


Figure 50: (a) Degraded $Rate(L_A) = 500Kbps$ and degraded $Rate(L_B) = 500Kbps$; Limited Memory ; Normal Rate = 5Mbps ; (b) Percentage of improvement in response time under two situations: (1) When only the first page from Source B is delayed and (2) when all pages from source B are delivered under degraded $Rate(L_B) = 500Kbps$. For both situations, I vary the amount of slow delivery observed for L_A, where degraded $Rate(L_A) = 500Kbps$.

Similar to the case of single connection delay, the same "staircase" effect is observed for the graph in Figure 50. This effect can be observed, for example, if I fix the number of delayed pages downloaded from Source B and vary the slow delivery duration observed for L_A. By doing this, I provide two interesting observation about the Ginga adaptation effectiveness.

The first observation is that not all alternative query plans are needed while coping with slow delivery. Consider the situation where only the first page of Source B is delayed as I vary the amount of slow delivery observed for L_A. The adaptation line for this scenario is the one that includes points T_1, T_2, and T_3 (see Figure 50(a)). Note that Ginga only uses up to alternative query plan P_{3ab}. Such phenomenon occurs because in this scenario

connections L_A and L_B do not degrade considerably. The smaller the degradation of a connection transfer rate, the fewer alternative query plans are needed to cope with slow delivery.

The second observation is that alternative query plans can be bypassed if I know that a given amount of slow data will certainly occur during query execution. Consider the case when all pages from B are delayed as I vary the slow delivery duration for L_A. The adaptation line would then include points T_4, T_5, T_6, and T_7. Observe that Ginga switches directly to alternative query plan P_{3ab}, bypassing plans P_{1ab} and P_{2ab}. This occurs because neither P_{1ab} nor P_{2ab} can fully absorb the total slow delivery observed on L_A and L_B. As the slow delivery on L_A increases, Ginga switches to alternative plan P_{4ab} and P_{5ab}. At 19 seconds of slow delivery on L_A, P_{5ab} can no longer fully absorb the total delay observed on connections L_A and L_B, and the response time with adaptation starts to increase linearly.

Figure 50(b) shows the comparative benefits of using Ginga versus no query adaptation for the two situations described above. As it can be observed, Ginga query adaptation is still a better solution than not having adaptation when end-to-end delays are detected over multiple network connections. Ginga is able to provide up to 39% of improvement on the query response time when compared with the no adaptation case.

6.3.1.2 Ginga Performance Boundaries

I now analyze Ginga's performance boundaries when significant variance is observed for the selectivity of the newly created joins used by some of the alternative query plans. I consider two situations: when the memory is *limited* and when memory is *unlimited*. With limited memory, some of the query plans require a large number of disk I/Os that can lead Ginga to worse performance than no adaptation. On the other hand, with unlimited memory, I/O costs are not incurred, but CPU power processing can be a bottleneck for Ginga's adaptation process.

Figure 51(a) shows the experimental results under *limited memory* when newly created joins used on query plans P_{5a}, P_{6a}, and P_{7a} (see Figure 47) produce 5,000; 25,000; and 50,000 objects. The '5,000'-line in the graph represents the situation studied previously. For

new joins resulting in 25,000 objects, the response time with adaptation is still acceptable. However, when compared with the '5,000'-line, the response time for 25,000 objects increases significantly (up to 32%) because for each new join, it is necessary to partition the relations due to the limited memory size. This situation is aggravated when the new joins produce 50,000 objects. In this case, there are moments when the response time with adaptation becomes worse than the response time with no adaptation. In other words, this means that waiting for $Rate(L_A)$ to resume to its expected rate is better than adaptation. Similar results were obtained for different degraded rates and having slow delivery first detected at different moments.

Figure 51(b) shows the experimental results under *unlimited memory*, which are analogous to the results in Figure 51(a). By assuming unlimited memory size, all the joins are executed in main memory eliminating I/O costs. However, note that unlimited memory still does not prevent the '5,000'-line from crossing the no-adaptation line. Intuitively, from this result, I conclude that Ginga's adaptation process is not only memory bounded, but also CPU bounded. Adapting the query execution to memory and CPU constraints is part of my ongoing research study.

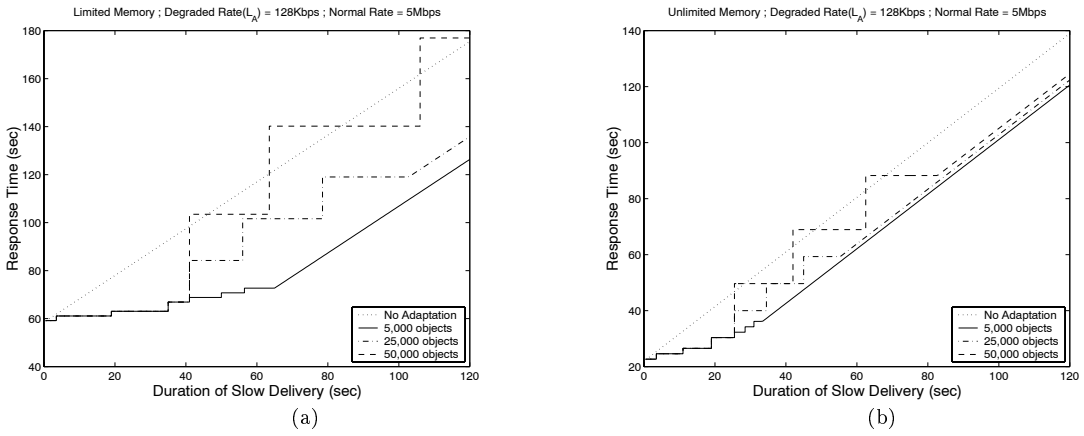


Figure 51: Ginga's performance under different result sizes for the newly created joins. (a) *Limited Memory* ; Normal Rate = 5Mbps ; Degraded $Rate(L_A) = 128Kbps$; (b) *Unlimited Memory* ; Normal Rate = 5Mbps ; Degraded $Rate(L_A) = 128Kbps$.

6.3.2 Keeping Download Alive

In this section, I analyze the performance of the data collector operator in conjunction with the "keeping download alive" (KDA) adaptation action. Similar to the experiments described in the previous section, for these experiments, I also model end-to-end delays in terms of degradations to the network transfer rate. Remote servers are expected to deliver data at $5Mbps$.

Figure 52 shows the workload used for the experimental results reported next: a single scan (or fetch) operator that retrieves remote relation A, where $|A| = 50MBytes$. Associated with `scan(A)` there is a data collector, which initially contacts two remote servers (left and right) containing relation A. The left server delivers relation A in ascending order and the right in descending order. Each server is assumed to crash after delivering 25% of relation A. In addition, each server is assumed to deliver data with a certain delay probability.[1]

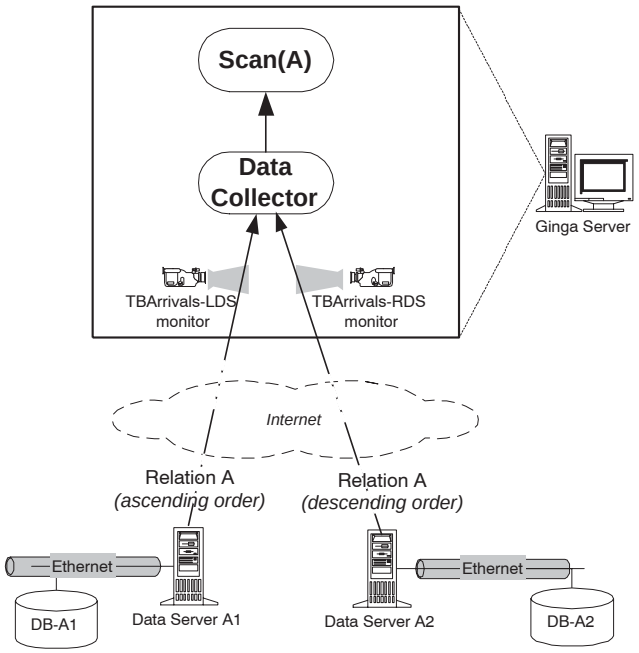
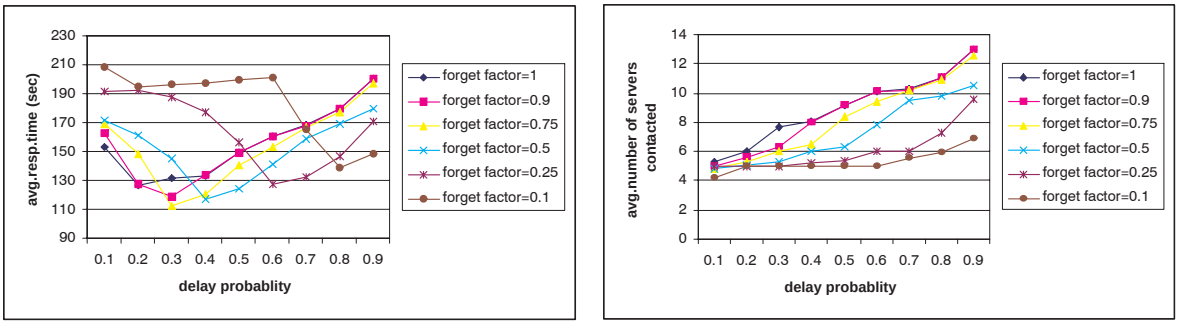


Figure 52: Workload for KDA experiments.

[1]For example, for delay probability equals to 0.9, every time a server delivers a page to the client, this page has 0.9 probability of being delivered with a degraded transfer rate.

The graph in Figure 53(a) shows the average response time[2] for executing `scan(R)` as a function of the delay probability associated with each server when DC uses six different forget factors, namely, 1, 0.9, 0.75, 0.5, 0.25, and 0.1. As expected, for small delay probability, the best response time is given by large forget factors. When the runtime environment is relatively stable (i.e., small delay probability), every small runtime environment change should be considered for adaptation. This means that Ginga should always remember the recent past of the timout history embedded in MTBT. Conversely, as the runtime environment becomes more unstable (i.e., delay probability increases), Ginga needs to be more conservative with respect to firing adaptations. Therefore, Ginga should forget less about the MTBT timeout history. In fact, in the graph of Figure 53(a), for delay probability equals to 0.9, the best response time is given by forget factor 0.1.



(a) (b)

Figure 53: Data collector performance: (a) Average response time; (b) average number of remote server contacted.

For delay probability (dp) equals to 0.1, forget factor $f = 1$ gives the best response time. However, it is interesting to observe that the smallest response time for $f = 1$ occurs when $dp = 0.2$. The reason for that is because for runtime environment with good stability (i.e., very few changes), a more tight (fine grained) monitoring may be necessary. Clearly, as currently implemented, the proposed approach for firing the adaptation process does not work satisfactorily for close-to-stable runtime environment. To address this problem, Ginga needs to reduce the timeout value used by the data collector. With a timeout smaller

[2]For all the experiments reported in this chapter, the average response time is calculated as the average response time of five runs of a query for each point in the x-axis of the graph.

than what Ginga currently uses, MTBT is updated more frequently, and consequently the adaptation process is fired sooner, providing better response time when $dp = 0.1$ and $f = 1$.

The graph in Figure 53(b) shows the average number of servers contacted by the data collector operator as a function of the delay probability when DC uses different forget factors. Given the workload used for the experiments reported in this section, it is expected that at least 4 servers should be contacted to fully collect relation A (each server crashes after delivering 25% of A). It is interesting to observe that for each forget factor line, before the data collector DC reaches its best response time shown in Figure 53(a), DC will contact on average five servers. Once the best response time is reached, then the number of servers contacted increases as the delay probability increases. The reason for that is because, before reaching its best performance, the data collector using a forget factor f is not able to react fast enough and contact alternative servers that could deliver the data faster. However, as delay probability increases and f becomes best suited for determining when to adapt, data collector is able to contact more servers, thus achieving better response time. By continuing to increase the delay probability, data collector using forget factor f will provide better response time than DC's response time using forget factors that are smaller than f. Now, the reason for that is because with forget factor smaller than f, DC is more aggressive (i.e., adapts more frequently) than it is with f. Consequently, the response time becomes worse.

In summary, I conclude that the proposed approach for adapting the download process of data from remote data servers was proved to be beneficial when the appropriate forget factor is chosen. However, the challenge yet to be addressed is how to determine which forget factor to use for a runtime environment with unpredictable changes, where in some moments it is close-to-stable and in other moments it is extremely unstable. One possible solution to address this problem is, for example, to have a counter in the `refreshMTBT` method (Figure 41) for the number of consecutive timeouts observed so far. If the counter becomes larger than a given threshold, this means that too many timeouts have occurred and Ginga should decrease the forget factor so that less adaptation would be fired. Conversely, if the *interval* between the occurrence of consecutive timeouts increases above certain threshold,

then Ginga should increase the forget factor. In this case, I have a situation where the runtime system was quite unstable and now it has achieved certain stability. Incorporating this solution into Ginga approach is part of my future research work.

6.3.3 Combination of Adaptation Actions

Finally, in this section I analyze the performance characteristics of Ginga when combining both concurrent caching and keeping download alive adaptation actions. The experimental results described next are grouped into three subsections. Subsection 6.3.3.1 analyzes the scenario of single join queries. Then, Subsection 6.3.3.2 considers the case when the query plan generated is a left-deep tree. Finally, Subsection 6.3.3.3 investigates the case for right-deep trees.

6.3.3.1 Single Join

For the experimental results reported in this section, I used the workload depicted in Figure 54: a join between remote relations A and B. The size of both relations is $50MBytes$ and the expected data transfer rate is $5Mbps$. The join is hash-based and it is executed without incurring I/O cost (i.e., the hash table for A, the build relation, can fit entirely in main memory). Only the servers providing relation A are considered to crash after downloading 25% of A. Servers for relation B are only delayed. When concurrent caching is executed, B, the probe relation, is cached. I executed a number of experiments using different workload configuration for the join considered in this section, and similar results were observed.

The graph in Figure 55(a) shows the average response time for executing the single join using only KDA adaptation action as a function of delay probability observed for all remote servers. The following six different forget factors for each DC involved in the join were considered: 1, 0.9, 0.75, 0.5, 0.25, and 0.1. For this scenario, I get similar results to those reported in Section 6.3.2. That is because the data collector of each base relation behaves similarly to the data collector shown in Figure 52 where no concurrent caching is being executed. Consequently, the final cost of the hash join in Figure 54 is the sum of each data collector cost plus the cost of executing the join itself.

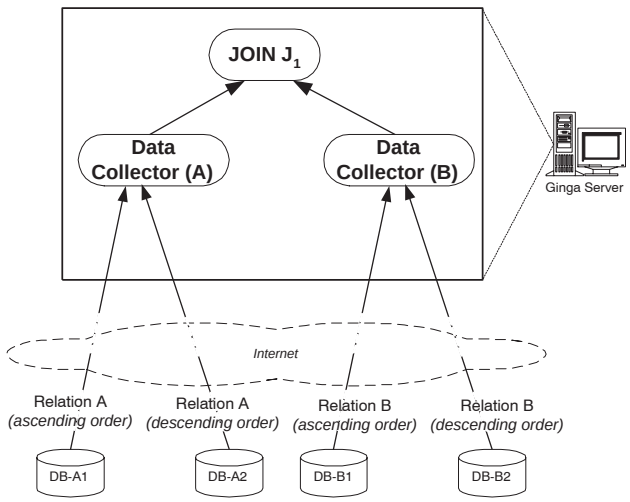
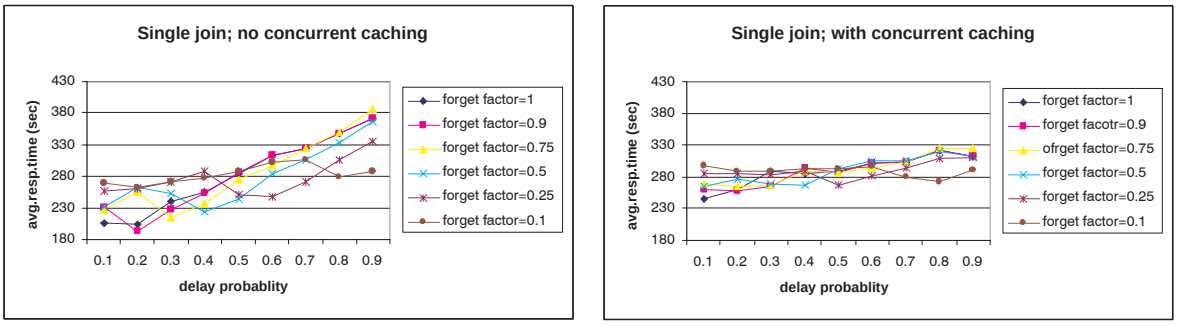


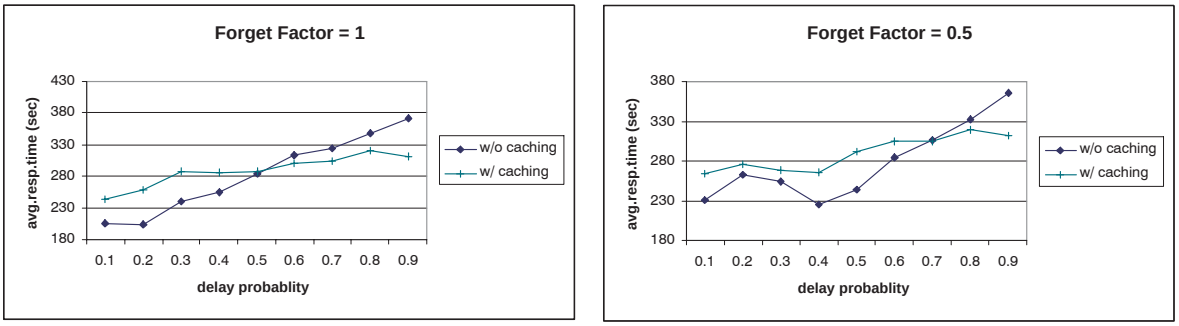
Figure 54: Single-join workload for experiments combining adaptation actions for coping with end-to-end delays.



(a) (b)

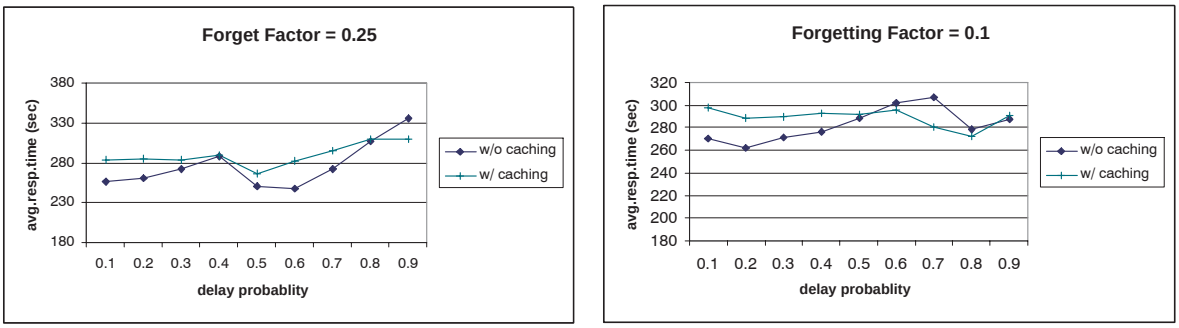
Figure 55: Single join: Average response time (a) without concurrent caching; and (b) with concurrent caching.

The graph in Figure 55(b) shows the average response time for executing the same join as function of the delay probability observed for all remote servers, but this time using both adaptation actions. When compared to the case using only KDA adaptation action, the response time with concurrent caching is worse for small delay probability. That is because hash join tolerates few timeouts and concurrent caching adaptation action ends up always been executed (see Section 6.2.2). For small delay probabilities, the extra I/O cost incurred for caching relation B is not masked by the delays and crashes incurred while downloading relation A.



(a) (b)

Figure 56: Single join: comparing the average response time of running with and without caching when the forget factor is equal to (a) 1, and (b) 0.5.



(a) (b)

Figure 57: Single join: caching versus no caching when forget factor is equal to (a) 0.25, and (b) 0.1.

As the delay probability increases, using concurrent caching provides better response time than without using concurrent caching. This situation can be clearly observed in the graphs of Figure 56 and Figure 57. These graphs only provide a better comparison for each pair of forget factor lines shown in Figure 55(a) and Figure 55(b). The challenge here is how to determine when concurrent caching should be used. The main complication is that not only Ginga cannot accurately predict the cost of downloading relation A, but also it is not possible to know exactly the cost of downloading B given that `DC(B)` also experiences delay in receiving B data. Consequently it becomes very difficult to decide whether or not to use concurrent caching. For the particular case described in this section, I argue that Ginga can always use concurrent caching because the response time for both cases are in

general close to each other.

In summary, I conclude that by combining both concurrent caching and KDA adaptation actions, Ginga can further improve (or maintain) the query execution performance.

6.3.3.2 Left-Deep Trees

For the experimental results reported in this section, I used the workload depicted in Figure 58(a): a two-way join left-deep query tree. The size of each relation is $50MBytes$ and the expected data transfer rate is $1Mbps$. The result size of join J_1 is $50MBytes$. The joins are hash-based and they are executed without incurring I/O cost (i.e., the hash table for A, the build relation, can fit entirely in main memory). Only the servers providing relation A are considered to crashed after downloading 10% of A. Servers for relations B and C are delayed by half of the delay probability applied to servers delivering relation A. For example, if servers for relation A deliver data with delay probability of 0.5, the servers for the other two relations will experience delay probability of 0.25. I executed a number of experiments using different workload configuration for the left-deep query tree considered in this section, and similar results were observed.

When concurrent caching is executed, first Ginga caches relation B. Then, relation C is cached, and finally the result of the join between B and C (both available at the local disk) is cached. However, these caching operations will be executed only when needed (see Section 6.2.1.1).

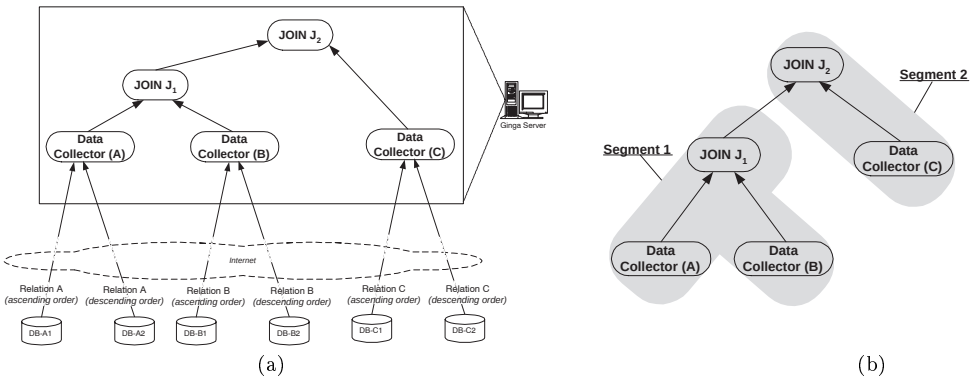
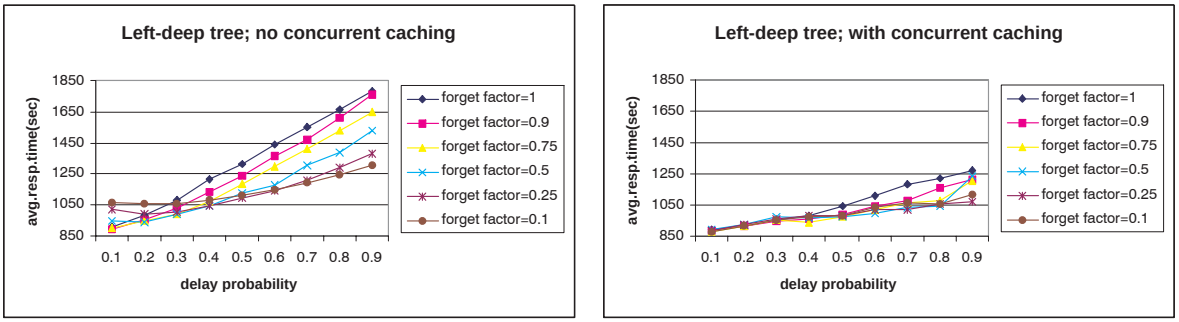


Figure 58: (a) Left-deep tree workload for experiments combining adaptation actions; (b) Left-deep tree decomposed in segments.

The graph in Figure 59(a) shows the average response time for executing the left-deep join using only KDA adaptation action as a function of the delay probability observed in each server delivering relation A. The following six different forget factors for each DC are considered: 1, 0.9, 0.75, 0.5, 0.25, and 0.1. For this scenario, I get similar results to those reported in Section 6.3.2. That is because the data collector of each base relation behaves similarly to the data collector shown in Figure 52 (see discussion in Section 6.3.3.1) where no concurrent caching is being executed.

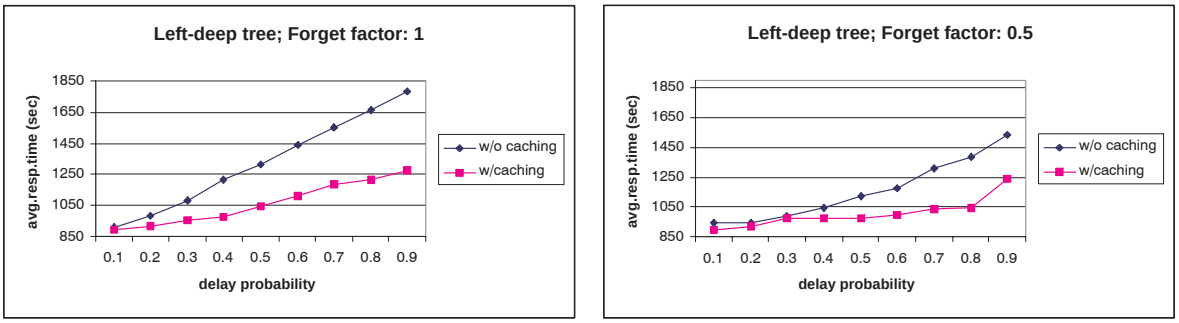


(a) (b)

Figure 59: Left-deep tree: Average response time (a) without concurrent caching; and (b) with concurrent caching.

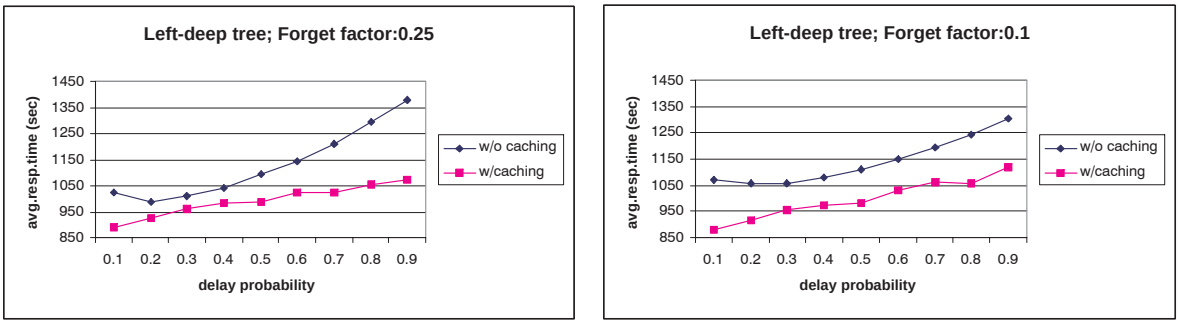
The graph in Figure 59(b) shows the average response time for executing the same left-deep query tree as a function of the delay probability observed in each server delivering relation A but this time using both adaptation actions. When compared to the case using only KDA adaptation action, the response time with concurrent caching is always better. The reason for that is because according to Segmented Execution Model (see Section 4.3.2), the left-deep tree workload used in my experiment will be divided into two segments as depicted in Figure 58(b). For small delay probabilities, it is likely that only relation B is cached. As discussed in Section 6.3.3.1, caching relation B when executing Segment 1 may not always be beneficial. However, this caching pays off when Ginga proceeds with the execution of Segment 2.

As the delay probability increases, the benefits of using concurrent caching versus not using it increases as the delay probability increases. This situation can be clearly observed



(a) (b)

Figure 60: Left-deep query tree: comparing the average response time of running with and without concurrent caching when the forget factor is equal to (a) 1 and (b) 0.5.



(a) (b)

Figure 61: Left-deep query tree: concurrent caching versus no concurrent caching when forget factor is equal to (a) 0.25, and (b) 0.1.

in the graphs of Figure 60 and Figure 61. These graphs only provide a better comparison for each pair of forget factor lines shown in Figure 59(a) and Figure 59(b).

6.3.3.3 Right-Deep Trees

I conclude the experimental results on combining concurrent caching and KDA adaptation actions by analyzing Ginga's performance when the optimized generated plan is a right-deep query tree. The workload use for this final set of experiments is depicted in Figure 62(a): a two-way join right-deep query tree. The size of each relation is $50MBytes$ and the expected data transfer rate is $1Mbps$. The result size of join J_1 is $50MBytes$. The joins are hash-based and they are executed without incurring I/O cost (i.e., the hash table for A, the

build relation, can fit entirely in main memory). Only the servers providing relation A are considered to crashed after downloading 10% of A. Servers for relations B and C are delayed by half of the delay probability applied to servers delivering relation A. I executed a number of experiments using different workload configuration for the right-deep query tree considered in this section, and similar results were observed.

When concurrent caching is executed, first Ginga caches relation B. Then, relation C is cached, and finally the result of the join between C and B (both available at the local disk) is cached.

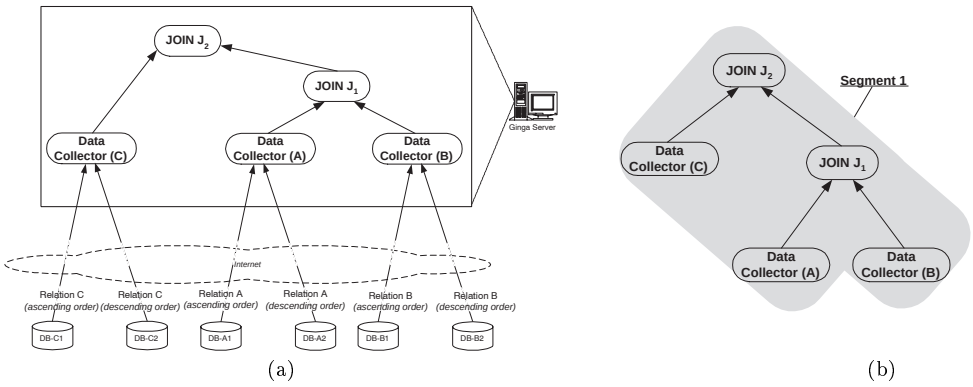
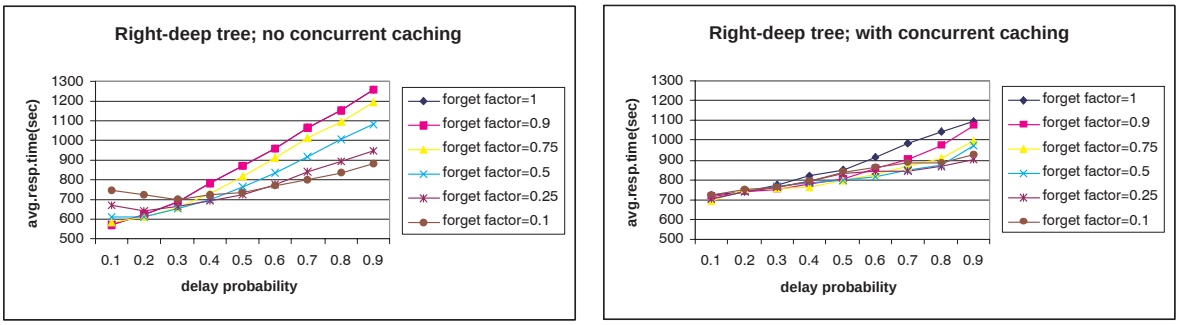


Figure 62: (a) Right-deep tree workload for experiments combining adaptation actions; (b) Segment decomposition for the right-deep tree.

The graphs in Figure 63(a) and Figure 63(b) show the average response time for executing the right-deep query plan when using only KDA adaptation action and when using both adaptation actions, respectively. The x-axis of these graphs represents the delay probability observed for the servers of relation A. The following six different forget factors were considered for each DC: 1, 0.9, 0.75, 0.5, 0.25, and 0.1. I observe that the experimental results reported in these graphs are similar to those presented in Section 6.3.3.1. By taking a close look at the single join scenario from Section 6.3.3.1 and the scenario considered in this section, it can be observed that both scenarios represent the same situation: the execution of a single segment. Therefore, similar results are obtained.

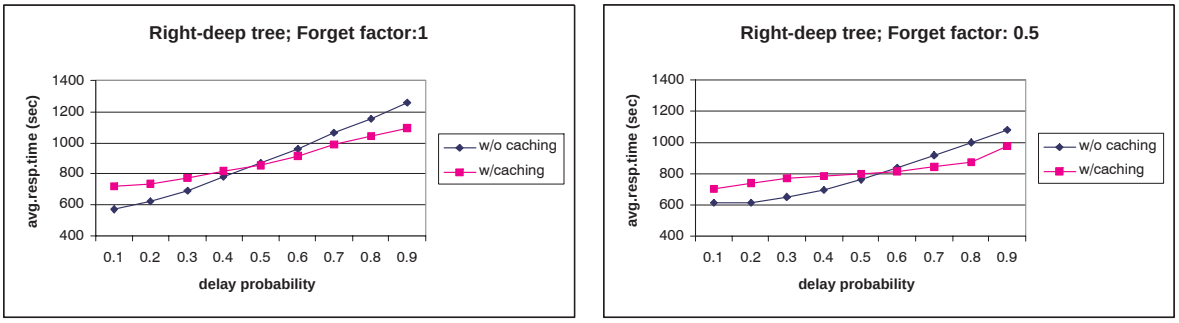
The key reason to decompose the query plan into right-deep segments is because right-deep trees allow Ginga to pipeline the execution of hash joins and achieve inter-operator



(a) (b)

Figure 63: Right-deep query tree: Average response time (a) without concurrent caching; and (b) with concurrent caching.

parallelism. However, if the initial probe relation in this segment has to be first cached into disk and then read back again, the benefits of the SEM is reduced. This situation can be observed in the graphs of Figure 64. These graphs only provide a better comparison for each pair of forget factor lines shown in Figure 63(a) and Figure 63(b). For small delay probability, without using concurrent caching Ginga can provide better response time than if it uses concurrent caching. This situation is changed as the delay probability increases.

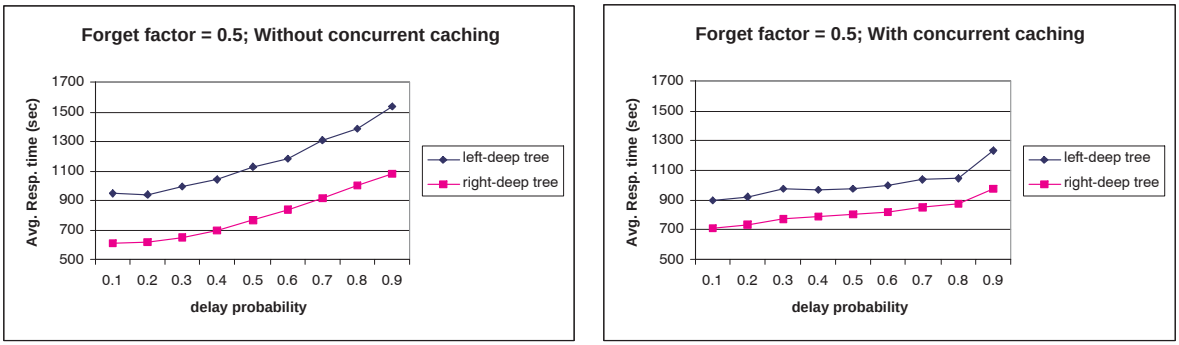


(a) (b)

Figure 64: Right-deep query tree: comparing the average response time of running with and without concurrent caching when the forget factor is equal to (a) 1, and (b) 0.5.

In Section 6.3.3.2, I observed that for left-deep trees, using concurrent caching in conjunction with KDA adaptation action was always better than using only KDA. Based on this observation and the discussion presented in the previous paragraph, one could intuitively conclude that having left-deep trees would be a better query tree shape than right-deep tree

to use for running distributed queries. However, this is not the case as I demonstrate in the graphs of Figure 65. Therefore using right deep trees is the best alternative.



(a) (b)

Figure 65: Comparing average response time for left-deep and right-deep query trees when executed (a) without using concurrent caching, and (b) with concurrent caching.

6.4 Summary of Adaptation to End-to-End Delays

In this chapter, I used Ginga to study the benefits and trade-offs of different adaptation strategies when query execution becomes suboptimal due to end-to-end delays (Section 6.2.1). While a number of unrelated adaptation techniques have been known to be effective in isolation, Ginga uses the concept of adaptation space to organize and combine them. In the case of end-to-end delays, I apply and combine two dynamic adaptation methods: concurrent caching and keeping download alive. Concurrent caching suggests that Ginga executes other parts of the query while waiting for the slow data to be delivered. With keeping download alive, for the sources known to be problematic with respect to the download process, Ginga contacts their mirror sites in parallel.

I used Ginga's query processing simulation system to evaluate the effectiveness of these two adaptation methods in isolation and in combination (Section 6.3). My experimental results confirm that each method improves query response time by efficiently masking unexpected end-to-end delays. More significantly, combined query adaptation can achieve significant performance improvements (up to 30% of response time gain for representative system and workload configurations) when compared to individual solutions. These results

show that it is a good idea to combine adaptive methods to address end-to-end delays at runtime, and Ginga offers a systematic approach to organize and combine these methods in an effective way. The Ginga approach also offers promise for the incorporation of other dynamic adaptation methods to handle runtime changes in end-to-end delays.

CHAPTER VII

ADAPTATION TO COMBINED FAILURE TYPES

7.1 Coping with Multiple Failures

In the last two chapters, I studied how Ginga adapts the execution of a query to each failure type (memory constraints and end-to-end delays) individually. As I demonstrated, for each failure type, having a single adaptation action is not appropriate, as the failure type can have a number of "side effects" in the query execution. I investigated how to combine the alternative adaptation actions associated with each failure type, and promising results were reported.

In this chapter, I study how to combine the adaptation actions to adapt the query execution to combinations of failure types [44]. One of the goals of this chapter is to investigate how the different adaptation actions are related to each other.

7.2 Adaptation Space Generation and Exploration: Memory Constraints and *End-to-End Delays*

In this section, I first discuss how to generate the adaptation cases for an adaptation space that supports query adaptation to both memory constraints and end-to-end delays. Then, I describe how this adaptation space is navigated at runtime.

7.2.1 Adaptation Cases

Adaptation cases are generated using the adaptation actions described in Chapter 5 and Chapter 6.

The decisions involved in creating the adaptation cases for memory constraints do not affect the decisions made in generating the cases for end-to-end delays. However, there is one situation where the decision made by adaptation actions for coping with end-to-end delays may cause changes in memory constraints. This situation occurs when new joins are created by the concurrent caching adaptation action. Memory blocks allocated

to the query plan are entirely devoted to the segment that is currently being executed. If a concurrent join needs to be executed, then extra memory will be required. If extra memory blocks are available in the system, Ginga can request these blocks to allocate to the concurrent join. However, if no spare memory blocks are available, there are at least two alternative approaches to handle the situation. The first approach is a simple one: do not execute the concurrent join. This approach may not be the best one because Ginga will be losing adaptation opportunities. The second approach is to pre-empt memory blocks from the operators in the current segment to allocate to the concurrent join. With this approach, Ginga can take advantage of the adaptation opportunity at the risk of degrading performance of the current segment execution.

In this dissertation, when the situation described in the previous paragraph is detected, I assume that there will be available in the system the minimum memory required for executing the concurrent join without having to pre-empt memory blocks from the operators in the current segment. Incorporating existing approaches into Ginga to handle the situation where memory blocks need to be pre-empted (e.g., [62]) is part of my future research work.

7.2.2 Adaptation Space

When constructing an adaptation space for combined runtime failures, the first step is to determine the order (total or partial) in which the adaptation cases for each failure type should be considered during the query execution. The next step is to use this order to organize the cases into the adaptation space.

For memory constraints and end-to-end delays, the order in which Ginga considers the associated adaptation cases is clear. Adaptation cases for memory constraints adapt the query plan *before* the execution of each segment, whereas cases for end-to-end delays are focused on adapting the plan *during* segment execution. Therefore, when constructing the adaptation space for the combination of these two failure types, Ginga first creates the cases for memory constraints followed by the creation of cases for end-to-end delays. Figure 66 depicts this adaptation space, and Figure 67 shows the respective navigation algorithm.

The navigation of the adaptation space is straightforward. Depending on the adaptation

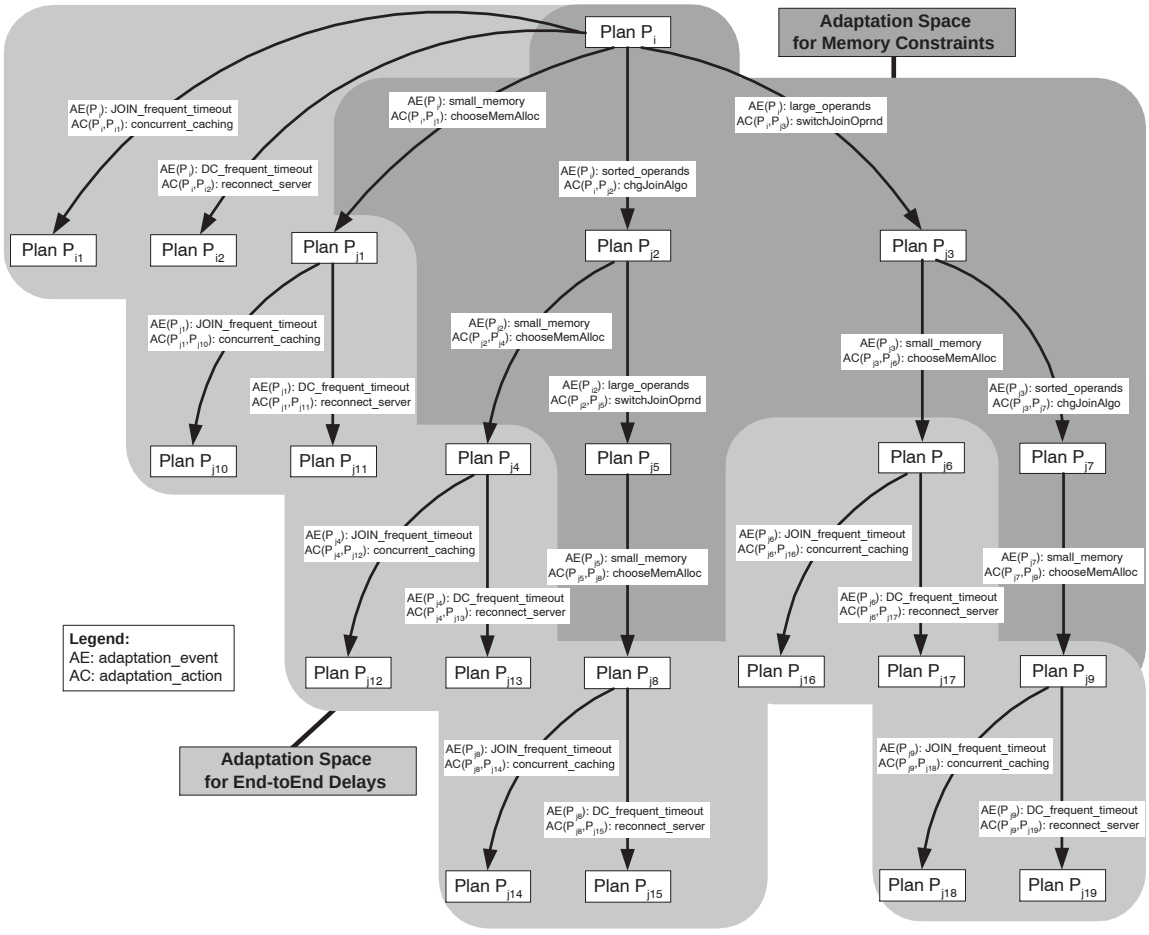


Figure 66: Adaptation Space for the combination of two failure types: memory constraints and end-to-end delays.

```
ADAPTATION SPACE MgtCombinedFailureTypes
Input: combined_adapt_space(P_i): adaptation space for P_i.
Output: P_j: plan P_i adapted.
Require: at least one of memory constraint or end-to-end predicates in adaptation_condition(P_i) is invalidate.
 1:
    // Adaptation event: memory constraints
 2: if (adaptation_event == sorted_operands or adaptation_event == large_operands or adaptation_event ==
    small_memory) then
 3:    P_j = MgtMemoryConstraints(combined_adapt_space(P_i));
 4:
    // Adaptation event: end-to-end delays
 5: if (adaptation_event == JOIN_frequent_timeout or adaptation_event == DC_frequent_timeout) then
 6:    P_j = MgtEndToEndDelay(combined_adapt_space(P_i));
 7:
 8: return P_j;
```

Figure 67: Adaptation Space for Managing the combination of memory constraints and end-to-end delays.

event generated, the respective navigation algorithm is invoked. Note that more than one adaptation event can be generated at the same time. The order in which simultaneous events should be handled is managed by the respective navigation algorithm.

7.3 Performance Analysis

For all of the experiments reported in this chapter, I use a simulator (based on the CSIM toolkit) that models a client-server system with one client running Ginga and n remote data servers. All base relations involved in the queries submitted to Ginga are assumed to be available at the remote servers. The size of the relations reflect the size of the answers provided by web services such as NCBI-Entrez discussed in Section 1.2.

Table 12: Simulation Parameters

Parameter	Value	Description
Speed	100	CPU speed (MIPS)
AvgSeekTime	8.9	average disk seek time (msecs)
AvgRotLatency	5.5	average disk rotational latency (msecs)
TranferRate	100	disk transfer rate (MBytes/sec)
DskPageSize	8192	disk page size (bytes)
MemorySize	$10 \cdots 100$	memory size (MBytes)
DiskIO	5000	instructions to start a disk I/O
Move	2	instructions to move 4 bytes
Comp	4	instructions to compare keys
Hash	25	instructions to hash a key
Swap	100	instructions to swap two tuples
F	1.2	incremental factor for hash join

Table 12 lists the classical parameters [10, 52, 61, 4] used in configuring the simulator. Client and server machines have the same hardware configuration. Disks are modeled as FIFO queues. I model end-to-end delays in terms of degradations to network bandwidth. Network connections are independent of each other, in the sense that the failure of a connection does not affect the others. The network transfer rates and their respective degradations considered in my experiments represent those typically observed in the Internet environment.

The experiments are divided into three groups: The first group (Section 7.3.1) analyzes Ginga's performance when executing alternative join algorithms in the presence of end-to-end delays. The second group (Section 7.3.2) investigates how end-to-end delays can affect Ginga's decision when selecting the appropriate memory allocation strategy. The third group (Section 7.3.3) combines all of the adaptation actions discussed in this dissertation.

7.3.1 Join Algorithms with End-to-End Delays

In this section, I analyze how end-to-end delays affect the performance of the join algorithms (hash join and sort-merge join) used by adaptation action "changing join algorithm" (see Section 5.3.1.1) for coping with memory constraints. In the experiments, I used the workload depicted in Figure 68: a join between remote relations A and B. The size of both relations is $50MBytes$, and the expected data transfer rate is $5Mbps$. In addition, both relations are assumed to be sorted, and $25MBytes$ of memory are allocated for the execution of this join. Only the servers providing relation A are considered to crash after downloading 25% of A. Servers for relation B are only delayed. When concurrent caching is executed, B is cached. I executed a number of experiments using different workload configurations for the join considered in this section, and similar results were observed.

The graphs in Figure 55 show the average response time[1] of executing hash join and sort-merge join as a function of the delay probability that each server experiences when delivering the requested data. In the graph of Figure 55(a), when no concurrent caching is used, sort-merge join performs better than hash join. However, as the delay probability

[1] For all of the experiments reported in this chapter, the average response time is calculated as the average response time of five runs of a query for each point in the x-axis of graph.

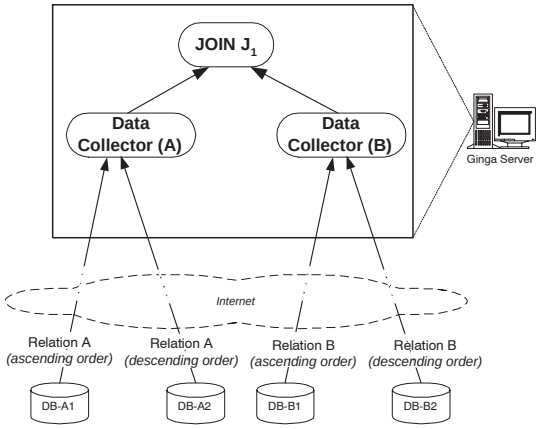
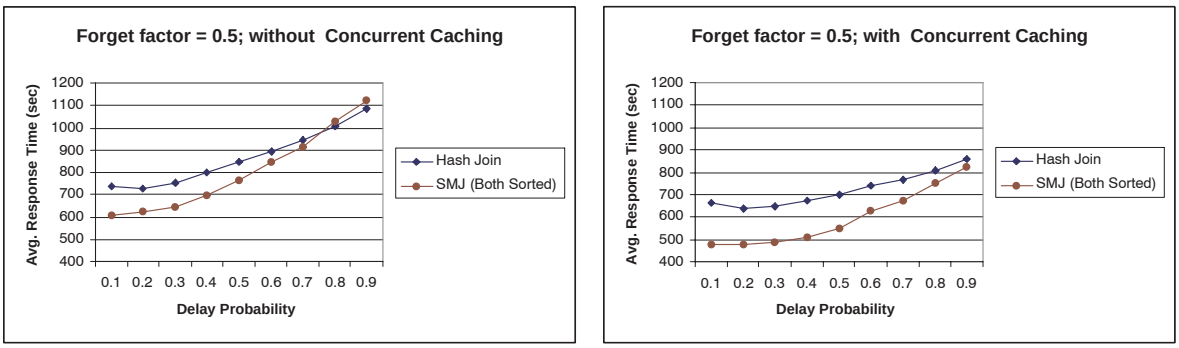


Figure 68: Single-join workload for experiments on alternative join algorithms.

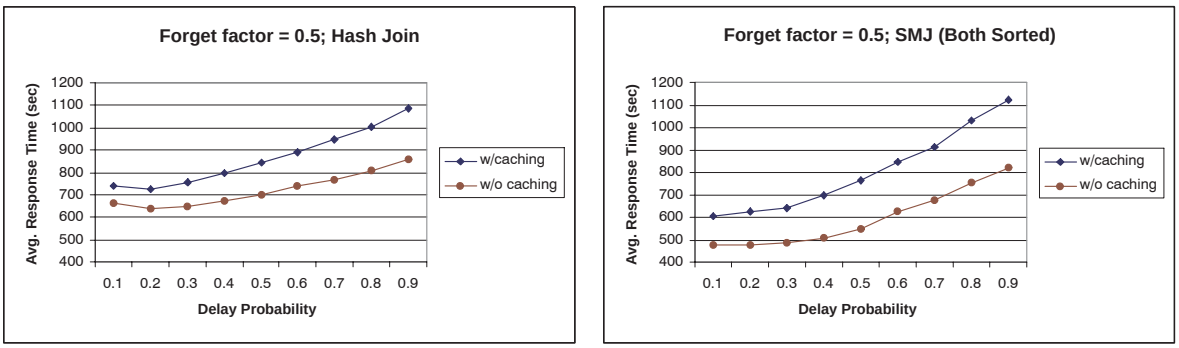
increases, both join algorithms have similar performance. In fact, for delay probability equal to 0.9, hash join outperforms sort-merge join by 36 seconds. Similar analysis applies to the graph in Figure 55(b), where concurrent caching is also used for adapting the execution of the two join algorithms. However, in this case, for delay probability equal to 0.9, hash join does not outperform sort-merge join.



(a) (b)

Figure 69: Single join: Average response time (a) without concurrent caching; and (b) with concurrent caching.

The graphs in Figure 70 compare the performance of each join algorithm when executed with and without concurrent caching. As discussed in Section 6.3.3.3, using concurrent caching is not advantageous when the query plan is decomposed into a single segment. The reason is that the extra I/O cost degrades the pipeline execution of the segment. The



(a) (b)

Figure 70: Single join: comparing the average response time of running with and without caching when the forget factor is equal to (a) 1, and (b) 0.5.

experimental results reported in Figure 70 confirm this observation.[2]

7.3.2 Memory Allocation Strategies with End-to-End Delays

I now turn my attention to experiments where I investigate how end-to-end delays can affect Ginga's decision on which memory allocation to use when coping with memory constraints (adaptation action "choosing memory allocation strategy," Section 5.3.1.3). The workload that I used for the experimental results reported in this section is depicted in Figure 71: a two-way join right-deep query tree, where $|A| = 50Mbytes$, $|B| = 100Mbytes$, and $|C| = 25Mbytes$. The expected data transfer rate is $1Mbps$. The resulting size of join J_1 is $50MBytes$. The joins are hash-based. Only the servers providing relation A are considered to crash after downloading 10% of A. Servers for relations B and C are delayed by half of the delay probability applied to servers delivering relation A. In particular, I fixed the delay probability at 0.9 for relation A servers and the forget factor at 0.25 for all the data collectors. I executed a number of experiments using different workload configurations for the right-deep query tree considered in this section, and similar results were observed.

When concurrent caching is executed, first Ginga caches relation B. Then, relation C is cached, and finally the result of the join between C and B (both available at the local disk) is cached.

[2] A query with a single join has only one segment.

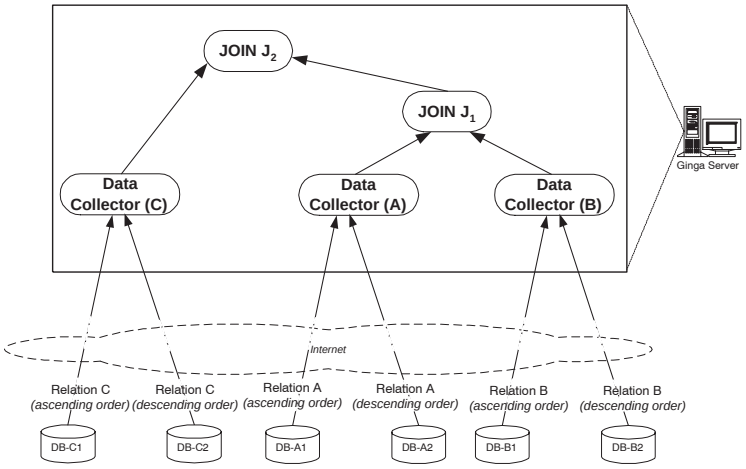
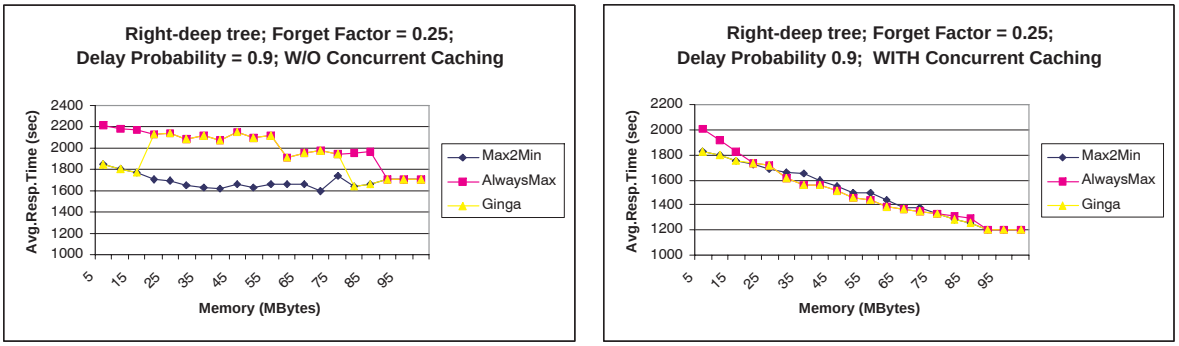


Figure 71: Right-deep tree workload for experiments on alternative memory allocation strategies.



(a) (b)

Figure 72: Right-deep tree: Average response time (a) without concurrent caching; and (b) with concurrent caching.

The graphs in Figure 72 show the average response time for executing the right-deep query plan P (Figure 71) when using three alternative memory allocation strategies (Max2Min, AlwaysMax, and Ginga memory allocation) as a function of the memory allocated for executing P. As described in Section 5.3.1.3, before executing each segment s of a query plan, Ginga memory allocation strategy chooses the allocation strategy that yields the best *estimated* response time for executing s. However, in the graph in Figure 72(a), where no concurrent caching is used, it is interesting to observe how the unpredictable end-to-end delays can affect Ginga's decision on the appropriate memory allocation strategy. For example, when memory allocated for executing P is equal to 15MBytes, instead of selecting

Max2Min as the appropriate allocation strategy, Ginga selects AlwaysMax. The reason for that is because when deciding which allocation strategy to use, the cost of reading the data from the operands of each operator in the segment is not taken into account. In this case, if no problem occurs while downloading the remote operands (i.e., no unpredictable delays or server crashes), Ginga is able to make the right decision about which memory allocation to use. Otherwise, there is the situation reported in the graph in Figure 72(a), where wrong decisions are made.

When concurrent caching is used for executing query plan P of Figure 71, Ginga is able to make the correct decision on which memory allocation to use. Figure 72(b) shows this situation. The reason for this is that with concurrent caching it is likely that by the time Ginga has finished collecting the slow delivered data for relation A, both relation B and relation C have already been cached. The unpredictable cost of downloading the data for these relations is then masked by the delays caused by the servers delivering relation A. Therefore, when Ginga finishes downloading A, relations B an C are already available locally, and no further unpredictable reading cost of these two relations is likely to occur.

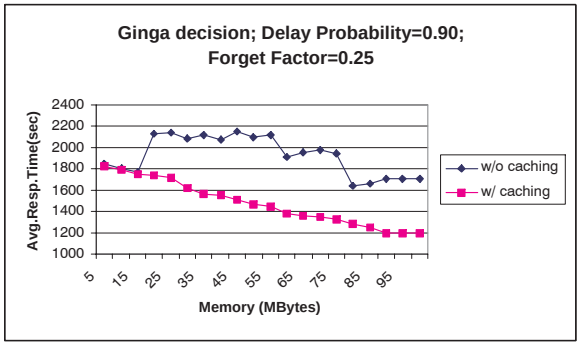


Figure 73: Comparing Ginga decision's effect with and without caching.

Finally, the graph in Figure 73 contrasts the response time of Ginga using concurrent caching and without concurrent caching. I observe that using concurrent caching is beneficial in the sense that it helps Ginga in guaranteeing that the correct decision is made with respect to which memory allocation strategy to use. However, it is important to note that this observation applies to the case where the maximum required memory for join operators in the segment is not provided. When operators are executed at their optimal performance

(see Section 5.2.1), using concurrent caching is not always a good solution for adaptation (see Section 6.3.3.3) when executing right-deep trees.[3]

7.3.3 Combining All Adaptation Actions

I conclude my experimental results on coping with the combination of memory constraints and end-to-end delays by analyzing Ginga's performance when all the adaptation actions discussed in this dissertation are combined. In particular, I revisit the experiments on memory constraints reported in Section 5.4.3, where all adaptation actions for memory constraints are combined, and combine these actions with the adaptation actions for end-to-end delays. This way, I have a scenario in which it is possible to analyze the benefits and trade-offs of adapting a query execution using the Ginga approach for coping with simultaneous multiple failure types.

In Section 5.4.3, I analyzed Ginga's performance when different adaptation paths are navigated within the adaptation space for memory constraints. More specifically, I considered the adaptation paths listed in Table 13. In the experiments reported next, I augment these paths with the adaptation events for end-to-end delays. Observe that by doing so, I am covering most of the adaptation paths within that adaptation space depicted in Figure 66.

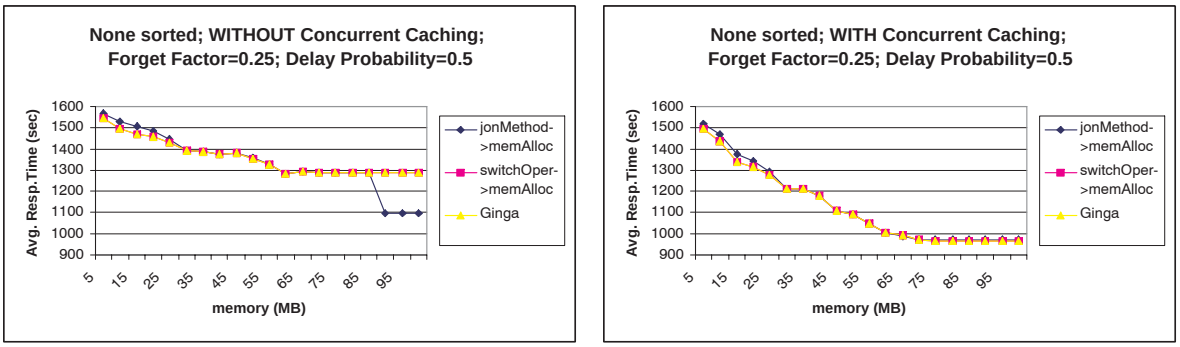
Table 13: Adaptation Paths for coping with memory constraints.

Adapt. Path	Events	Sequence of actions
1	(*sorted_operands, memory_size*)	Changing Join Algorithm → Choosing Memory Allocation Strategies
2	(*large_operands, memory_size*)	Switching Hash Join Operands → Choosing Memory Allocation Strategies

For this final set of experiments, I used a workload similar to the one used for the experiments described in Section 7.3.2. The only difference is with respect to the fixed delay probability and forget factor. For the experiments reported in this section, I fixed delay probability at 0.5 for relation A servers and forget factor at 0.25 for all of the data collectors. I executed a number of experiments using different workload configurations for

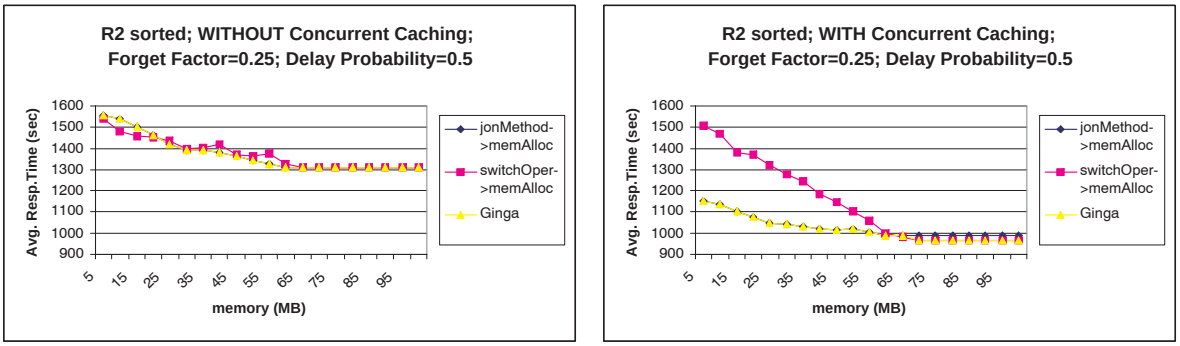
[3]When Ginga allocates the maximum required memory for each join operator, it does not matter which memory allocation strategy it uses. All operators will receive all of the memory they need to perform at their optimal performance.

the right-deep query tree considered in this section, and similar results were observed.



(a) (b)

Figure 74: None Sorted: (a) WITHOUT concurrent caching; (b) WITH concurrent caching.

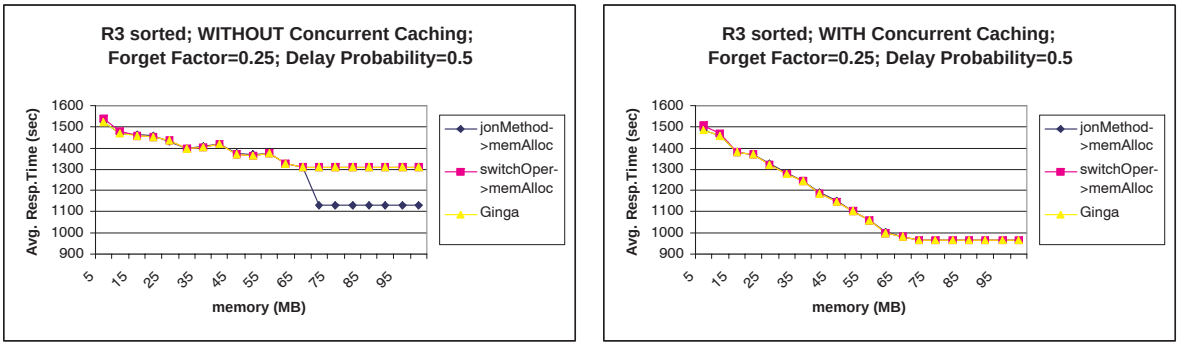


(a) (b)

Figure 75: Only R2 Sorted: (a) WITHOUT concurrent caching; (b) WITH concurrent caching.

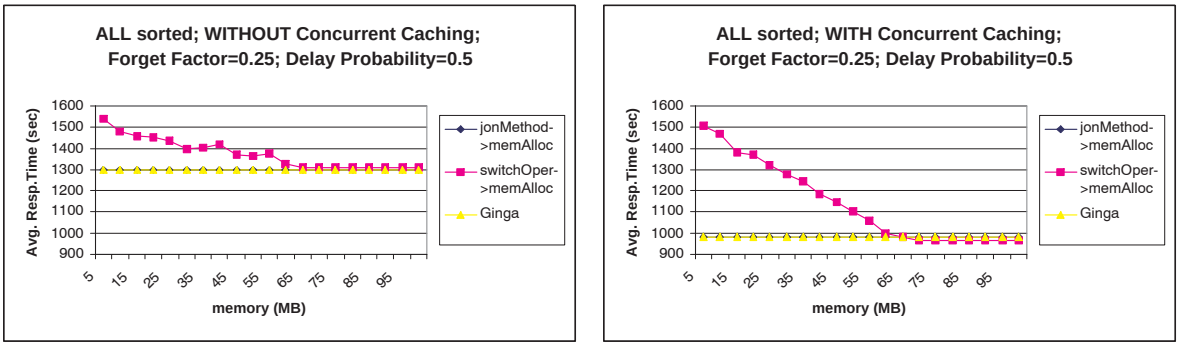
All of the graphs presented in this section show the average response time of executing the right-deep query plan tree depicted in Figure 71 as a function of the size of the memory allocated to it when Ginga navigates the adaptation paths listed in Table 13, augmented by the adaptation events generated due to end-to-end delays.

The graphs in Figure 74 contrast the response time of Ginga using concurrent caching and without concurrent caching when all input relations are not sorted. In accordance with what I discussed in the previous section, when no concurrent caching is used, Ginga still makes wrong decisions about the adaptation process. For example, in Figure 74(a), when 90MBytes are allocated to the query, Ginga should follow the adaptation path (*sorted_operands,*



(a) (b)

Figure 76: Only R3 Sorted: (a) WITHOUT concurrent caching; (b) WITH concurrent caching.



(a) (b)

Figure 77: ALL Sorted: (a) WITHOUT concurrent caching; (b) WITH concurrent caching.

memory_size) instead of (*large_operands, memory_size*) (see Table 13). However, when concurrent caching is used, Ginga is able to cope correctly with the runtime variations. Similar analysis applies to the pair of graphs in Figure 75, Figure 76, and Figure 77. In each of these figures, the graph on the left side reports the average response time when the query is executed without concurrent caching, and the graph on the right side considers the query execution with concurrent caching.

I presented in Section 5.4.3 a detailed discussion on the curves for the different adaptation paths for memory constraints considered in the experiments reported in this section. Therefore, I omit the discussion on these curves here and refer the reader to Section 5.4.3. The point that I want to make with the experimental results of this section is that when I

use adaptation actions for memory constraints combined with adaptation actions for end-to-end delays, I get results similar to those presented in Section 5.4.3. This can be clearly observed when I compare the graphs on the right side of Figure 75, Figure 76, and Figure 77 with the graphs shown in Figure 31 and Figure 32 of Chapter 5. This means that for the runtime change scenarios considered in this dissertation, Ginga is able to successfully adapt the query execution plan to cope with these changes.

7.4 Summary of Adaptation to Combined Failure Types

In this chapter, I demonstrated how to combine the adaptation spaces for memory constraints and end-to-end delays, and how Ginga navigates the combined adaptation space at runtime for coping with the combination these two failure types. To evaluate Ginga's performance, I used workloads similar to those used in Chapter 5 (Adaptation to Memory Constraints) and Chapter 6 (Adaptation to End-to-End Delays) so that I could determine whether Ginga would, in fact, be effective. From the experimental results, I conclude that when Ginga combines all adaptation actions, improved performance can be achieved when coping with the failure types addressed in this dissertation.

CHAPTER VIII

CONCLUSION

8.1 Dissertation Summary

Processing and optimizing ad-hoc and continual queries in an open environment with distributed, autonomous, and heterogeneous data servers (e.g., the Internet) pose several technical challenges. First, it is well known that optimized query execution plans constructed at compile time make some assumptions about the environment (e.g., network speed, data sources' availability). When such assumptions no longer hold at runtime, how can I guarantee the optimized execution of the query? Second, it is widely recognized that runtime adaptation is a complex and difficult task in terms of cost and benefit. How to develop an adaptation methodology that makes the runtime adaptation beneficial at an affordable cost? Last, but not the least, are there any viable performance metrics and performance evaluation techniques for measuring the cost and validating the benefits of runtime adaptation methods?

To address the new challenges posed by Internet query and search systems, several areas of computer science (e.g., database and operating systems) are exploring the design of systems that are adaptive to their environment. However, despite the large number of adaptive systems proposed in the literature until now, most of them present a solution for adapting the system to a specific change in the runtime environment. Typically, these solutions are not easily "extendable" to allow the system to adapt to other runtime changes not predicted in their approach.

In this dissertation, I study the problem of how to construct a framework where the known solutions to query processing adaptation can be cataloged and how to develop an application that makes use of this framework. I call the solution to these two problems the Ginga approach.

I provide in this dissertation three main contributions:

First Contribution: The adoption of the Adaptation Space concept (Chapter 3) combined with feedback-based control mechanisms for coordinating and integrating different kinds of query adaptations to different runtime failure types, such as memory constraints and end-to-end delays.

An adaptation space is a powerful abstraction and framework that describes a collection of possible adaptations of a software component or system and provides a uniform way of viewing a group of alternative software adaptation processes. Adaptation spaces provide application designers with a framework in which to reason about various adaptation methods. By using such a framework, designers can enhance their confidence that all adaptation methods have been considered, and they can have a better understanding of what will happen to the system in each case.

Second Contribution: The Ginga approach (Chapter 4) to adaptive query processing. This approach has three interesting features: First, it utilizes the Adaptation Space concept to manage the predefined generation of query plans. These parameterized plans will serve as alternatives to react to unexpected shortages of runtime resources. Second, it provides a feedback-based control mechanism that allows the query engine to switch to alternative plans upon detecting runtime environment variations. Third, it describes a systematic approach to integrate the adaptation space with feedback control that allows the combination of predefined and reactive adaptive query processing, including policies and mechanisms for determining when to adapt, what to adapt, and how to adapt.

Third Contribution: A detailed performance study, using Ginga, of how adapt to two failure types, and their combination, encountered during the execution of distributed queries: memory constraints and end-to-end delays. The results reported in Chapter 6, Chapter 5, and Chapter 7 show that Ginga holds promise both as an approach and as a system for adaptive query processing. Ginga is a good approach since Adaptation Space supports a variety of adaptation actions through a uniform framework. Ginga is a good system due to the effectiveness of its adaptation policies and mechanisms, as demonstrated by my experimental evaluation.

8.2 Discussion of Open Issues

I conclude with a discussion of interesting open issues that are not addressed in this dissertation but that are relevant to the context of the Ginga approach.

Adaptation Space

- Develop an algorithm for automatically generating adaptation spaces. In this dissertation, I focused on adaptation spaces for the execution of distributed queries. It would be interesting to develop an algorithm that is generic enough so that it could be specialized to different types of "application domain" (e.g., query processing domain, real-time application domain, resource allocation domain). Ultimately, it would be interesting to study how to combine these different specialized (domain-specific) adaptation spaces so that a system that is intelligent enough to survive almost any runtime change could be developed. Runtime change does not only refer to failure types like those that I have studied in my dissertation work. One can think of runtime changes due to system upgrades, which require application re-configurations to adapt to the new environment.

- Investigate how Volcano Dynamic Plans (VDP) [23, 12] and Parametric Query Optimization (PQO) [27] could be extended to generate the adaptation space. There are two main limitations with these two approaches that would need to be addressed: First, they were designed to address the changes of one specific resource parameter. VDP is targeted to the case where selectivities are unknown, and PQO is restricted to memory constraints. Scaling their plan generation algorithm to any number of resource parameters is not a trivial task. The second limitation is that neither VDP nor PQO considers changes to the runtime environment during query execution. They provide only a one-time adaptation (at start-up time) for a query execution. They do not address the issue of changing from one plan to another.

- Develop an algorithm for checking when adaptation cases are equivalent. By using this algorithm as a primitive, an algorithm for determining when two adaptation spaces are

comparable could be developed. Being able to determine when two adaptation spaces are similar can help when determining which adaptation spaces should be stored in the Ginga Adaptation Space repository.

Ginga Approach

- Develop a domain specific language (DSL) for query adaptation using the Ginga approach.[1] In this dissertation, the definition of a number of adaptation actions for coping with memory constraints and end-to-end delays is presented. The manner in which these actions have been described is generic enough to be applied to any input query (provided that queries are executed according to the segmented execution model). With a language in which the primitives are these adaptation actions, the system administrator could program Ginga to adapt to specific runtime changes, as needed. In addition, as new adaptation action primitives become available, the administrator could re-program Ginga to use these new adaptation processes.

- Extend the Ginga approach to accept query inputs expressed in query languages other than SQL. One immediate query language that would be interesting to consider is XQuery. However, this would probably result in having to develop new adaptation actions for adapting the execution of XML queries.

- Develop a query optimizer for creating the initial query plan P_0 that Ginga uses for generating the adaptation space. One possible, and immediate, approach is to extend the Distributed Query Scheduler (DQS) [32] prototype to generate P_0. DQS was developed as an extension to the traditional cost-based query optimizer for processing distributed queries. An alternative approach to generate P_0 would be the *Least Expected Cost* query optimization (LEC) [11], which assumes that there is a probability distribution of the possible runtime parameter values. When using LEC, optimal query plans are chosen based on their expected cost instead of their cost at a specific parameter setting. This approach is motivated by the following two observations:

[1] I would call this language "Balance" because it would allow the system to maintain its "balance" as the runtime changes unpredictably.

First, if the goal is to minimize the expected running time of the plan, then choosing a specific value is not necessarily the right thing to do. In general, the expected best plan is not the plan that is best under the assumption that the parameter takes on its expected value. Second, if the performance of the plan chosen can vary significantly, depending on the parameter value, then the plan chosen by the optimizer might be far from optimal.

- Construct a query optimizer for incrementally generating the alternative query plans associated with each adaptation case. This optimizer should generate alternative plans in a way that yields "smooth" transition from one plan to another at runtime, in case adaptation is deemed necessary. The goal is to minimize, as much as possible, the need for transferring state from one plan to another. Otherwise, the cost of executing the adaptation could be considerably high. This incremental query optimizer should be based on adaptation actions, such as those described in this dissertation.

- Investigate when would be the best moment to construct the adaptation space for an input query. Analogous to the division of adaptation space (AdaptS) generation phase before query execution and AdaptS exploration phase during query execution, the alternative query plans P_i can be generated statically, before the query execution starts, or dynamically, during the query execution. Static generation offers fast adaptation by simple selection at runtime, but it takes more time and space at the beginning (usually an offline process). In contrast, dynamic generation saves time and space at the beginning, but it may slow down the adaptation process at runtime.

- Investigate when to adapt. Determining exactly when to adapt is one of the most difficult challenges of my work. In the Ginga approach, I address this problem as follows: Each adaptation trigger has a *wait_time* component (see Section 4.3.1), which indicates the length of time the trigger condition must hold before the adaptation action occurs. This *wait_time* is used to prevent oscillations in feedback-based adaptation, when, e.g., temporary network latency and bandwidth fluctuations (detected as end-to-end delay) cause repeated transitions back and forth between two query

plans. As in all feedback-based adaptation, a short *wait_time* results in fast adaptation soon after the onset of end-to-end delay. A long *wait_time* slows down the adaptation process, but it decreases the probability of an oscillation. A precise setup for *wait_time* is a difficult problem yet to be solved as it implies being able to predict the future state of a runtime environment.

- Study query adaptation to other failure types. In this dissertation, I study two classical failure types in the execution of distributed queries, namely, end-to-end delays and memory constraints. Another possible failure type would be disk contention, when there are a large number of queries trying to access data from the same local disk. The side effect on the query processing side would be similar to end-to-end delays. However, how one could detect this situation? In addition, caching other parts of the query for coping with this failure type may not be the best solution. This is because the same failure type may re-occur later when the query engine reads the cached result from the disk.

Adaptation to End-to-End Delays

- Integrate Ginga approach with existing monitoring tools for end-to-end delays. One of the possible causes of end-to-end delay is the delay of the network connection due to, e.g., unexpected latency fluctuation. Detecting this type of failure is possible if we use some of the network monitoring tools currently being developed under projects like Web100,[2] Net100,[3] and NetLogger.[4]

- Extend the data collector (DC) operator (see Section 6.2.1.2) to consider the situation where multiple data streams are arriving. In this dissertation, I assume that DC had only two streams of data arriving at any given moment. However, if more data streams are used, the time to download the requested data from a given remote server with a large number of mirror sites can be improved. The major challenge is how to identify

[2] `http://www.web100.org/`
[3] `http://www.net100.org/`
[4] `http://www-didc.lbl.gov/NetLogger/`

and discard duplicated data efficiently.

Adaptation to Memory Constraints

- Construct adaptation actions for adapting a segment to changes in memory constraints *during* its execution. In this dissertation, I adapt a segment only *before* its execution, by choosing sort-merge join when operands are sorted, switching the operands for hash joins when the left operand is found to be larger, and selecting memory allocation strategies, e.g., between Max2Min and AlwaysMax as appropriate. However, if I have a segment with at least two join operators and the size of intermediate results within the segment are detected to be larger than expected, then adaptation may be required. One possible approach to address this problem is to re-allocate the memory blocks among the operators in the segment to minimize the amount of extra paging due to the oversized intermediate result. However, implementing this approach is not a trivial task because it requires preempting memory blocks from one operator to allocate to another operator. The work described in [62] proposes a possible solution to this approach. It would be interesting to investigate how to incorporate this solution into the Ginga approach so that segments can be adapted not only *before*, but also *during*, its execution.

Adaptation to Combined Failure Types

- Investigate the scenario where more than one adaptation action can be fired to cope with unexpected changes in the runtime environment. In this dissertation, I study the situation in which, when a failure type is detected, there is only *one* adaptation action to be fired. In addition, I consider only the scenario where at a given moment, only one failure type is detected. However, when more than one failure type is detected at the same time, which adaptation action should Ginga fire?

REFERENCES

[1] Adali, S., Candan, K. S., Papakonstantinou, Y., and Subrahmanian, V. S., "Query caching and optimization in distributed mediator systems," in *ACM SIGMOD*, 1996.

[2] Amsaleg, L., Franklin, M. J., Tomasic, A., and T.Urhan, "Scrambling query plans to cope with unexpected delays," in *PDIS*, 1996.

[3] Amsaleg, L., Bonnet, P., Franklin, M. J., Tomasic, A., and Urhan, T., "Improving responsiveness for wide-area data access," *Data Engineering Bulletin*, vol. 20, no. 3, 1997.

[4] Amsaleg, L., Franklin, M. J., and Tomasic, A., "Dynamic query operator scheduling for wide-area remote access," *Journal of Distributed and Parallel Databases*, vol. 6, no. 3, 1998.

[5] Antoshenkov, G., "Dynamic query optimization in rdb/vms," ICDE, 1993.

[6] Avnur, R. and Hellerstein, J. M., "Eddies: Continously adaptive query processing," in *ACM SIGMOD*, 2000.

[7] Bernstein, P. A., Goodman, N., Wong, E., Reeve, C. L., and Jr., J. B. R., "Query processing in a system for distributed databases (sdd-1)," *ACM Transactions on Database Systems*, vol. 6, no. 4, 1981.

[8] Bodorik, P., Riordon, J. S., and Jacob, C., "Dynamic distributed query processing techniques," in *ACM 17th Annual Computer Science Conference*, 1989.

[9] Bouganim, L., Fabret, F., Mohan, C., and Valduriez, P., "Dynamic query scheduling in data integration systems," in *ICDE*, 2000.

[10] Bouganim, L., Kapitskaia, O., and Valduriez, P., "Memory-adaptive scheduling for large query execution," in *CIKM*, 1999.

[11] Chu, F., Halpern, J. Y., and Seshadri, P., "Least expected cost query optimization: An exercise in utility," in *PODS*, 1999.

[12] Cole, R. and Graefe, G., "Optimization of dynamic query evaluation plans," in *Proceedings of the ACM SIGMOD*, 1994.

[13] Consel, C., Hornof, L., Noel, F., Noye, J., and Volanschi, E., "A uniform approach for compile-time and run-time specialization," in *Partial Evaluation, International Seminar*, 1996.

[14] Cornell, D. W. and Yu, P. S., "Integration of buffer management and query optimization in relational database environment," in *VLDB*, 1989.

[15] Cowan, C., Black, A., Krasic, C., Pu, C., and Walpole, J., "Specialization classes: an object framework for specialization," in *International workshop on object-oriented in Operating Systems*, 1996.

[16] DeWitt, D. J., Katz, R. H., Shapiro, F. O. L. D., Stonebraker, M., and Wood, D. A., "Implementation techniques for main memory database systems," in *ACM SIGMOD*, 1984.

[17] Du, W., Shan, M., and Dayal, U., "Reducing multidabase query response time by tree balancing," in *ACM SIGMOD*, 1995.

[18] Elmasri, R. and Navathe, S. B., *Fundamentals of Database Systems.* Addison-Wesley, 2003.

[19] Epstein, R., Stonebraker, and Wong, E., "Distributed query processing in relational database systems," in *Proc. ACM SIGMOD*, 1978.

[20] Garcia-Molina, H., Ullman, J. D., and Widom, J., *Database System Implementation.* Prentice-Hall, 2000.

[21] Graefe, G., "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, vol. 25, no. 2, 1993.

[22] Graefe, G. and McKenna, W. J., "The volcano optimizer generator: Extensibility and efficient search," in *ICDE*, 1993.

[23] Graefe, G. and Ward, K., "Dynamic query evaluation plans," in *ACM SIGMOD*, 1989.

[24] Hellerstein, J., Haas, P., and Wang, H., "Online aggregation," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (Tucson, Arizona), May 1997.

[25] Hellerstein, J. M., Franklin, M. J., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V., and Shah, M. A., "Adaptive query processing: Technology in evolution," *Data Engineering Bulletin*, vol. 23, no. 2, 2000.

[26] Ioannidis, Y. and Kang, Y., "Randomized algorithms for optimizing large join queries," in *ACM SIGMOD*, 1990.

[27] Ioannidis, Y. E., Ng, R. T., Shim, K., and Sellis, T. K., "Parametric query optimization," in *VLDB*, 1992.

[28] Ives, Z. G., Florescu, D., Friedman, M., Levy, A., and Weld, D. S., "An adaptive query execution system for data integration," in *ACM SIGMOD*, 1999.

[29] Kabra, N. and DeWitt, D., "Efficient mid-query re-optimization of sub-optimal query execution plans," in *ACM SIGMOD*, 1998.

[30] Liu, L., "Query routing in large-scale digital library systems," in *ICDE*, 1999.

[31] Liu, L., Pu, C., and Lee, Y., "An adaptive approach to query mediation across heterogeneous information sources," in *International Conference on Cooperative Information Systems (CoopIS)*, 1996.

[32] Liu, L., Pu, C., and Richine, K., "Distributed query scheduling service: An architecture and its implementation," *JCIS*, vol. 7, no. 2-3, 1998.

[33] Liu, L., Pu, C., and Tang, W., "Continual queries for internet-scale event-driven information delivery," *IEEE Knowledge and Data Engineering*, 1999. Special Issue on Web Technology.

[34] Liu, L., Pu, C., Tang, W., Biggs, J., Buttler, D., Han, W., Benninghoff, P., and Fenghua, "CQ: A Personalized Update Monitoring Toolkit," in *Proceedings of ACM SIGMOD Conference*, 1998.

[35] Lu, H., Tan, K.-L., and Dao, S., "The fittest survives: An adaptive approach to query optimization," in *VLDB*, 1995.

[36] Maier, D., "Adaptation spaces: Concepts and realization," tech. rep., OGI CSE Technical Report, 1997.

[37] McNamee, D., Walpole, J., Pu, C., Cowan, C., Krasic, C., Goel, A., Wagle, P., Consel, C., Muller, G., and Marlet, R., "Specialization tools and techniques for systematic optimization of system software," *ACM Transactions on Computer Systems*, vol. 19, no. 2, 2001.

[38] Nag, B. and DeWitt, D., "Memory allocation strategies for complex decision support queries," in *CIKM*, 1998.

[39] Ozcan, F., Nural, S., Koksal, P., Evrendilek, C., and Dogac, A., "Dynamic query optimization on a distributed object management platforms," in *CIKM*, 1996.

[40] Ozsu, T. and Valduriez, P., *Principles of Distributed Database Systems*. Prentice-Hall, 1999.

[41] Paques, H., Liu, L., and Pu, C., "Ginga: A self-adaptive query processing system," in *ACM-CIKM*, 2002.

[42] Paques, H., Liu, L., and Pu, C., "Adaptation space: A design framework for adaptive web services," in *International Conference on Web Services - Europe*, 2003.

[43] Paques, H., Liu, L., and Pu, C., "Distributed query adaptation and its trade-offs," in *ACM Symposium in Applied Computing*, 2003.

[44] Paques, H., Pu, C., and Liu, L., "Adapting distributed queries to multiple failure types," tech. rep., Georgia Institute of Technology, *in progress*, 2003.

[45] Paques, H., Pu, C., and Liu, L., "Dynamic adaptation in query execution under variable memory constraints," tech. rep., Georgia Institute of Technology, 2003.

[46] Plale, B. and Schwan, K., "dquob: Managing large da flows using dynamic embedded queries," in *IEEE International Conference on High Performance Distributed Computing*, 2000.

[47] Plale, B. and Schwan, K., "Optimizations enabled by a relational data model view to querying streams," in *IEEE International Conference on Parallel and Distributed Processing Symposium*, 2001.

[48] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K., "Optimistic incremental specialization: Streamlining a commercial operating system," in *SOSP*, 1995.

[49] Pu, C. and Walpole, J., "A case for adaptive os kernels," in *OOPSLA*, 1994.

[50] Rosu, D., Schwan, K., and Jha, R., "On adaptive resource allocation for complex real-time applications," in *IEEE Real-Time Systems Symposium*, 1997.

[51] Selinger, P. and Adiba, M., "Access path selection in distributed database management systems," *VLDB*, 1980.

[52] Shapiro, L. D., "Join processing in database systems with large main memory," *ACM TODS*, vol. 11, no. 3, 1986.

[53] Shekita, E. J., Young, H. C., and Tan, K.-L., "Multi-join optimization for symmetric multiprocessors," in *VDLB*, 1993.

[54] Steere, D. C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., and Walpole, J., "A feedback-driven proportion allocator for real-rate scheduling," in *OSDI*, 1999.

[55] Stonebraker, M., "The design and implementation of distributed ingres," *The INGRES Papers*, vol. M. Stonebraker (ed.), no. Addison-Wesley, 1986.

[56] Stonebraker, M., Aoki, P. M., Litwin, W., Pfeffer, A., Sah, A., Sidell, J., Staelin, C., and Yu, A., "Mariposa: A wide-area distributed database system," *VLDB Journal*, vol. 5, no. 1, 1996.

[57] Tip, F. and Sweeney, P., "Class hierarchy specialization," in *OOPSLA*, 1997.

[58] Urhan, T. and Franklin, M. J., "Xjoin: A reactively-scheduled pipelined join operator," *Data Engineering Bulletin*, vol. 23, no. 2, 2000.

[59] Urhan, T., Franklin, M. J., and Amsaleg, L., "Cost-based query scrambling for initial delays," in *SIGMOD*, 1998.

[60] Volanschi, E. N., Consel, C., and Cowan, C., "Declarative specialization of object-oriented programs," in *OOPSLA*, 1996.

[61] Yu, P. S. and Cornell, D. W., "Buffer management based on return on consumption in a multi-query environment," *VLDB Journal*, vol. 2, 1993.

[62] Zeller, H. and Gray, J., "An adaptive hash join algorithm for multiuser environments," in *VLDB*, 1990.