Ginga: A Self-Adaptive Query Processing System

Henrique Paques
Georgia Inst. of Technology
College of Computing
paques@cc.gatech.edu

Ling Liu
Georgia Inst. of Technology
College of Computing
lingliu@cc.gatech.edu

Calton Pu
Georgia Inst. of Technology
College of Computing
calton@cc.gatech.edu

Categories and Subject Descriptors: H.2.4 [**Systems**]: Query Processing; H.1.0 [**Models and Principles**]: General.**General Terms:** Design, Measurement, Reliability, Experimentation, Performance. **Keywords:** Query Adaptation, Distributed Query Processing.

1. INTRODUCTION

Adaptive query processing in wide area distributed systems has received significant attention due to the dynamics of the Internet. *Ginga* is a project currently ongoing at Georgia Institute of Technology aiming at studying the research issues in building a self-adaptive query processing system. We identify three important failure modes for executing queries on the Internet: network delays [1], memory shortage, and sudden unavailability of remote data servers. This extended abstract will give a brief overview of the design of Ginga system and illustrate the proposed adaptation framework using a network delay scenario as example.

A query processing system is called *adaptive* if it has the following two characteristics. First, it collects information about its runtime environment and uses this information to determine its subsequent behavior. Second, this information collection and behavior revision process iterates over time, generating feedback loop between the runtime environment and the behavior of the system. For each query, information about its runtime environment may include statistics on the specific data sources accessed, the current state of the network at the time that such access is attempted, and the control information entered by the users.

We call a query processing system *self-adaptive* if it satisfies the following two properties: (1) information and control feedback is collected or learned automatically by the system rather than provided through interaction with users, and (2) the system reacts to specified environment changes immediately upon receiving feedback.

We use *self-adaptive* to characterize the *Ginga* system to distributed query processing. The *Ginga* query adaptation engine collects information about its runtime environment with high frequency and full automation, and reacts to any environment change of interest immediately. We argue that to deal with frequent and unexpected delays experienced when accessing data across the Internet, it is important for an Internet search and query system to be self-tunable within a short feedback loop between the runtime environment and the system.

The Ginga project has three interesting features. First, it utilizes the Adaptation Space concept [4] to manage the proactive generation of query plans. Adaptation space offers a simple, yet powerful abstraction and framework to create the main query plans, define system condition changes that would render the current plan sub-optimal and trigger a change to a new query plan, and manage the change process. 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 us to combine proactive and reactive adaptive query processing, including policies and mechanisms for determining when to adapt, what to adapt, and how to adapt.

Our experimental results show that Ginga query adaptation can achieve significant performance improvements (up to 40% of response time gain) for processing distributed queries over the Internet.

CIKM'02, November 4–9, 2002, McLean, Virginia, USA.

2. SYSTEM ARCHITECTURE: OVERVIEW

The Ginga system 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 network 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 rhythm and movements of samba, the query processing system equipped with Ginga can 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 1 shows a sketch of the Ginga system architecture. A query submitted to Ginga 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 [5], 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.

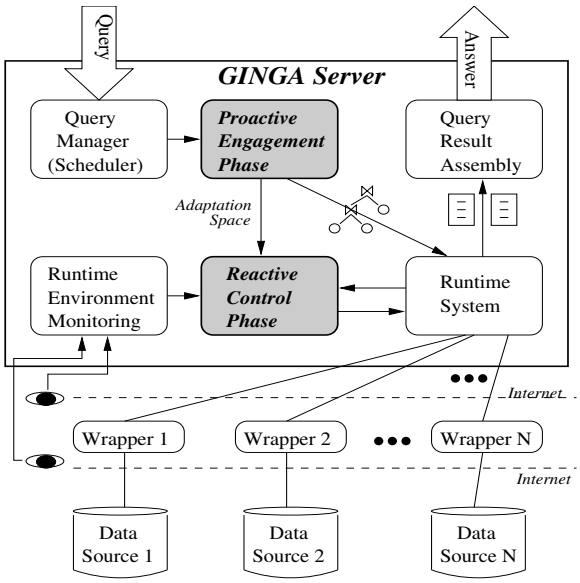

Figure 1: Ginga System Architecture.

Ginga query adaptation engine 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 a proactive *adaptation engagement* phase, before query execution, with a reactive *adaptation control* phase during the execution. At the engagement phase, an initial 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.

3. PROACTIVE ENGAGEMENT PHASE

In the proactive engagement phase, 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 (*e.g.*, [11, 6, 9]). In our experimental study, we use the distributed query scheduler developed in [6]. The second task is to generate query adaptation alternatives ($\{P_i, i = 1, \ldots\ n\}$).

Each of the alternative query plans generated by Ginga consists of two parts. The first part is a trigger, describing a significant runtime environmental change that may degrade the query execution according to the original plan P_0 (or more generally, the current plan on execution, P_{i-1}). The second part is a new query execution plan P_i, optimized for the new runtime parameters, taking into account the change detected. Since there are many potential alternative plans, we organize them into an adaptation space [4].

Adaptation Space Model

In general, an adaptation space consists of two main components: a set of adaptation triggers and a set of adaptation cases associated with the triggers. Formal definitions for the adaptation space model are presented in [4].

In Ginga, adaptation cases are the query plans P_i, where $i = 0, \ldots, n$, each optimized for a set of environmental parameters. An adaptation trigger is defined as a quadruple $< P_{from}, AT\_condition,\ P_{to}, wait\_time >$. Assuming that Ginga is currently executing the query plan P_{from}, when $AT\_condition$ (a significant runtime environment change such as the network delay reaches certain threshold) happens, Ginga adapts by making the transition from P_{from} to P_{to}. The $wait\_time$ component indicates for how long $AT\_condition$ 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 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 network 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.

It is important to note that constructing an adaptation space that covers adaptation cases for every single change to the runtime environment is not only unrealistic but also unnecessary due to a number of reasons. First, it is very unlikely that all possible changes will indeed occur. Second, the overhead of constructing such a complete 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, the initial adaptation space for each query is generated at the adaptation engagement phase, prior to the query execution. New adaptation cases can be inserted to the initial adaptation space as new triggers are identified from the feedbacks about the runtime environment behavior over multiple runs of the same query. We gather information of the runtime environment by monitoring the usage of the resources allocated during the query execution.

An adaptation space can also be described by a *transition graph* – a *lattice* with nodes representing query plans and edges representing the transition from one query plan to another. We illustrate next the construction of an adaptation space by using a simple distributed query.

Example 1 (Adaptation Space Construction): Suppose that we want to execute query Q (Figure 2(a)) where all data sources are remotely located and connected to the Ginga Server through network links (L_i, $1 \leq i \leq 3$) with transfer rate ($Rate(L_i)$) of 1Mbps. Also, assume that we are interested in adapting the execution of Q whenever $Rate(L_i)$ drops below $w \times 1Mbps$, $0 < w \leq 1$.

The construction of an adaptation space for executing Q has three main steps. First, we generate the initial query plan P_0 and record the assumptions made about the runtime environment used by P_0. Second, we identify the set of important adaptation triggers $AT\_conditions$, specifying the conditions when adaptation should be turned on to deal with the delays experienced. Third, we generate the alternative optimized query plan for each adaptation trigger $AT\_condition$. Then we construct the adaptation space in the format of a transition graph. Figure 2(b) shows an example adaptation space for Q. The root adaptation case P_0 (labeled as $ac\_initialP$) is the initial optimized query plan, and the other nodes represent alternative query plans when

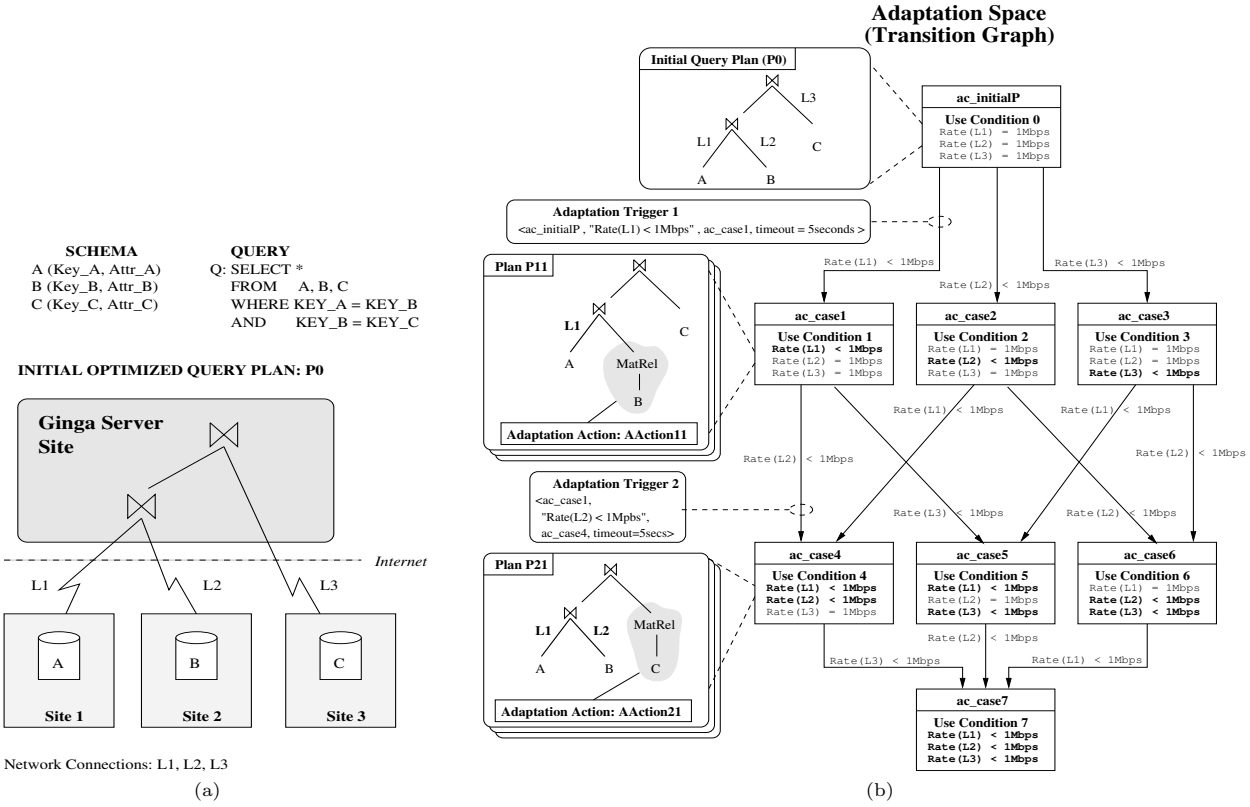

Figure 2: (a) Initial optimized plan query P_0; (b) Adaptation Space for Q.

network delays occur. For each non-root node, we may have more than one alternative query plan where each plan addresses different network delay duration (see Example 2).

4. REACTIVE CONTROL PHASE

When query Q is executed, Ginga (reactive control phase) monitors the *AT_conditions* specified in Q's adaptation triggers that are relevant to the current query plan P_{from} (*e.g.*, transfer rate of network connections). The adaptation trigger is fired when its *AT_condition* becomes true (*e.g.*, network delay). The transition from plan P_{from} to P_{to} is called *adaptation action*.

When an adaptation trigger is fired, Ginga checks the current stage of the query execution.[1] If the adaptation action is found to be still beneficial, Ginga proceeds to change the query plan to P_{to}. Determining if an adaptation action is beneficial depends directly on how much of the query plan has already been processed. For example, if Adaptation Trigger 1 from Figure 2(b) is fired after we have collected most of the data from Source A (Site 1), scheduling adaptation action $AAction11$ may not provide significant gains to the final query response time.

Using the Adaptation Space model, we now discuss the concrete steps taken by Ginga to cope with environmental changes, particularly network delays caused by slow delivery.

[1]We assume that the Runtime System will keep the Reactive Control Phase informed of its current execution.

In order to cope with network delays, Ginga first schedules adaptation actions that involve materializing independent subtrees from the original query tree while continuing to process the data retrieval through the slow network connection[2]. When no more independent subtrees can be materialized and the problematic connection is still slow, Ginga schedules adaptation actions that create and materialize new joins between the relations that were previously materialized.

Example 2 (Adaptation Actions): Suppose that as the query processor starts executing P_0 (Figure 2(a)), the transfer rate of connection L_1 degrades from 1Mbps to 500Kbps causing the data from A to be slowly delivered. Consequently, Ginga schedules the first adaptation action to start the new query plan P_{11}, which executes concurrently the materialization of remote data source B (*MatRel(B)*). If we can finish processing A [3] while concurrently running *MatRel(B)*, then no further adaptation is necessary. However, if the slow delivery on L_1 delays the processing of A beyond the completion of *MatRel(B)*, then another adaptation action should be scheduled. In our example, the new query plan P_{12} (Figure 3) starts concurrently the materialization of the data from source C (*MatRel(C)*). The goal of both

[2]We call *independent subtree* a query tree from the original plan that does not depend on the input from the delayed source.

[3]Processing A consists of retrieving, hashing, and partitioning the data from A.

P_{11} and P_{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 P_0 to plan P_{11} or from P_{11} to P_{12}, the operator schedule is changed accordingly.

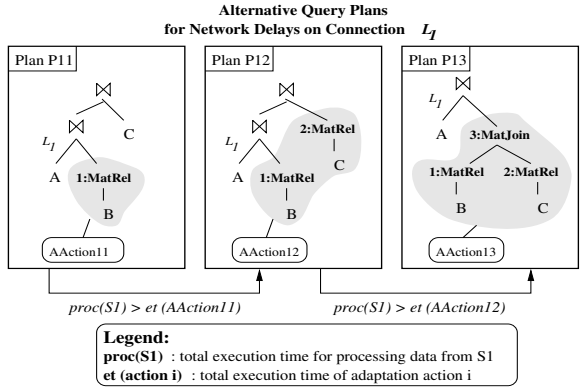

Figure 3: Summary of adaptation actions when delays are detected on network connection L_1.

When there are no more *independent* query subtrees from the original plan to be materialized and we still have not finished processing A, the next adaptation action is to start with joins. The new query plan P_{13} will create and materialize a new join with the relations that P_{12} materialized. In this example, P_{13} will start $MatRel(B) \bowtie MatRel(C)$. Ginga creates new joins by following the query graph associated with P_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.

The eager execution of these expensive new join operators is a departure from the initial optimized plan P_0. This means that P_{13} was considered sub-optimal when compared to P_0 originally. Therefore, P_{13} may or may not finish before $P_{1i}, i < 3$, depending on the completion time of processing A. 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 [3].

The scheduling of P_{13} results in changing the right operand of the originally scheduled join between A and B. In our current implementation of Ginga, this scenario is possible because we assume that all joins operators are hash-based [10] and the execution of a query plan follows the model presented in [8], where the left operand from the hash join is first materialized before starting the probe of the right operand in a pipeline fashion.

Figure 3 summarizes the adaptation actions and *AT_conditions* (represented by the labels on the edges connecting the query plans) used for coping with network delays detected on connection L_1. For each action, the materializations are numbered in the order they should the executed. Observe that these new materialization operations are scheduled in a way that allows a smooth transition from one query plan to another at runtime.

5. RELATED WORK AND CONCLUSION

Representative examples of relevant adaptive query processing research include *Query Scrambling* [1] and *Dynamic Scheduling Execution*(DSE) [2]. Query Scrambling uses materialization and operator synthesis to adapt the execution of distributed queries in the presence of network delays. DSE also adapts the execution of distributed queries to the network delays by concurrently scheduling the execution of independent subtrees. In comparison, Ginga uses a unified framework (adaptation space model) to support both approaches. In addition, Ginga's framework is powerful enough for us to study not only query adaptation to network delays, but also query adaptation to memory shortage and sudden unavailability of remote sources.

In this extended abstract, we outline some of the interesting features of Ginga, primarily an experimental study of the trade-offs in adapting the execution of distributed query to network delays. Our ongoing research efforts include other resource constraint trade-offs such as memory shortage, sudden unavailability of remote sources, and combined failure modes. Further technical details of Ginga are described in [7].

6. ACKNOWLEDGEMENTS

The first author was partially supported by CAPES - Brasilia, Brazil - and DoE SciDAC grants. The other two authors were partially supported by DARPA, DoE, and NSF grants.

7. REFERENCES

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

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

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

[4] L. Liu. Adaptation cases and adaptation spaces: Notation, issues, and applications (part i: Concepts and semantics). Technical report, OGI CSE Heterodyne Working Paper, 1998.

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

[6] L. Liu, C. Pu, and K. Richine. Distributed query scheduling service: An architecture and its implementation. *JCIS*, 7(2-3), 1998.

[7] H. Paques, L. Liu, and C. Pu. Distributed Query Adaptation and Its Trade-offs. Technical Report, Georgia Institute of Technology, 2002.

[8] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor databas machines. *VLDB'90*.

[9] P. Selinger and M. Adiba. Access path selection in distributed database management systems. *VLDB'80*.

[10] L. D. Shapiro. Join processing in database systems with large main memory. *ACM TODS*, 11(3), 1986.

[11] M. Stonebraker. The design and implementation of distributed ingres. *The INGRES Papers*, M. Stonebraker (ed.)(Addison-Wesley), 1986.