

SPECULATIVE PLAN EXECUTION FOR
INFORMATION AGENTS

by

Greg Barish

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

December 2003

Copyright 2003

Greg Barish

Dedication

*To my parents Tina and Frank,
my first and most influential teachers.*

For their encouragement, understanding, and love.

Acknowledgements

I would very much like to thank my thesis advisor Craig Knoblock for the many enjoyable years of mentorship, support, and friendship. Craig has always given me the freedom to explore my own paths towards solving a problem, encouraged me to take chances, while at the same time challenging me to back up my claims and to sometimes consider alternative approaches. Through him, I learned how to read research papers as well as how to write them. His thoughts and advice greatly influenced and improved this thesis. I am extremely grateful for his guidance and I know that it will continue to inspire me as I work with and mentor others.

I would also like to thank the members of my thesis committee: Dr. Steven Minton, Professor Paul Rosenbloom, Professor Cyrus Shahabi, and Professor Jean-Luc Gaudiot. Their suggestions and comments on this work improved it significantly.

In addition to being an excellent committee member, Steve Minton has been both very supportive and influential since my first days as a graduate student. Steve contributed a great deal to the design of the Theseus agent executor and had many important comments and suggestions with regards to learning for speculative execution. He has the unique ability to listen to a complex problem and not only quickly understand and analyze it, but also to provide useful feedback. I have greatly enjoyed working with him over the course of my research and have been lucky to have his advice. I am also very excited to join him at Fetch.com.

I am very grateful for the thorough review that Paul Rosenbloom gave my dissertation. During our meetings together, I was often amazed at how quickly Paul was able to identify the key issues of the research. The questions he asked were always important ones that really made me think. Along with Steve Minton, Paul helped me to better understand the relationship between my work and past research in speedup learning. The quality of this thesis is much improved because of his feedback.

I have really enjoyed working with Cyrus Shahabi since my first years as a graduate student. Back then, we spent many hours discussing, designing, and then deploying the TheaterLoc information integration application. Later, Cyrus included me in a research effort related to haptic data gathering and I enjoyed learning about some of the challenges associated with immersidata data management. I am grateful for his support and advice.

I owe many thanks to Jean-Luc Gaudiot for not only being a great committee member, but for also playing a significant role in educating me about dataflow architectures. His discussions with me and his class on functional programming, dataflow computing, and multi-threaded systems allowed me to better understand both the theoretical origins as well as the current state of the art. What he taught me influenced the implementation of the Theseus executor in several important respects.

The Information Sciences Institute (ISI) in Marina del Rey is an outstanding place to do research as a graduate student. While there, I was able to work on my own research as well as integrate it into larger software systems and projects. I feel

very fortunate to have been a member of the Intelligent Systems Division (ISD) family at ISI, which is a supportive and inspiring group of researchers. As division leader, Yigal Arens has been very accessible, thoughtful, and always supportive. Others in ISD contributed directly to my own research. I very much appreciate discussions with Kevin Knight on the use of transducers to generate value predictions. I am also indebted to Yolanda Gil for her very timely and insightful comments about my defense presentation slides.

I have had the great pleasure to work with many fine colleagues in the Information Agents Group, including Jose-Luis Ambite, Naveen Ashish, Matthew Ho, Salim Khan, Kristina Lerman, Martin Michalowski, Ion Muslea, Maria Muslea, Jean Oh, Andrew Philpot, Sheila Tejada, Snehal Thakkar, and Rattapoom (Pipe) Tuchinda. I learned a lot from them and had a great time working together. Many of them – including Maria, Jean, Martin, Snehal, and Pipe – contributed to the development and testing of the Theseus agent executor.

I am also very much indebted to three alumni of the group, Dan DiPasquo, Dan Rosenberry, and Parag Samdadiya, as well as several people from Fetch Technologies, notably Gary Hirschhorn, Cenk Gazen, and Bryan Pelz. All contributed to the design and development of Theseus and I really enjoyed working with them. In addition, I would like to thank Claude Nanjo from Fetch for useful feedback on parts of this thesis.

Other friends in graduate school made life on campus interesting and I enjoyed getting to know them over the years as well as celebrating and lamenting grad school together. Thanks to Yi-Shin Chen, Chad Jenkins, Mohammad Kohladozhuan, Jay Modi, Joonseok Park, Jae Wook Shin, Nan-Kyung Suh, and Didi Yao.

While a graduate student, I was lucky to receive the Intel Corporation Graduate Fellowship. In addition to financial support, Intel provided me with a first-class laptop computer. That computer increased my productivity immensely and I would like to thank Intel for donating such a wonderfully useful gift that had a real impact on my research.

Before graduate school, I was employed at Oracle Corporation and Healthon/WebMD Corporation. Many people I worked with or for inspired me to become a better engineer and I believe this better prepared me for some of the challenges I faced during my PhD research. These people included Sarah Groves Hobart, Jon Veach, Curt Bennett, Mark Moore, Kittu Kolluri, and Theron Tock.

While at UCLA, I studied Cognitive Science. Some of my earliest interests in AI stem from classes I took and people I met in this program. In particular, I would like to thank Keith Holyoak and Barbara Spellman for giving me an opportunity to work on their semantic priming research as an undergraduate.

I was fortunate to spend many of my pre-college years with friends Sven, Justin, and Tom, who all shared my interests in computers and video games. I have vivid memories of typing in programs from Compute magazine, desperately trying to get some games on cassette tape to load, and later beginning to design our own games. Those were very enjoyable years that played a role in where I am today.

To my wonderful family – Mom, Dad, Heather, Lisa, Chris, Jack, Zoey, and Max. They have always been with me, every step of the way. Their love, support, and advice over the years means more to me than I know how to express.

Finally, to my dear wife Seong Rim. She was always interested in my work, understood the nights and weekends I spent writing code and running experiments, put up with my San Francisco to Los Angeles commute, and was there with love and inspiration every day just the same.

This material is based upon work supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contract/agreement numbers F30602-01-C-0197 and F30602-00-1-0504, in part by the Air Force Office of Scientific Research under grant numbers F49620-01-1-0053 and F49620-02-1-0270, in part by the United States Air Force under contract number F49620-02-C-0103, in part by gifts from the Intel and Microsoft Corporations. The U.S. Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copy right annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

Contents

Dedication	ii
Acknowledgements	iii
List of Tables	ix
List of Figures	x
Abstract	xiii
1. Introduction	1
1.1 The Challenge of Performance	1
1.2 Motivating example.....	4
1.3 Approach.....	7
1.3.1 Streaming dataflow plan execution.....	7
1.3.2 Speculative plan execution.....	9
1.3.3 Learning to predict data for speculative plan execution	10
1.4 Thesis statement	11
1.5 Contributions.....	11
1.6 Thesis organization	12
2. Efficient execution of information agent plans.....	13
2.1 The nature of information agent plans	13
2.1.1 Example Web information gathering tasks	14
2.2 Background.....	15
2.2.1 Dataflow Computing.....	15
2.2.2 Web-based information gathering and integration.....	17
2.3 An Expressive and Efficient Agent Plan Language	18
2.3.1 Plan representation	18
2.3.2 Data structures.....	21
2.3.3 Plan Operators	22
2.3.4 Subplans	28
2.4 Using the plan language to build information agents	30
2.4.1 The CarInfo agent plan.....	30
2.4.2 Homeseekers: a more complicated type of information agent.....	30
2.4.3 The Homeseekers agent plan	34
2.5 An efficient plan execution architecture	35
2.5.1 Dataflow executor	36

2.5.2 Data streaming	39
2.6 Experimental results	41
2.6.1 The Theseus information agent system.....	42
2.6.2 Hypothesis 1: Efficient information agents.....	42
2.6.3 Hypothesis 2: Achieving certain information gathering goals more easily	46
2.6.4 Hypothesis 3: Increased expressivity does not impact performance	48
2.7 Summary	50
3. Speculative Plan Execution	51
3.1 Exceeding the dataflow limit with speculative plan execution.....	52
3.2 Speculation and confirmation	53
3.2.1 Safety and fairness	55
3.2.2 Optimistic performance benefits	56
3.3 Achieving better speedups.....	56
3.3.1 Cascading speculation.....	57
3.3.2 Simultaneous speculation.....	62
3.3.3 Leveraging antecedent and subsequent functional dependencies	63
3.4 Automatic plan transformation	66
3.4.1 The set of candidate transformations	66
3.4.2 Heuristics to reduce the number of possible transformations	67
3.4.3 The SPEC-REWRITE algorithm	68
3.5 Experimental results	71
3.5.1 Web agent plans	72
3.5.2 Database query plans.....	83
3.6 Summary	90
4. Value Prediction for Speculative Execution	91
4.1 Value prediction strategies	91
4.1.1 Caching	92
4.1.2 Classification.....	93
4.1.3 Transduction.....	94
4.1.4 Comparison of techniques.....	97
4.2 A unifying learning algorithm	97
4.2.1 Value Transducers.....	97
4.2.2 Learning templates of string sets.....	98
4.2.3 Learning hint transducers.....	99
4.2.4 Detailed example of predictor learning.....	100
4.3 Experimental results	103

4.3.1 The learning cycle	104
4.3.2 Measurements of predictor accuracy	106
4.3.3 Measurements of predictor space-efficiency	108
4.3.4 Discussion	110
4.4 Summary	113
5. Related Work.....	115
5.1 Expressive and efficient plan execution	115
5.1.1 Network query engines	115
5.1.2 General purpose plan execution systems	117
5.1.3 Other dataflow computing languages.....	117
5.2 Speculative execution.....	119
5.2.1 Execution based on partial and approximate results	119
5.2.2 Executing anticipated actions in advance	121
5.2.3 Prefetching data.....	122
5.2.4 Speculative execution at the operating system and database level	123
5.2.5 Speculative execution at the hardware level.....	123
5.3 Value prediction	124
5.3.1 Value prediction as speedup learning	124
5.3.2 Other techniques for value prediction.....	125
5.3.3 Other approaches to learning transducers	126
6. Conclusion and Future Work	128
6.1 Limitations	129
6.2 Future Work.....	130
6.2.1 Learning to choose good values for speculative overhead.....	130
6.2.2 Classifier compression / probabilistic classification.....	131
6.2.3 SMT Benchmarking.....	131
6.2.4 Integrating additional value prediction techniques	132
Bibliography	134

List of Tables

Table 2.1a: Data gathering operators	23
Table 2.1b: Data manipulation operators	24
Table 2.1c: Data storage operators	24
Table 2.1d: Conditional execution operator	24
Table 2.1e: Asynchronous notification operators	24
Table 2.1f: Task administration operators	24
Table 2.1g: Extensibility operators	24
Table 2.2: The benefits of streaming dataflow for three other plans	44
Table 2.3: Data sources used in (Raman and Hellerstein 2002)	48
Table 2.4: Query used by (Raman and Hellerstein 2002)	49
Table 3.1: Optimistic execution times for CarInfo flows shown in Figure 3.6	58
Table 3.2: Independent probabilities of each speculative opportunity	61
Table 3.3: Likelihood of various fear execution schedules	61
Table 3.4: Execution schedule probability and normal/contributing performance ..	61
Table 3.5: Operator execution times in CarInfo	80
Table 3.6: Path execution times in CarInfo	80
Table 3.7: Path execution times after transformation for speculative execution	80
Table 4.1: Cache for the Edmunds wrapper in CarInfo after one example	92
Table 4.2: Cache for Edmunds based on three examples	93
Table 4.3: Comparing value prediction strategies	97
Table 4.4a: The sequence of source examples	101
Table 4.4b: The sequence of target examples	101
Table 4.5: VTs for the ConsumerGuide search predictor after two examples	102
Table 4.6: VTs for the ConsumerGuide search predictor after three examples	103
Table 4.7: VTs for the ConsumerGuide search predictor after four examples	103
Table 4.8: Summary of predictors learned	105
Table 4.9: Average number of examples required to learn	108
Table 4.10a: Space efficiency of classification-based predictors vs. caches	109
Table 4.10b: Space efficiency of transduction-based predictors vs. caches	110

List of Figures

Figure 1.1: Von Neumann vs. dataflow execution.....	2
Figure 1.2: Non-streaming vs. streaming execution	3
Figure 1.3a: Edmunds car search results page	5
Figure 1.3b: NHTSA safety ratings page.....	5
Figure 1.3c: ConsumerGuide car reviews page	6
Figure 1.4: The CarInfo plan.....	6
Figure 1.5: Execution time chart for CarInfo.....	7
Figure 1.6 Dataflow-style version of CarInfo information agent plan.....	8
Figure 1.7 Execution time chart for CarInfo under streaming dataflow	8
Figure 1.8 Upper bound execution time chart of CarInfo for a single CPU	9
Figure 1.9 Optimistic time chart for CarInfo under speculative execution.....	10
Figure 2.1: Graph form of example_plan.....	20
Figure 2.2: Text form of example_plan	21
Figure 2.3: The Null operator.....	25
Figure 2.4: Task scheduling subsystem	27
Figure 2.5: Text form of parent_plan.....	28
Figure 2.6: Graph form of persistent_diff plan	29
Figure 2.7a: The query_search_engine plan	30
Figure 2.7b: The recursive subplan gather_and_follow.....	30
Figure 2.8: Text representation of CarInfo agent plan.....	31
Figure 2.9a: Homeseekers search screen	32
Figure 2.9b: Homeseekers results list screen	33
Figure 2.9c: Homeseekers detail page screen	33
Figure 2.10: Abstract Homeseekers plan	34
Figure 2.11a: Homeseekers agent plan get_houses	35
Figure 2.11b: Homeseekers recursive subplan get_urls.....	35
Figure 2.12: Detailed architecture of executor.....	38
Figure 2.13a: Text form of the Homeseekers get_houses plan.....	42
Figure 2.13b: Text form of the Homeseekers get_urls recursive subplan	43
Figure 2.14: Average Homeseekers performance results	44
Figure 2.15: Theseus vs. theoretical network query engine for Homeseekers.....	45
Figure 2.16: Graph form of the plan to monitor Homeseekers.....	46
Figure 2.17: Comparing Theseus and Telegraph performance.....	50
Figure 3.1: The CarInfo plan, modified for speculative execution	54
Figure 3.2: The Speculate operator	54
Figure 3.3: The Confirm operator	55
Figure 3.4: A longer sequence of operators	57
Figure 3.5: Cascading speculation of the sequence in Figure 3.4.....	57
Figure 3.6: CarInfo modified for cascading speculation.....	58
Figure 3.7: Short FD example flow	63
Figure 3.8: Flow in Figure 3.7 modified for speculative execution.....	64

Figure 3.9: Leveraging the determinism of the Format operator	64
Figure 3.10: Leveraging the determinism of the Project operator	65
Figure 3.11: Sample plan that meets (i), (ii), and (iii)	66
Figure 3.12a: The SPEC-REWRITE algorithm	69
Figure 3.12b: The GET-MEP-INFO helper function.....	70
Figure 3.12c: The GET-LHS-INFO helper function.....	70
Figure 3.12c: The GET-RHS-INFO helper function	71
Figure 3.12d: The CALC-FLOW-TIMES helper function.....	71
Figure 3.13a: Congress.org Web page	73
Figure 3.13b: Yahoo News Web page	73
Figure 3.13c: Open Secrets Web page	74
Figure 3.14a: The RepInfo agent plan.....	74
Figure 3.14b: The modified RepInfo agent plan.....	74
Figure 3.15a: Yahoo Movies web page	75
Figure 3.15b: Dine.com web page	75
Figure 3.15c: TIGER Mapping Service web page.....	76
Figure 3.16a: The TheaterLoc agent plan	76
Figure 3.16b: The modified TheaterLoc agent plan.....	76
Figure 3.17a: Delta Airlines web page.....	77
Figure 3.17b: US Naval Time Details web page	77
Figure 3.18a: The FlightStatus agent plan	78
Figure 3.18b: The modified FlightStatus agent plan.....	78
Figure 3.19a: MarketWatch profile page	79
Figure 3.19b: MarketWatch industry page.....	79
Figure 3.20a: The StockInfo agent plan.....	79
Figure 3.20b: The modified StockInfo agent plan	79
Figure 3.21a: Performance improvement of time to first tuple.....	81
Figure 3.21b: Performance improvement of time to last tuple	81
Figure 3.22a: Speedup increases related to the time to first tuple	82
Figure 3.22b: Speedup increases related to the time to last tuple	82
Figure 3.23: Converting a schema to a distributed database.....	84
Figure 3.24: SQL for TPC-H query #17	85
Figure 3.25: The Oracle EXPLAIN PLAN for TPC-H query #17	85
Figure 3.26: Dataflow graph of the explained plan.....	85
Figure 3.27: Theseus plan based on dataflow graph	86
Figure 3.28a: Average speedup of TPC-H queries (s=0.2, concurrency=5).....	87
Figure 3.28b: Average speedup of TPC-H queries (s=0.6, concurrency=5).....	87
Figure 3.28c: Average speedup of TPC-H queries (s=0.2, concurrency=100).....	87
Figure 3.29: Speedups obtained for TPC-H queries vs. theoretical maximums	88
Figure 4.1: Full review URL transduction is part extraction, part production.....	95
Figure 4.2: Value transducer for the full-review URL in CarInfo	96
Figure 4.3: The LEARN-VALUE-TRANSDUCER algorithm	98
Figure 4.4: The LEARN-SD-TEMPLATE algorithm	99
Figure 4.5: The LEARN-HINT-TRANSDUCER algorithm	100
Figure 4.6: Sample hint transducer for the names example	100

Figure 4.7: The Phone Info agent plan.....	104
Figure 4.8: Speculative version of PhoneInfo.....	104
Figure 4.9: Accuracy of Carsummary, Replist, and Phonestate classifiers	107
Figure 4.10: Frequency of “no predictions possible” by Carsummary	108
Figure 4.11: Impact of learning on CarInfo agent execution performance.....	111
Figure 4.12: Impact of learning on RepInfo agent execution performance	112
Figure 4.13: Impact of learning on PhoneInfo agent execution performance.....	112
Figure 5.1: Basic multiplexer logic.....	118
Figure 5.2: Verilog module that represents the multiplexer logic in Fig 5.1	119

Abstract

While information agents make it possible to gather, combine, and process data on networks like the Internet, execution performance often suffers due to remote source latencies. Agents do not control remote sources and must wait an undetermined amount of time for a query to be answered. The problem becomes worse when an agent plan requires that the answers provided by one source be used as a basis for querying another source.

In this dissertation, I make three related contributions that address these problems and significantly improve information agent performance. The first is an expressive agent plan language and a streaming dataflow execution system. The combination of both allows agent plans to be described and efficiently executed, realizing the maximum parallelism allowable by the data dependencies in the plan. My experimental results confirm that execution is efficient and that the plan language is expressive enough to support tasks beyond those supported by traditional network query engines, such as recursive information gathering and monitoring.

A second contribution is a strategy for speculative plan execution within a streaming dataflow architecture. Under speculative execution, certain operators are issued ahead of schedule, using data predicted from experience. Through this technique, remaining costly data dependencies between I/O-bound operators can be broken, leading to parallelism beyond the normal dataflow limit. My experimental results demonstrate that speculative execution can lead to significant speedups in both Web agent plans as well as certain types of queries for distributed database systems.

A third contribution is a technique for learning how to predict data for speculative plan execution. This approach combines caching with classification and transduction as a means for predicting future values from prior hints. Classification and transduction are more space efficient than caching and can improve the accuracy of prediction because each is capable of responding to new hints. The resulting improved accuracy increases the utility of speculative execution and leads to greater average plan speedups. My experimental results for a set of Web agent plans confirm these space-efficiency and accuracy claims.

Chapter 1

Introduction

As the twenty-first century begins, there is an enormous amount of information publicly accessible through electronic means. The availability of digital information networks and the ubiquity of connectivity allows people to easily access terabytes of data on almost every conceivable subject. People now regularly use networks like the Internet for a variety of daily tasks – from reading the daily newspaper, to booking travel reservations, enrolling in university courses, browsing encyclopedias, searching classified ads, donating funds to relief organizations, and monitoring stock portfolios. The enabling technology for all of this consists of a handful of communication standards and tens of thousands of internetworking devices, linking together data across all sorts of national and international boundaries.

Ever since their emergence, there has been great interest in the creation of technology capable of automatically querying information networks. In part, this is due to the vast amount of information online and the tediousness involved in some of the tasks that one would like to accomplish. In recent years, an intersection of the AI and database research communities have developed approaches to *information integration*, the combining and processing of data from multiple sources. Among other things, this research has shown that it is possible to query multiple heterogeneous sources through a unified view, to extract data from semi-structured sources such as Web sites, to deal with the unpredictability of remote sources, and to resolve semantic relationships between similar data from different sources.

Information integration techniques are often used by *information agents*, computer programs that gather, combine, and process data from one or more network sources, possibly on some regular schedule. Agents execute information gathering *plans* that can be more complicated than traditional database query plans. These plans can involve monitoring sources, interacting with local databases, providing non-interactive periodic feedback to users, and interacting with other agents. For example, it is possible to build an agent that uses online sources to continuously monitor travel status (Ambite et al. 2002), coordinate project activities (Chalupsky et al. 2001), or search for houses for sale that meet a certain search criteria (Barish and Knoblock 2002).

1.1 The Challenge of Performance

An ongoing challenge of information agent research has been improving plan execution performance. In general, performance is determined by the time it takes to execute various plan operators and the ordering constraints between these operators.

Typically, the slowest operations are those that involve gathering data from a remote source. During such instances, the agent is generally waiting for data and has little computational demands. Remote source latencies are generally greater than the time it takes to complete processor-intensive operations, such as filtering data. As a result, agent plans are often I/O-bound.

While an agent is waiting for data, local resources are being wasted. For example, a typical desktop computer today in an office environment contains a 2.66 GHz processor, 256MB of RAM, and has access to 10Mbps bandwidth. When an agent waits for data from a remote source, most of these resources – such as the processor, which can execute over 4 billion integer operations per second – sit idle.

The cost of wasting resources while waiting for data will only get worse. While Moore's Law continues to forecast the periodic doubling of computing power capable by a CPU, a trend expected to continue until at least 2013 (Moore 2003), the speed of light is not changing. Fundamental principles of physics ensure that it will always take the same minimum amount of time for a packet to travel from point A to point B.

One strategy to combat the network latencies inherent in information gathering plans is to extract as much parallelism as possible from the plan during execution. For example, plan operations that are independent can be scheduled in parallel through *dataflow-style execution*. In the dataflow model, the availability of data schedules which instruction(s) to execute next and, in theory, supports concurrent execution. Dataflow computing stands in contrast to the more well-known von Neumann model, which schedules execution using a program counter and does not, in theory, support concurrency.

Figure 1.1 illustrates the difference between the dataflow and von Neumann approaches in terms of computing the set of instructions required to process the expression $((a+b)*(b*c))$. The program requires the multiplication of two independent additions. Under the von Neumann model of execution, the ADD operations must be executed sequentially, even though they are independent of each other, because a program counter schedules only one instruction at a time. In

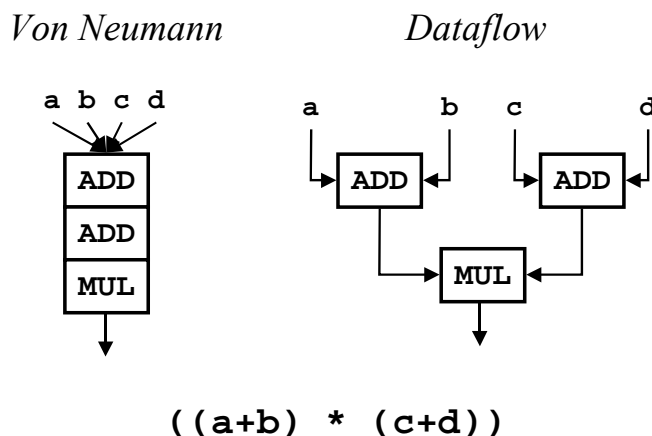


Figure 1.1: Von Neumann vs. dataflow execution

contrast, since the availability of data drives the scheduling of a dataflow machine, both ADD operations can be executed as soon as their input dependencies are fulfilled – thus, in theory, both operations can be executing at the same time.

A second method to extract parallelism from execution is to pipeline or *stream* data between plan operators that are part of the same sequential flow. Streaming allows multiple operators to concurrently process a single set of data. Figure 1.2 shows the difference between non-streaming and streaming for a single producer and a single consumer that process a small set of data. The figure shows how a non-streaming mode requires that all of the elements in a logical set of data be processed before transmitting results to any consumers. As a result, consumer operators are often idle, waiting for producers to complete processing of the entire set. In contrast, a streaming mode enables producers to emit data as soon as it is processed. Thus, consumers like Op_2 can begin subsequent processing as soon as possible.

In summary, dataflow execution maximizes the degree of *horizontal parallelism* available while data streaming maximizes the degree of *vertical parallelism* during execution. By employing these strategies, an executor can process as many independent operations in parallel on the available data, as soon as that data becomes available.

Despite the benefits of streaming dataflow, agent plan execution can still remain very slow because of data dependencies between remote sources. For example, a plan that gathers reviews of automobiles in a certain price range often must first query the source that provides the set of automobiles in the price range specified and then query the source that provides automobile reviews. The minimum execution time of this plan is the sum of the time required by a single query to each source. More complex plans usually have longer chains of dependencies. Worse, if even a single source in such a chain is slow, execution of the rest of the plan bottlenecks until the latent source responds. Such data dependencies between sources are also known as *binding patterns*.

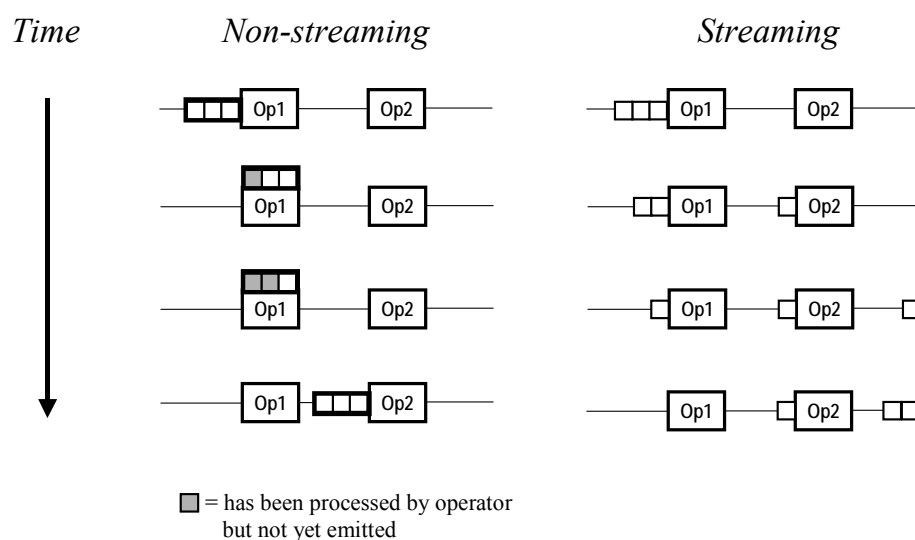


Figure 1.2: Non-streaming vs. streaming execution

In this thesis, I introduce a novel approach to agent execution that addresses the ongoing problem of poor performance and in particular the penalties of sequential execution due to binding patterns. My approach, *speculative plan execution*, combines streaming dataflow execution with a new type of parallelism – speculative parallelism – that executes plan operators ahead of their normal schedule. Speculative plan execution relies on knowledge gained from prior plan executions to predict future executions. In particular, it observes the relationships between the data consumed and produced by I/O-bound operators and reasons about data likely produced from future executions of those operators. It then uses the predicted data to trigger the early execution of operators not yet scheduled. When speculation is correct, plan speedups can be significant, up to a factor roughly equal to the length of the longest, most latent dataflow within a plan. To increase the likelihood that predictions made will be accurate, I also introduce an approach to value prediction that combines machine learning with caching to issue predictions under a variety of conditions, even when the hint driving prediction has not been seen and when the value to be predicted has never been previously predicted.

1.2 Motivating example

To better understand the challenge of efficient information agent execution, and to set the stage for an overview of the approach this thesis proposes, let us consider the details of an example Web information agent plan.

CarInfo is an agent that collects reviews and safety ratings of used cars that meet a specific set of user search criteria. The criteria is composed of car type, year of original production, and a desired price range. The user also specifies a list of car makers to avoid. Once it receives its input data, CarInfo uses a collection of Web sources to gather the appropriate results. In particular, three different Web sources are used:

- **Edmunds.com**, to get a list of used car models meeting the initial search criteria.
- **ConsumerGuide.com**, to obtain the reviews for those models.
- **NHTSA.gov** (National Highway Traffic Safety Association), for crash safety ratings of those models.

The Web pages for each of these sources is shown in Figures 1.3a-c. CarInfo first gathers the list of cars from Edmunds, filters out those automakers that the user would like to avoid (Edmunds does not allow this to be specified through its search interface), gathers the safety reports from NHTSA for the filtered set of cars, combines this result with reviews gathered at ConsumerGuide and then outputs the results. The plan for CarInfo that performs these operations is shown in Figure 1.4.

Figure 1.3a: Edmunds car search results page

Make & Model	Frontal Star Rating		Side Star Rating		R R e I
	Driver	Passenger	Front Seat	Rear Seat	
2002 Acura 3.2 CL 2-DR. w/SAB	Not Tested	Not Tested	Not Tested	Not Tested	Not Ra.
2002 Acura 3.2 TL 4-DR. w/SAB	★★★★★	★★★★★	★★★★★	★★★★★	★★
2002 Acura NSX-T Convertible	Not Tested	Not Tested	Not Tested	Not Tested	Not Ra.
2002 Audi A4 4-DR. w/SAB	★★★★★	★★★★★	★★★★★	★★★★★	★★
2002 Audi A4 Avant 4-DR. w/SAB	★★★★★	★★★★★	Not Tested	Not Tested	Not Ra.
2002 Audi A6 4-DR. w/SAB	Not Tested	Not Tested	Not Tested	Not Tested	Not Ra.
2002 Audi A6 Avant/S6 Avant 4-DR. w/SAB	Not Tested	Not Tested	Not Tested	Not Tested	Not Ra.
2002 Audi S4 4-DR. w/SAB	Not Tested	Not Tested	Not Tested	Not Tested	Not Ra.

Figure 1.3b: NHTSA safety ratings page

Note that to gather the detailed car reviews from ConsumerGuide, additional navigation is required. CarInfo must first query ConsumerGuide through its search interface to find a pointer to the summary page for that car. It then queries the summary page to find the detailed review page. Finally, it gathers the review text from the detailed review page. Engaging in additional navigation in order to extract the desired information is a common subtask for Web agents in particular, since Web sites are designed to be visually browsed and may not support the direct querying of all the information they provide.

As a detailed example of CarInfo execution, consider the case where the initial search criteria is (*Midsize sedan, year 2002 model, minimum price \$4000, maximum price \$12000*) and the cars to avoid are those by the auto maker (*Oldsmobile*).



Figure 1.3c: ConsumerGuide car reviews page

During execution, the first Wrapper operator returns (*Oldsmobile Alero*, *Dodge Stratus*, *Pontiac Grand Am*, *Mercury Cougar*). From these, filtering out of Oldsmobile models results in the subset (*Dodge Stratus*, *Pontiac Grand Am*, *Mercury Cougar*). The safety reports and full reviews of these cars are then queried. For example, for the first tuple (*Dodge Stratus*), the URL for the summary review of that car is (<http://cg.com/summ/20812.htm>) and the URL for the full review is (<http://cg.com/full/20812.htm>). Once at the full review URL, the review text can be extracted and joined with the safety report.

The CarInfo plan is one common type of information agent plan. Similar plans that extract data from two or more distinct sources and then combine them together are common throughout the literature (Friedman et al. 1999; Ives et al. 1999; Barish et al. 2000; Barish and Knoblock 2002). Like CarInfo, these plans also involve extracting and combining data from multiple sources using relational-style operations.

To understand the performance challenges of a plan like CarInfo, let us calculate the non-optimized execution time of the plan shown in Figure 1.4. As the figure shows, only one operator is executed at a time. Each operator receives its input from the previous operator (or plan input), performs its function on the set of input data, and routes the set of results to its consumer operator (or plan output).



Figure 1.4: The CarInfo plan

Let us assume that each Wrapper operation takes 1000ms per tuple and all other CPU-bound operations (i.e., Select) take 100ms per tuple. For the search criteria given earlier, the baseline execution time is $(1000 + 4 \cdot 100 + 3 \cdot 1000 + (3 \cdot 1000 + 3 \cdot 1000 + 3 \cdot 1000) =)$ 13400ms. Figure 1.5 shows the corresponding execution time chart.

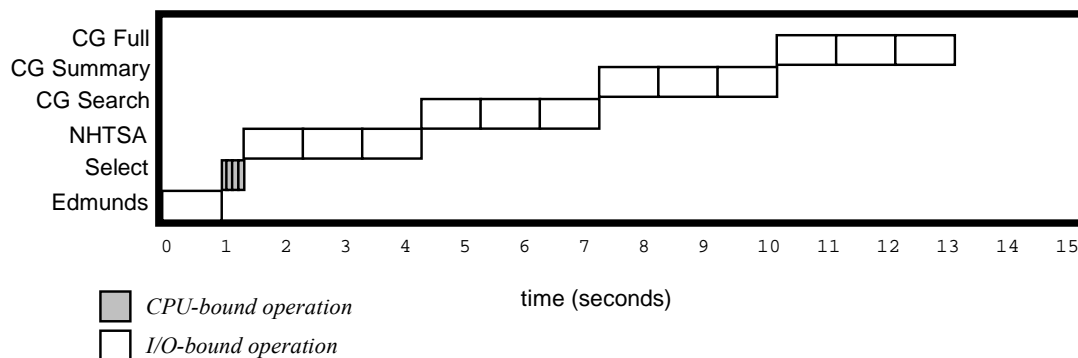


Figure 1.5: Execution time chart for CarInfo

1.3 Approach

This thesis presents an approach to agent execution that seeks to dramatically improve the performance of agents like CarInfo. The solution proposed consists of three major parts: a streaming dataflow language and execution system for information agents, a strategy for speculative execution in a streaming dataflow architecture, and an accurate and space-efficient value prediction strategy. Generally speaking, streaming dataflow increases the degree of run-time parallelism, speculative execution allows this to be potentially increased beyond the theoretical dataflow limit of the plan, and the learning increases the probability that speculative execution will realize its potential benefits. I now provide an overview of each of these components, each of which is covered in greater detail in successive chapters of this thesis.

1.3.1 Streaming dataflow plan execution

The first part of my approach involves a streaming dataflow language and execution system that is both expressive and efficient. The language supports the expression of information agent tasks that range from simple information gathering, to integration, to recursive gathering. The execution system efficiently processes agent plans through a streaming dataflow architecture, maximizing both the available horizontal and vertical degrees of parallelism.

For example, the language allows the CarInfo plan shown in Figure 1.4 to be transformed from a serial sequence of steps to a partially ordered graph of data dependent operations, shown in Figure 1.6. During execution, operators process data as it arrives (independent of any global schedule), iterating on each tuple of data, so that results can be streamed to consumers as early as possible. Note that an additional operation – a Join – has been added in order to correctly combine results

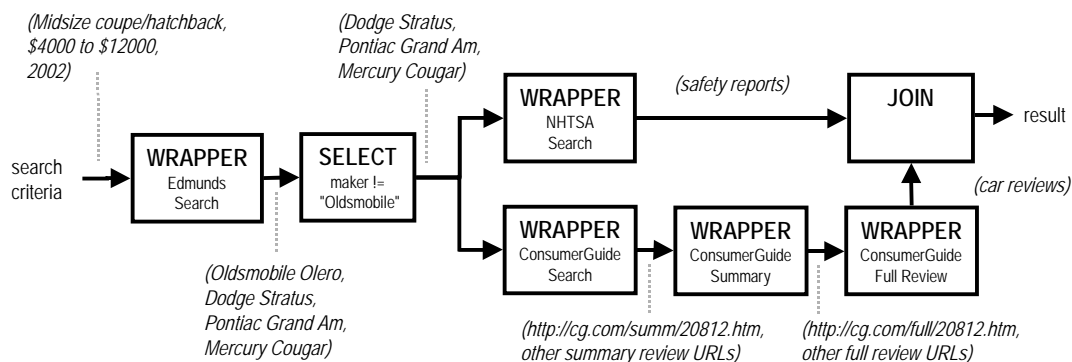


Figure 1.6 Dataflow-style version of CarInfo information agent plan

gathered in parallel from different sources. With dataflow execution, the safety report and the car reviews can be gathered in parallel. This reduces the original execution time of 13400ms to $(1*1000 + 4*100 + (3*1000 + 3*1000 + 3*1000) + 3*100 =) 10700\text{ms}$. With data streaming, there is no need for the plan to stalled at each operation for all tuples input to that operation, and the execution time can be reduced further to $(1000 + 100 + (1000 + 1000 + 3*1000) + 100 =) 6200\text{ms}$. Thus, the streaming dataflow model enables a speedup of 2.16 over the original serial, non-streaming execution model. Figure 1.7 shows the detailed time chart for the streaming dataflow version of CarInfo execution.

Note that if tuples can be independently processed by each operator and enough execution resources (threads and/or processors) exist, a true dataflow machine would not require an operator to serialize its work queue. This means that any tuple could be executed as soon as it arrives. If we think of execution as a series of time elements $E = \{E_1..E_m\}$, that each have a degree of parallelism $D(E_i)$, and the set of processors $P = \{P_1..P_k\}$, then this can only be practically true if:

$$\forall E_i \in E, k \geq \max(D(E_i)), 1 \leq i \leq m$$

When this is not the case, the true degree of parallelism depends on how well k processors use available threads to manage the additional request for parallelism. Let us simplify and assume that, for computers with a single processor, (a) a CPU-bound operator must serialize the processing of its inputs while (b) an I/O-bound operator can parallelize the processing of its inputs. In practice, use of multiple threads on modern CPUs do not starve CPUs as much as (a) while only closely

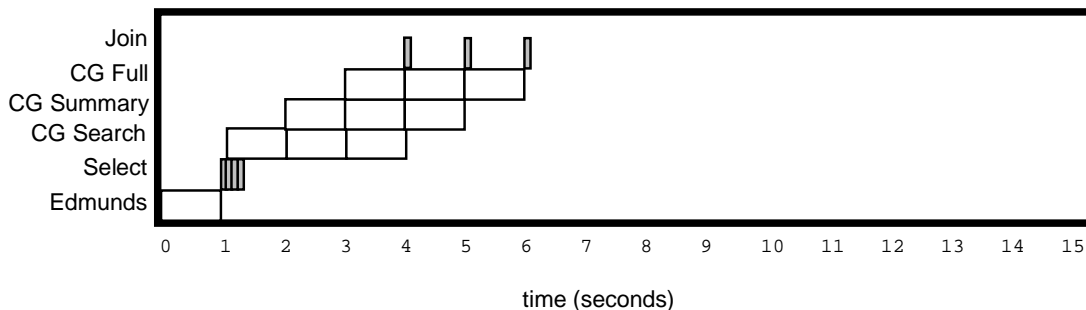


Figure 1.7 Execution time chart for CarInfo under streaming dataflow

approaching (b). With these assumptions, the execution time of CarInfo can be further reduced to 4200ms, as the execution time chart in Figure 1.8 shows.

While the above example shows that streaming dataflow can be an effective technique for significantly improving plan performance, it is important to note that the plan remains I/O bound. In particular, note that $(4000\text{ms}/4200\text{ms} =)$ over 95% of the execution time still consists of waiting for data. The key issue is the need to query sources in sequence. For example, for each tuple in CarInfo, the Edmunds query must be completed before the ConsumerGuide Search can be performed, a step that needs to be completed before the ConsumerGuide Summary query can be executed, and so on. The binding patterns between sources continues to force serialized execution of I/O-bound operations.

1.3.2 Speculative plan execution

To overcome the limits imposed by binding patterns between sources, speculative execution can be employed. The general process of speculative execution involves issuing operations ahead of their normal schedule, based on data (hints) received earlier in the plan. Because more operators can execute in parallel at a given time, with some executing speculatively, a higher degree of concurrency is possible. If predictions are correct, significant plan speedups are possible.

One way to execute the CarInfo plan would be to use the car search criteria as a hint for what kind of cars Edmunds is likely to produce (based on prior executions). We can then use these predicted results to predict what ConsumerGuide review URLs are likely and thus gather this information ahead of its normal schedule. In short, the initial search criteria can directly lead to predictions (and predictions based on predictions) that, if correct, facilitate the gathering of data based on a likely list of cars while the actual list of cars is being retrieved.

Under the scenario proposed, execution could proceed as follows. Input data, such as (*Midsize coupe/hatchback, 2002, \$4000, \$12000*), would result in the retrieval of the initial search results from Edmunds.com in parallel with the retrieval of reviews and safety ratings based on the makes and models Edmunds is predicted to return. Furthermore, these predicted makes and models would also drive the predictions of the ConsumerGuide Search and ConsumerGuide Summary URLs, significantly increasing the parallelism of plan execution. Figure 1.9 shows the corresponding theoretical execution time chart. In short, if all predictions are correct, the resulting optimistic execution time is only $(100 + 1000 + (3*100)) =$

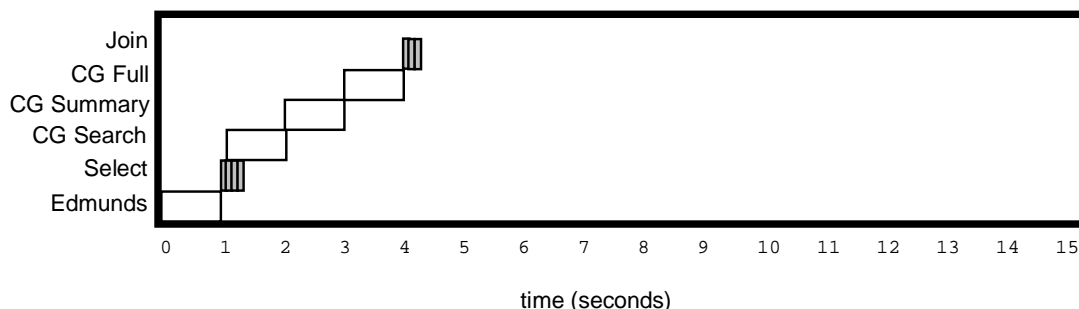


Figure 1.8 Upper bound execution time chart of CarInfo for a single CPU

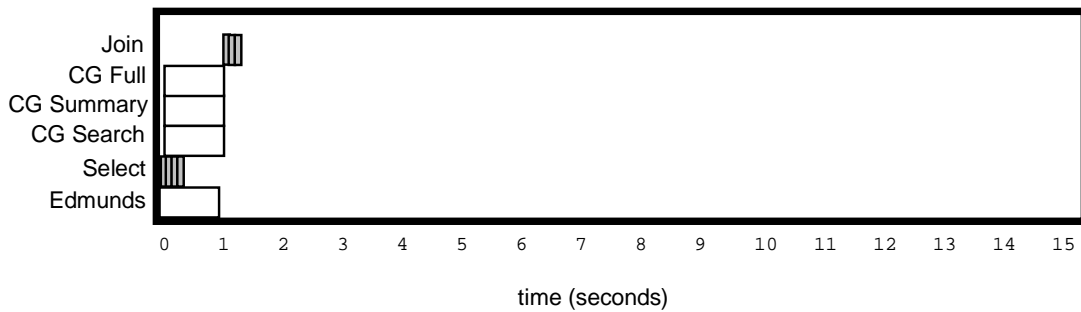


Figure 1.9 Optimistic execution time chart for CarInfo under speculative execution

1400ms plus the overhead to speculate, a potential speedup of 4.42 over the upper bound of streaming dataflow. All techniques thus combine to enable an overall speedup of $(13400\text{ms}/1400\text{ms} =)$ 9.57 in the CarInfo example.

1.3.3 Learning to predict data for speculative plan execution

The third part of my approach focuses on learning to predict data for speculative plan execution. While the average speedup due to speculative execution varies somewhat depending source latencies and speculation overhead, a more important issue is predictive accuracy. Had all predictions been incorrect, execution would have actually been slightly slower (due to overhead) than the original execution time under streaming dataflow. Thus, maintaining a high average accuracy of prediction is key to the success of speculative execution: the greater this accuracy, the higher the average speedup.

The value prediction challenge involves choosing one or more values to predict for a hint, given data on prior executions. The easiest way to do this is to cache information from prior executions. Upon future executions, the re-appearance of hints that have been seen before can be used to generate the predictions that those hints were associated with in past executions. For example, if a future execution of the CarInfo plan involved the same search criteria (*Midsize, 2002, \$4000, \$12000*) seen earlier, a caching value prediction strategy would generate the same results confirmed last execution: (*Oldsmobile Alero, Dodge Stratus, Pontiac Grand Am, Mercury Cougar*).

However, if the hint was slightly different, say (*Midsize, 2002, \$5000, \$12000*), a caching strategy would be unable to make any kind of prediction, even though it seems that (*Oldsmobile Alero, Dodge Stratus, Pontiac Grand Am, Mercury Cougar*) might still be a good prediction. Similarly, a summary URL of (<http://cg.com/summ/2978.htm>) that had not been seen in earlier executions would not be associated with any prior cached prediction, and thus speculative execution would not be possible. This is unfortunate, since from even a single past execution, intuition suggests that a good prediction would be (<http://cg.com/full/2978.htm>).

To address these issues, and to make value prediction more accurate and possible more often, my approach includes a set of algorithms that use machine learning techniques to induce value predictors that combine caching with classification and transduction. By integrating classification into a value prediction scheme, we can learn which features of the hint are most telling of the likely

subsequent result – thus, we can learn that a slight alteration in the minimum price does not make any difference as far as the likely set of cars that will be returned. By integrating transduction into the value prediction process, we can learn how to extract and manipulate the key piece of dynamic information from a hint, so that we can learn the general process of data transformation.

Classification and transduction not only allow the system to make recurring predictions based on recurring hints, but also to make *recurring predictions based on new hints*, and *novel predictions based on new hints*. Both features increase the accuracy of value prediction. Furthermore, the hybrid approach to value prediction I describe is almost always more space efficient than caching. Classifiers need only to store the values for the features that distinguish one prediction from another (as well as discretize continuous values, instead of storing all of them). Transducers are even more efficient: once learned, a transducer is small and its size remains finite because it is a function valid for all examples.

1.4 Thesis statement

In summary, this thesis introduces a technique for speculative plan execution in a streaming dataflow architecture that significantly increases the runtime performance of information agent plans. The thesis of this dissertation is:

Speculative execution of streaming dataflow plans increases the degree of runtime parallelism realizable by information agents, leading to better average execution performance.

1.5 Contributions

Overall, in this thesis, I make following contributions:

1. **An efficient language and execution system for information agent plans.** The language allows complicated information agent tasks to be easily specified through its support for modularity, recursion, and extensibility. The execution system allows true streaming dataflow execution to occur: operators are scheduled independently and producers asynchronously transmit tuples to consumers as soon as possible. A thread pooling approach is used to obtain significant degrees of horizontal and vertical parallelism without exhausting resources and without repetitive overhead costs for thread creation.
2. **An approach for speculative plan execution that yields arbitrary speedups while ensuring safety and fairness.** I introduce algorithms that automatically transform any information gathering plan into one capable of speculative execution. I present empirical results of speculative plan execution to a set of common information agent plans as well as plans generated by queries contained in a well-known database benchmark standard.
3. **An approach to value prediction that combines caching, classification, and transduction to yield predictors that are accurate, generally applicable after only a few examples, and space efficient.** I introduce algorithms that build hybrid predictors by identifying patterns in the data being predicted and identify the right general strategy to adopt for future predictions.

In addition to being able to re-issue predictions for recurring hints, the value predictors learned can generate predictions for new hints and can synthesize novel predictions (manufacture data not yet seen).

1.6 Thesis organization

The remainder of this thesis is organized as follows. In chapter 2, I describe a framework for expressive and efficient information gathering through a novel agent language and an execution system based on a streaming dataflow model. In chapter 3, I show how the streaming dataflow architecture can be augmented to automatically support speculative plan execution, in a manner that is safe, fair, and transparent to the end-user. In chapter 4, I introduce a technique for value prediction that combines caching, classification, and transduction to issue more accurate predictions, which in turn lead to greater average plan speedups. In chapter 5, I survey the related work. Finally, in chapter 6, I conclude and discuss directions for future work.

Chapter 2

Efficient execution of information agent plans

In this section, I describe a novel language and execution system for information agents. Both the language and execution system are an important foundation for speculative plan execution. The language allows agent plans to be easily described in a way that facilitates efficient execution. The executor implements a true streaming dataflow architecture and maximizes the amount of run-time horizontal and vertical parallelism possible.

This section is organized as follows. I first motivate the need for an expressive information agent plan language, and in particular how information networks like the Web involve tasks that are usually more complicated than traditional database-style queries. I then describe the plan language in detail, focusing on the features that promote expressivity. Next, I introduce the detailed design for a streaming dataflow execution system that can process information agent plans. Finally, I present evidence of how the combination of the language and execution system effectively marry the efficiency of modern network query engine with the generality and flexibility of a traditional agent executor.

2.1 The nature of information agent plans

Information agent tasks can include fetching data from one or more sources, integrating data, filtering and other types of data processing, and communicating data to end-users. Agent plans are usually more complex than traditional database query plans for a number of reasons. Unlike database queries, agent plans are often non-interactive, instead providing periodic notification to end-users. Agent plans also may integrate data from heterogeneous types of sources, instead of combining tables from a collection of structured relational databases. Control flow might also be more complex: for example, agent plans may loop over a set of tasks, whereas a database query plan is a set of flows executed once.

To better understand the challenges of Web information gathering, let us consider automating various types of online tasks. The Web is a useful example because while in some respects it is just a large distributed database, in other respects it is not. For one, the Web is federated: one does not have administrative control over all sources. Second, most Web data sources are semi-structured: the desired data is embedded within Web pages that were originally meant for visual consumption.

2.1.1 Example Web information gathering tasks

One type of difficult Web information gathering task involves interleaved gathering and navigation. For the benefit of people that use a Web browser to access online data, many Web sources display large sets of query results spread over a series of Web pages connected through “Next Page” links. For example, querying an online classified listings for automobiles for sale can generate many results. Instead of displaying the results on a single very long Web page, many classified listings sites group sets of results over series of hyperlinked pages. In order to automatically collect this data, a system needs to interleave navigation and gathering: that is, it needs to collect results from a given page, navigate to the next, gather the next set of results, navigate, and so on. While there has been some work addressing how to theoretically incorporate navigation into the gathering process (Friedman et al. 1999), no attention has been given to the efficient execution of plans that engage in this type of interleaved retrieval.

A second example has to do with monitoring a Web source. Since the Web does not contain a built-in trigger facility, one is forced to manually check sources for updated data. When updates are frequent or the need to identify an update immediately is urgent, it becomes desirable to automate the monitoring of these updates, notifying the user when one or more conditions are met. For example, suppose we want to be alerted as soon as a particular type of used car is listed for sale by one or more online classified ad sources. Repeated manual checking for such changes is obviously tedious. Mediators and network query engines can automate the query, but additional software in programming languages such as Java or C must be written to handle the monitoring process itself, something that requires conditional execution, comparison with past results, possible notification of the user, and other such actions. Even network query engines that support continuous queries, such as NiagaraCQ (Chen et al. 2000), do not provide much flexibility in how monitoring is to be done and also require that the user program actions using a separate stored procedure language, which does not necessarily run as efficiently as the query plan. In short, no existing systems support a rich enough query language make tasks like monitoring simple enough to integrate directly into the query plan.

These examples show that automatically querying the Web can be difficult and beyond the capabilities of existing systems. Even when some type of integrated automation is possible, overall execution is inefficient. The root of the problem is the lack expressivity provided by traditional query languages. More complicated tasks, such as those described here and in the Electric Elves project (Chalupsky et al. 2001; Ambite et al. 2002), usually involve actions beyond those needed for merely querying (i.e., beyond filtering and combining) – they require agents that are capable of a variety of actions, such as conditional execution, integration with local databases, and asynchronous notification (e.g., e-mail or FAX) to users. At the same time, efficiency is also important, so that information from large and/or slow data sources can be processed as fast as possible.

One possible alternative to existing Web query systems is to use more general robot or agent executors, such as the RAP system (Firby 1994) or PRS-Lite (Myers 1996). These systems are attractive because each offers support for more expressive

plans that are still capable of executing concurrent actions. For example, PRS-Lite supports conditional actions via the IF goal and concurrent execution via the AND and SPLIT goal modalities. Unfortunately, because they are not designed to efficiently route large sets of information (such as relations) between plan operators, these systems are of limited use in the domain of Web information gathering. For example, one way that network query engines combat Web source latencies is to stream information between operators – in this way, data that trickles in from a slow source can be processed in parallel and delivered to the user as fast as possible. While a substantial amount of previous work has gone into building operators that exploit this potential for database systems (Wilschut and Apers 1993) and network query engines (Ives et al. 2002), more general executors like the RAP system and PRS-Lite do not offer such support. Thus, while existing agent execution systems are attractive because of support for more expressive plans, these systems are not designed for high-performance information gathering and are inherently less efficient solutions.

2.2 Background

The plan language and executor that is the focus of this chapter builds on a foundation of prior research related to dataflow computing and Web information integration. Although seemingly orthogonal disciplines, they are effective complements in that the parallelism and asynchrony provided by dataflow computing lends itself to the performance problems associated with Web information gathering.

2.2.1 Dataflow Computing

The pure dataflow model of computation was first introduced by (Dennis 1974) as an alternative to the standard von Neumann execution model. Its foundations share much in common with past work on computation graphs (Karp and Miller 1955), process networks (Kahn 1974), and communicating sequential processes (Hoare 1978). Dataflow computing has a long theoretical and experimental history, with the first machines being proposed in the early 1970s and real physical systems being constructed in the late 1970s and throughout the 1980s and early 1990s (Arvind and Nikhil 1990; Papadopoulos and Traub 1991; Gurd and Snelling 1992).

The dataflow model of computation describes program execution in terms of data dependencies between instructions. A dataflow graph is a directed acyclic graph (DAG) of nodes and edges. The nodes are called actors. They consume and produce data tokens along the edges that connect them to other actors. All actors run concurrently and each is able to execute, or fire, at any time after its input tokens arrive. Input tokens can come from initial program input or as a result of earlier execution (i.e., the output of prior actor firings). The potential overall concurrency of execution is thus a function of the data dependencies that exist in the program, a degree of parallelism referred to as the *dataflow limit*.

The key observation to be made about dataflow computing that execution is inherently parallel – actors function independently (asynchronously) and fire as necessary. In contrast, the von Neumann execution model involves the sequential processing of a pre-ordered set of instructions. Thus, execution is inherently serial.

When comparing dataflow to von Neumann, a more subtle difference (yet one at the heart of the distinction between the two) to be noted is that the scheduling of instructions is determined at run-time (i.e., dynamic scheduling), whereas in a von Neumann system it occurs at compile-time (i.e., static scheduling).

Dataflow systems have evolved from the classic static (Dennis 1974) model to dynamic tagged token models (Arvind and Nikhil 1990) that allowed multiple tokens per arc, to hybrid models that combine von Neumann and traditional dataflow styles of execution (Iannucci 1988; Evripidou and Gaudiot 1991; Gao 1993). Other models that have been applied to digital signal processing include boolean dataflow and synchronous dataflow (Lee and Messerschmitt 1987), resulting in architectures known as “dataflow networks”. The work described in this paper is most relevant to a specific hybrid dataflow approach, known as threaded dataflow (Papadopoulos and Traub 1991), which maintains a data-driven model of execution but associates instruction streams with individual threads that execute in a von Neumann fashion. It is distinct from pure von Neumann multithreading in the sense that data, not an instruction counter, remains the basis for scheduling instructions (operators). But it is also distinct from pure dataflow in the sense that execution of instruction streams is a statically scheduled sequential task, unlike the typical dynamic scheduling found in dataflow machines. As a result, threaded dataflow can also be viewed as data-driven multithreading.

Recent advances in processor architecture, such as the Simultaneous Multithreading (SMT) project (Tullsen et al. 1995) have demonstrated the benefits of data-driven multithreading. SMT-style processors differ from conventional CPUs (such as the Intel Pentium) by partitioning on-chip resources so that multiple threads can execute concurrently, making better use of available functional units on the same amount of chip real estate. The resulting execution reduces “vertical waste” (the wasting of cycles) that can occur when a sequence of instructions is executed using only one thread, as well as “horizontal waste” (the wasting available functional units) that can occur when executing multiple threads. To do so, the technique effectively trades instruction-level parallelism (ILP) benefits for thread-level parallelism (TLP) benefits. Instead of having a deep processor pipeline (which becomes less useful as its depth increases), SMT processors contain multiple shorter pipelines, each associated with a single thread. The result can, for highly parallel applications, substantially improve the scheduling of on-chip resources that, on conventional CPUs, would normally be starved as a result of both I/O stalls as well as thread context-switching.

The work described here applies a threaded dataflow design to a higher level of execution – the information agent plan level. Instead of executing fine-grained instructions, we are interested in the execution of coarse-grained operators. Still, threaded dataflow is generally an efficient strategy for executing I/O-bound information gathering plans that integrate multiple remote sources because it allows coarse-grained I/O requests (such as network requests to multiple Web sources) to be automatically scheduled for parallel execution. Such plans are similar to other systems that maintain high degrees of concurrent network connections, such as a Web server or database system. Prior studies on such Web servers (Redstone et al.

2000) and database systems (Lo et al. 1998) have already shown that such systems run very efficiently on SMT-style processors; I believe the same will hold true for the execution of dataflow-style information gathering plans.

2.2.2 Web-based information gathering and integration

Generic information integration systems (Chawathe et al. 1994; Arens et al. 1996; Levy et al. 1996; Genesereth et al. 1997) are concerned with the problem of allowing multiple distributed information sources to be queried as a logical whole. These systems typically deal with heterogeneous sources – in addition to traditional databases, they provide transparent access to flat files, information agents, and other structured data sources. A high-level domain model maps domain-level entities and attributes to underlying sources and the information they provide. An information mediator (Wiederhold 1996) is responsible for query processing, using the domain model and information about the sources to compile a query plan. In traditional databases, query processing involves three major phases: (a) parsing the query, (b) query plan generation and optimization and (c) execution. Query processing for information integration involves the same phases but builds upon traditional query plan optimization techniques by addressing cases that involve duplicate, slow, and/or unreliable information sources.

Web-based information integration, such as that described in (Knoblock et al. 2001), differs from other types of information integration by focusing on the specific case where information sources are Web sites. This adds two additional challenges to the basic integration problem: (1) that of retrieving structured information (i.e., a relation) from a semi-structured source (Web pages written in HTML) and (2) querying data that is often organized to facilitate human visual consumption, and not necessarily in a strictly relational manner (e.g., a single logical table may be associated with multiple Web pages). To address the first challenge, Web site wrappers are used to convert semi-structured HTML into structured relations, allowing Web sites to be queried as if they were databases. Wrappers take queries (such as those expressed in a query language like SQL) and process them on data extracted from a Web site, thus providing a transparent way of accessing unstructured information as if it were structured. Wrappers can be constructed manually or automatically, the latter using machine learning techniques (Kushmerick 2000; Knoblock et al. 2002; Muslea 2002). While wrappers can be used to extract data from many Web sites, other sites are problematic because of how the data to be extracted is presented. One common case is where the Web site distributes a single logical relational answer over multiple physical Web pages, such as in the case of the online classified ads example described earlier. Automating interleaved navigation with gathering has received considerably less attention. One exception is (Friedman et al. 1999), which describes how to extend traditional query answering for information integration systems to incorporate the capability for navigation. However, that solution addresses the query processing phase and thus it remains an open issue regarding how to execute these types of information gathering plans efficiently.

A more recent technology for querying the Web is the network query engine (Ives et al. 1999; Hellerstein et al. 2000; Naughton et al. 2001). While these systems are, like mediators, capable of querying multiple Web sources, there has been a greater focus on the challenges of efficient query plan execution, robustness in the face of network failure or large data sets, and operators for processing XML. Many network query engines rely on adaptive execution techniques, such as dynamic reordering of tuples among query plan operators (Avnur and Hellerstein 2000) and the double pipelined hash join (Ives et al. 1999), to overcome the inherent latency and unpredictable availability of Web sites.

An important aspect of network query engine research has been its focus on dataflow-style execution. Research on parallel database systems has long regarded dataflow-style query execution efficient (Wilschut and Apers 1993). However, when applied to the Web, dataflow-style processing can yield even greater speedups because (a) Web sources are remote, so the base latency of access is much higher than that of accessing local data and (b) Web data cannot be strategically pre-partitioned, such as in the case of shared-nothing designs (Dewitt and Gray 1992). Thus, because the average latency of Web data access is high, the parallelizing capability of dataflow-style execution is even more compelling than it is for traditional parallel database systems because the potential speedups are greater.

2.3 An Expressive and Efficient Agent Plan Language

In this section, I present an information gathering plan language that goes beyond what traditional database query languages allow and makes it possible to describe a wide variety of information gathering tasks.

2.3.1 Plan representation

In the agent plan language, *plans* are textual representations of dataflow graphs describing a set of *input data*, a series of operations on that data (and the intermediate results it leads to), and a set of *output data*. As discussed earlier, dataflow is a naturally efficient paradigm for information gathering plans. Graphs consist of a set of operator sequences called *flows* where plan input data is iteratively processed by a succession of operators until further propagation of data halts (either because the flow stops or because the last operator produces one of the plan outputs). Formally, I define the following:

Definition 1: An information gathering plan P is described as a directed acyclic graph (DAG) where a set of operators Ops are the nodes that are connected through a set of variables $Vars$ that are the edges. Furthermore, a subset of $Vars$ are plan input variables $PlanIn$ and another subset of variables are plan output variables $PlanOut$. More specifically, let a plan P be represented as the tuple

$$P = \langle Vars, Ops, PlanIn, PlanOut \rangle$$

where

$$\begin{aligned} Vars &= \{v_1, \dots, v_n\}, n > 0 \\ Ops &= \{Op_1, \dots, Op_m\}, m > 0 \end{aligned}$$

$$PlanIn = \{v_{a1}, \dots, v_{ax}\}, x > 0, s.t. \{v_{a1}, \dots, v_{ax}\} \in Vars$$

$$PlanOut = \{v_{b1}, \dots, v_{by}\}, y \geq 0, s.t. \{v_{b1}, \dots, v_{by}\} \in Vars$$

Definition 2: Each operator Op encapsulates a function $Func$ that computes a set of output variables $OpOut$ from a set of input variables $OpIn$. More specifically, let each operator Op_i in P be represented as the tuple

$$Op_i = \langle OpIn, OpOut, Func \rangle$$

where

$$OpIn = \{v_{i1}, \dots, v_{ic}\}, c > 0, s.t. \{v_{i1}, \dots, v_{ic}\} \in Vars$$

$$OpOut = \{v_{o1}, \dots, v_{og}\}, g \geq 0, s.t. \{v_{o1}, \dots, v_{og}\} \in Vars$$

$$Func = \text{Function that computes } \{v_{o1}, \dots, v_{og}\} \text{ from } \{v_{i1}, \dots, v_{ic}\}$$

Definition 3: The schedule of execution for any operator Op_i is described by a firing rule Ψ_i that depends on $OpIn$, an optional second set of input wait variables $OpWait$, and results in the generation of $OpOut$ and an optional second set of output $OpEnable$ variables. The initial firing of an operator is conditional on the availability of at least one of $OpIn$ and all of $OpWait$. After the initial firing, any $OpEnable$ variables are also produced. All other $OpOut$ variables are produced in accordance with the semantics of the operator. More specifically, let us define:

$$\Psi_i (Op_i) = \langle OpIn, OpWait, OpOut, OpEnable \rangle$$

where

$$OpWait = \{v_{w1}, \dots, v_{wd}\}, d \geq 0, s.t. \{v_{w1}, \dots, v_{wd}\} \in Vars$$

$$OpEnable = \{v_{e1}, \dots, v_{eh}\}, h \geq 0, s.t. \{v_{e1}, \dots, v_{eh}\} \in Vars$$

Operators have a predefined number of input and output *slots*. When specified in a plan, input and output variables map directly to these slots. In addition, operator slots can support a variable number of arguments. For example, a standard Union operator would have two standard input slots, *lhs* and *rhs*, and a standard output slot named *out*. In a plan, a particular instance of Union might be $Op_{union} = \langle (x,y), \emptyset, (z), \emptyset, Union \rangle$. Argument matching is positional, thus x maps to *lhs*, y maps to *rhs*, and z maps to *out*. However, consider a VarUnion operator that unions a variable number of inputs to produce a single output. In this case, the operator would have a single variable argument input slot and a single standard output slot. For example, an instance might be $Op_{varunion} = \langle ((a,b,c,d)), \emptyset, (z), \emptyset, VarUnion \rangle$ or $Op_{varunion} = \langle ((a,b,x)), \emptyset, (z), \emptyset, VarUnion \rangle$. Both are legal since variable argument input slots can accept one or more variables. The operator implementations handle the management of variable arguments.

Wait and enable variables are useful synchronization mechanisms that allows operator execution to be conditional beyond its normal set of input data variables. Before I describe how, let us first distinguish between a standard data variable and a synchronization variable. A standard data variable is one that contains information that is meant to be interpreted, or more specifically, processed by an operator. For example, $PlanIn$, $PlanOut$, $OpIn$, and $OpOut$ all consist of normal data variables. A

synchronization variable is one that consists of data not meant to be interpreted – rather, such variables are used as additional conditions to execution. Since control in dataflow systems is driven by the availability of data, synchronization variables in dataflow style plans are useful because they provide more control flow flexibility. For example, if a certain static operation should occur each time a given data flow is active, synchronization variables allow us to declare such behavior.

The *OpWait* and *OpEnable* variable sets are the only ones that can contain synchronization variables. These sets are not part of an operator’s definition – they are only relevant to a particular instance of an operator within a particular plan. Plan operators that have wait variables, contained in the *OpWait* set, cannot execute until all of them have been received. After it executes (i.e., following the iterative processing of the last input tuple), an operator then produces all of its enable variables (if any), contained in the *OpEnable* set. Enable variables are exclusively synchronization variables: they are only consumed by downstream operators as wait variables for purposes of conditional execution. In contrast, the set of wait variables can contain a mix of standard data variables and synchronization variables. Thus, it is possible for the data output by an operator can be consumed as a wait variable by another downstream operator: in effect, the execution of that downstream operator is conditional on the production of output by the upstream operator.

Example plan

To better illustrate the representation of a plan, let us consider an example. Figure 2.1 illustrates the dataflow graph form of a plan named *example_plan*. It shows that the plan consists of six nodes (operators) connected with a set of edges (variables). As defined earlier, each operator instance consumes one or more inputs and produces zero or more outputs. The plan appears as a standard dataflow graph, except that one arc (from Op3 to Op5) is denoted as a synchronization arc.

Figure 2.2 shows the text form of the same plan. As shown, a header part consists of the name of the plan (*example_plan*), the set of input variables (*a* and *b*), and the set of output variables (*g*). The body section of the plan contains the set of operators. The set of inputs for each operator appears to the left of the colon delimiter and the set of outputs appears to the right of the delimiter. Wait and enable variables are denoted within curly braces that follow an operator. Recall that, like output variables, wait and enable variables are optional and are thus not necessary part of an operator declaration.

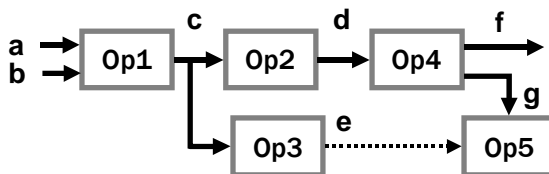


Figure 2.1: Graph form of *example_plan*

```

PLAN example_plan
{
  INPUT: a, b
  OUTPUT: f

  BODY
  {
    Op1 (a, b : c)
    Op2 (c : d)
    Op3 (c : ) {ENABLE: e}
    Op4 (d : f, g)
    Op5 (g : ) {WAIT : e}
  }
}

```

Figure 2.2: Text form of *example_plan*

Both the graph and text forms of the example plan describe the following execution. Variables *a* and *b* are plan input variables. Together, they trigger the execution of Op1, which produces variable *c*. Op2 fires when *c* becomes available, and this leads to the output of variable *d*. Op3 also fires upon the availability of *c* and produces the synchronization variable *e*. Op4 uses *d* to compute *f* (the plan output variable) and *g*. Finally, the availability of *g* and the synchronization variable *e* triggers the execution of Op5.

Note that although the body part of the text form of the plan lists operators in a linear order, this does not affect when those operators are actually executed. Per the dataflow model of processing, operators fire whenever their individual data dependencies are fulfilled. For example, although Op3 follows Op2 in the order specified by the plan text, it actually executes at the same logical time as Op2. Also note that plan output, *f*, can be produced while the plan is still running (i.e., while Op5 is still processing).

2.3.2 Data structures

Operators process and transmit data in terms of relations. Each relation *R* consists of a set of attributes (i.e., columns) $a_1..a_c$ and a set of zero or more tuples (i.e., rows) $t_1..t_r$, each tuple t_i containing values $v_{i1}..v_{ic}$. We can express relations with attributes and a set of tuples containing values for each of those attributes as:

$$R(a_1, \dots, a_c) = \{ (v_{11}, \dots, v_{1c}), (v_{21}, \dots, v_{2c}), \dots, (v_{r1}, \dots, v_{rc}) \}$$

Each attribute of a relation can be one of five types: *char*, *number*, *date*, *relation* (embedded), or *document* (i.e., a DOM object).

Embedded relations (Schek and Scholl 1986) within a particular relation R_x are treated as opaque objects v_{ij} when processed by an operator. However, when extracted, they become a separate relation R_y that can be processed by the rest of the system. Embedded relations are useful in that they allow a set of values (the non-embedded objects) to be associated with an entire relation. For example, if an operator performs a COUNT function on a relation to determine the number of tuples

contained in that relation, the resulting tuple emitted from the operator can consist of two attributes: (a) the embedded relation object and (b) the value equal to the number of rows in that embedded relation. Embedded relations thus allow sets to be associated with singletons, rather than forcing a join between the two. In this sense, they preserve the relationship between a particular tuple and a relation without requiring the space for an additional key or the repeating of data (as a join would require).

XML data is supported through the document attribute type. XML is one type of document specified by the Document Object Model (DOM). The proposed language here contains specific operators that allow DOM objects to be converted to relations, for relations to be converted to DOM objects, and for DOM objects that are XML documents to be queried in their native form using XQuery. Thus, the language supports the querying of XML documents in their native or flattened form.

2.3.3 Plan Operators

The available operators in the plan language represent a rich set of functions that can be used to address the challenges of more complex information gathering tasks, such as monitoring. Specifically, the operators support the following classes of actions:

- **Data gathering:** retrieval of data from both the network and from traditional relational databases, such as Oracle or DB2.
- **Data manipulation:** including standard relational data manipulation, such as Select and Join, as well as XML-style manipulations such as XQuery.
- **Data storage:** the export and updating of data in traditional relational databases.
- **Conditional execution:** routing of data based on its contents at runtime.
- **Asynchronous notification:** communication of intermediate/periodic results through mediums/devices where transmitted data can be queued (e.g., e-mail).
- **Task administration:** the dynamic scheduling or unscheduling of plans from an external task database.
- **Extensibility:** the ability to embed any special type of computation (single-row or aggregate) directly into the streaming dataflow query plan.

Though operators differ on their exact semantics, they do share some similarities in how they process input and generate output. In particular, there are two modes worth noting: the automatic joining of output to input (a dependent join) and the packing/unpacking (embedding/extracting) of relations.

In information gathering plans, it is common to use data collected from one source as a basis for querying additional sources. Later, it often becomes desirable to associate the input to the source with the output it produces. However, doing this join as a separate step can be tedious because it requires the creation of another key on the existing set of data and then the cost of a join. To simplify plans and improve the efficiency of execution, many of the operators in the language perform a

dependent join of input tuples onto the output tuples that they produce. A dependent join simply combines the contents of the input tuple with any output tuple(s) it generates, preserving the parity between the two. For example, the operator ROUND converts a floating point value in a column to its nearest whole integer value. Thus, if the input data consisted of the tuples $((Jack, 89.73), (Jill, 98.21))$ then the result after the ROUND operator executes would be of $((Jack, 89.73, 90), (Jill, 98.21, 98))$. Without a dependent join, a primary key would need to be added (if one did not already exist) and then a separate join would have to be done after the ROUND computation. Thus, dependent joins simplify plans – they reduce the total number of operators in a plan (by reducing the number of decoupled joins) and eliminate the need to ensure entity integrity prior to processing.¹

Another processing mode of operators involves the packing and unpacking of relations. These operations are relevant in the context of embedded relations. Instead of creating and managing two distinct results (which often need to be joined later), it is cleaner and more space-efficient to perform a dependent join on the packed version of an input relation with the result output by an aggregate-type operator. For example, when using an AVERAGE operator on the input data above, the result after a dependent join with the packed form of the original relation would be: $((((Jack, 89.73), (Jill, 98.21)), 93.97))$. Unpacking would be necessary to get at the original data. In short, embedded relations make it easy to associate aggregates with the values that led to their derivation. Packing and unpacking are useful data handling techniques that facilitate this goal.

Tables 2.1a-e shows the entire set of operators in the proposed language, grouped by function. Some of these (such as Select and Join) have well-known semantics (Abiteboul et al. 1995) and are used in other database and information gathering systems. As a result, I will not discuss them here in any detail. However, many of the operators are new and provide the ability to express more complicated types of plans. I now focus on the purpose and mechanics of some of these other types of operators.

Operator	Purpose
Wrapper	Fetches and extracts data from web sites into relations.
DbQuery	Queries local database relations using SQL.

Table 2.1a: Data gathering operators

¹ Entity integrity refers to the existence of a primary key in the relation being processed. A relation that will be joined with results generated by that relation (as input to an operator) requires a primary key in order for the join to be correct. Thus, without the dependent join, some relations may have to be “pre-tagged” with a primary key (such as a row ID) before being processed by an operator.

<i>Operator</i>	<i>Purpose</i>
Select	Filters data from a relation.
Project	Filters attributes from a relation.
Join	Combines data from two relations, based on a specified condition.
Union	Performs a set union of two relations.
Intersect	Finds the intersection of two relations.
Minus	Subtracts one relation from another.
Distinct	Returns tuples unique across one or more attributes.
GroupBy	Groups tuples by attributes and any selected aggregate measures
Pack	Embeds a relation within a new relation consisting of a single tuple.
Unpack	Extracts an embedded relation from tuples of an input relation.
Format	Generates a new formatted text attribute based on tuple values.
Rel2xml	Converts a relation to an XML document.
Xml2rel	Converts an XML document to a relation.
XQuery	Queries an XML document attribute of tuples of an input relation using language specified by the Xquery standard, returning an XML document result attribute contained in the tuples of the output relation.

Table 2.1b: Data manipulation operators

<i>Operator</i>	<i>Purpose</i>
DbAppend	Appends a relation to an existing table – creates the table if none exists.
DbExport	Exports a relation to a single table.
DbUpdate	Executes a SQL-style update query; no results returned.

Table 2.1c: Data storage operators

<i>Operator</i>	<i>Purpose</i>
Null	Conditionally routes one of two streams based on existence of tuples in a third

Table 2.1d: Conditional execution operator

<i>Operator</i>	<i>Purpose</i>
Email	Uses SMTP to communicate an email message to a valid email address.
Phone	Sends a text message to a valid cell phone number.
Fax	Faxes data to a recipient at a valid fax number.

Table 2.1e: Asynchronous notification operators

<i>Operator</i>	<i>Purpose</i>
Schedule	Adds a task to the task database with scheduling information.
Unschedule	Removes a task from the database.

Table 2.1f: Task administration operators

<i>Operator</i>	<i>Purpose</i>
Apply	Executes a user-defined function on each tuple of a relation.
Aggregate	Executes a user-defined function on an entire relation.

Table 2.1g: Extensibility operators

Interacting with local databases

There are two major reasons why it is useful to be able to interact with local database systems during plan execution. One reason is that the local database may contain information to be integrated with other online information. A second reason has to do with the ability for the local database to act as “memory” for plans that run continuously or when a plan run at a later time needs to use the results of a plan run at an earlier time.

To address both needs, the database operators DbImport, DbQuery, DbExport, and DbAppend are provided. A common use for these operators is to implement a monitoring-style query. For example, suppose we wish to gradually collect data over a period of time. To accomplish this, DbQuery can be used to bring previously queried data into a plan so that it can be compared with newly queried data (gathered by a Wrapper operator) by using any of the set-theoretic operators (such as Minus) and the result or difference can be written back to the database through DbAppend or DbExport.

Supporting conditional execution

Conditional execution is important for plans that need to perform different actions for data based on the run-time value of that data. To analyze and conditionally route data in a plan, the language supports the Null operator. Null acts as a switch, conditionally routing one set of data based on the status of another set of data.

For example, suppose it is desired to have stock quotes automatically communicated to a user every 30 minutes. Normally, quotes should be retrieved and then e-mailed. However, if the percentage price change of any stock in the portfolio is greater than 20%, then all quotes should be sent via cell phone messaging (since such communication can be more immediate). Null would be useful in such a case because it would allow a Select condition to process the check on price changes and – if there exist tuples that match the filtering criteria – allow that data to trigger an operator that communicated those results via cell phone. Otherwise, Null would route the data to an operator that communicated the information via e-mail. In short, Null is powerful because it is a dynamic form of conditional execution in that it can be used with other operators (like Select) to activate/deactivate flows based on the runtime content of the data.

The input and output to Null is summarized in Figure 2.3. The input is data to be analyzed d , data to be forwarded upon true (null) dt , and the data to be forwarded upon false df . If d is null (i.e., contains zero tuples), then dt is copied as output



Figure 2.3: The Null operator

variable t . Otherwise, df is copied as output f . For example, if d contains three tuples $\{x_1, x_2, x_3\}$ and if dt contains five tuples $\{t_1, t_2, t_3, t_4, t_5\}$ and df contains two tuples $\{f_1, f_2\}$, then only a variable f containing $\{f_1, f_2\}$ is output. Consumers of t will never receive any data.

Calling user-defined functions

Because of the wide variety of tasks they perform, there are times when agents need to execute some special logic (e.g., business logic) beyond that supported by the operators listed in Table 2.1 during execution. Often, this logic does not involve relational information processing and the plan writer simply wants to be able to code in a standard programming language (such as Java or C). For example, some of the plans written for the Electric Elves travel agents (Ambite et al. 2002) required the agent to send updates to users via the DARPA CoAbs Agent Grid (Thompson et al. 1999). Other plans required normalization date strings formats produced by different Web sources. Instead of expanding the operator set for demands of each case, it was more convenient to have two special operators that allowed plans to make calls to arbitrary functions written in standard programming languages. The goal of consolidating this functionality in extensibility operators was to (a) make it easier to write plans that required special calculations or library calls, (b) encourage non-relational information processing (which does not benefit from the efficiency of dataflow style processing) to be modularized outside of the plan, and (c) to simplify plans.

The two operators, Apply and Aggregate, provide extensibility at both the tuple and relation level. Apply calls user-defined single-row functions on each tuple of relational data and performs a dependent join on the input tuple with its corresponding result. For example, a user-defined single-row function called SQRT might return a tuple consisting of two values: the input value and its square root. The user defined function is written in a standard programming language, such as Java, and is executed on a per-tuple basis. Thus, this type of external function is very similar to the use of stored procedures or UDFs in commercial relational database systems

The Aggregate operator calls user-defined multi-row functions and performs a dependent join on the packed form of the input and its result. For example, a COUNT function might return a relation consisting of a single tuple with two values: the first being the packed form of the input and the second being the count of the number of distinct rows in that relation. As with Apply, the user-defined multi-row function is written in a standard programming language like Java. However, in contrast to being called on a per-tuple basis, it is executed on a per-relation basis.

XML integration

For purposes of efficiency and flexibility, it is often convenient to package or transform data to/from XML in mid-plan execution. For example, the contents of a large data set can often be described more compactly by capitalizing on the tree structure of an XML document. In addition, some Web sources (such as Web services) already provide query answers in XML format. To analyze or process this

data, it is often simpler and more efficient to deal with it in its native form rather than to convert it into relations, process it, and convert it back to XML. However, in other cases, a relatively small amount of XML data might need to be joined with a large set of relational data.

To provide flexible XML manipulation and integration, the language supports the Rel2xml, Xml2rel, and XQuery operators. The first two convert relations to XML documents and vice-versa. To allow XML to be processed in their native form, the language supports the XQuery operator, based on the evolving XQuery standard (Boag et al. 2002).

Asynchronous notification

Many continuously running plans do not involve interactive sessions with users. Instead, users request that a plan be run on a given schedule and expect to receive updates from the periodic execution of that plan. These updates are delivered through asynchronous mediums, such as e-mail, cell-phone messaging, or facsimile. To enable this kind of notification, the language includes operators such as Email, Fax, and Phone that communicate data via these devices.

Each of these operators works in a similar fashion. Input data received by the operator is re-formatted into a form that is suitable for transmission using the desired medium. The data is then transmitted: Email sends an e-mail message, Fax contacts a facsimile server with its data, and Phone routes data to a cell phone capable of receiving messages.

Automatic task administration

The overall system that accompanies the language includes a task database and a daemon process that periodically reads the task database and executes plans according to their schedule. This architecture is shown in Figure 2.4. Task entries consist of a plan name, a set of input to provide to that plan, and scheduling information. The latter data is represented in a format similar to the UNIX crontab

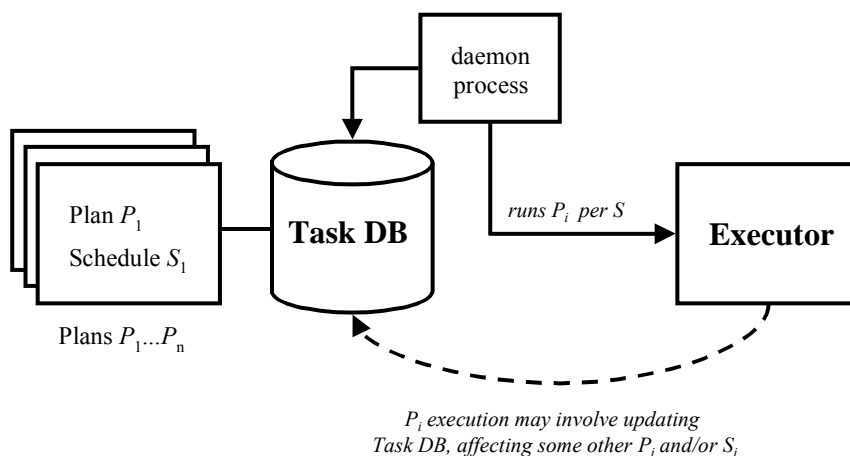


Figure 2.4: Task scheduling subsystem

entry. This format allows the minute, hour, day of the month, month, and year that a plan is supposed to be run. For example, a task entry of

*05 08-17 1,3,5 * * example.plan*

means: run the *example.plan* at five minutes after every hour between 8am and 5pm on the 1st, 3rd, and 5th days of every month of every year.

While tasks can be scheduled manually, the language also allows plans to automatically update the scheduling of other plans, including itself. To do so, the language supports two special scheduling operators, *Schedule* and *Unschedule*. The former allows a plan to register a new plan to be run. It creates or updates plan schedule data in the task database. *Unschedule* removes a scheduled task from the task database. *Unschedule* can be used by a plan to remove itself from a monitoring activity and is often used in tandem with a notification operator. For example, a plan can monitor the set of available houses for sale by various real estate Web sites for the entire month of September, send an email at the end of that month to the user containing the results, *unschedule* itself from execution, and then schedule a new plan (perhaps, for example, to clean up the database that stored the monitoring data).

2.3.4 Subplans

To promote reusability, modularity, and the capability for recursion, the plan language supports the notion of subplans. Recall that all plans are named, consist of a set of input and output streams, and a set of operators. If we consider that the series of operators amounts to a complex function on the input data, then plans present the same interface as do operators. In particular, using our earlier definitions, it is possible that $Op_i = P$ in that $OpIn = PlanIn$, $OpOut = PlanOut$, $OpWait = \emptyset$, $OpEnable = \emptyset$, and $Func = \{Op_1 \dots Op_n\}$. Thus, a plan can be referenced within another plan as if it were an operator. During execution, a subplan is called just like any other operator would – as inputs of the subplan arrive, they are executed within the body of the subplan by the operators of that subplan.

For example, consider how the *example_plan*, introduced earlier, can be referenced by another plan called *parent_plan*. Figure 2.5 illustrates how the text form of *parent_plan* treats *example_plan* as merely another operator. Subplans

```

PLAN parent_plan
{
  INPUT: w, x
  OUTPUT: z

  BODY
  {
    Op6 (w : y)
    example_plan (x, y : z)
  }
}

```

Figure 2.5: Text form of *parent_plan*

encourage modularity and re-use. Once written, a plan can be used as an operator in any number of future plans. Complicated manipulations of data can thus be abstracted away, making plan construction simpler and more efficient.

For example, one could develop a simple subplan called *persistent_diff*, shown in Figure 2.6, that uses the existing operators DbQuery, Minus, Null, and DbAppend to take any relation, compare it to a named relation stored in a local database. This plan determines if there was an update, appends the result, and returns the difference. Many types of monitoring style plans that operate on updated results can incorporate this subplan into their existing plan.

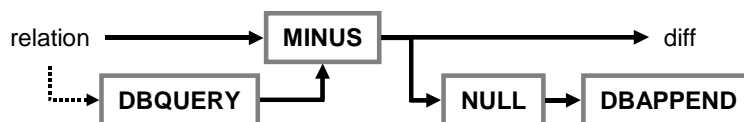


Figure 2.6: Graph form of *persistent_diff*

Recursion

In addition to promoting modularity and re-use, subplans make another form of control flow possible: recursion. As described earlier, a number of online information gathering tasks require some sort of looping-style (repeat until) control flow. Results from a single query can be spanned across multiple Web pages. Recursion provides an elegant way to address this type of interleaved information gathering and navigation in a streaming dataflow environment.

For example, when processing results from a search engine query, an automated information gathering system needs to collect results from each page, follow the "next page" link, collect results from the next page, collect the "next page" link on that page, and so on – until it runs out of "next page" links. If we were to express this in von Neumann style programming language, a *Do...While* loop might be used accomplish this task. However, implementing these types of loops in a dataflow environment is problematic because it requires cycles within a plan. This leads to data from one loop iteration possibly colliding with data from a different iteration. In practice, loops in dataflow graphs requires a fair amount of synchronization and additional operators.

Instead, this problem can be solved simply with recursion. We can use subplan reference as a means by which to repeat the same body of functionality and we can use the Null operator as the test, or exit condition. The resulting simplicity and lack of synchronization complexity makes recursion an elegant solution for addressing cases where navigation is interleaved with retrieval and when the number of iterations for looping style information gathering is not known until runtime.

As an example of how recursion is used, consider the abstract plan for processing the results of a search engine query. A higher level plan called *query_search_engine*, shown in Figure 2.7a, posts the initial query to the search engine and retrieves the initial results. A subplan called *gather_and_follow*, shown in Figure 2.7b, is then called to process these results. To do so, the subplan routes



Figure 2.7a: The *query_search_engine* plan

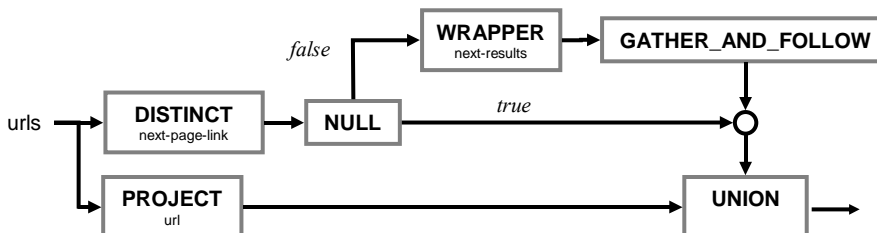


Figure 2.7b: The recursive subplan *gather_and_follow*

the current page results to a Union operator and then calls itself again to process results available via the next page link. The results of this recursive call are then combined at the Union operator with the first flow. This process continues until no more next page links are found.

2.4 Using the plan language to build information agents

Having introduced the agent plan language in the previous section, let us turn to how this language can be used to build information agents. We start with the main example used in the paper, the CarInfo agent, which was described in Chapter 1.

2.4.1 The CarInfo agent plan

Recall that the purpose of the CarInfo agent is to return safety ratings and car reviews for automobiles that match a particular set of user criteria, including car type, year, and price range. Figure 1.6 of Chapter 1 showed the dataflow graph for the CarInfo agent. Using the language introduced in this chapter, Figure 2.8 shows the corresponding text form of the plan described by that graph.

The figure is a straightforward translation of the graph form of the plan. Again, the ordering of the operators has nothing to do with the execution schedule – operators fire whenever their input data becomes available. Literals shown in Figure 2.8 are treated as streams containing a single tuple.

2.4.2 Homeseekers: a more complicated type of information agent

While Figure 2.8 shows how the language I have described can be used to specify tasks like CarInfo, some of these tasks can also be expressed in database query languages like SQL. For example, the following SQL expresses the CarInfo task:

```

PLAN car_info
{
  INPUT: criteria
  OUTPUT: reviews-and-ratings

  BODY
  {
    Wrapper ("Edmunds", criteria : cars)
    Select (cars, "maker != 'Oldsmobile'" : filtered-cars)
    Wrapper ("NHTSA", filtered-cars : safety-ratings)
    Wrapper ("CG Search", filtered-cars : summary-urls)
    Wrapper ("CG Summary", summary-urls : full-urls)
    Wrapper ("CG Full", full-urls : car-reviews)
    Join (safety-ratings, car-reviews,
         "l.make = r.make and l.model = r.model" :
         reviews-and-ratings)
  }
}

```

Figure 2.8: Text representation of CarInfo agent plan

```

SELECT E.make, E.model, N.safety_rating, CFULL.review_text
FROM edmunds as E, nhtsa as N, cg_search as CSEARCH,
     cg_summary as CSUMM, cg_full as CFULL
WHERE E.type = 'Midsize coupe/hatchback'
     AND E.price between 5000 and 7000
     AND E.year = 2002
     AND E.make = N.make AND E.make = CSEARCH.make
     AND E.model = N.model AND E.model = CSEARCH.model
     AND CSEARCH.summary_url = CSUMM.url
     AND CSUMM.full_url = CFULL.url
     AND E.make <> 'Oldsmobile'

```

Existing query processing technology would construct a plan similar to the one in Figure 2.8 in order to return an answer. However, more complicated Web data gathering tasks cannot always be expressed in languages like SQL. As a result, traditional database-like approaches to Web information gathering that use such languages, for example mediators and network query engines, are limited in what tasks they can perform.

As a concrete example, let us consider the “Homeseekers” task. This example involves using the Web to search for a new house to buy. Suppose that we want to use an online real estate listings site, such as Homeseekers (<http://www.homeseekers.com>), to locate houses that meet a certain set of price, location, and number of rooms constraints. In doing so, we want the search query to run periodically over a medium duration of time (say a few weeks) and have any new updates (i.e., new houses that meet specified criteria) e-mailed to us as they are found. Finally, we would also like these summaries to include references (URLs) to large pictures of each house.

To understand how to automate the gathering part of this task, let us first discuss how users would complete it manually. Figures 2.9a, 2.9b, and 2.9c show the

interface and result pages for Homeseekers. To query for new homes, users initially fill the criteria shown in Figure 2.9a – specifically, they enter information that includes city, state, maximum price, etc. Once they fill in this form, they submit the query to the site and an initial set of results are returned – these are shown in Figure 2.1b. However, notice that this page only contains results 1 through 15 of 22. To get the remainder of the results, a "Next" link (circled in Figure 2.9b) must be followed to the page containing results 16 through 22. Finally, to get the details of each house, users must follow the URL link associated with each listing. A sample detail screen is shown in Figure 2.9c. The detail screen is useful because it often contains pictures and more information, such as the MLS (multiple listing services) information, about each house. In this example, the detailed page for a house must be investigated in order to identify the URL for the large image of each house. In this example, the detail page is also important because it specifies the exact number of rooms in each house – the search facility provided by the site only allows the user to specify “*n* or more” rooms (e.g., 2 or more rooms).

To turn this one time search into a monitoring process, users would then repeat the above procedure with some frequency over the desired number of days, weeks, or months. Note that the user must both query the site periodically and keep track of new results by hand. This latter activity can require a great deal of work – users must discern which houses in each result list are new entries and identify changes (e.g., selling price updates) for houses that have been previously viewed.

Figure 2.9a: Homeseekers search screen

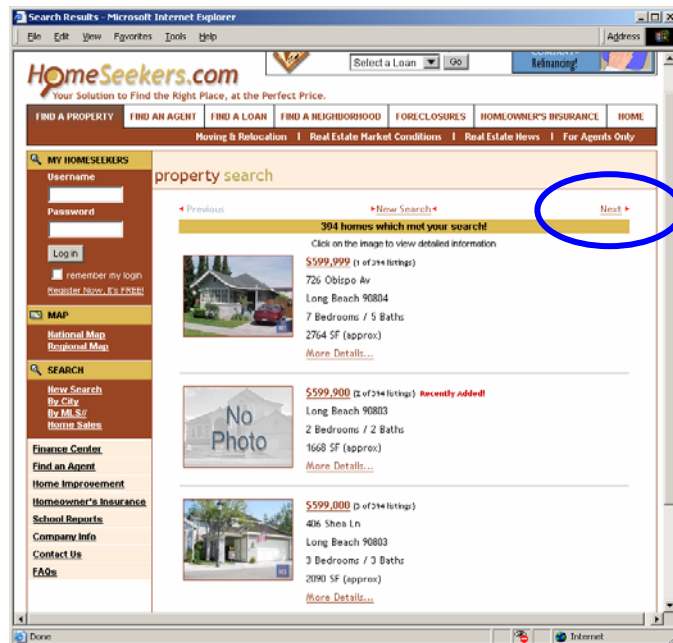


Figure 2.9b: Homeseekers results list screen

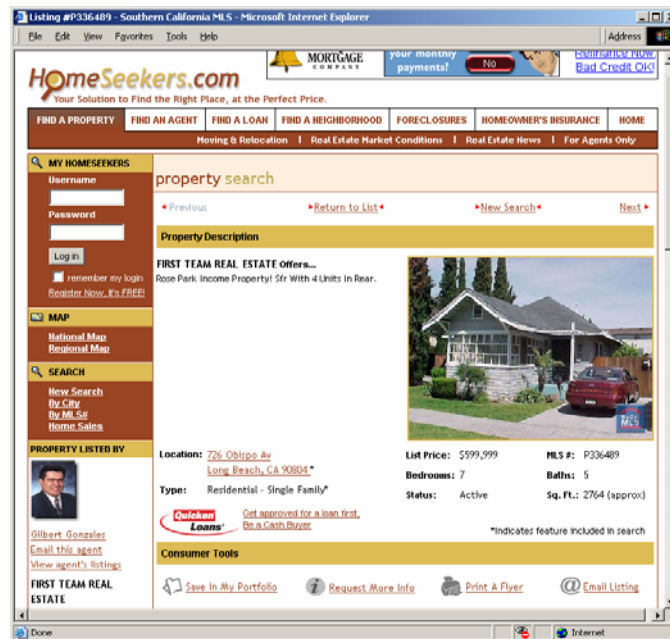


Figure 2.9c: Homeseekers detail page screen

As I have already discussed, it is possible to accomplish part of the task using existing Web query techniques, such as those provided by mediators and network query engines. However, notice that the task requires actions beyond gathering and filtering data. It involves periodic execution, comparison with past results, conditional execution, and asynchronous notification to the user. These are not actions that traditional Web query languages support and involve more than

gathering and filtering. Instead of a query plan language, what is needed is an agent plan language that supports the operators and constructs necessary to complete the task.

We can consider how such agent plans generally might look. Figure 2.10 shows an abstract plan for monitoring Homeseekers. As the figure shows, search criteria is used as input to generate one or more pages of house results. The URLs for each house from each results page are extracted and compared against houses that already existed in a local database. New houses – those on the Web page but not in the database – are then queried for their details and appended to the database so that future queries can distinguish new results. During the extraction of houses from a given Homeseekers results page, the "Next" link (if any) on that page is followed and the resulting new houses go through the same process. This cycle stops when the last result page, the page without a "Next" link, has been reached. Then, after the details of the last house has been gathered, an update on the set of new houses found are e-mailed to the user.

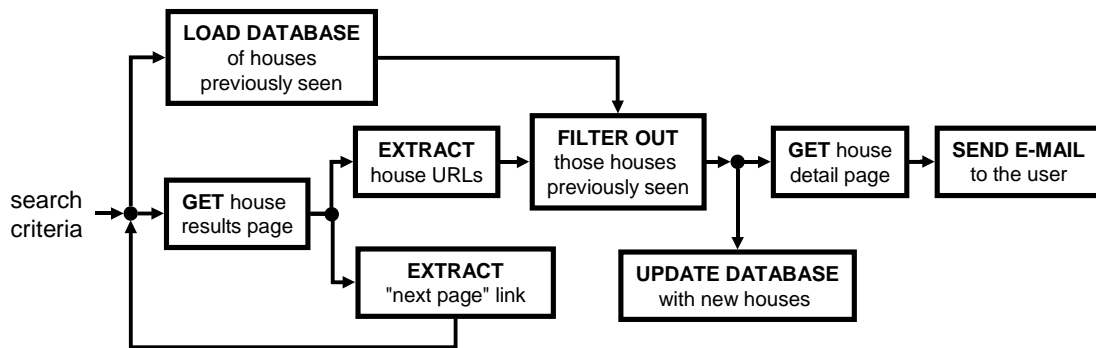


Figure 2.10: Abstract Homeseekers plan

2.4.3 The Homeseekers agent plan

To solve the Homeseekers task in the agent plan language I have described, let us first notice that it is somewhat similar to the search engine example described earlier in section 2.3.4. The basic idea is to post the query to the Web site, gather the initial results, gather the "next page" link, and continue gathering results from successive pages. Then, for each result page, we would drill down on each result URL in order to extract the details about each house.

The details of the solution are shown in the Figure 2.11a and Figure 2.11b. The former illustrates the *get_houses*, required to implement the abstract real estate plan in Figure 2.10. *get_houses* calls the subplan *get_urls* shown in Figure 2.11b, which is nearly identical to the plan *gather_and_follow*, described above. The rest of *get_houses* works as follows:

1. A Wrapper operator fetches the initial set of houses and link to the next page (if any) and passes it off to the *get_urls* recursive subplan, which continues this gathering process recursively. The *get_urls* subplan terminates after it reaches the final page of search results.

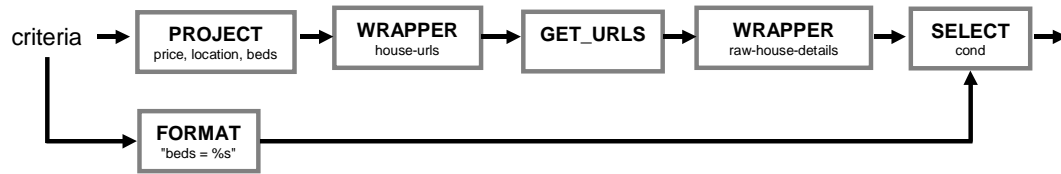


Figure 2.11a: Homeseekers agent plan *get_houses*

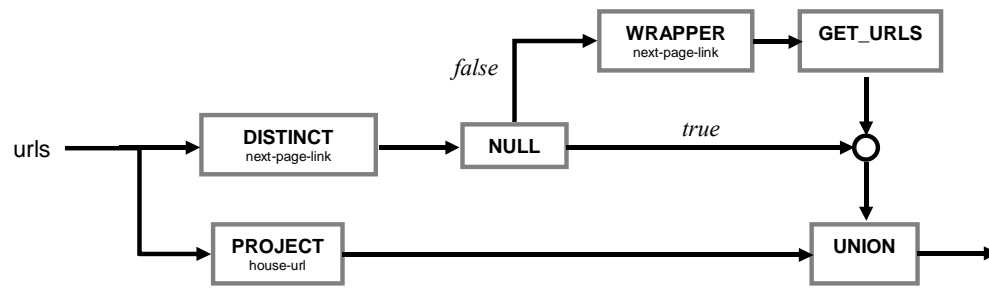


Figure 2.11b: Homeseekers recursive subplan *get_urls*

2. Another Wrapper operator investigates the detail link for each house so that the full set of criteria (including picture) can be returned.
3. Using these details, a Select operator filters those satisfying the search criteria.
4. The result is output from the plan.

2.5 An Efficient Plan Execution Architecture

By definition, Web information gathering involves processing data gathered from remote sources. During the execution of an information gathering plan, it is often the case that multiple independent requests are made for different sets of remote data. Those data are then independently processed by a series of operations and then combined or output. Network latencies, bandwidth limitations, slow Web sites, and queries that yield large result sets can dramatically curtail the execution performance of information gathering plans. This is especially the case when plan operators are executed serially: any one of the issues mentioned can bottleneck the execution of an entire plan.

From an efficiency standpoint, there are two problems with standard von Neumann execution of information gathering plans. One is that it does not exploit the independence of data flows in a common plan: for example, multiple unrelated requests for remote data cannot be parallelized. The plan language described in this chapter addresses this problem to some extent by allowing plans to be expressed in terms of their minimal data dependencies: still, that does not dictate how those operators are actually executed.

The second efficiency problem is that von Neumann execution does not exploit the independence of tuples in a common relation: for example, when a large data set is being progressively retrieved from a remote source, the tuples that have already been retrieved could conceivably be operated on by successive operators in the plan.

This is often reasonable, since the CPU on the local system is often under-utilized while remote data is being fetched.

Both problems are addressed through a streaming dataflow execution system for information agent plans. The system allows plans to realize significant operator and data parallelism at runtime by executing multiple operators concurrently and pipelining data between operators throughout execution. Other network query engines have implemented designs that bear some similarity. However, the architecture I propose below is novel in two ways:

- A thread-pooling approach is applied to streaming dataflow execution, where multiple threads are shared by all operators in a plan. This allows significant parallelism without exhausting resources.
- Recursive streaming dataflow execution is addressed using a data coloring approach.

2.5.1 Dataflow executor

While the plan language described here allows dataflow-style plans to be coded in text, it does not specify how the actual execution process works. Thus, to complement the language and to efficiently execute plans, I have developed a true dataflow-style execution component. The executor allows plans to realize opportunities for concurrency between independent flows of data, thus enabling greater horizontal parallelism at runtime.

The executor functions as a virtual threaded dataflow machine. It assigns user-level threads to execute operators that are ready to fire. This type of execution is said to be “virtual dataflow” because thread creation and assignment is not done natively by the CPU, nor even in kernel space by the operating system, but by an application program (the executor) running in user space. By using threads to parallelize execution of a plan, the executor can realize better degrees of true parallelism, even on single CPU machines. This is because the use of threads reduces the impact of any I/O penalties caused by a currently executing operator. That is, multiple threads reduces the effect of vertical waste that can occur when single-threaded execution reaches an operation that blocks on I/O.

For example, consider the case where a plan containing two independent Wrapper operators is being executed on a machine with a single CPU. Suppose that both Wrapper operators have their input and can fire. Both operators will be assigned distinct threads. The single CPU will execute code that issues the network request for the first Wrapper operator, not wait for data to be returned, and finish issuing the network request for the second Wrapper operator. Thus, in a matter of microseconds, both operators will have issued their requests (which typically take on the order of hundreds of milliseconds to complete) and retrieval of the data (on the remote sites) will have been parallelized. Thus, the overall execution time will be equal to the slowest of the two requests to complete. This contrasts with the execution time required for serial execution, which is equal to the sum of time required for each request.

Promoting and bounding parallelism with thread pools

While using threaded dataflow has its benefits, past research in dataflow computing and operating systems has shown that there are cases when parallelism must be throttled or the overhead of thread management (i.e., the creation and destruction of threads) can be overly taxing. For example, if threads are created whenever an operator is ready, the cost to create them can add up to significant overhead. Also, if there is significant parallelism during execution, the number of threads employed might result in context switching costs that outweigh the parallelism benefits. To address both issues, I describe a thread pooling architecture that allows the executor to realize significant parallelism opportunities within fixed bounds.

There are advantages to using a pooling approach to thread management for streaming dataflow execution as opposed to other types of thread management, such as on-demand creation or permanent operator/thread assignment. In particular, two key benefits are that (a) significant parallelism can be realized during execution while at the same time (b) ensuring that the parallelism does not exceed some limit. In contrast, if threads are created on demand, the amount of data determines the maximum degree of parallelism – this can easily exceed machine limits. Alternatively, if a fixed number of threads are associated per operator, then many available threads may go unused at any given point in execution. Furthermore, since threads in a thread pool are created once, there is practically no overhead issue. In summary, thread pools ensure a low-overhead means to achieve significant, but bounded parallelism during streaming dataflow execution.

Let us now turn to the details of how a thread pool is used by the executor. At the start of plan execution, a finite number of threads are created (this number is easily adjustable through an external configuration file) and arranged in a thread pool. Once the threads have been created, execution begins. When data becomes available (either via input or through operator production), a thread from the pool is assigned to execute a method on the consuming operator with that data. Each time that operator produces output, it hands off the output to zero or more threads so that its consumer(s), if any, can process the output. If the pool does not contain any available threads, the output is queued in a spillover work queue, to be picked up later by threads as they return to the queue. This same behavior occurs for all operator input events. Thus, parallelism is both ensured by the existence of multiple threads in the pool and bounded by it – in the latter case, the degree of true parallelism during execution can never exceed the pool size. Demands on parallelism beyond the number of threads in the pool is handled by the work queue.

Figure 2.12 illustrates the details of how the thread pool is used by the executor at runtime. The figure shows that there are four key parts to the executor:

- **The thread pool:** This is a collection of threads ready to process the input collected in the queue. There can be a single thread pool or it can be partitioned so that certain sources have a guaranteed number of threads available to operators that query those sources. All available threads wait for new objects in the queue. Typically, contention for the queue on machines with a single CPU is not an issue (even with

hundreds of threads). However, configuration options do exist for multiple work queues to be created and for the thread pool to be partitioned across queues.

- **The spillover work queue:** All data received externally and transmitted internally (i.e., as a result of operator execution) that cannot be immediately assigned to an available thread is collected in this queue. As threads return to the pool, they check if there are objects in the queue: if there are, they process them, otherwise the thread waits to be activated by future input. The queue itself is an asynchronous FIFO queue implemented as a circular buffer. When the queue is full, it grows incrementally as needed. The initial size of the queue is configurable. The structure of a queue element is described in detail below.
- **The routing table:** This data structure describes the dataflow plan graph in terms of producer/consumer operator method associations. For example, if a Select operator produces data consumed by a Project operator, the data structure that marshals output from the Select is associated with the Project input method that should consume this data. The table is computed once – prior to execution – so that the performance of operator-to-operator I/O is not impacted at runtime by repetitive lookups of consumers by producers. Instead, pre-computation allows the data structure associated with a producing method to immediately route output data to the proper set of consuming input methods.
- **The set of operator objects:** These are the collection of operator

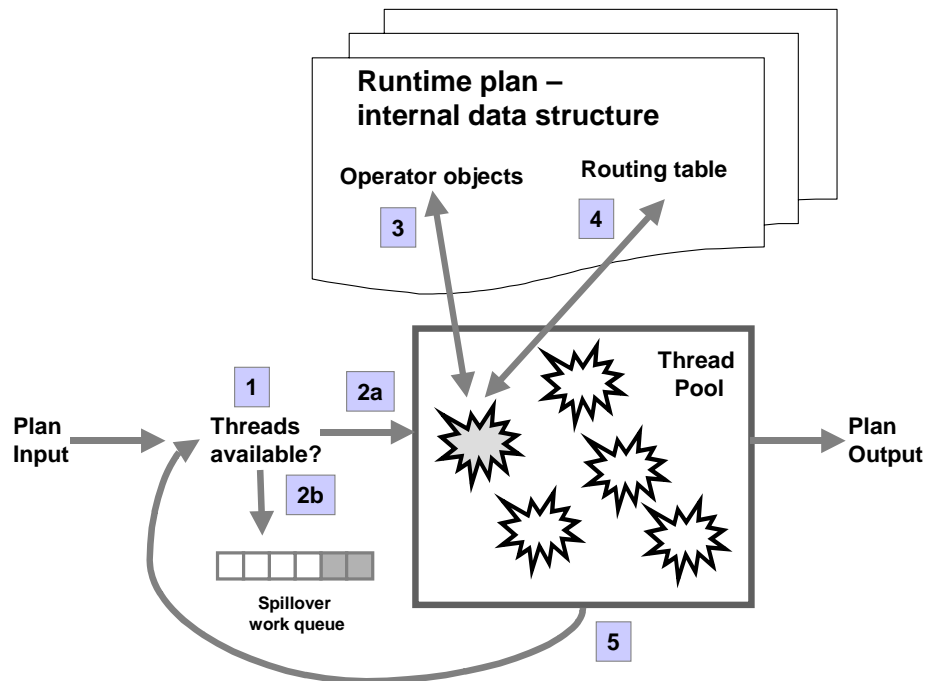


Figure 2.12: Detailed architecture of executor

classes (including their input/output methods and state data structures). There exists one operator object per instance in the plan.

Each queue object consists of a tuple that describes:

- the session ID
- the iteration ID
- the content (i.e., the data)
- destination operator interface (i.e., a function pointer).

The session ID is used to distinguish independent sessions (external invocations of the plan) and the iteration ID to distinguish current call-graph level of a session, which ensures safety during concurrent re-entrancy at runtime. These IDs provide a unique key for indexing operator state information. For example, during recursive execution, these IDs ensure that the concurrent firing of the same operator at different levels of the call graph do not co-mingle state information. Finally, the destination operator interface is the pointer to the code that the thread assigned to a queue object will run.

At runtime, the system works as follows. Initial plan input arrives and is assigned to threads from the thread pool (#1 in Figure 2.12), one thread for each input tuple (#2a), or if no threads are available the data is added to the spillover work queue (#2b). Each assigned thread from the pool takes its queue object and, based on the description of its target, fetches the appropriate operator object so that it can execute the proper function on the data (#3). During the execution of the operator, state information from previous firings may be accessed using the (session ID, iteration ID) pair as a key. The result of an operator firing may result in output. If it does, the operator uses the routing table (#4) to determine the set of consumers of that output. It then composes new data queue objects for each consumer and hands off those objects (#5) to either an available thread in the thread pool (#2a) or deposits them to the work queue (#2b) if no threads are available. To reduce memory demands, producers only deep-copy data they produce if there are multiple consumers. Finally, operators that produce plan output data route that data out of the plan as it becomes available.

2.5.2 Data streaming

At a logical level, each of the variables in the plan language I describe are relations. However, to provide more parallelism and thus efficiency at runtime, tuples of a common relation are streamed between operators. Each stream consists of stream elements (the tuples in a relation), followed by an end of stream (EOS) marker. Thus, when communicating a relation from producer to consumer, producing operators communicate individual tuples to consumer operators and follow the final tuple with an EOS token.

Streaming relations between operators increases the degree of vertical parallelism during plan execution. In revisiting the firing rule described earlier, it can be further clarified to read:

An operator may fire upon receipt of any input tuple, providing it has received the first tuple of all of its wait variables.

Thus, when an operator receives a single tuple on any of its inputs, it can consume and process that tuple. Afterwards, it can potentially emit output that, in turn, can be consumed by a downstream operator or output from the plan. The resulting parallelism is “vertical” in the sense that two or more operators (e.g., one producer and one or more consumers) can concurrently operate on the same relation of data. Remote sources that return significant amounts of data can be more efficiently processed through streaming, since the operator that receives the network transmission can pass along data for processing as it becomes available and before the rest of the data has been received.

Support for any kind of streaming implies that state must be kept by operators between firings. This is because the operation being performed is logically on an entire relation, even though it physically involves each tuple of that relation. If the operator does not maintain state between firings, it cannot necessarily produce correct results. For example, consider the set-theoretic Minus operator that takes two inputs, *lhs* and *rhs*, and outputs the result of *lhs* - *rhs*. This operator can begin emitting output as soon as it has received the *rhs* EOS token. However, the operator must still keep track of *rhs* data until it receives the EOS from both; if not, it may emit a result that is later found to be incorrect. To see how this could happen, suppose that the order of input received by an instance of the Minus operator was:

```
lhs: (Thing1)
lhs: (Thing2)
rhs: (Thing3)
rhs: (Thing2)
rhs: EOS
lhs: (Thing3)
lhs: EOS
```

The correct output, *lhs* - *rhs*, should be

```
lhs-rhs: (Thing1)
lhs-rhs: EOS
```

However, this can only be achieved by waiting for the EOS before emitting any output and also by keeping track (i.e., maintaining state) of both inputs. For example, if only *lhs* data is retained, then the *rhs* instance of (Thing3) would not be in memory when the *lhs* instance of (Thing3) occurred and this tuple would be incorrectly emitted.

In summary, streaming is a technique that improves the efficiency of operator I/O by increasing the degree of vertical parallelism that is possible at runtime. By allowing producers to emit tuples as soon as possible – and by not forcing them to wait for consumers to receive them – both producers and consumers can work as fast as they are able. The main tradeoff is increased memory, for the queue required to facilitate streaming and for the state that needs to be maintained between firings.

Recursive streaming: simplicity + efficiency

Streaming can complement the simplicity of many types of recursive plans with highly efficient execution. Looping in theoretical dataflow systems is non-trivial because of the desire for single-assignment and because of the need for

synchronization during loop iterations. Streaming further complicates this: data from different loop iterations can collide, requiring some mechanism to color the data for each iteration. As a result, looping becomes an even more difficult challenge.

To address this problem, I introduce a data coloring approach. Each time that data enters a flow, it is given a color (which can be thought of as a transaction ID) and an iteration value (initially 0). Upon re-entrancy, the iteration value is incremented. When leaving a re-entrant module, the iteration value is decremented. If the new value is equal to 0, the flow is routed out of the recursive module; otherwise, the data flow continues to unravel until its iteration value is 0. For tail-recursive situations, the system optimizes this process and simply decrements the iteration value to 0 immediately and exits the recursive module. The two pronged data-coloring approach, which is similar to strategies used in dataflow computing literature, maintains the property of single assignment at each the level of the call graph. Streaming easily fits into this model without any other changes. As a result, many levels of the call graph can be active in parallel – effectively parallelizing the loop.

To see how this works, let us return to the *get_houses* example of Figures 2.10a and 2.10b. When the input tuple arrives, the initial page of houses is fetched. When that happens, the “Next” link is followed in parallel with the projecting of the house URLs to the Union operator and then to the Minus operator. Since the Union operator can emit results immediately, and the Minus operator will have both of its inputs, data flow continues until the next Wrapper operator, which queries the URL and extracts the details from the house. Thus, the details of the houses from the first page are queried in parallel with the following of the “Next” link, if it exists. Data from the next page is then extracted in parallel with the following of the “Next” link from this second page and so on. Meanwhile, the results from the *get_urls* subplan (the house URLs) are streamed back to the first level of the plan, to the Union operator. They continue on through and their details are gathered in parallel.

2.6 Experimental results

I conducted a set of experiments that highlight the increased expressivity and efficient execution supported by the agent plan language and execution system. This method consists of verifying three hypotheses that are fundamental to the key claims:

Hypothesis 1: Efficient information agents. The language and execution system supports the efficient execution of information agent plans.

Hypothesis 2: Expressive plan language supports information gathering loops and source monitoring, actions not supported by other network query engine languages. The agent plan language described here allows certain types of information gathering goals, such as those that require interleaved gathering or source monitoring, to be achieved more easily than would be possible using languages supported by network query engines.

Hypothesis 3: Increased expressivity does not impact performance. The additional expressivity permitted by the plan language described here does not negatively impact the efficiency of the system.

After a brief introduction about the implemented system used in the experiments, the rest of this section is divided into three subsections, each of which focuses on verifying each of these hypotheses.

2.6.1 The Theseus information agent system

I implemented the approach described in this chapter in a system called Theseus. Theseus is written entirely in Java (approximately 30,000 lines of code) and thus runs on any operating system to which the Java virtual machine (JVM) has been ported. I ran the experiments described here on an Intel Pentium III 833MHz machine with 256MB RAM, running Windows 2000 Professional Edition, using the JVM and API provided by Sun Microsystems' Java Standard Edition (JSE), version 1.4.1. This machine was connected to the Internet via a 10Mbps Ethernet network card.

2.6.2 Hypothesis 1: Efficient information agents

To verify the first hypothesis, that the language and execution system supports the efficient execution of information agent plans, I measured the efficiency of the Homeseekers information agent. The experiments show that without the parallelism benefits of the plan language and execution system, agents such as Homeseekers would perform much more slowly than with these features.

The graphical plans for Homeseekers are the same as shown in Figures 2.11a and 2.11b, without the operators for monitoring (the DbQuery, Minus, DbAppend, and Email). Thus, this is an interactive agent that simply gathers data once. The textual plans required for this are simply translations of Figures 2.11a and 2.11b using the plan language described in this paper. The textual form of the *get_houses* plan is shown in Figure 2.13a and the textual form of the *get_urls* plan is shown in Figure 2.13b.

To demonstrate the efficiency that streaming dataflow provides, the Homeseekers *Get_houses* plan was run under three different configurations of

```

PLAN get_urls
{
  INPUT: result-page-data
  OUTPUT: combined-urls

  BODY
  {
    project(result-page-data, "house-url" : curr-urls)
    distinct(result-page-data, "next-page-url" : next-status)
    null (next-status, next-status, next-status :
          next-page-url, next-urls)
    wrapper ("result-page", next-page-url : next-page-data)
    get_urls (next-page-data : next-urls)
    union (curr-urls, next-urls : combined-urls)
  }
}

```

Figure 2.13a: Text form of the Homeseekers *get_houses* plan

```

PLAN get_urls
{
  INPUT: result-page-data
  OUTPUT: combined-urls

  BODY
  {
    project(result-page-data, "house-url" : curr-urls)
    distinct(result-page-data, "next-page-url" : next-status)
    null (next-status, next-status, next-status :
        next-page-url, next-urls)
    wrapper ("result-page", next-page-url : next-page-data)
    get_urls (next-page-data : next-urls)
    union (curr-urls, next-urls : combined-urls)
  }
}

```

Figure 2.13b: Text form of the Homeseekers *get_urls* recursive subplan

Theseus. The first configuration (*D-*) consisted of a thread pool with one thread – effectively preventing true multi-threaded dataflow execution and also makes streaming irrelevant. The resulting execution is thus very similar to the case where the plan had been programmed directly (without threads) using a language like Java or C++. A second Theseus configuration (*D+S-*) used multiple threads for dataflow-style processing, but did not steam data between operators. Finally, the third configuration (*D+S+*) consisted of running Theseus in its normal streaming dataflow mode, enabling both types of parallelism. For the *D+S-* and *D+S+* cases, the number of threads was set to 15.

Each configuration was run three times (interleaved, to negate any temporary benefits of network or source availability) and averaged the measurements of the three runs. The search constraints consisted of finding “houses in Irvine, CA that are priced between \$500,000 and \$550,000”. This query returned 72 results (tuples), spread across 12 pages (6 results per page). Figure 2.14 shows the average performance results for these three configurations in terms of the time it took to obtain the first tuple (beginning of output) and the time it took to obtain the last tuple (end of output).

As the figure shows, the parallelism provided by streaming dataflow has a significant impact. Typical von Neumann style execution, such as that in (*D-*), cannot not leverage opportunities for parallelism and suffers heavily from the cumulative effects of I/O delays. While the *D+S-* fares better because concurrent I/O requests can be issued in parallel, the inability to stream data throughout the plan prevents all result pages from being queried in parallel. Also, because of the lack of streaming, results obtained early during execution (i.e., the first tuple) cannot be communicated until the last tuple is ready. Finally, the *D+S+* case shows that streaming can alleviate both problems, allowing the first tuple to be output as soon as possible, while supporting the ability to query all result pages in parallel (and process the detail pages as soon as possible, in parallel). In short, Figure 2.14 shows that

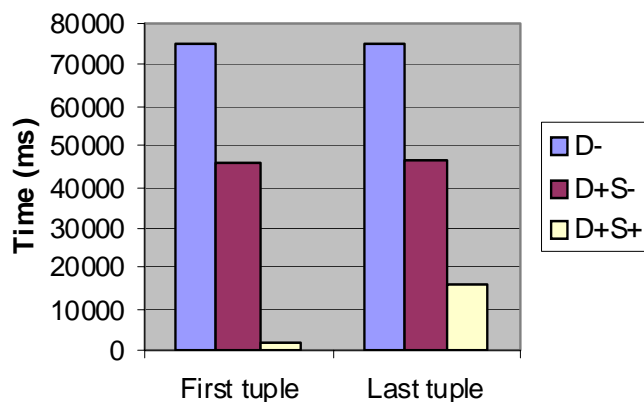


Figure 2.14: Average Homeseekers performance results

streaming dataflow is a very efficient execution paradigm for I/O-bound Web-information gathering plans that require interleaved navigation and gathering.

Note that the performance benefits of streaming dataflow are not limited to recursive streaming dataflow plans. Even for other common types of data integration plans, where agents gather information from multiple sources and/or where queries to some sources are dependent on answers from others (i.e., sources with binding patterns), streaming dataflow produces noticeable benefits. Table 2.2 shows how streaming dataflow execution is more efficient than standard von Neumann-style execution for three common types of plans: CarInfo, RepInfo, and StockInfo². CarInfo has been discussed earlier and a details about the RepInfo and StockInfo agent plans can be found in Chapter 3.

To further demonstrate the benefit of streaming dataflow for recursive plans, I sought to compare the execution performance of the Homeseekers *Get_houses* plan against the performance achieved when using another type of information gathering system, such as a network query engine. However, since none of these systems support the ability to express loops or recursive information gathering, it was not

Plan	Scenario	First Tuple (ms)	Last Tuple (ms)
Car-Info	D-	10578	10578
Car-Info	D+S-	7438	7438
Car-Info	D+S+	4833	5879
Rep-Info	D-	11104	11104
Rep-Info	D+S-	4125	4125
Rep-Info	D+S+	3682	4238
Stock-Info	D-	7283	7283
Stock-Info	D+S-	6615	6615
Stock-Info	D+S+	6151	6151

Table 2.2: The benefits of streaming dataflow for three other plans

² **Note:** Performance numbers for some of these plans, such as CarInfo, differ slightly from what has been presented elsewhere in this document. This is because Table 2.2 measurements were taken at a different time and reflect changes in the performance of the remote sources (i.e., Web sites) since the time of the earlier measurements.

possible to simply run the same plan in these other executors. Instead, Theseus can only be to the theoretical performance of an ad-hoc solution – a streaming dataflow network query engine integrated into software that loops over the result pages.

More precisely, to complete an equivalent Homeseekers-like task, these systems would need to gather data from one result page at a time. Note that while loops or recursion for these systems is not possible (i.e., not possible to gather data spread across a set of pages in parallel), given the type of intermediate plan language they support, they can still be used to “drill down” on the details of a particular result (i.e., gather data below a set of pages) in parallel. Thus, a network query engine could leverage its dataflow and streaming capabilities to process a single page, but could not be used to parallelize the information gathering from a set of linked result pages. Each page (and its details) would have to be processed separately.

To simulate this behavior, I used Theseus to extract house URLs and the details one page at time, for each of the twelve pages of results obtained by the initial query. The average time required to gather the details of all six housing results was 3204 ms. Note again that the time to retrieve the first detailed result was the same as in the Theseus *D+S+* case: 1852ms. If we take the time to extract all six detailed results and multiply it by the number of pages in the query (12), the time of last tuple is equal to $(3204 * 12 =)$ 38448ms. Figure 2.15 shows how these results compare to the *D+S+* case of Theseus.

Thus, while an ad-hoc solution using a network query engine could allow the first tuple of results to be returned just as fast as the system described in this paper, the inability for the “Next” links to be navigated to immediately would result in less loop parallelism and, as a result, would lead to slower production of the last tuple of data. Therefore, while network query engines could be used to gather results spread across multiple hyperlinked Web pages, their inability to natively support a mechanism for looping negates the potential for streaming to further parallelize the looping process.

In summary, to verify the first hypothesis, I have described how the expressivity of the plan language presented enables more complex queries (like Homeseekers) to be answered efficiently. These results apply not just to Homeseekers, but to any type of site that reports a result list as a series of hyperlinked pages.

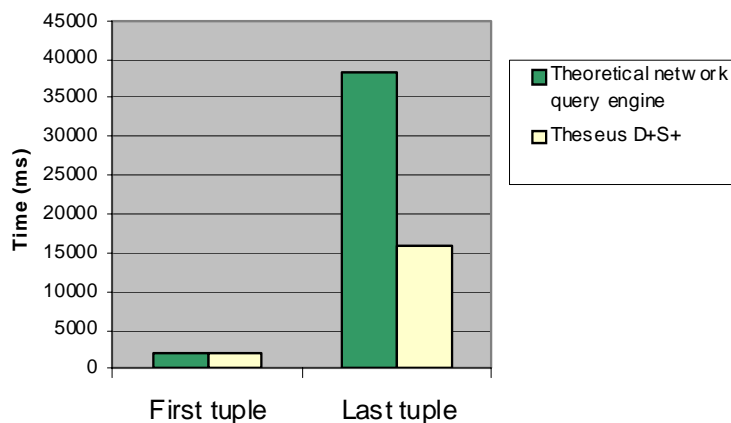


Figure 2.15: Theseus vs. theoretical network query engine

2.6.3 Hypothesis 2: Expressive plan language supports information gathering loops and source monitoring, actions not supported by other network query engine languages

To validate the second hypothesis, that the agent plan language described here supports plans that either (a) are not simple to describe using the languages of other network query engines or (b) cannot be represented at all by these query languages, I compared how the task of monitoring Homeseekers could be accomplished using the approach described in this chapter versus existing Web query systems. I have previously described why monitoring in cases such as this would be useful – searching for a house is a process that requires weeks, if not months of executing the same kind of query. Thus, a corresponding information gathering plan would query Homeseekers once per day and send newly found matches to the end user over e-mail. Again, this type of problem is general – it is often desirable to be able to monitor many Internet sites that produce lists of results. However, to do so requires support for plans that are capable of expressing the monitoring task, the persistence of monitoring data, and the ability to notify users asynchronously.

The graph form of the plan to monitor Homeseekers is shown in Figure 2.16. It simply leverages the existing Homeseekers plan and uses a few additional operators to support monitoring. In particular, it uses two database operators (DbQuery and DbAppend) to integrate a local commercial database system for the persistence of results. This allows future queries to only return new results and stored all past results. Notice that initial DbQuery is triggered by a synchronization variable. The plan also communicates new results asynchronously to users via an Email operator.

To measure expressivity, let us consider a comparison of the plan in Figure 2.15 with those capable of being produced by the Telegraph and Niagara network query engines. The comparison focuses on TelegraphCQ (Chandrasekaran et al. 2003) and NiagaraCQ (Chen et al. 2000), both of which are modifications of their original systems to support continuous queries for the monitoring streaming data sources. Since the TelegraphCQ and NiagaraCQ query languages are very similar, I present a detailed comparison with the former and a general comparison with the latter.

TelegraphCQ provides a SQL-like language with extensions for expressing operations on windows of streaming data. Specifically, the language allows one to express Select-Project-Join (SPJ) style queries over streaming data and also includes support for “for” loop constructs to allow the frequency of querying those streams. For example, to treat Homeseekers as a streaming data source and to query it once per day (for 10 days) for houses in Manhattan Beach, CA, that are less than \$800,000:

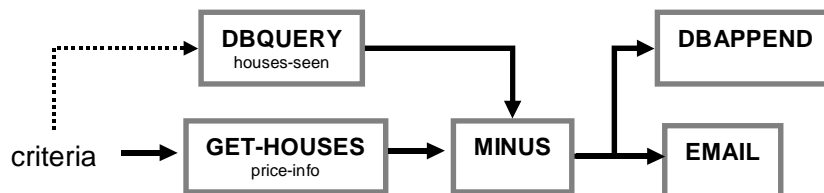


Figure 2.16: Graph form of the plan to monitor Homeseekers

```

Select street_address, num_rooms, price
  From Homeseekers
 Where price < 800000
   and city = 'Manhattan Beach'
   and state = 'CA'
 for (t=ST; t<ST+10; t++) {
   WindowIs(Homeseekers, t-1, t)
 }

```

From this, it is clear that TelegraphCQ supports some ability to monitor sources. Unfortunately, the above example would not work the way the plan in Figure 2.16 does, for two major reasons:

- The Homeseekers source cannot simply have its data streamed, as the query above indicates. Rather, there needs to be some way to express the need to gather multiple results spread over multiple pages.
- Unlike the plan in Figure 2.16, the query above does not allow one to specify that recurring queries only communicate the “diff” of prior findings. As a result, a consumer of the above query will receive multiple notifications about the same houses. Worse, it is possible for the user to receive updates even if nothing has changed on the source. In contrast, the plan shown in Figure 2.16 only updates the user if new houses have been listed.

In addition to both of the points above, there is no mention of how periodic results are to be communicated to users. In contrast, the Email operator in Figure 16 allows updates to be asynchronously transmitted to users.

Finally, it is worth noting that TelegraphCQ queries only terminate based on temporal conditions – that is, the only way to control continuous queries are through declarations of time windows. For example, it is not possible to express that a continuous query to Homeseekers should be terminated after, say, one hundred total results have been found, or after no new houses have been found in the last week. In contrast, simple modifications of the plan in Figure 2.15 using the Aggregate and database operators could achieve both goals.

The NiagaraCQ query language, like TelegraphCQ, also allows continuous SPJ queries to be expressed. NiagaraCQ also allows more complicated operations, such as Email, to be accomplished by calling out to a function declared in a stored procedure language. The format of a NiagaraCQ query is:

```

CREATE CQ_name XML-QL query
DO action
{START s_time} {EVERY time_interval} {EXPIRE e_time}

```

In the example, the “query” part would consist of the XML-QL equivalent of selecting house information for those that met the desired search criteria. The “action” part would be something similar to “MailTo:user@example.com”.

Generally, the NiagaraCQ query language has the same limitations that the TelegraphCQ language does when it comes to the flexible monitoring of sources. There is no ability to interleave gathering of data with navigation (in fact,

NiagaraCQ assumes that Homeseekers can be queried as an XML source that provides a single set of XML data). There is also no support for actions (like e-mail) based on differentials of data monitored over some period of time. Although NiagaraCQ allows one to write a stored procedure that could accomplish this task, it requires a separate programming task and its execution is not necessarily as efficient as the rest of the query. Finally, as is the case in TelegraphCQ, there is no way to terminate a query other than by temporal constraints.

In summary, I have provided a qualitative analysis of how the continuous query languages of network query engines compare to the one presented here. As described, while languages such as TelegraphCQ and NiagaraCQ do allow some ability to monitor sources via continuous queries, these languages are not as flexible as the one presented in this paper. Specifically, these languages lack the ability to terminate monitoring for non-temporal reasons, cannot easily notify a user about source updates, and lack the ability to query sources that require more complicated types of data gathering (i.e., that requires interleaved navigation and gathering).

2.6.4 Hypothesis 3: Increased expressivity does not impact performance

Though it has been demonstrated that Theseus performs well on more complex information gathering tasks, it is useful to assess whether the increased expressivity in Theseus impacts its performance on simpler tasks – in particular, ones that network query engines typically process. To do this, I measured the performance of Theseus on a more traditional, database style query plan for online information gathering and compared it to the same type of plan executed by a network query engine.

To measure the performance of their partial results query processing technique (Raman and Hellerstein 2002) ran a query that gathered data from three sources and then joined them together. The specific query involved gathering information on contributors to the 2000 U.S. Presidential campaign, and then combined this information with neighborhood demographic information and crime index information. Table 2.3 lists the sources and the data they provide. “Bulk scannable” sources are those where the data to be extracted can be read directly (i.e., exists on a static Web page or file). “Index” sources are those that provide answers based on queries via Web forms. Index sources are thus sources which require binding patterns. Table 2.4 shows the query that was used to evaluate the performance of Telegraph.

Source	Site	Type of data
FEC	www.fec.gov	Bulk scannable source that provides information (including zip code) on each contributor to a candidate in the 2000 Presidential campaign.
Yahoo Real Estate	realestate.yahoo.com	Index source that returns neighborhood demographic information for a particular zip code.
Crime	www.apbnews.com	Index source that returns crime level ratings for a particular zip code.

Table 2.3: Data sources used in (Raman and Hellerstein 2002)

Query
<pre>SELECT F.Name, C.Crime, Y.income FROM FEC as F, Crime as C, Yahoo as Y WHERE F.zip = Y.zip and F.zip = C.zip</pre>

Table 2.4: Query used by (Raman and Hellerstein 2002)

It is important to note that (Raman and Hellerstein 2002) measured the performance of the query in Table 2.4 under standard pipelined mode and compared this with their JuggleEddy partial results approach. We are only interested in the results of the former, since this is a measure of how well a *non-optimized* network query engine – what I call the “baseline” – gathers data when processing a traditional, database-style query. Any type of further optimization, such as the JuggleEddy, are complementary to the system described here. Since both types of systems rely on streaming dataflow execution consisting of tuples routed through iterative-style query operators, it would not be difficult to extend the system described here to support this and other types of adaptive query processing techniques.

I wrote a simple Theseus plan that allowed the query in Table 2.4 to be executed. The same sources were used; however, it was clear that the latency of the Crime source had increased substantially, as compared to the time when (Raman and Hellerstein 2002) ran their tests. Instead, I used another source (Yahoo Real Estate) but added an artificial delay to each tuple processed by that source, so that the new source performed similarly. The results of (Raman and Hellerstein 2002) show that the performance of their pipeline plan was as slow as the Crime source, and about 250ms per tuple (Raman 2002). To match this, I added a 150ms delay to each tuple of processing for this new source, Yahoo, which was normally fetching data at about 100ms per tuple. The results are shown in Figure 2.17.

The results show that Theseus was not only able to execute the same plan at least as fast as the “baseline” Telegraph plan, the non-optimized result shown in Figure 8 of (Raman and Hellerstein 2002), but Theseus execution can be more efficient depending on the number of threads in the thread pool. For example, Theseus-3 describes the case where the Theseus thread pool contains 3 threads. The result from this run performs slightly worse than the Telegraph baseline – such minor differences could be due to changes in source behavior or in different proximities to network sources.

However, running Theseus with more threads in the thread pool (i.e., Theseus-6 and Theseus-10) shows much better performance. This is because the degree of vertical parallelism demanded during execution can be better accommodated with more threads. More specifically, the thread pool architecture prevents operator input from queuing up the way it would when only a single thread was permanently assigned to each operator. It should be noted that the reason Telegraph does not perform as well as Theseus-6 and Theseus-10 is likely because that system only assigned a single thread to each operator (Raman 2002). That is, Theseus-6 and

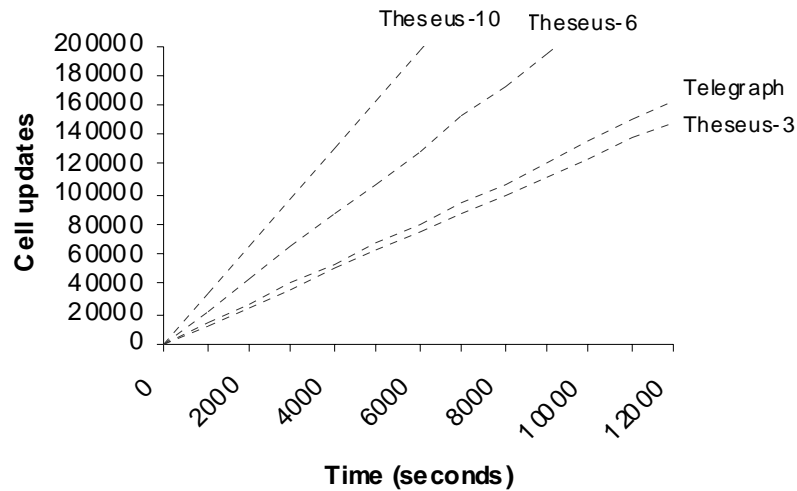


Figure 2.17: Comparing Theseus and Telegraph performance

Theseus-10 execution uses 6 and 10 concurrent threads on a single processor, respectively, whereas the Telegraph plan uses only 3 concurrent threads.

2.7 Summary

In this section, I have described an information gathering plan language and execution system that, taken together, enable more expressive and efficient information gathering plans. The plan language is unique because of the expressivity it provides. Through its rich operator set and support for subplans and recursion, plans that address more complex information gathering tasks can be built. The language represents an improvement over plan languages found in other network query engines because of its expressivity and its accessibility. While somewhat related to older dataflow-style languages and more recent embedded systems languages, the language presented in this paper is unique because it focuses on high-level operators that process incoming streams of relational data.

In addition to the language, I have presented a design for an efficient executor. The executor functions as a true streaming dataflow machine that enables plans to realize significant levels of horizontal and vertical parallelism at runtime. In doing so, the executor uses a thread pool to achieve significant concurrency without exhausting resources.

Chapter 3

Speculative Plan Execution

The streaming dataflow model of execution for information gathering plans generates significant parallelism and, as a result, improves plan performance. For example, as described in the last chapter, the original CarInfo execution time of 13400ms can be reduced to less than a third, about 4200ms. The combination of horizontal and vertical parallelism thus yields a speedup of about 3.19.

Despite the benefits of streaming dataflow, agent plans often remain significantly I/O-bound. For example, almost all of the remaining 4200ms second execution time in CarInfo is devoted to waiting for data from remote sources. This is not unusual for Web information agent plans, which focus on gathering and combining data from multiple online sources. Incurring network latencies for plans like CarInfo that query remote sources are unavoidable: if we want the data from a particular source, and we have no administrative control over that source, then we are forced to wait for as long as the source takes. Usually, querying a single source does not cause a noticeable degree of latency during execution. However, querying multiple data-dependent sources in sequence can often lead to a noticeable aggregate latency.

Unfortunately, the nature of information integration is such that there are often data dependencies, or *binding patterns*, between sources: that is, plans often need to gather data from one source and then use it to query another. Furthermore, information networks like the Web are designed to be browsed interactively by the user, requiring additional navigation in order to obtain a final answer (such as the details of a house or the full review of a car). Additional navigation typically involves chasing “Next Page” or “Details” links from a previous page, translating into even more data-dependent remote fetches. This further increasing the sequential nature of the plan, leading to greater overall aggregate latencies and slower plans.

Thus, one of the primary remaining challenges associated with increasing the performance of Web query plans has to do with improving the extent to which flows that contain binding-pattern relationships can be parallelized. For example, in the CarInfo plan, it is not normally possible to query NHTSA safety ratings and ConsumerGuide car reviews until Edmunds returns the list of cars that meet the initial search criteria. If we could somehow parallelize the gathering of ratings and reviews with the Edmunds search, the overall execution time would be dramatically improved. Unfortunately, this does not make logical sense: we cannot gather safety ratings and car reviews until we know which cars that we need ratings and reviews

for. In short, the data dependencies between operators in a plan determine its performance barrier. The maximum parallelism of a plan given its data dependencies is better known as the *dataflow limit*.

3.1 Exceeding the dataflow limit with speculative plan execution

To combat the natural dataflow limit of a plan, I introduce a new form of run-time parallelism: *speculative plan execution*. The general intuition behind this technique is the use of hints received at earlier points in execution to generate speculative input data to dependent operators that occur later in a plan and execute them ahead of their normal schedule. Through this method, consumer operators that are dependent on slow producers can be executed in parallel with those producers, using the input to those producers as hints about how to execute.

In speculative plan execution, the knowledge of how hints are associated with predictions occurs over time. This relationship can either be cached or learned from earlier executions. As more knowledge is gained, predictive accuracy (recall) can improve.

Speculative execution can be an effective technique for overcoming the natural dataflow limit of information gathering plans and obtaining significantly better speedups. To better illustrate the general idea, let us return to the CarInfo plan example presented earlier. Consider the retrievals of the car reviews from ConsumerGuide and the safety ratings from NHTSA. Both activities occur in parallel, but both are dependent on the cars returned from Edmunds based on the user search criteria. As observed earlier, if Edmunds is slow, performance of the rest of the plan suffers.

With speculative execution, however, the input to Edmunds (the price range, the year, the type of car, mileage specifications, etc.) can be used to predict the inputs for the ConsumerGuide and NHTSA wrappers. For example, it could be learned that certain features of the search criteria (such as car type, year, and price range) are good predictors of the car makes and models that Edmunds will return. This would provide a reasonable basis upon which to predict queries to ConsumerGuide and NHTSA – even for input never previously seen. For example, once the system has seen the cars that the search criteria of (*Midsize coupe/hatchback, 2002, \$4000, \$12000*) returns, it is possible to make reasonable predictions about the cars that the criteria (*Midsize coupe/hatchback, 2002, \$5000, \$11000*) will return.

In this example, note that there is no limitation related to speculation about only one set of cars – in fact, there is no reason why the system cannot speculatively execute retrievals for multiple sets of cars to improve the chances for success. For example, from prior executions, the system could learn that a price range of \$4000-\$12000 returns a result set RS_1 and a price range of \$8000-\$16000 returns a result set RS_2 . When given a new criteria of \$6000-\$14000, the system could predict both RS_1 and RS_2 . Identifying exactly the correct subset occurs during the processing of the search at Edmunds. However, the capability to issue multiple sets of predictions at once allows us to have the best of both worlds – hedging both predictions – and confirming only those speculations that turn out to be correct. Speculatively

executing the same path with multiple data can thus often be useful when hints map to multiple answers.

Speculative plan execution can enable the fetching of data from Edmunds, NHTSA, and ConsumerGuide to be run in parallel. Since all three tasks are almost entirely I/O-bound, using separate threads for each can result in almost true concurrent execution. It is important to note that we cannot speculate without caution, however. In particular, it is important to be careful about how the output from the final Join operator is handled – that is, data should not exit the plan until the earlier predictions that led to it have been verified as correct.

In summary, this discussion of speculatively executing information agent plans has raised three important requirements. Specifically, for any approach, it is important to:

- **Define a process for speculation and confirmation:** It is important to specify how speculative execution actually works – what triggers it, how are predictions made, etc.
- **Ensure safety:** Speculative execution must be prevented from triggering an unrecoverable action (such as the generation of output or the execution of an operator affecting the external world) until earlier predictions has been verified. Thus, all speculation must be *confirmed*.
- **Ensure fairness:** Speculative execution should not be prioritized at the same level as normal execution. Its resources demands should be secondary. For example, the CPU should not be processing speculative instructions while normal instructions await execution.

In the next subsections, I describe an approach in terms of each of these three requirements. The subject of value prediction, which directly affects the utility of speculative execution, is addressed in detail in Chapter 4.

3.2 Speculation and confirmation

The process I introduce for enabling speculative plan execution involves augmenting a standard information agent plan with two additional operators. The first, **Speculate**, is a mechanism for using hints to predict inputs to future operators, and later for correcting or confirming those predictions. The second operator, **Confirm**, halts the flow of speculative data beyond “safe points” in a plan until earlier predictions can be confirmed or corrected.

Figure 3.1 shows how these operators are deployed in a transformation of CarInfo for speculative execution. As the figure shows, a Speculate operator receives its hint (the search criteria) and uses it to generate predictions about car models. These cars, in turn, drive the remainder of execution, while the first part of execution continues. Note that the final Join can also be executed – the only requirement is that a Confirm operator be the last operator in the plan. This prevents speculative results from exiting the plan until Speculate has confirmed its predictions.

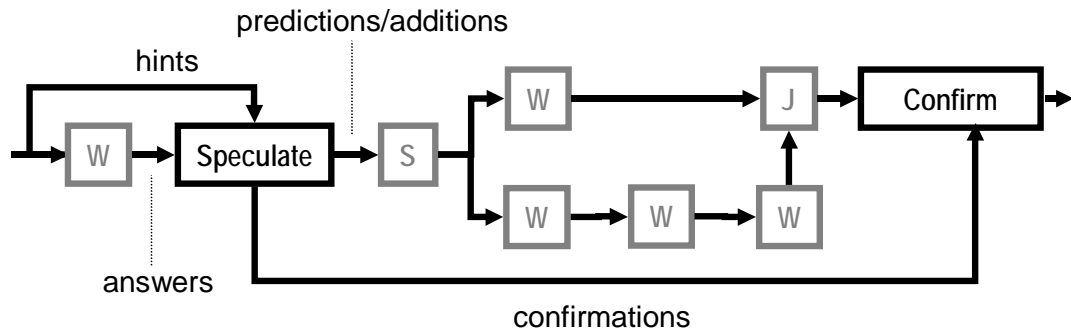


Figure 3.1: The CarInfo plan, modified for speculative execution

The inputs and outputs of the Speculate operator are summarized in Figure 3.2. As the figure shows, this operator receives *hints* (input data to an earlier operator in the plan) and uses those hints to generate data *predictions* (used as input to operators later in the plan). These predictions are tagged as speculative; any further results they lead to are also tagged. Later, Speculate receives *answers* to its earlier predictions from the operator normally producing this data. Using these answers, *confirmations* can be generated to validate prior predictions. Any data errantly predicted is not confirmed and data that was never predicted is eventually forwarded via the predictions/additions output, without being tagged.

For example, in Figure 3.1, search criteria is used to predict cars *X* and *Y*. This triggers the gathering and combining of safety ratings and car reviews, with the combination (joining) of this data held up at the Confirm operator. At the same time, suppose that the Speculate operator receives an answer that indicates that the real cars were *X* and *Z*. It can subsequently route confirmation for *X* to the Confirm operator. In contrast, *Y* is not confirmed because no such answer was received from Edmunds. In addition, *Z* is not tagged speculative and is propagated through to the ConsumerGuide, NHTSA, and Join operators. Note that *Z* does not require confirmation because it was never predicted (Confirm allows tuples not tagged for confirmation to pass through). As this example demonstrates, because Speculate operates at the tuple level, corrections to its predictions are fine-grained and require only the minimum amount of additional work be done to correct a mistaken prediction.

The behavior of the Confirm operator is similar to that of a relational Select – it acts as a filter on a set of incoming tuples. Figure 3.3 illustrates its inputs and outputs: *probable_results* are the incoming speculative tuples, *confirmations* are generated by the Speculate operator, and *actual_results* are the filtered (correct) results. The role of Confirm is to guard against the release of unconfirmed or errant

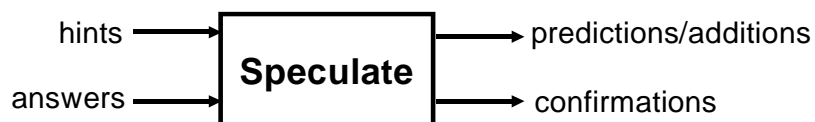


Figure 3.2: The Speculate operator



Figure 3.3: The Confirm operator

tuples beyond a safe point in the plan. The main way it differs from a relational Select operator is in how it uses the confirmations data as a filter to halt *probable_results* tuples until each has been confirmed.

Note that this approach exploits the fine-grained property of execution that data steaming provides. By basing production of verified results on confirmations – instead of errors – correct data can be output as soon as possible, without waiting for the remaining corrections to be processed. Confirm will continue to wait for corrections until it receives an EOS (controlled and propagated by Speculate).

A final note about the Confirm operator has to do with the nature of the confirmations input. In Figure 3.3, it is shown as a single input. However, as supported by the language in Chapter 2, this input is actually a variable stream input. That is, it accepts multiple producers of the same data (each producer sending its own EOS) and unions together all of these streams. In this way, multiple producers of confirmations (i.e., multiple Speculate operators) can share the same Confirm operator. The advantage of this will become clear in later sections of this chapter.

3.2.1 Safety and fairness

Ensuring safety during speculative execution means preventing errant predictions from affecting the external world in unrecoverable ways. As described above, the Confirm operator ensures safety by only producing verified results as long as it is correctly placed in a transformed plan. To maximize the benefits of speculative execution while ensuring correctness, Confirm is placed as far as possible along a speculative path, occurring just prior to plan output or an “unsafe operator”. This allows speculation to parallelize sequential flows as much as is safely possible. For example, in Figure 3.1, Confirm is located just prior to plan output.

Ensuring fairness means guaranteeing that normal execution is prioritized over speculative execution in terms of access to resources. For information gathering plans, the primary three resources to be concerned about are processing power (CPU), physical memory (RAM), and network bandwidth. Using existing technology, fairness with respect to the CPU can be ensured by the operating system. During execution, operators for information gathering systems are associated with threads and processing occurs at the tuple-level. By maintaining a pool of standard-priority “normal threads” and a pool of lower-priority “speculative threads”, the former can be used to handle the firing of operators under normal execution while the latter can be used for speculative execution. Standard operating system thread scheduling thus ensures that speculative CPU use never supersedes normal CPU use.

Memory can be metered by pooling objects. Operators can be written such that they draw memory from different pools, based on whether the objects being processed have been tagged as speculative. If so, new objects can be allocated from the speculative pool of those objects. The sizes of these pools can be adjusted as

necessary, based on how much physical memory is allocated for speculative processing.

In terms of bandwidth, the goal is again to make sure that speculative use of bandwidth does not interfere with normal requests for bandwidth. Bandwidth reservation schemes such as RSVP (Zhang et al. 1993) are one way to provide such guarantees. In addition to hardware-based (e.g., network switch bandwidth provisioning) and software-based (e.g., TCP/IP socket configuration) methods, network resources can also be controlled by limiting the number of speculative threads and handles to network connection objects. This is similar to the solution for limiting memory use. A fixed number of threads and connection objects limits the number of simultaneous speculative use of resources and thus can assist in bounding the amount of speculative bandwidth (or any other resource) concurrently demanded.

3.2.2 Optimistic performance benefits

The maximum, or optimistic performance benefit resulting from speculative execution is equal to the minimum possible execution time of a transformed plan. Calculating this requires computing the minimum execution times for each of the independent sequential flows of the plan and then choosing the maximum value of that set. Using the minimum execution time for each flow implies all predictions are correct and no further additions are needed.

For example, consider the optimistic performance of the plan in Figure 3.1. This plan shows three paths of concurrent execution: the Edmunds flow f_a , the NHTSA speculative flow f_b , and the ConsumerGuide speculative flow f_c . If we again assume that all network retrievals take 1000ms per tuple and all computations (Select, Join, Speculate, and Confirm) each take 100ms per tuple, the resulting flow performance for the first tuple is:

$$\begin{aligned} f_a &= 1000 + 100 + 100 = 1200 \text{ ms} \\ f_b &= 100 + 100 + 1000 + 100 + 100 = 1400 \text{ ms} \\ f_c &= 100 + 100 + 1000 + 1000 + 1000 + 100 + 100 = 3400 \text{ ms} \end{aligned}$$

Since the original time to first tuple (using these assumed values) would have been 4200ms, the potential speedup due to speculative execution in this case is $4200\text{ms}/3400\text{ms} = 1.24$. Note that if Edmunds had been very slow, say 3200ms per tuple, overall original performance would have been slower (6400ms) and potential speedup ($6400\text{ms}/3400\text{ms} = 1.88$) greater.

3.3 Achieving better speedups

While a speedup of about two allows the execution time to be nearly halved, producing noticeable results, there is room for improvement. At first, it might not seem possible – since all speculation must be confirmed, execution time appears bound by either the time to perform speculative work or the time to process its confirmation. For example, in Figure 3.1, we are either bound by the time required by initial and confirming flow f_a or the speculative flows f_b or f_c .

However, three additional techniques can be used to increase the degree of speculative parallelism and the level of predictive accuracy, both leading to

significantly better speedups. The first involves using earlier speculation to drive later speculation, which increases the degree of speculative parallelism at runtime. The second is the concept of speculating multiple times per hint, which increases the level of average predictive accuracy for a particular speculative opportunity. Finally, the third involves leveraging some types of deterministic operators, in order to generate predictions earlier than usual. I discuss all three in detail below.

3.3.1 Cascading speculation

We are not limited to speculating about only one input at a time. In fact, it is possible for speculation about one input to trigger speculation about another input and so on, an effect I call *cascading speculation*. When the results of an initial prediction are known, this can trigger confirmation of the second prediction and so on, in effect cascading confirmations.

The performance benefit of cascading is the increase in speculative parallelism it allows, thus making it possible to achieve very high speedups. To illustrate, consider a longer sequence of operators, such as that in Figure 3.4. Using the earlier execution time assumptions, processing 10 Wrapper operators in succession would normally require $(10 \times 1s =)$ 10 seconds. Let us also assume that each operator consumes a single tuple of input and produces a single tuple of output. Predicting input f in Figure 3.4, which occurs midway in the sequence, allows the first and last halves of the plan to execute concurrently, resulting in a new execution time of 5 seconds and a speedup of 2. With a single Speculate operator, this is the maximum speedup possible.

However, suppose that we wanted to use a to speculate about the input b to second Wrapper, use the speculation of b to predict c , and so on. This is shown in Figure 3.5 (each Speculate operator is denoted by an S ; Confirm by a C). Note that in the case of cascading speculation, one Confirm is still all that is required, as this operator is used to generally verify speculative tuples and requires no knowledge of when or why the speculation occurred³. It simply determines if each answer tuple

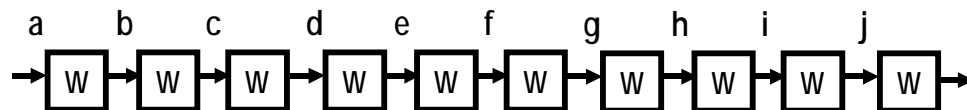


Figure 3.4: A longer sequence of operators

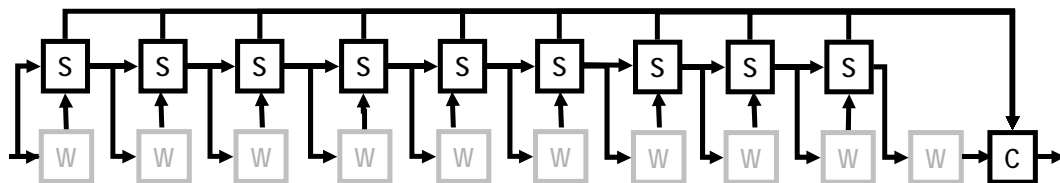


Figure 3.5: Cascading speculation of the sequence in Figure 3.4

³ Recall that the Confirm operator can take a variable number of confirmation inputs. For dataflow plan languages that do not support variable inputs, cascading speculation would still be possible by arranging a sequence of Confirm operators in place of the single Confirm operator shown in Fig 3.5.

either speculative output or a product of earlier speculative output. If so, the tuple is held up until the confirmation(s) for that tuple have arrived.

Since all wrappers require the same amount of time to execute and are all I/O-bound, they would act simultaneously and their confirmations could be processed at once. Thus, the resulting execution time would simply be the duration of a single wrapper call plus the overhead for speculation and the time to process confirmation. Even if we assume that the overhead and confirmation somehow requires an additional 100ms, execution would still only require $1000+100+100=1200\text{ms}$, a speedup of 8.33.

Optimistic performance benefits (revisited)

Figure 3.6 shows a version of the speculative CarInfo plan in Figure 3.1 further modified for cascading speculation. Using earlier timing assumptions, then the five flows require the execution times shown in Table 3.1. Since execution time would be limited to the slowest of these flows, the optimistic speedup for the first tuple would be $(4200\text{ms}/1600\text{ms} =) 2.63$.

Plan flow	Execution time (ms)
Edmunds + Spec + Confirm	1200
Spec + Select + CG Search + Spec + Confirm	1400
Spec + Select + Spec + CG Summary + Spec + Confirm	1500
Spec + Select + Spec + Spec + CG Full + Join + Confirm	1600

Table 3.1: Optimistic execution times for CarInfo flows shown in Figure 3.6

Intuitively, cascaded speculation seems to make the most sense for navigational sequences, such as the three successive fetches from ConsumerGuide in the CarInfo plan. Many Web sources present a visual view of an underlying relational database schema. HTML pages are programmatically generated and thus navigation to certain data often tends to follow some simple URL patterns. Once prediction to the initial page is confirmed, all subsequent navigation is almost always verified because it predictably follows from the first page. Thus, for information gathering plans that speculate about interleaved navigation, cascading speculation can often overcome the cost of interleaved navigation.

This specific case occurs in the CarInfo plan. Consider the lower half of the plan in Figure 3.1, where ConsumerGuide is queried for car reviews. Once the

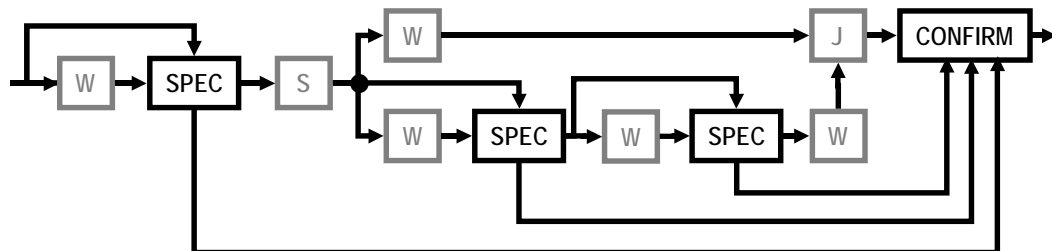


Figure 3.6: CarInfo modified for cascading speculation

dynamic part of the target URL is discovered (the car ID, “20812” in the case of the Dodge Stratus example earlier), the subsequent navigational pages are predictable. As a result, use of cascading speculation can easily yield a speedup of 3 for this interleaved navigation sequence.

Average performance benefits

Earlier, I discussed the optimistic performance benefits of a single speculation during execution. Although those simple calculations give us some idea as to the potential maximum benefit of the technique, we must also consider cases where predictions are incorrect, and where the total mix is defined by a subset of correct predictions and a subset of incorrect predictions.

To do so, we must figure in the probabilities of a particular speculation being correct. Let us start by defining the conditional probability that a prediction v will be correct given hint h as:

$$\Pr(v | h)$$

Recall from Chapter 2.4 that a plan flow is any sequence of operators that spans the entire width of the plan (from consumption of some plan input to production of some plan output – or production of no output). Thus, a given plan P with a set of operators $Ops = \{Op_1...Op_m\}$ contains a set of data flows:

$$F = \{f_1...f_n\} \text{ s.t. each flow } f_x = \{Op_b...Op_c\}, 1 \leq b, c \leq m$$

The average time T it normally takes to execute a particular flow f_x is:

$$T(f_x) = \sum_{y=b}^c T(Op_y) \quad (1)$$

Per Amdahl’s Law, the average performance of a plan is the most expensive path f_{mep} , which is:

$$\forall f_x \in F, f_{mep} = \mathbf{max} (T(f_x)) \quad (2)$$

However, speculative execution of operators in that flow creates new flows, per the number of speculations occurring along that flow. Furthermore, some of these new flows are non-standard because the Speculate operator may play one of two roles: in some cases, it may act as the termination point for a tuple (if a prior prediction made was correct) and in other cases it might not. In short, there is a probability associated with the pursuance of some flows in plan transformed for speculative execution.

For each of these new flows, we must consider that the probability of any one speculation in a cascade is, to an extent, dependent on the probability of any prior prediction. That is, the *prediction probabilities cascade*. In basic probability theory, two events A and B are said to be *independent* if

$$\Pr(A \cap B) = \Pr(A) * \Pr(B) \quad (3)$$

where $\Pr(A \cap B)$ is the probability that both events occur. Under speculative execution, the likelihood of each prediction being correct is independent, and thus

the likelihood of a particular set of speculative outcomes is based on the product of all probabilities involved.

More precisely, there is an **independent probability** and **dependent probability** of a prediction being correct. Intuitively, the former refers to the likelihood that a prediction will be correct *given a correct hint*. In contrast, the dependent probability of a prediction being correct is the likelihood that a prediction will be correct *given a hint that may not be correct*. More formally, the independent probability IPr_i for the i^{th} speculation in a cascading set of speculations $i=\{0..n\}$ is simply the likelihood of that particular prediction being correct assuming that the hint is correct. The dependent probability DPr_i is the likelihood that the hint will be correct multiplied by the independent probability, which can be expressed as:

$$\text{DPr}_i(v_i | h_i) = \text{IPr}_i(v_i | h_i) \cdot \prod_{j=0}^{i-1} \text{IPr}_j(v_j | h_j) \quad (4)$$

Thus, the dependent probability is the overall likelihood that a particular prediction will be correct in a cascading speculation environment.

Another way to express this is in terms of the flow schedules. Assume that a particular flow f_x contains a Speculate operator. This flow will have two execution schedules, S_{succ} and S_{fail} where the former is the flow schedule followed during a correct prediction and the latter is the flow scheduled during a failed prediction. For example, in Figure 3.1, $S_{\text{succ}}=\{\text{Wrapper}(\text{Edmunds}), \text{Select}, \text{Speculate}, \text{Confirm}\}$. This is the optimistic case – it is the same as flow f_a defined in 3.2.2. However, the other flow schedule S_{fail} is where the predictions made by the Speculate operator are not correct. In this case, $S_{\text{fail}}=\{\text{Wrapper}(\text{Edmunds}), \text{Select}, \text{Speculate}, \text{Wrapper}(\text{ConsumerGuide Search}), \text{Wrapper}(\text{ConsumerGuide Summary}), \text{Wrapper}(\text{ConsumerGuide Full}), \text{Join}, \text{Confirm}\}$.

There are two important notes to make with respect to the execution times of each schedule, defined as $T(S_{\text{succ}})$ and $T(S_{\text{fail}})$. One is that the failed speculative execution schedule is the same as the successful schedule with the important exception that the operators following the Speculate operator will need to be executed again – this time, for the correct data. Thus:

$$T(S_{\text{succ}}) < T(S_{\text{fail}}) \quad (5)$$

The second is that S_{succ} and S_{fail} are the only two possible execution schedules possible for each speculative opportunity along a flow. Thus, the average time it takes the original flow f_x to execute is the sum of the probabilities of each schedule multiplied by the execution time of each. More precisely:

$$T(f_x) = \text{Pr}(S_{\text{succ}}) \cdot T(S_{\text{succ}}) + \text{Pr}(S_{\text{fail}}) \cdot T(S_{\text{fail}}) \quad (6)$$

As a detailed example of how one can calculate the average performance of a flow that is speculatively executed, consider the flow in CarInfo that includes the Edmunds wrapper, the Select, the three wrappers for ConsumerGuide and the Join. Hereafter, this flow is referred to as f_{car} . As Figure 3.6 shows, three speculations are performed among the four wrappers on f_{car} . To simplify this example, let us focus on the case where the user search criteria returned a single car from Edmunds.

To calculate the average performance of f_{car} , let us first recall that the original streaming dataflow plan execution time is 4200ms, as described when the example was introduced in Chapter 1.

The next step is to assign probabilities to each event (speculation). For purposes of example, let us suppose that the independent probabilities of correct prediction for each of the three wrappers are as shown in Table 3.2. For example, the table shows that there is a 93% probability that, based on past results, the list of cars will be correctly predicted based on the specified search criteria.

Name	Prediction	Probability of success
P1	<i>Car list given (user search criteria)</i>	0.93
P2	<i>CG summary URL given car list</i>	0.95
P3	<i>CG full review URL given CG summary URL</i>	0.91

Table 3.2: Independent probabilities of each speculative opportunity

Next, let us enumerate the possible execution schedules of the modified f_{car} , in terms of the success of the various predictions possible. This is shown in Table 3.3, along with the probability of each flow schedule occurring. The flow schedule probability is simply the aggregate product of independent probabilities shown in Table 3.2, as specified by (4). Note that Table 3.3 does not include “lucky” cases where a missed earlier prediction may still somehow lead to a correct prediction later down the line. Such cases are unlikely.

Schedule	P1	P2	P3	Probability
S1	Y	Y	Y	0.804
S2	Y	Y	N	0.080
S3	Y	N	N	0.047
S4	N	N	N	0.070

Table 3.3: Likelihood of various f_{car} execution schedules

We can then use the probability of each execution schedule to determine its “contributing” time to the overall execution time. This is done by multiplying the probability shown in Table 3.3 with the normal execution time of each possible schedule. This is shown in Table 3.4.

Schedule	Normal	Contributing
S1	1500	1206
S2	2500	199
S3	3500	163
S4	4600	322

Table 3.4: Execution schedule probability and normal/contributing performance

Finally, the average execution time of the entire flow f_{car} is simply the summation of the contributing values: $1206+199+163+322 = 1890ms$, representing an average speedup of $(4200ms/1890ms =) 2.22$.

3.3.2 Simultaneous speculation

A second technique that can lead to better speedups for speculative plan execution is *simultaneous speculation*, the concept of making multiple sets of predictions. This technique acts as a “hedging” device for a Speculate operator; even if predictions about some tuples are incorrect, others may be correct and the additional number of predictions can increase the likelihood of 100% accuracy. Note that perfect accuracy does not mean that only the necessary set of predictions were made – rather, it means that the complete set of correctly predicted tuples was a subset of those actually predicted.

The net effect on multiple predictions per hint is equivalent to a summation of the probabilities that each hint is correct:

$$\Pr(V | h) = \sum_{i=0}^k \Pr(v_i | h), \text{ where } V = \{v_0 \dots v_k\} \quad (7)$$

Though simultaneous speculation can increase the degree of predictive accuracy, it is important to limit how many additional speculations are made on behalf of a single hint. Too many speculations can increase the overhead of speculative execution in several ways. First, each speculation leads to additional speculative work by one or more threads. In the case of CarInfo, each extra prediction of what Edmunds might return requires work by at least 6 threads (one for each normal operator) + 3 additional threads (two additional Speculate and one Confirm operator), a total of 9 threads. Note that if any operator subsequent to the first Speculate had generated multiple outputs per input, this would have been much worse. Overall, the key point is that multiple speculations consume more speculative resources.

A second way that multiple speculations can increase overhead is by severely impacting a resource. For example, if a 100 different cars from Edmunds are predicted based a single hint (when in fact there are only 3 or 4 actual answers), the NHTSA and ConsumerGuide websites might be adversely affected by the additional load placed on their servers, which in turn affects the execution of the CarInfo plan. Obviously, the issue of load is one that varies tremendously among resources like Web sites. Thus, it is not possible to use a single static cost or static cost function – the cost for each resource must be assessed separately.

However, for certain scenarios, multiple speculations are a reasonable and effective way to increase predictive accuracy. For example, if a Speculate operator needed to predict the result from a weather forecasting site, there may only be a few possible predictions (e.g., “sun”, “clouds”, “rain”, “snow”, or “wind”). If the forecasting site is slow, it may be worthwhile to predict all five, knowing that only one will eventually be confirmed. By predicting all five, there is a guarantee that predictive accuracy will be 100%.

3.3.3 Leveraging antecedent and subsequent functional dependencies

A third technique for improving speedups involves leveraging antecedent and subsequent functional dependencies of a predicted operator, enabling more efficient hint generation and prediction. Leveraging antecedents involves using the *earliest possible form of a hint* to elicit a prediction. Leveraging subsequents involves using the *latest possible form of a prediction* during speculation.

Functionally dependent operator inputs and outputs

Recall that if a database schema is considered as a single relation R consisting of attributes $A_1..A_n$, a *functional dependency* (FD) $A_i \rightarrow A_j$ describes a constraint between A_i and A_j , specifically that the latter is functionally dependent on the former. Simply put, this means that A_j is determined by A_i . Several inference rules about FDs exist, including the transitive rule, which states:

$$\forall \text{ attributes } A_x, A_y, A_z : \{A_x \rightarrow A_y, A_y \rightarrow A_z\} \models A_x \rightarrow A_z$$

Let us extend the concept of FDs to an information agent plan. Recall that all plan operators work by performing a function over the set of input, leading to a set of output. Let us focus on the subset of operators that are deterministic and produce a single stream of output where each tuple of output is a function of each tuple of input. More precisely, the cardinality between the main input and output streams is one to one (1:1). For example, in the plan language described in Chapter 2, the Project, Format, Apply, and Null operators have this characteristic. The same is also true for the Speculate operator introduced in this chapter, but only when the cardinality of hints to predicted values is 1:1 (this cardinality, in turn, depends on the cardinality of the operator being predicted). We can then say that, for these operators, the main output A_y of an operator Op_i is functionally dependent on its main input stream A_x , or that the FD $A_x \rightarrow A_y$ exists.

Leveraging functional dependencies

Functional dependencies are useful for speculative plan execution because they allow predictions to be issued as far ahead as possible and confirmations to be issued as early as possible. More specifically, we can make predictions based on the earliest form of a hint from a sequence of FDs. We can also route the predictions issued past any FD sequence that follows the operator to be predicted. This means that predictions can reach later parts of the plan faster. Also, it allows us to reorganize the flows that are executed in parallel, potentially reassigning the subsequent FD operators on the right-hand side (RHS) to the left-hand side (LHS) of the predicted operator for greater efficiency.

As a detailed example, consider the case where a Format operator feeds a Wrapper operator, which feeds a Project operator, which feeds another Wrapper operator. This is shown in Figure 3.7. For purposes of example, let us assume that

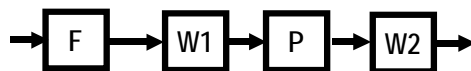


Figure 3.7: Short FD example flow

W1 requires 1000ms, W2 requires 1200ms and the remaining CPU-bound operators require 100ms per tuple. Thus, this short flow requires $(100+1000+100+1200=)$ 2400ms to run.

Next, suppose that we wish to speculate about the output of the first Wrapper operator W1. Normally, as described earlier in this chapter, we would use the output from Format as the basis for hints and the Project that follows W1 as the target for prediction. This is shown in Figure 3.8. Under optimistic circumstances, this requires $\max((100+1000+100+100=1300), (100+100+1200+100=1500)) = 1500$ ms, a speedup of $(2200\text{ms}/1500\text{ms}=)$ 1.47.

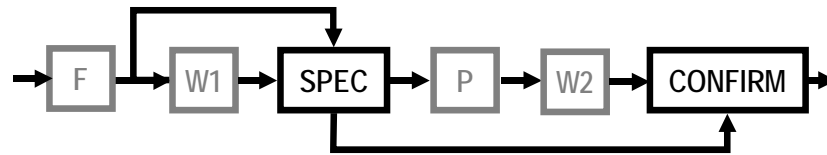


Figure 3.8: Flow in Figure 3.7 modified for speculative

As described above, Format is deterministic and has a 1:1 input/output cardinality between its input and output, using the attribute A_0 to produce A_1 . Thus, the FD $A_0 \rightarrow A_1$ exists. If W1 also has a 1:1 input/output ratio, then so will the Speculate operator and we can establish that the FD $A_1 \rightarrow A_2$ will exist. As a result, we can infer $A_0 \rightarrow A_2$. This is important because it allows us to use the *input* to the Format as the basis for issuing predictions from Speculate. Thus, we can issue predictions earlier, as shown in Figure 3.9. As a result, the new optimistic performance is 100ms better (execution of the speculative flow does not require the initial Format processing time) at 1400ms, an improved speedup of $(2200\text{ms}/1400\text{ms}=)$ 1.59.

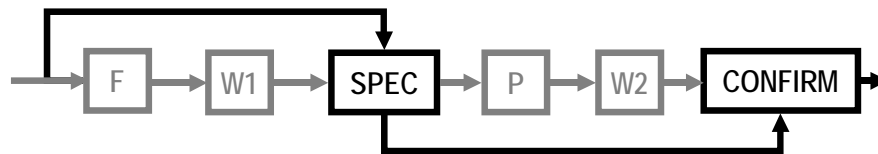


Figure 3.9: Leveraging the determinism of the Format operator

In addition, since the Project operator is deterministic and 1:1, we know that $A_2 \rightarrow A_3$ exists and can thus infer $A_1 \rightarrow A_3$. Since the speculative flow (which includes W2) is slower than the confirming flow (which includes W1), we can move as many subsequent FD operators from the RHS to the LHS. In this case, there is only one such operator, the Project, and its repositioning is shown in Figure 3.10. This allows us to further reduce the time required by the speculative (but still dominant) subflow, down to 1300ms and thus gives us the improved speedup of $(2200\text{ms}/1300\text{ms}=)$ 1.69, roughly a 15% increased speedup over the initial speculative speedup.

It is important to note that while saving execution time, leveraging antecedents and subsequents may also result in changes to the cost of speculative execution,

specifically in terms of the amount of memory required. For example, in Figure 3.8, the hint values were the result of a Format, and were thus larger than the hint values in Figure 3.9, which occur before the Project. Thus, potentially less memory is required for speculation. The same is true when the RHS Project can be reassigned, as shown in Figure 3.10, as the tuples output from Project require less space than the tuples input. In general, the amount of extra/less memory due to leveraging antecedent/subsequent determinism is a product of the nature of the deterministic operators and the number of tuples processed.

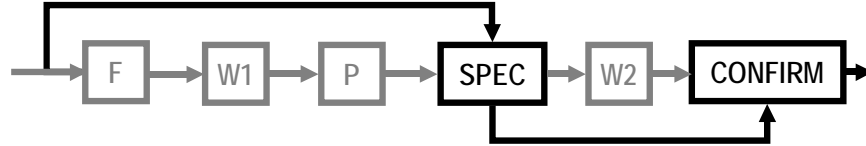


Figure 3.10: Leveraging the determinism of the Project operator

The benefits of exploiting functional dependencies

In terms of execution time savings and increased speedup, as the above example demonstrates, the benefit of leveraging deterministic antecedent and subsequent FDs depends on the cost of the flows prior to and following the Speculate operator. I now describe these potential benefits more generally.

Let t_{lhs} be the time required by the “hint/confirmations flow”, set of operators prior to the Speculate operator. Next, suppose t_{det_lhs} is the time required by the sequence of the FD operators on the answer flow that immediately precede the predicted operator (e.g., W1 in Figure 3.8). Thus, $t_{det_lhs} < t_{lhs}$. Similarly, suppose that t_{rhs} is the “predictions flow”, the sequence of all flow operators following the Speculate operator. Also, let t_{det_rhs} be the time required by the sequence of deterministic operators on the predictions flow following the predicted operator. Finally, let Op_{pred} be the predicted operator and t_{pred} be the time to execute that operator. Leveraging antecedent FDs is only worthwhile when:

$$\begin{aligned} t_{det_lhs} &> 0 \\ t_{rhs} &> t_{pred} \end{aligned} \quad (8)$$

Leveraging subsequent FDs is only worthwhile when the RHS is as slow or slower than the LHS plus the FD sequence on the RHS. This can be expressed as the case where:

$$\begin{aligned} t_{det_rhs} &> 0 \\ t_{rhs} - t_{det_rhs} &\geq t_{pred} + t_{det_rhs} \end{aligned} \quad (9)$$

Thus, the potential overall gain g and improved execution time ΔT from leveraging both antecedents and subsequents can be summarized as:

$$g = \mathbf{min} (\mathbf{max} (t_{\text{pred}}, (t_{\text{rhs}} - t_{\text{det_lhs}})), \mathbf{max} (t_{\text{pred}} + t_{\text{det_rhs}}, (t_{\text{rhs}} - t_{\text{det_lhs}} - t_{\text{det_rhs}}))) \quad (10)$$

$$\Delta T = (\mathbf{max} (t_{\text{pred}}, t_{\text{rhs}})) - g \quad (11)$$

Equation (10) basically determines if the cost of the predicted operator or the flow that follows is dominating. If the former is the case, no benefits are realized because confirmation is the bottleneck. Otherwise, the use of predictions is the bottleneck. In this case, leveraging antecedents and subsequents is clearly beneficial.

3.4 Automatic plan transformation

In the previous section, I described how speculative plan execution can yield significant performance gains. However, in that example, augmentation of the CarInfo plan was done manually. In this section, I introduce algorithms that enable the automatic transformation of any information gathering plan into one capable of speculative execution.

The overall goal is to maximize the theoretical average performance gain resulting from speculative execution. At the same time, we also need to be wary of the overhead (cost) of speculative execution. Thus, we would like to identify the best speculative transformation P'_i of a plan P , from some larger set of possible transformations $P'_1..P'_m$, that are different transformations of P for speculative execution. More specifically, if the time to execute a particular transformation P'_i is $T(P'_i)$, we want to find the plan P_{best} where:

$$\forall P'_i \in \{P'_1..P'_m\}: P_{\text{best}} = \mathbf{min} (T(P'_i)), 1 \leq i \leq m$$

3.4.1 The set of candidate transformations

One natural way to approach the problem is to first generate the set of all possible speculative transformations and then iterate through this set, applying the equation above to identify the speculative transformation with the best theoretical execution time. Unfortunately, this approach is impractical because the set of all possible speculative transformations is huge.

To demonstrate why this is the case, let us consider how to calculate the number of possible speculative transformations for certain class of very simple information gathering plans that is a subset of the larger set of all possible plans. The class of plans considered are those that:

- (i) are composed of a single, unbroken chain of n operators
- (ii) consist of operators that all have monadic input and output
- (iii) have one plan input and one plan output

For example, the plan shown in Figure 3.11 meets these requirements.

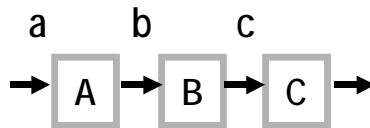


Figure 3.11: Sample plan that meets (i), (ii), and (iii)

To calculate the number of possible speculative transformations of a particular plan, it is assumed that we are only interested in transformations where:

- all speculations involve using the input of an upstream operator as a hint for predicting the input of a downstream operator
- there can be one or more speculations in the plan (i.e., cascading speculation)
- the same downstream input is not predicted by multiple upstream inputs

For example, there are five possible transformations for the plan shown in Figure 3.11, which can be summarized as:

$$((b|a), (c|a), (b|a, c|a), (c|b), (b|a, c|b))$$

This list denotes the set of possible transformation. Each transformation involves one or more instances of using a particular variable as a hint for issuing predictions about another variable. The list above simply describes the hint/prediction pairs for each transformation. The “|” means that the left-hand side variable could be predicted by the right-hand side variable (which always precedes the left-hand side in the plan). For example, the transformation $(b|a, c|b)$ is one where “a” is used to predict “b” and “b” (speculative “b”, that is) is used to predict “c”. Thus, in this example, there are two Speculate operators and one Confirm.

To consider the total number of potential speculative transformations, we observe that for operator sequences of lengths 2, 3, and 4, the total possible number of transformations is 1, 5, and 23, respectively. Generally speaking, the number of transformations for a sequence of length n consists of the number of transformations required for a sequence of $n-1$ plus the additional set of speculative schedules that involve the added operator. Specifically, the total number of possible speculative transformations $ST(n)$ for a particular sequence of n operators for plans is roughly equal to the factorial series for n ; it can be calculated precisely as:

$$ST(n) = (n-1) + n*ST(n-1), \quad ST(1) = 0 \quad (12)$$

As the equation suggests, even simple plans of moderate length can quickly generate a very large number of candidate transformations to evaluate. For example, even under the fairly strict set of assumptions described earlier, a sequence of 10 operators has 3,628,799 possible speculative transformations.

3.4.2 Heuristics to reduce the number of possible transformations

The problem with using a brute force approach to identify the most profitable plan transformation is the factorial blowup of the number of candidate transformations. The problem obviously worsens for larger plans and even more dramatically when we relax earlier assumptions, such as that plans can only consist of a single flow. At the same time, intuition suggests that it is better to focus on how speculation might reduce the impact of major bottleneck operators in a plan, instead of considering every possible speculative opportunity for every possible pair of operators.

We can reduce the size of the candidate transformation set substantially by leveraging Amdahl’s Law, which states that program execution time is a function of

its most latent sequence of instructions. In effect, this suggests that it is not worthwhile to consider transformations that involve operators which do not exist along this sequence because any potential improvement cannot have any impact on overall execution time.

Instead, Amdahl's Law suggests that performance optimization should be focused on the costliest flow in the plan. In particular, we can use a most-expensive-path (MEP) approach that identifies the most latent sequence of operators in an information gathering plan and focuses the generation of candidate transformations on that path⁴. An MEP-based transformation algorithm for a given plan P consists of the following key steps:

1. Find all paths of P and their execution costs.
2. Identify f_{mep} .
3. Identify all possible speculative transformations of f_{mep} , ignoring transformations on operators that execute faster than the overhead of speculating.
4. If at least one transform was found, apply the most profitable transform to the plan and repeat the process. Otherwise, stop.

The key parts of this algorithm are step 2, which reduces the number of possible paths to consider, and step 3, which reduces the number of operators along that path to consider. In addition, the iterative refinement approach gives the above algorithm an anytime property and thus allows refinement to be bounded by some fixed time, if necessary.

3.4.3 The SPEC-REWRITE algorithm

I present a detailed form of the above plan transformation process in the algorithms shown in Figures 3.12a-d. The main algorithm, SPEC-REWRITE, shown in Figure 3.12a and it calls the other three helper functions shown in Figures 3.12b-d. I now described the details of these algorithms and relate their operation to the approach discussed throughout this chapter.

The overall purpose of the SPEC-REWRITE algorithm is to iteratively find the MEP, attempt to optimize it for speculative execution, and then continue the process with respect to the new MEP in the transformed plan. In optimizing the transformation of the MEP, SPEC-REWRITE identifies uses the GET-LHS-INFO and GET-RHS-INFO functions to locate the earliest hint/confirmation producer and the latest possible deployment of predictions, per the discussion on functional dependencies in section 3.3.3.

```

01 Function SPEC-REWRITE
02   Input: oldPlan
03   Returns: newPlan
04   {
05     newPlan  $\leftarrow \emptyset$ 
06
07   do
08     newMep  $\leftarrow \emptyset$ 

```

⁴ The terms “path” and “flow” are henceforth used interchangeably in this chapter.

```

09  bestSpeedup ← 1
10  planPaths ← GET-ALL-PATHS(oldPlan)
11  mepInfo ← GET-MEP-INFO(planPaths)
12
13  foreach operator op ∈ mepInfo.mep
14    lhsInfo ← GET-LHS-INFO(op, mepInfo.mep)
15    rhsInfo ← GET-RHS-INFO(op, mepInfo.mep)
16    opTime ← CALC-OPERATOR-EXECUTION-TIME(op)
17    opOverheadTime ← (2 * PTO) * GET-OPERATOR-TUPLES(op)
18    srcDstInfo ← CALC-SRC-DST-INFO(opTime, lhsInfo, rhsInfo)
19    newMepTime ← srcDstInfo.lhsTime + MAX(opTime, srcDstInfo.rhsTime) + opOverheadTime
20    candSpeedup ← mepInfo.time / newMepTime
21    if candSpeedup > bestSpeedup then
22      newMep ← GENERATE-TRANSFORM-PATH(mepInfo.mep, op, srcDstInfo.srcOp, srcDstInfo.dstOp)
23      bestSpeedup ← candSpeedup
24    endif
25  end
26
27  if bestSpeedup > 1 then
28    if newPlan == ∅ then
29      newPlan ← oldPlan
30    endif
31    newPlan ← REPLACE-PATH(newPlan, mepInfo.mep, newMep)
32  endif
33
34  while newMep != ∅
35
36  return newPlan
37 }

```

Figure 3.12a: The SPEC-REWRITE algorithm

Specific key parts of the SPEC-REWRITE algorithm are:

- **Line 07:** The start of the main loop that iteratively refines the plan based on the current MEP.
- **Lines 10-11:** Find the MEP from the set of all possible paths in the plan.
- **Line 13:** Loop through all operators on the MEP, searching for the most profitable operator to speculate about.
- **Lines 14-15:** Get timing statistics (overall execution time, FD time savings) from both LHS and RHS parts of the flow.
- **Lines 16-17:** Calculate operator execution time and overhead time. CALC-OPERATOR-EXECUTION-TIME returns the average execution time of an operator based on the number of tuples it typically processes and its average execution time per tuple. The predefined constant **PTO** stands for per-tuple overhead, the additional time required per-tuple for context switching and speculation/confirmation processing. It is multiplied by 2 in the Spec-Rewrite algorithm to account for the overhead associated with both Speculation and Confirmation per tuple.
- **Line 18:** Compute times for revised LHS and RHS parts of the MEP, based on whether FDs can be leveraged.
- **Lines 20-24:** Compare resulting speedup to current best speedup and revise best speedup as necessary.
- **Lines 27-32:** Transform MEP into LHS and RHS flows if better speedup can be achieved. Doing this will ensure that iterative refinement of the plan continues.

The GET-MEP-INFO algorithm, shown in Figure 3.12b is one of the helper functions called by SPEC-REWRITE. It returns an object called *mepInfo* that contains information on the most expensive path and the cost of that path. This function is called during each iteration of original plan transformation to locate which flow is the primary plan bottleneck.

```

01 Function GET-MEP-INFO
02 Input: planPaths
03 Returns: mepInfo
04 {
05   mepInfo  $\leftarrow$  new MepInfo
06
07   mepInfo.mep  $\leftarrow$   $\emptyset$ 
08   mepInfo.mepCost  $\leftarrow$   $\emptyset$ 
09
10   foreach path p  $\in$  planPaths
11     curCost  $\leftarrow$  0
12     foreach operator op  $\in$  p
13       curCost  $\leftarrow$  curCost + CALC-OPERATOR-EXECUTION-TIME(op)
14     end
15     if mep =  $\emptyset$  or curCost > mepCost then
16       mepInfo.mep  $\leftarrow$  p
17       mepInfo.mepCost  $\leftarrow$  curCost
18     endif
19   end
20
21   return mepInfo
22 }

```

Figure 3.12b: The GET-MEP-INFO helper function

The functions GET-LHS-INFO and GET-RHS-INFO, shown in Figures 3.12c and 3.12d are very similar, but both are included for completeness. Each function examines the LHS or RHS part of the MEP from the standpoint of the operator to be predicted, calculating overall execution time as well as the execution time savings due to any 1:1 FDs, like Project.

```

01 Function GET-LHS-INFO
02 Input: op, mep
03 Returns: lhsInfo
04 {
05   lhsInfo  $\leftarrow$  new LhsInfo
06
07   lhsInfo.time  $\leftarrow$  0
08   lhsInfo.fdTime  $\leftarrow$  0
09   fdActive  $\leftarrow$  true
10   lhsInfo.srcOp = lhsInfo.origSrcOp = op.pred()
11
12   foreach predecessor operator predOp  $\in$  mep s.t. predOp < op
13     curOpTime  $\leftarrow$  CALC-OPERATOR-EXECUTION-TIME(predOp)
14     lhsInfo.time  $\leftarrow$  lhsInfo.time + curOpTime
15     if fdActive and predOp.succ() input is functionally dependent on predOp output then
16       lhsInfo.fdTime  $\leftarrow$  lhsInfo.fdTime + curOpTime
17       lhsInfo.srcOp  $\leftarrow$  predOp.pred()
18     else
19       fdActive  $\leftarrow$  false
20     endif
21   end
22
23   return lhsInfo
24 }

```

Figure 3.12c: The GET-LHS-INFO helper function

```

01 Function GET-RHS-INFO
02 Input: op, mep
03 Returns: rhsInfo
04 {
05   rhsInfo ← new RhsInfo
06
07   rhsInfo.time ← 0
08   rhsInfo.fdTime ← 0
09   fdActive ← true
10   rhsInfo.dstOp = rhsInfo.origDstOp = op.succ()
11
12   foreach successor operator succOp ∈ mep s.t. succOp > op
13     curOpTime ← CALC-OPERATOR-EXECUTION-TIME(succOp)
14     rhsInfo.time ← rhsInfo.time + curOpTime
15     if fdActive and succOp input is functionally dependent on succOp.pred() output then
16       rhsInfo.fdTime ← rhsInfo.fdTime + curOpTime
17       rhsInfo.dstOp ← succOp.succ()
18     else
19       fdActive ← false
20     endif
21 end
22
23 return rhsInfo
24 }

```

Figure 3.12c: The GET-RHS-INFO helper function

Finally, the CALC-SRC-DST-INFO function is shown in Figure 3.12d. The main purpose of this function is to adjust the hint source and predictions target operator based on the FD information found in GET-LHS-INFO and GET-RHS-INFO, as well as the operator time calculated in SPEC-REWRITE (the caller).

```

01 Function CALC-SRC-DST-INFO
02 Input: opTime, lhsInfo, rhsInfo
03 Returns: flowTimes
04 {
05   srcDstInfo ← new SrcDstInfo
06
07   srcDstInfo.lhsTime = lhsInfo.time
08   srcDstInfo.rhsTime = rhsInfo.time
09   srcDstInfo.lhsOp = lhsInfo.origSrcOp
10   srcDstInfo.rhsOp = rhsInfo.origDstOp
11
12   if opTime < srcDstInfo.rhsTime then
13     srcDstInfo.rhsTime ← srcDstInfo.rhsTime - lhsInfo.fdTime
14     srcDstInfo.lhsOp = lhsInfo.srcOp
15     if opTime < srcDstInfo.rhsTime then
16       srcDstInfo.rhsTime ← srcDstInfo.rhsTime - rhsInfo.fdTime
17       srcDstInfo.lhsTime ← srcDstInfo.lhsTime + rhsInfo.fdTime
18       srcDstInfo.dstOp = rhsInfo.dstOp
19     endif
20   endif
21
22 return srcDstInfo
23 }

```

Figure 3.12d: The CALC-FLOW-TIMES helper function

3.5 Experimental results

To measure the impact of speculative plan execution on the information gathering process, I conducted two sets of experiments. The first involved measuring the impact of speculation for a set of typical Web information agent plans. The goal of this experiment was to discover how useful the technique would be for the types of information integration plans that are common to Internet information gathering.

A second set of tests involved applying the technique to standard database query plans that were generated from the Transaction Processing Council's TPC-H benchmark, a set of ad-hoc business-style queries to an order-entry schema. I was able to use this benchmark by generating a query plans for each of a subset of the 22 specified TPC-H queries and then, when executing that plan, simulating latencies for each table accessed (thus, the schema becomes a distributed database with noticeable latencies). I measured the effectiveness of speculative execution for different degrees of latency and database sizes. The goal of this second type of experiment was to measure the utility of speculative execution for a more traditional type of distributed database using a standard schema and standard set of queries.

Both experiments were conducted using Theseus, the implementation of the streaming dataflow execution system for information agents described earlier in Chapter 2. Theseus was modified to support the automatic transformation of plans using the SPEC-REWRITE algorithm. In addition, Theseus was instrumented to count the average number of tuples per operator, per transaction as well as the average time it took to process each tuple. Using these numbers, Theseus iteratively transformed the MEPs in each plan, until no further transformations were possible (or profitable). For the second and successive runs, Theseus issued predictions when possible using data acquired from past executions. It also collected source/target data for each Speculative opportunity from to improve its predictive accuracy for future runs. I now focus on the tests run for each group of plans – the Web agent plans and the TPC query plans.

3.5.1 Web agent plans

To measure the utility of speculative execution on online information gathering, I looked at how the technique affected the performance of five different types of Web agent plans that integrate information between multiple Internet sources. These plans included:

- **CarInfo**: The main example, introduced in Chapter 1.
- **RepInfo**: An agent described in (Barish and Knoblock 2002) that allows users to specify an U.S. nine-digit zip code to query multiple Web sources that identify the set of corresponding U.S. federal congressional members (representative and senators), along with funding charts and recent news corresponding to each member.
- **TheaterLoc**: An agent described in (Barish et al. 2000) that combines restaurant and theater data for a particular city and emits a dynamically generated map that plots their locations.
- **FlightStatus**: An agent described in (Ambite et al. 2002) that queries the status of a particular flight, and then e-mails the user/hotel with updates as necessary.
- **StockInfo**: An agent that takes a particular company name, identifies the stock symbol associated with it, locates profile information on that company, finds out what industry sector that company is in, identifies the largest competitor (based on market capitalization) and retrieves a

chart that compares the 1 year performance of that competitor with the input company and the sector.

I now describe the details of each of these agent plans, except CarInfo (which has been shown earlier in this chapter), including the original and transformed versions of the plans.

RepInfo

This agent uses Congress.org (<http://www.congress.org>) to identify the congressional members based on zip code, Yahoo News (<http://news.yahoo.com>) for headlines about each member, and Open Secrets (<http://www.opensecrets.org>) for funding charts for each member. Figure 3.13a-c show the Web pages corresponding to each of these sources.



Figure 3.13a: Congress.org Web page



Figure 3.13b: Yahoo News Web page

Figure 3.14a shows the original RepInfo plan while Figure 3.14b shows the plan modified for speculative execution. Note that querying both Congress.org and the chart from Open Secrets requires navigating from links derived from an initial query

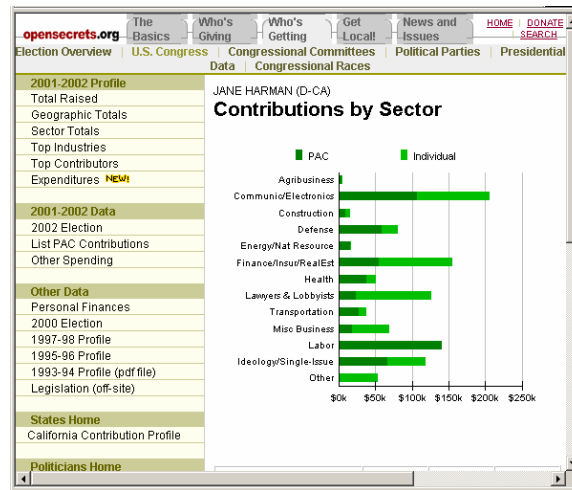


Figure 3.13c: Open Secrets Web page

– thus, interleaved navigation is required in order to obtain an answer during plan execution.

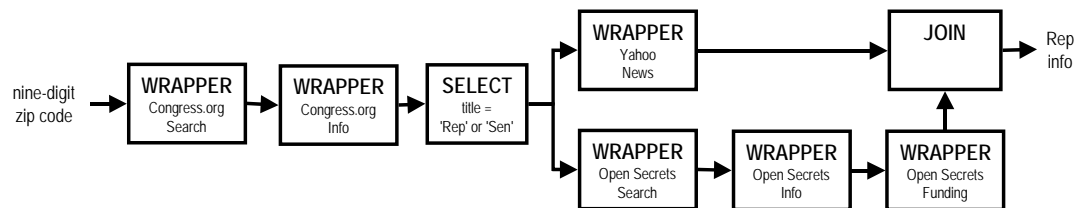


Figure 3.14a: The RepInfo agent plan

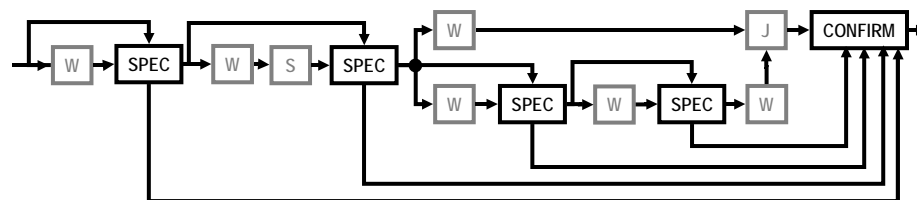


Figure 3.14b: The modified RepInfo agent plan

TheaterLoc

The TheaterLoc agent plan queries for restaurant and theater data from Dine.com (<http://www.dine.com>) and Yahoo Movies (<http://movies.yahoo.com>), respectively. It combines that data and uses it to query a local geocoder, which is very similar to the one provided by E-Tak (<http://www.etak.com>) and then plots the locations based on their resulting latitude and longitude on a map dynamically generated by U.S. Census Tiger service (<http://tiger.census.gov>). Figures 3.15a-c show the Web pages for some of these sources.

YAHOO! MOVIES

Travel + Study | Earn Your Degree While You Travel! | Learn more

Programs offered in: Accounting, Education, Management, Business, Information Technology, Nursing

Search: Entire movie database for

Home | In Theaters | Times & Tickets | Greg's Previews | Trailers | DVD/Video | News

Showtimes: 90292 [Change location]

View by: [Movie] [Theater]

Loews Cineplex Marina Marketplace | Laemmle's Monica Fourplex | Pacific Culver Stadium 12
 UA Marina Del Rey | Loews Cineplex Broadway | Nuwlishing
 The Bridge: cinema de lux | Mann Criterion | More Theaters...
 AMC Santa Monica 7

June 15, 2003 [Sun | Mon | Tue | Wed | Thu | Fri | Sat]

Buy Tickets from MovieTickets.com by clicking on a linked showtime.

Loews Cineplex Marina Marketplace Phone: (310)827-9588
 13455 Maxella Avenue - Suite 270, Marina Del Rey, CA 90292 Theater Info | Map It

2 Fast 2 Furious PG-13 1 hr. 40 min.
 11:00 AM, 12:50, 1:30, 3:20, 4:00, 5:50, 6:30, 7:30, 8:30, 9:30, 10:10 **

Bruce Almighty PG-13 1 hr. 34 min.
 11:20 AM, 1:50, 4:30, 7:20, 10:00 **

Matrix Reloaded, The R 2 hrs. 18 min.
 12:30 PM, 3:40, 6:50, 9:50 **

Rugrats Go Wild PG 1 hr. 21 min.
 11:30 AM, 12:00, 2:10, 2:30, 4:40, 5:00, 7:10, 9:20 **

Figure 3.15a: Yahoo Movies web page

DINE.com | FriendFinder.com | Meet your spec

Browse: men seeking women for: love

Home | Top Reviewers | Browse by Location | Search | New Reviews

Italian Restaurants in and near 90292

Rating	Name	Type	Address	City
3.80	MAP Valentino Italian Restaurant	Italian	3115 Pico Blvd	Santa Monica, CA
3.25	Guidos	italian	11980 Santa Monica Blvd	Los Angeles, CA
3.25	Alejo's Presto Trattoria	italian, pasta, pizza, seafood	4002 Lincoln Blvd	Marina Del Rey, CA
3.20	MAP C & O Trattoria	Italian	31 Washington Blvd	Marina Del Rey, CA
2.75	MAP Villa Italian Restaurant	Italian	3973 Sepulveda Blvd	Culver City, CA
2.67	MAP Bistro of Santa Monica	Bistro, Italian	2301 Santa Monica Blvd	Santa Monica, CA
2.60	MAP Mario's Italian Restaurant	Italian	1444 3rd Street Promenade	Santa Monica, CA
2.33	MAP Earth Wind & Flour	Italian	2222 Wilshire Blvd	Santa Monica, CA
2.25	MAP Matteo's Italian Restaurant	Italian	2323 Westwood Blvd	Los Angeles, CA
Not Reviewed	MAP Anna's Italian Restaurant	Italian	10929 W Pico Blvd	Los Angeles, CA

Member Login: Email, Password, Login, Forget Password?

Post reviews and earn a free Dine.com T-shirt or premium memberships on any Friend Finder Network personals site!

The first step is to become a Dine.com member and get personalized email reviews, and restaurant suggestions!

Figure 3.15b: Dine.com web page



Figure 3.15c: TIGER Mapping Service web page

Figure 3.16a shows the original TheaterLoc plan while Figure 3.16b shows the plan modified for speculative execution.

Note that the Confirm operator appears *before* the final wrapper (the call to the U.S. Census Tiger Service). This is because the operator was tagged (in the plan) as being “unsafe”, since it needs to write to the file system before calling the Tiger Service (which produces dynamic maps from locally defined URLs – Web pages that must be generated by the agent).

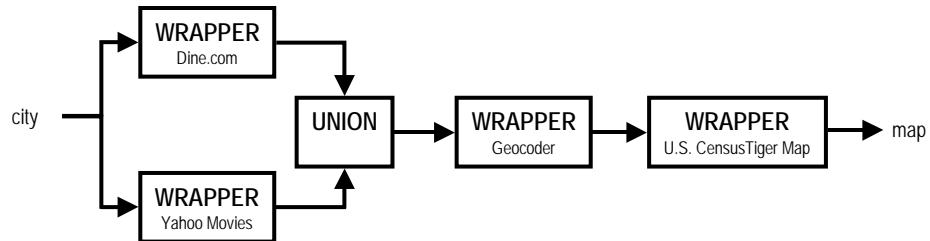


Figure 3.16a: The TheaterLoc agent plan

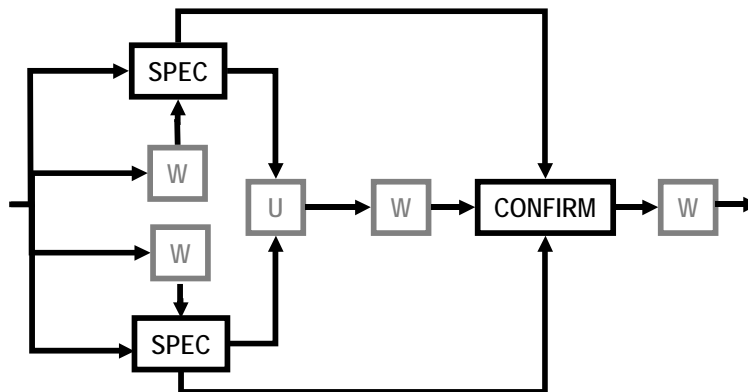


Figure 3.16b: The modified TheaterLoc agent plan

FlightStatus

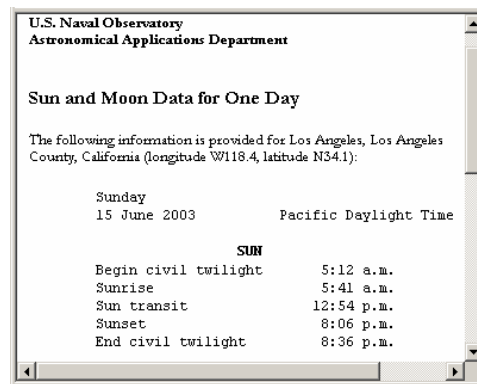
The FlightStatus agent queries the status of a particular flight, and then e-mails the user/hotel with updates as necessary. The plan first queries a flight status Web site (I used Delta Airlines, at <http://www.delta.com>) and then takes the resulting destination city and converts the destination time based on the time zone of that city, via an online time conversion source (U.S. Naval Observatory Astronomical Applications Department, at <http://aa.usno.navy.mil>). An e-mail may also be sent to the user in certain cases (i.e., if a flight status changes to “Cancelled”) or the agent may be unscheduled in other (i.e., if the flight status has changed to “Arrived”). Figures 3.17a and 3.17b show the Delta Airlines and US Naval Time zone conversion Web pages.



The screenshot shows the Delta Airlines website's flight information page. The page title is "Flight Information" and it displays details for "Flight 1982". The information is organized into two main sections: "Departing" and "Arriving".

Departing			Arriving			
City	Scheduled	Estimated (actual)	City	Scheduled	Estimated (actual)	Status
Los Angeles (LAX)	June 15 08:20am	June 15 08:16am	New York (JFK)	June 15 04:44pm	June 15 04:20pm	In Flight

Figure 3.17a: Delta Airlines web page



The screenshot shows the U.S. Naval Observatory's "Sun and Moon Data for One Day" page. The page provides information for Los Angeles, California (longitude W118.4, latitude N34.1) for Sunday, June 15, 2003, in Pacific Daylight Time.

SUN	
Begin civil twilight	5:12 a.m.
Sunrise	5:41 a.m.
Sun transit	12:54 p.m.
Sunset	8:06 p.m.
End civil twilight	8:36 p.m.

Figure 3.17b: US Naval Time Details web page

Figure 3.18a shows the original version of the FlightStatus agent plan. Figure 3.18b shows the one modified for speculative execution.

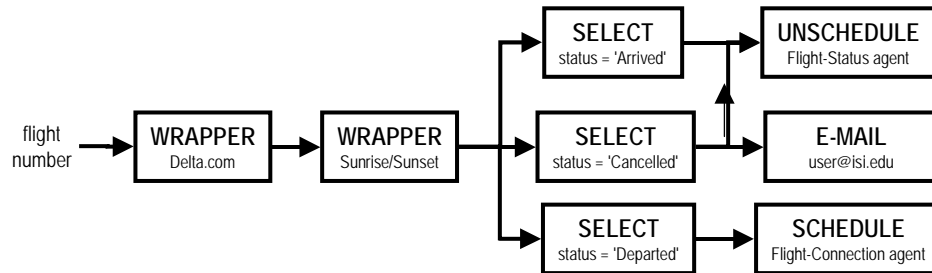


Figure 3.18a: The FlightStatus agent plan

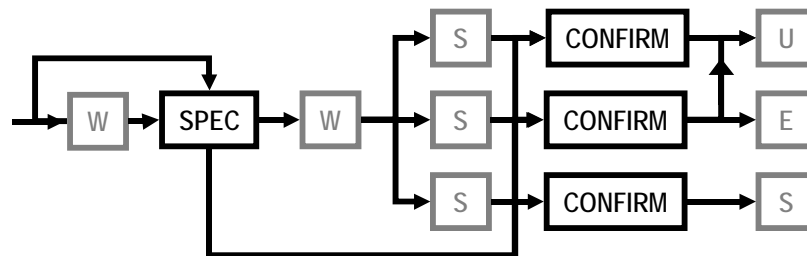


Figure 3.18b: The modified FlightStatus agent plan

Note that three Confirm operators are needed since there are three different paths on which speculative results can propagate. Since E-Mail, Unschedule, and Schedule are also operations that are unsafe and affect the external world, the Confirm operators were placed prior to their execution.

StockInfo

To return stock information for a collection of stocks in a portfolio, this agent uses a single site – CBS MarketWatch (<http://www.marketwatch.com>) – for all of its queries. One of the reasons I chose this plan was to investigate the impact of speculative execution on a more extreme case, a dataflow graph that was completely sequential due to data dependencies.

The plan involves collecting competitor information about a particular stock. To do that requires first looking up the symbol, going to the profile page for that stock, identifying the business industry name (e.g., software) and then gathering competitor information in that industry. Figures 3.19a and 3.19b show two web pages in this sequence – the profile page and the industry page.

Figure 3.20a shows the StockInfo agent plan. Figure 3.20b shows the version of the plan transformed for speculative execution.

Example plan transformation

To better illustrate the details of plan transformation using SPEC-REWRITE, I describe the process of optimization on the real CarInfo plan, using actual operator execution times. In practice, the initial run of this plan took 6900 seconds and yielded the operator execution times shown in Table 3.5.



Figure 3.19a: MarketWatch profile page



Figure 3.19b: MarketWatch industry page



Figure 3.20a: The StockInfo agent plan

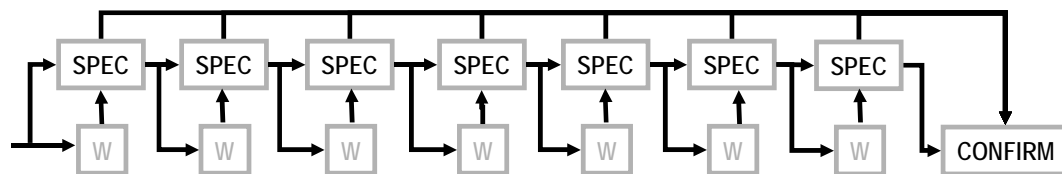


Figure 3.20b: The modified StockInfo agent plan

Operator	Time (ms)
Join	10
Select	153
Wrapper (NHTSA)	359
Wrapper (Consumer Guide - Summary)	1912
Wrapper (Consumer Guide - Full Review)	2175
Wrapper (Consumer Guide - Search)	1478
Wrapper (Edmunds)	812
Total	6900

Table 3.5: Operator execution times in CarInfo

From this, the path execution times shown in Table 3.6 were calculated.

Path	Path operators	Time (ms)
P1	Edmunds + Select + NHTSA + Join	1334
P2	Edmunds + Select + CG-Search + CG-Summary + CG-Full + Join	6900

Table 3.6: Path execution times in CarInfo

The SPEC-REWRITE algorithm then used the above statistics to transform the plan for speculative execution. It first determined that the MEP of the plan was path *P2*. Initially, the most profitable operator to speculate about was the Consumer Guide Search wrapper. Parallelizing its execution through speculation with operators on the MEP leading up to it theoretically saved just over 1900ms (assuming 100% correct predictions). Note that even though the Consumer Guide Full Review wrapper took longer, parallelizing its execution with the rest of the plan would save little time, since only a very fast Join follows. By continuing with the algorithm, the original MEP was reduced further by speculating about both the Consumer Guide Summary wrapper and Edmunds wrapper. In short, the algorithm transformed the plan so that instead of only two long parallel paths (as in Table 3.6), there were now many short parallel paths, as shown in Table 3.7.

Path operators	Estimated Time (ms)
Edmunds + Spec	812
Select + NHTSA + Join + Confirm	522
Select + CG-Search + Spec	1631
CG-Summary + Spec	1912
CG-Full + Join + Confirm	2185

Table 3.7: Path execution times after transformation for speculative execution

Thus, the estimated execution time of the plan would be equal to the new MEP, the $\{CG\ Full, Join, Confirm\}$ path, of about 2200ms. Note that Table 3.7 does not take into account the overhead of speculation. For example, if we assume that it costs roughly 100ms per tuple for each of execution of Speculate and Confirm, then an optimistic execution of the new MEP would be $(2200+100+100) = 2400ms$, representing a speedup of $(6900ms/2400ms) = 2.88$ for the time to first tuple.

Overall results

I compared the performance of normal execution to speculative execution for all five agent plans, focusing specifically on the speedups associated with the time to first and last tuple. When comparing normal execution to speculative execution, I looked at three cases of speculative execution:

- **Optimistic:** 100% of the predictions made were correct
- **Average:** 50% of the predictions made were correct
- **Pessimistic:** none of the predictions made were correct

I chose to measure these three cases of speculative execution to show the impact of prediction quality on plan speedup. Figures 3.11a and 3.11b show the average performance of the different predictive accuracy scenarios. Figure 3.11a shows the affect of speculative execution on the time to first tuple (start of output), while Figure 3.21b shows the impact on the time to the final tuple (end of output). The resulting average speedups for each of the plans, for both the 100% correct and 50% correct predictions cases, are shown in Figures 3.22a and 3.22b.

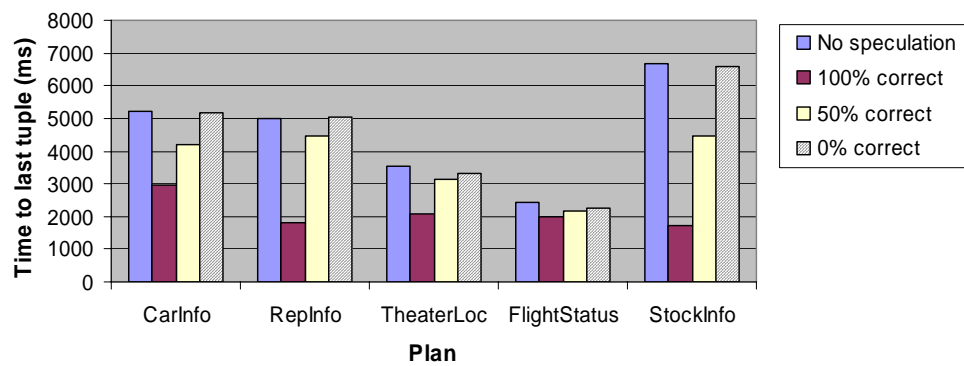


Figure 3.21b: Performance improvement of time to last tuple

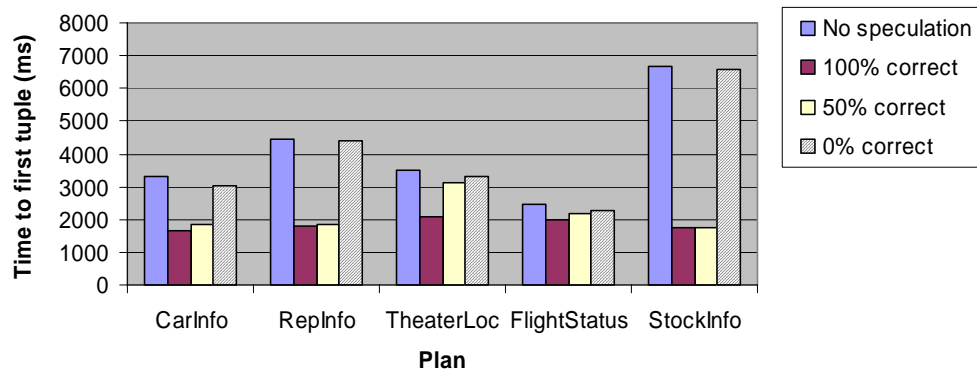


Figure 3.21a: Performance improvement of time to first tuple

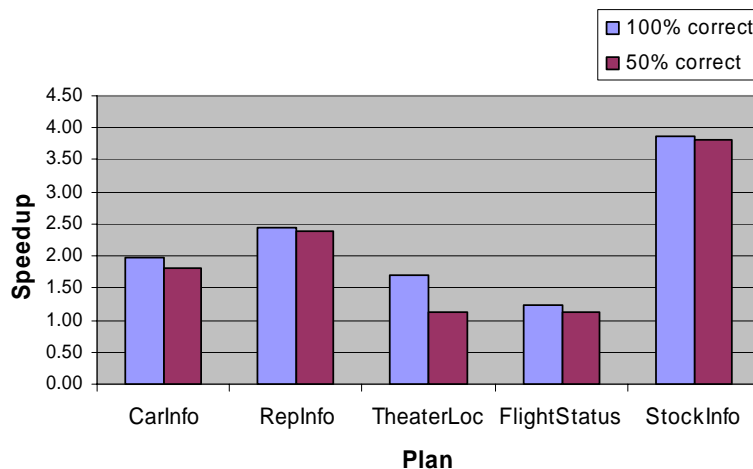


Figure 3.22a: Speedup increases related to the time to first tuple

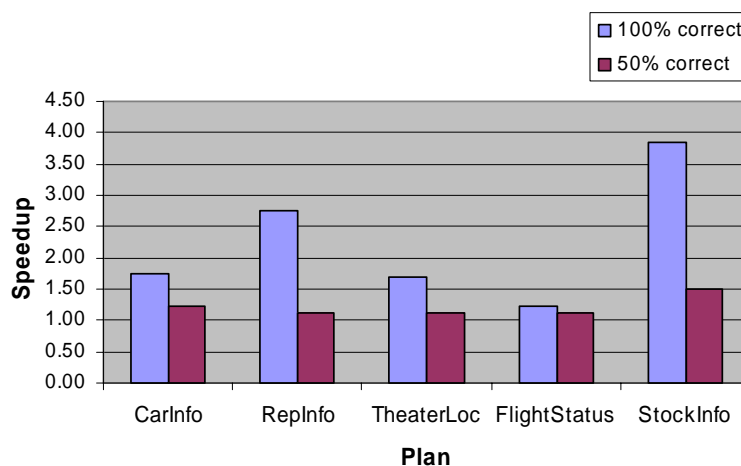


Figure 3.22b: Speedup increases related to the time to last tuple

Discussion

There were two interesting findings worth noting from the Web information gathering results. The first was that speculative execution reduced average execution time significantly for CarInfo, RepInfo, TheaterLoc, StockInfo, and less significantly for FlightStatus. Clearly, this difference in the impact of speculative execution has to do with two factors: (a) the number of binding patterns between Wrapper operators in plan and (b) the latency of the sources used.

For example, the StockInfo plan had an MEP parallelizable to a degree of seven. Correspondingly, its average speedup was just under 4, the difference likely due to the overhead of speculation. The same is true for CarInfo and RepInfo, which had MEPs parallelizable up to 3 and 4, respectively, and yielded average speedups of 2 and 2.5. In contrast, the maximum possible speedup for FlightStatus – if the sources were equally latent – was 2.0. However, since one of the sources (the U.S. Naval Time source) was very fast, execution time was dominated by the slower source (Delta airlines).

A second notable finding was the difference in speedups between first and last tuple as a function of predictive accuracy. For example, for a predictive accuracy of 100%, we see that the speedups of the time to first and the time to last tuple due to speculative execution roughly correspond. Consider CarInfo, where the first tuple and last tuple speedups were 1.98 and 1.76, respectively, a standard deviation of 0.16. However, for predictive accuracies less than 100%, there were significant differences between first and last tuple speedups. For example, the CarInfo first and last tuple speedups when 50% accurate were 1.80 and 1.24, respectively, a standard deviation of 0.39. The difference in deviations can be explained by the fact that, in cases where the predictive accuracy was less than 100%, the last tuple(s) will have required traveling through the normal path of execution – that is, since confirmation failed at an earlier stage, some tuples needed to pass through some or all of the plan. However, note that speedups on the last tuple were still possible because (a) execution was more “spread out” (smaller groups of tuples required concurrent processing by Wrapper operators) and (b) although speculation failed some percentage of the time, it was rare that a tuple which failed but was corrected in the middle of the plan, failed again at a later point in the plan.

Finally, for purposes of clarity, it is useful to revisit the definitions of “optimistic” and “average” in the experiments. Note that for cascading speculation, the “optimistic” case assumed that *all* predictors in the modified plan are 100% accurate in their predictions, all of the time. In contrast, the “average” case assumed that *all* predictors are 50% correct. This is equivalent to having said that (a) the plan input data is repeated 100% (or 50%) of the time and that (b) no generalization (such as learning, discussed in Chapter 4) is performed. This means that, to an extent, the boundaries can be viewed as somewhat “over optimistic” and “over pessimistic”, depending on the application. Nevertheless, these assumptions allow us to get some feel for the impact of speculative execution given different degrees of predictive accuracy, underscore how important good predictions are during speculative execution, and help motivate the discussion on learning in Chapter 4.

3.5.2 Database query plans

A second set of experiments focused on applying speculative execution to plans generated by SQL queries from the TPC-H ad-hoc query benchmark. The goal was to understand the impact of speculation on complex queries that involve a very common type of schema, where each of the entities (tables) of that schema were independent sources on a network. Figure 3.23 summarizes how the TPC-H schema was converted to a distributed database.

The TPC-H benchmark is composed of a schema and 22 queries over this schema. The schema approximates a typical order-entry system, composed of customers, suppliers, orders, line items, parts, and nation as well as regional information about customers and suppliers. The 22 queries developed by the TPC organization target typical ad-hoc queries that analysts invoke on such schemas. These queries commonly require gathering data from multiple tables and then computing some aggregate value (such as a SUM or AVERAGE) from the combined data, or grouping and/or ordering of those results. The TPC also provides tools for

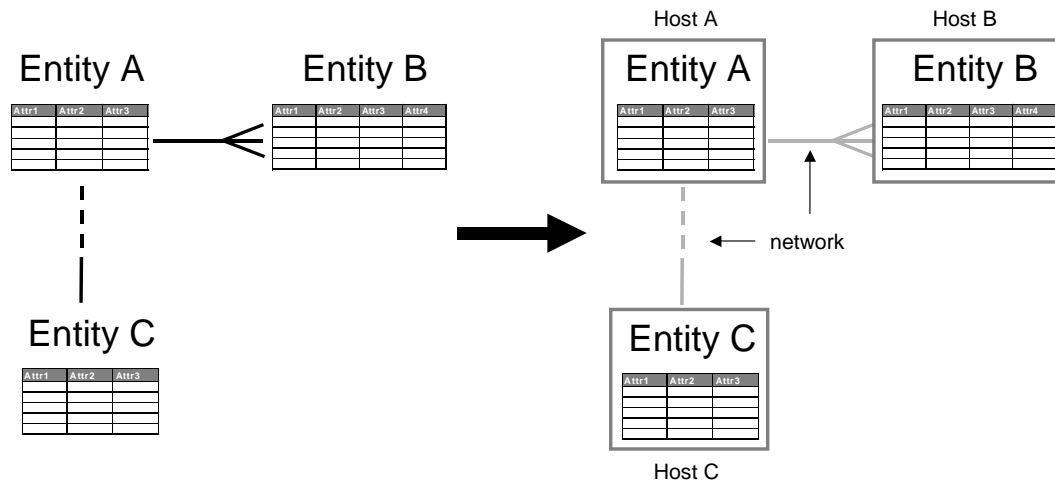


Figure 3.23: Converting a schema to a distributed database

generating databases based on the TPC-H schema (at a variety of scales) and for randomizing certain parts of each of the 22 queries (such as randomizing the content of SELECT WHERE clauses).

I generated Theseus plans for each of these queries using the results of plan generation by a commercial relational database management system, Oracle 8.1.6. To do this, I loaded the sample TPC-H database into Oracle. Then, I used Oracle's EXPLAIN PLAN feature to generate the query plan produced by each of the queries. As the TPC-H benchmark suggests, I did not pre-index any attributes of any tables and cleared all prior statistics from Oracle's data dictionary, so no other types of cost-based optimizations were used during plan generation. I then took the plan produced by EXPLAIN PLAN and converted it into a Theseus plan, which was then executed using Theseus.

Example query transformation

To illustrate the entire process of converting a SQL query to a Theseus plan, let us consider the steps required in the conversion of TPC-H query #17. The SQL for this query is shown in Figure 3.24:

When TPC-H schema was loaded into an Oracle 8.1.6 database, the Oracle EXPLAIN PLAN feature returned the query plan in Figure 3.25. This plan corresponds to the dataflow graph shown in Figure 3.26. Based on this graph, and mindful of the results of EXPLAIN PLAN as well as the details of the query, the Theseus plan shown in Figure 3.27 was constructed. This plan performs the operations suggested by EXPLAIN PLAN to address the original SQL query.

Overall results

In running the experiments, 13 of the 22 queries from the TPC-H benchmark were tested. I believe that the subset of queries tested contain enough evidence of the impact of speculative execution on database-style query plans in order to reach some reasonable conclusions. The queries not evaluated fall into two classes:

```

select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    part
where
    p_partkey = l_partkey
    and p_brand = 'Brand#45'
    and p_container = 'WRAP CAN'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
    );

```

Figure 3.24: SQL for TPC-H query #17

SELECT STATEMENT ()		1
SORT (AGGREGATE)		2
FILTER ()		3
NESTED LOOPS ()		4
TABLE ACCESS (FULL)	LINEITEM	5
TABLE ACCESS (BY INDEX ROWID)	PART	5
INDEX (UNIQUE SCAN)	PART_PK	6
SORT (AGGREGATE)		4
TABLE ACCESS (FULL)	LINEITEM	5

Figure 3.25: The Oracle EXPLAIN PLAN for TPC-H query #17

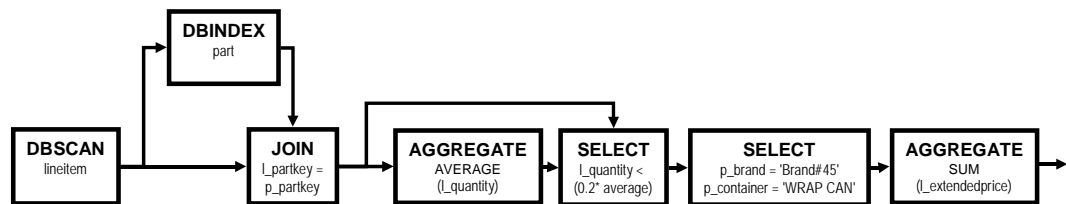


Figure 3.26: Dataflow graph of the explained plan

- **Those not applicable:** Of the 22 TPC-H queries, query 1 and 6 were the only ones that theoretically could not benefit from speculative parallelism. For both queries, this was due to the fact that they consisted of calculations based on only a single table scan (no joins with other tables).
- **Those requiring operations not currently supported by Theseus:** Some of the TPC-H queries involve more advanced SQL functionality not currently supported by Theseus. The queries that fall into this category include queries 11, 13, 14, 15, 18, 21 and 22. The operations not supported include the SQL HAVING clause, outer joins, SQL ELSE clause, and SQL EXISTS and NOT EXISTS clauses.

```

PLAN q17
{
  INPUT:
  OUTPUT: stream answer

  BODY
  {
    dbscan("lineitem" : lineitems)
    dbindex(lineitems, "part", "l_partkey" : parts)
    join(lineitems, parts, "l_partkey = p_partkey" : lp)
    aggregate(lp, "avg(l_quantity) the_avg" : avg)
    format(avg, "l_quantity < %s", "the_avg", "cri" : cri)
    select(lp, cri : selected-lp)
    aggregate(selected-lp, "sum(l_extendedprice) the_sum" : answer)
  }
}

```

Figure 3.27: Theseus plan based on dataflow graph

I measured the impact of speculative execution on the 13 other queries under several conditions. Specifically, the following parameters were varied:

- **Database size:** The TPC-H tool DBGEN allow databases of various sizes to be generated. Sizes are based on a “scale” factor input to DBGEN. A scale factor of 1 corresponds to about 1 GB of data. The experiments used databases with scale sizes of 0.2 and 0.6, to demonstrate how speculative execution performance was impacted by an increased amount of data.
- **Maximum source concurrency:** In a distributed federated database (such as the Web), there are often limits on how concurrently a source can be queried (i.e., maximum number of simultaneous connections allowed). To measure the impact of source concurrency on speculative execution, the maximum number of concurrent connections to a particular source (i.e., table) was varied between 5 and 100.
- **Average source latency:** Source latencies can vary based on source capability, current load, and the complexity of query being performed. To measure its impact on speculative execution, source latencies of 2000ms, 4000ms, 6000ms, 8000ms, and 10000ms were tested.
- **The details of the query:** TPC-H tools allow the literals used in queries (such as the details of a SQL WHERE clause) to be varied. Each test used three customized versions of the same query, each version semi-randomly generated by the tools.

The results are shown in Figures 3.28a, 3.28b, and 3.28c.

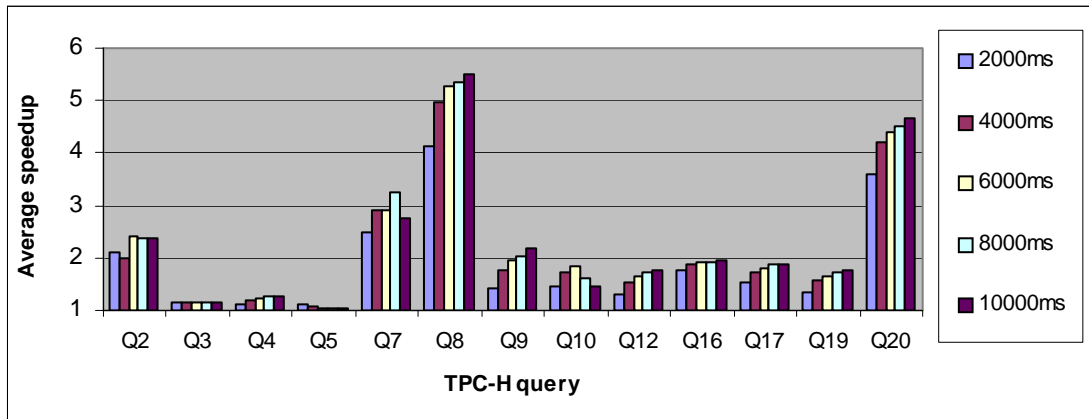


Figure 3.28a: Average speedup of TPC-H queries (database scale = 0.2, concurrency=5)

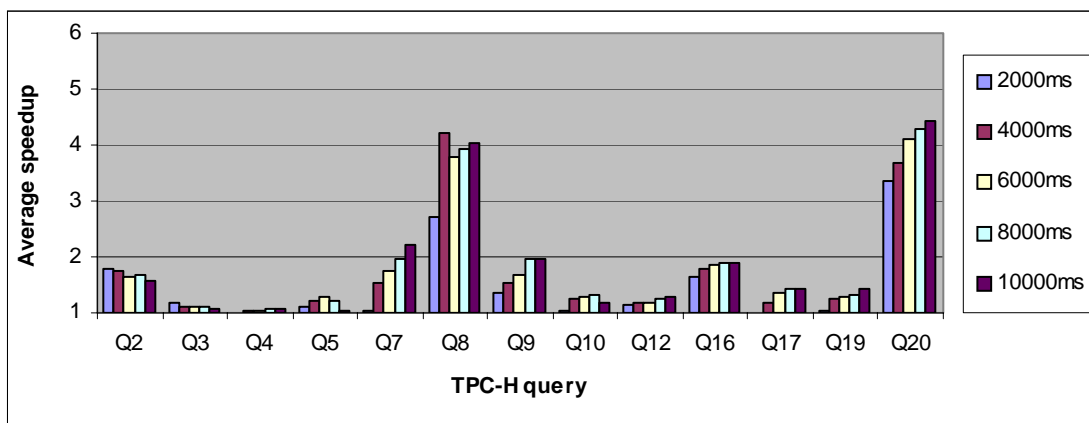


Figure 3.28b: Average speedup of TPC-H queries (database scale = 0.6, concurrency=5)

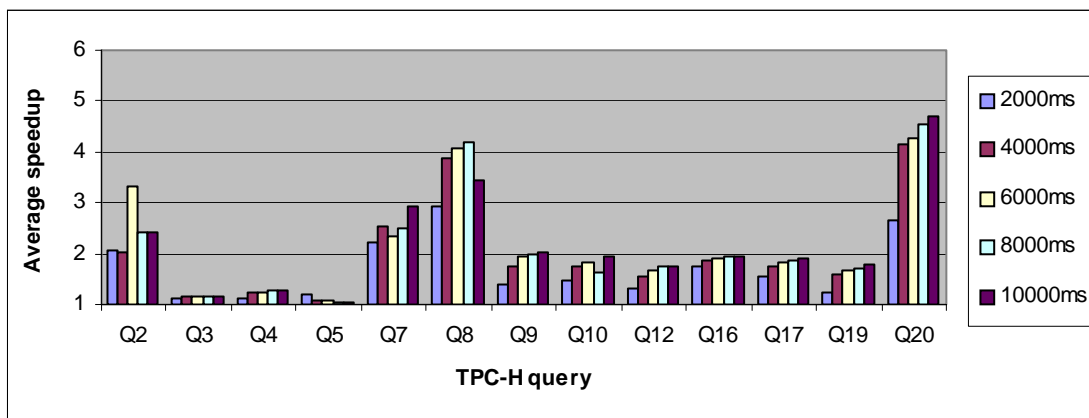


Figure 3.28c: Average speedup of TPC-H queries (database scale = 0.2, concurrency=100)

Figure 3.28a shows the average speedups of all queries for a variety of latencies when the scale of the data was set to 0.2 and the maximum number of concurrent connections to each source was limited to 5. Figure 3.28b shows the same type of

data for a scenario where the scale of the data was 0.6 and the max concurrency remained at 5. Finally, Figure 3.28c shows the same type of data for the case where the scale was 0.2 and the max concurrency was 100.

Discussion

Just as was the case for Web information gathering plans, speculative execution also appears useful for traditional distributed database query plans, such as those generated by TPC-H queries. In two dramatic cases (queries 8 and 20), the technique yielded speedups above 3 (at both data scales). For many other cases, speedups above 1.5 were attained, most at the lower data scales. In addition, speedups were greater in some cases (queries 7 and 8) when constrained on the number of concurrent connections, because operators were blocked more often, leading to greater CPU idle time and thus more speculative opportunities.

As the graphs in Figures 3.28a-c illustrate, the entire set of speedups observed ranged from just over 1 to nearly 6. This is because, as was the case for Web information gathering, the maximum speedup possible for each of the queries was different. For example, a query that involved a scan of one table with a (pipelined) join of data indexed from another has a maximum speedup of 2 (i.e., the best we can do is to query the second based on what we predict will be returned from the first). Thus, to better understand the impact speculative execution had on the TPC-H queries, it is useful to compare the speedups measured with the maximum speedups possible. Figure 3.29 shows the theoretical maximum speedups for each plan alongside the results from Figure 3.28a.

It should be noted that the theoretical maximum speedups for all queries is an upper bound. This is because the nature of the types of retrieval vary during the execution of a query. Typically, the plan generated from a SQL query consists of scanning a first source (reading in an entire table) and then using that source as a “driving table” for index queries (binding pattern-style queries) to all other sources in the plan. In the experiments, scan latencies are assumed to be a one time occurrence (there is a delay and then all of the data is available) while index latencies are per tuple. When speculating about data returned from a scanned source, it is

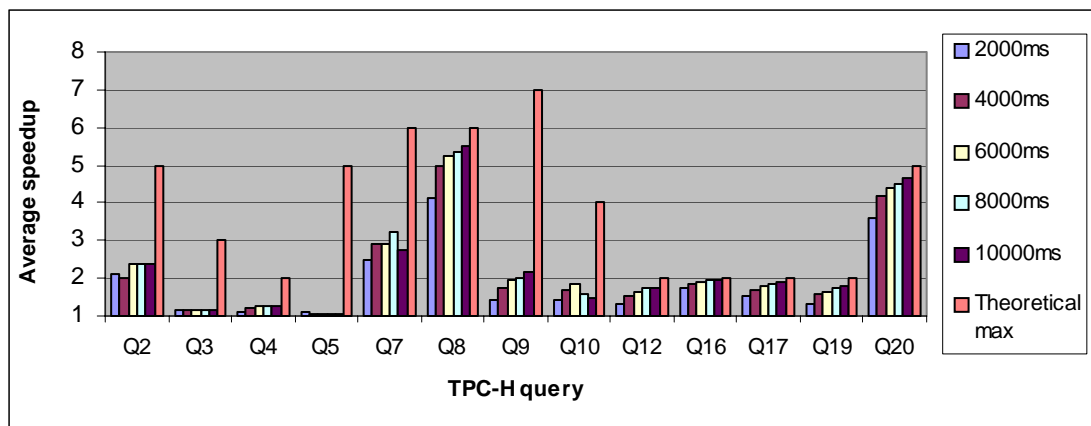


Figure 3.29: Speedups obtained for TPC-H queries vs. theoretical maximums

necessary to be able to process the subsequent index retrievals in parallel in order to obtain the maximum speedup of 2. For example, consider a simple plan that scans source A and uses the data scanned to index source B. If all sources have a latency of 1000ms, and scanning A returns 50 tuples, obtaining a speedup of 2 for speculative transformation of that plan will require that the corresponding 50 index retrievals will need to be performed from B in parallel with the scan of A. If that source has limitations on the number of concurrent connections, or if the source is affected by the load, or if the number of threads/processes available for parallel querying are limited, speedups of 2 will not be possible because the 50 index retrievals will take longer than the scan. Thus, for plans that scan large amounts of data, the benefits of speculative parallelism can be inhibited by local or remote concurrency constraints.

With this fact in mind, we now turn to further analysis of the results. In considering Figure 3.29 in conjunction with the results from Figures 3.28a-c, we see that speculative execution of about half of the queries (8, 12, 16, 17, 19, and 20) approached the theoretical maximum speedups for those queries. In other cases (2, 7, 9, and 10), significant speedups were obtained, but appeared to approach their theoretical maximum more slowly. Finally, in other cases (3, 4, and 5) speedups were moderate to minimal, ranging from just over 1 to about 1.3.

Generally, the variance in the differences between speedups due to speculative execution and the theoretical max speedup is due to the difference in the number of speculative tuples generated during execution – thus, the overhead of speculation. This is best demonstrated by the differences between the speculative execution of queries 20 and 5. Speculative execution of query 20 yielded speedups that were very close to the theoretical maximum. The query involved first scanning a table called PART and then using that as the driving table for indexing other sources. The PART table is small, containing just over 2% of the total number of rows in the aggregate set of tables. Thus, speculation about that scan involves prediction of only a few tuples and thus a lower overhead of speculation. In contrast, query 5 involved an initial scan of the LINEITEM table, which contained 67% of the total number of rows in the aggregate set of tables. Thus, speculation about that scan required a much higher overhead, which worked against any gains speculative execution provided.

Another important result to note was that speedup generally increased as source latencies increased. The main reason for this is that greater latencies compensate for the overhead of speculation (both prediction generation and confirmation). Recall that the SPEC-REWRITE algorithm produces a transformed plan if there is any improvement, not just if the improvement will completely compensate for the overhead required. However, as latencies increase, the effect of the overhead (which is a fixed amount, unaffected by the latencies) is lessened.

A final result worth noting is the effect of database scale. In considering Figures 3.28a-c, we see that the utility of speculative execution for the TPC-H queries generally decreases as the size of the database increases. The only queries that seemed relatively impervious to this were queries 9, 16, and 20. Despite being affected by increased database size, queries 2, 7, and 8 still yielded significant

speedups; for example, for a latency of 2000ms and a scale of 0.6, speculative execution of query 8 yielded average speedups of over 2.5 (down from a speedup of 4 for a database scale of 0.2).

In summary, the experiments indicate that speculative execution can have a significant impact on several TPC-H queries, with a few caveats. Generally, the utility of speculative execution is associated with database scale, intermediate data size, the number of sources queried, and the average latency of those sources. The most important factor is the number of sources joined together: the more sources, the greater the theoretical max speedup. The second most important factor is the size of the intermediate data – when it is small to modest, significant speedups can be obtained and the theoretical max can be within reach. Finally, the utility of speculative execution generally improves with longer latencies (which compensate for speculative overhead) and small databases (less speculative overhead for scans, etc). Overall, speculative execution can have a modest to significant impact for TPC-H queries, suggesting that it is a technique applicable to improving the performance of distributed database query processing.

3.6 Summary

In this chapter, I have introduced an approach to the speculative execution of information agent plans. I have shown how this approach represents a new form of run-time parallelism that allows plan execution to exceed its normal dataflow limit, leading to significant execution speedups without sacrificing fairness or safety during execution. In addition, I have presented algorithms that enables any pre-existing plan to be automatically transformed into one capable of speculative execution.

Overall, the results indicate that speculative execution is an effective technique for improving the performance of Web information gathering plans as well as more traditional type of query plans for distributed database systems. In analyzing the effects of speculative execution in the latter scenario, it is clear that its impact is tied closely to the total number of distinct sources queried, the amount of data needed for speculation, the size of the database, and the latency of the individual sources. Because the method of speculative execution proposed ensures fairness, transformed plans are usually just as fast or faster than without speculation, with speedups at least in the 1.20-1.50 range. However, when the number of sources being speculated about is large and the amount of data to be speculated is modest, very large speedups (4 and greater) can be obtained.

Chapter 4

Value Prediction for Speculative Execution

Thus far, I have described how speculative execution can lead to better parallelism and subsequently better average plan speedups. However, the utility of speculative execution is fundamentally linked with the ability to issue accurate value predictions. The more accurate the predictions, the closer the average speedup of a plan approaches its maximum theoretical performance. Thus, a good value prediction strategy is important.

The basic problem of value prediction involves being able to leverage knowledge about the set of past hints when making a prediction about a new hint. More specifically, the goal is to use some source tuple h as hint for issuing a predicted target tuple v . One approach to value prediction is simple caching: we can note that particular hint h_x corresponds to a particular target v_y so that future receipt of h_x can lead to prediction of v_y . Caching is one simple and safe solution to the problem of value prediction. It requires no new algorithms and can be applied to any opportunity for value predictions.

However, since the type of speculative execution that I have described occurs at the agent level, where the values being predicted are related tuples of data, there are often opportunities where it is possible to do better. For example, in the CarInfo plan, the full review URL is simply just a transformation of the summary URL. If possible, it would be more desirable to learn this transformation function because such a predictor would be useful towards never-before-seen hints (that would not have otherwise been cached). In addition, this type of function-style predictor would also be smaller and bounded in the space it demands.

In this chapter, I introduce a technique for value prediction that combines caching with two machine learning techniques, classification and transduction. The resulting predictors learned are not only capable of both predicting values based on recurring past hints, but are also capable of making predictions for never-before-seen hints and synthesizing new predictions if necessary. As a result, the predictors learned can make predictions more often and thus increase the average accuracy of prediction. This, in turn, leads to better average plan speedups due to speculative execution.

4.1 Value prediction strategies

There are several potential methods that can be used to predict values for speculative plan execution. These strategies differ in terms of their design complexity, space efficiency, and predictive capabilities. The last metric is especially important

because better predictions at runtime translate into better speedups. To better compare methods of prediction, there are three scenarios to consider:

- **Predictions of past values based on recurring hints:** Given the past association of an input with an output, future receipt of that prior input can be treated as a hint h_{xi} justifying prediction of that prior output value v_{yi} . More compactly, this can be described as the case where $(v_{yi} | h_{xi})$.
- **Predictions of past values based on new hints:** In cases where a many-to-one or many-to-many relationship exists between hints and predictions, receipt of a new hint $h_{xq} \in H$, where $H = \{h_{x1}..h_{xm}\}$ and $q > m$ can lead to a prediction $v_{yi} \in V$, a previously collected set of predictions $V = \{v_{y1}..v_{yn}\}$, where $1 \leq i \leq n$. Equivalently, this is the case $(v_{yi} | h_{xq})$.
- **Predictions of novel values based on new hints:** In cases where it can be observed that $(v_{yi} | h_{xi})$ and that $v_{yi} = \mathbf{F}(h_{xi})$, it is desirable to be able to infer function \mathbf{F} and therefore be able to compute a prediction for some new $h_{xj} \notin H$, specifically to compute $\mathbf{F}(h_{xj}) = v_{yj}$. Thus, this is the case $(\mathbf{F}(h_{xj}) | h_{xj})$.

In this section, I discuss three strategies for value prediction – caching, classification, and transduction – and evaluate their accuracies with respect to these three categories.

4.1.1 Caching

The most basic strategy for value prediction involves caching input and output values for the operator to be predicted, and using future instances of input to predict output. A cache is simply a table that associates hint with predicted value(s). In cases where multiple hints can map to the same prediction, a slightly more efficient cache would associate a list of hints with one or more predictions. After it is built, the table is consulted during future executions. In general, over time, the accuracy of the cache increases (as does its size).

For example, consider using a cache in the CarInfo example to predict the output of (*Oldsmobile Alero, Dodge Stratus, Pontiac Grand Am, Mercury Cougar*) from the Edmunds wrapper based on the input (*Midsize coupe/hatchback, 2002, \$4000, \$12000*). Based on this input, the cache would simply consist of a one row, two column table that paired these two values:

Hint	Prediction
Midsize coupe/hatchback, 2002, \$4000, \$12000	Oldsmobile Alero, Dodge Stratus, Pontiac Grand Am, Mercury Cougar

Table 4.1: Cache for the Edmunds wrapper in CarInfo after one example

Future observations that did not already exist in the cache would be added. For example, the input (*Midsize coupe/hatchback, 2002, \$16000, \$18000*) that returns (*Honda Accord, Pontiac Grand Prix, Toyota Camry, Chevrolet Camaro*) would be appended. Note that this process also applies to cases where a similar (but not

exactly identical) hint leads to the same predicted value. For example, it is also true that the input (*Midsize coupe/hatchback, 2002, \$5000, \$12000*) – which differs from the first hint only on the minimum price – returns the same result as the first hint. If we now take all three instances and store them in the cache, the result is Table 4.2.

Hint	Predictions
Midsize coupe/hatchback, 2002, \$4000, \$12000	Oldsmobile Alero, Dodge Stratus, Pontiac Grand Am, Mercury Cougar
Midsize coupe/hatchback, 2002, \$16000, \$18000	Honda Accord, Pontiac Grand Prix, Toyota Camry, Chevrolet Camaro
Midsize coupe/hatchback, 2002, \$5000, \$12000	Oldsmobile Alero, Dodge Stratus, Pontiac Grand Am, Mercury Cougar

Table 4.2: Cache for Edmunds based on three examples

From these examples, it should be clear that caching is limited in that it can only respond to past hints. Furthermore, the minimum size of the cache required to store Table 2 is 184 bytes (counting only the unique data values needing storage) plus the data required to store information about the structure of the cache. However, from the examples seen, storing all of this data is not necessary – the same predictions can be made if we store only the key parts of information that distinguish one prediction from the others. I now describe alternative techniques to caching that can also be used for value prediction.

4.1.2 Classification

Classification involves extracting knowledge from a set of data (instances) that describes how the attributes of those instances are associated with a set of target classes. Given a set of instances, classification rules can be learned so that recurring instances can be classified correctly. Once learned, a classifier can also make reasonable predictions about new instances, even instances that are a combination of attribute values which had not previously been seen. The ability for classification to accommodate new instances makes it an intriguing technology for the speculative execution of information gathering plans because, unlike caching, classification rules allow predictions to be made about novel hints.

As an example, consider again the prediction of the make and model of a car in the CarInfo plan. It turns out that Edmunds returns the same answer (*Oldsmobile Alero, Dodge Stratus, Pontiac Grand Am, Mercury Cougar*) for the criteria (*Midsize coupe/hatchback, 2002*) that also includes any minimum price of \$9912 or less and any maximum price of \$11944 or more. This explains why the third hint in the example above, which had a minimum price of \$5000, returned the same answer as the first. Thus, we see that in the case of the Edmunds wrapper, multiple search criteria can be associated with the same result.

Intuitively, we know that certain features of the hint will always lead a different result than previous hints. For example, if we had altered the type or class of car, we know that we would not get the same set of results returned (and, in fact, we do not). However, intuition also suggests that there are ranges of prices that will return the

same result of (*Oldsmobile Alero, Dodge Stratus, Pontiac Grand Am, Mercury Cougar*), but we do not know exactly what those ranges are. More important is the issue of encoding this knowledge into a predictor. A cache does not support any way to express rules under which hints can map to certain predictions – in contrast, this is exactly how classifiers work.

Given a set of examples, a classifier can be used to learn rules for prediction that are based on features of the hint. The basic idea involves calculating the information gain hint attributes provide in terms of determining an association to a particular target class (the prediction). The more closely associated a particular feature of a set of training instances is with the target classes for each of those instances, the better that feature is at classifying the instances. For example, when considering the examples described in the caching section above, a classifier like Id3 (Quinlan 1986) could induce the following decision rules:

min ≤ 5000: *Oldsmobile Alero, Dodge Stratus, Pontiac Grand Am, Mercury Cougar*

min > 5000: *Honda Accord, Pontiac Grand Prix, Toyota Camry, Chevrolet Camaro*

When presented with an instance previously seen, such as (*Midsized coupe/hatchback, 2002, \$4000, \$12000*), both the cache and the classifier would result in the same prediction. However, when presented with a new instance, such as (*Midsized coupe/hatchback, 2002, \$4500, \$12000*), the cache would be unable to make a prediction whereas the classifier would issue the correct prediction. Note that even when classification leads to an errant prediction, the Confirm operator would prevent errant data from leaving the plan.

The decision tree above is also more space efficient than a cache for the same data. Recall that the cache requires storing at least 184 bytes. The decision tree above requires storing only 132 bytes (nearly a 30% improvement) plus the information required to describe tree structure and attribute value conditions (i.e., price < 18000). In short, classifiers such as decision trees can potentially function as better, more space-efficient predictors. And in the worst case, where the source tuple consists of only a single non-continuous attribute and corresponds to a unique target class, a classifier roughly emulates a cache.

4.1.3 Transduction

Transducers are finite state machines that transform input to output by using the former to iteratively proceed through a series of states that progressively produce the latter. One type of transducer is a string-to-string *sequential transducer*, defined by (Mohri 1997) as $T = (Q, i, F, \Sigma, \Delta, \delta, \sigma)$, where Q is the set of states, $i \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, Σ and Δ are finite sets corresponding to input and output alphabets, δ is the state-transition function that maps $Q \times \Sigma$ to Q , and σ is the output function that maps $Q \times \Sigma$ to Δ^* .

A more general type of subsequential transducer is the *p-subsequential transducer* which extends the definition of a sequential transducer by allowing the final state to

include p additional output arcs. This simply allows the transducer to append on additional characters (i.e., a suffix). Transducers are used in many sub-disciplines of computer science, including natural language processing, where they have been applied to the problem of automatically translating a source string to a target string.

Value prediction by transduction makes sense for Web information gathering plans primarily because of how Web sources organize information and how Web requests (i.e., HTTP queries) are standardized. In the case of the former, Web sources often use predictable hierarchies to catalog information. For example, in the CarInfo example, the summary URL for the Dodge Stratus was *http://cg.com/summ/20812.htm* and the full review was at *http://cg.com/full/20812.htm*. Notice that the second URL uses the key piece of dynamic information (20812) in the first URL. More specifically, it extracts that information from the first URL and combines it with other static data, as shown in Figure 4.1. By learning the full review URL transduction, we can then predict future full review URLs for corresponding summary URLs we have never previously seen.

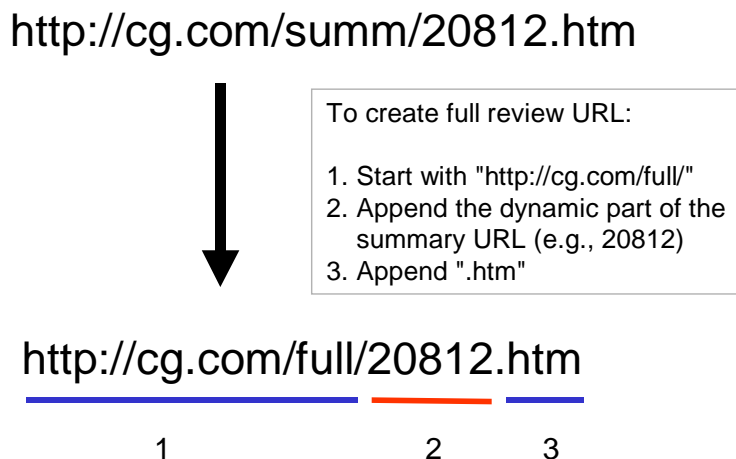


Figure 4.1: Full review URL transduction is part extraction, part

In addition to URLs, transducers can also be used to predict HTTP queries. For example, an HTTP GET query for the IBM stock chart is *http://finance.yahoo.com/q?s=ibm&d=c*. By exploiting the regularity of this URL structure, the system can predict the URL for the Cisco Systems (CSCO) chart.

In this chapter, I define two new types of transducers that extend the traditional definition of p -subsequential transducers. The first is a high-level transducer, called a *value transducer* that constructs the a predicted value based on the regularity and transformations observed in a set of examples of past hints and values. Value transducers build the predicted value through substring-level operations **{Insert, Cache, Classify, Transduce}**. **Insert** constructs the static parts of predicted values. **Cache** recalls past values associated with the hint key. **Classify** categorizes hint information into part of a predicted value. Finally, **Transduce** transforms hint information into part of a predicted value. **Transduce** uses a second type of special

transducer, called a *hint transducer*, in which the operations {**Accept**, **Copy**, **Replace**, **Upper**, **Lower**} all function on individual characters of the hint and perform the same transformation as their name implies, with respect to the predicted value.

To illustrate, consider the process shown in Figure 4.2, which can be applied to predicting the full-review URL in the CarInfo example. The figure shows two transducers. The upper one, the value transducer, performs high-level operations including the insertion of substrings and the call to a lower-level transduction process. The second transducer (in abbreviated form) is a hint transducer. The example shown uses the **Accept** and **Copy** operations to transform the part of the hint value into its proper point in the predicted value. In summary, the value transducer builds the “*http://cg.com/full/*” part, the hint transducer fills in the dynamic part “20812” via copying it from the hint value, and finally the third value transducer operation appends the “.htm” suffix.

The key idea this example shows is that synthesis of a prediction can consist of several sub-operations. Some of these sub-operations, such as Insert, are independent of the hint value. Others, such as transduction, classification, or caching are a function of the hint value. Together, both types of sub-operations enable values to be generated, even from never-before-seen hints.

Transducers lend themselves to value prediction because of the way information is stored by and queried from Web sources. They are a natural fit because URLs are strings that are often the result of simple transformations based on earlier input. Thus, for sources that provide content that cannot be queried directly (instead requiring an initial query and then further navigation), transducers serve as predictors that capitalize on the regularity of Web queries and source structure.

In terms of space efficiency, a learned transducer is generally very compact because what is learned is a set of transformation rules for the hint. For example, once the value transducer shown in Figure 4.2 is learned, it can be applied to all new hints. It should be noted that transducers in other areas of computer science, such as natural language processing, are not always compact and do grow as more examples are seen. In contrast, the types of transducers common to Web information gathering

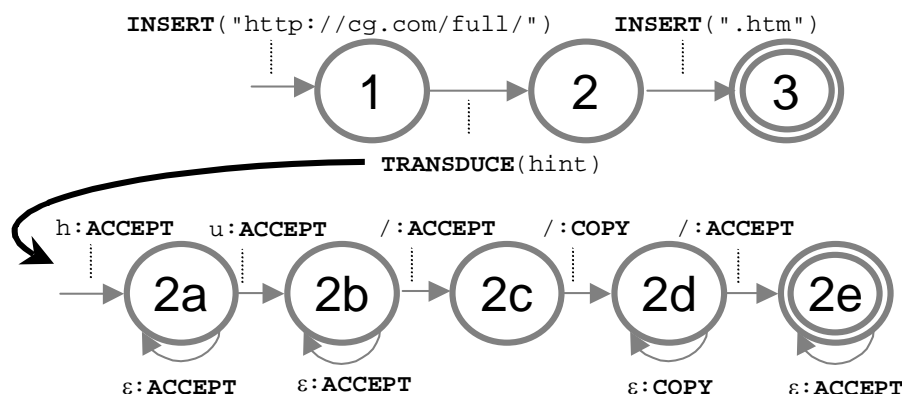


Figure 4.2: Value transducer for the full-review URL in CarInfo

plans, in particular those useful for URL prediction, tend to be more like small functions. Thus, the space demands for a transducer stay fixed over time.

4.1.4 Comparison of techniques

In this section, I have discussed three value prediction techniques, caching, classification, and transduction. Each has its advantages and disadvantages. Caching is simple, always works when given a recurring hint, but is unable to deal with new hints nor generate novel predictions and has the worst space efficiency of the three. Nevertheless, it is a good alternative when no other learning algorithm can be applied.

Classification has better space efficiency, can deal with new hints, and – if necessary – can roughly emulate a cache for cases where all hint features are equally good/bad in terms of prediction. One minor disadvantage is that it is possible for a classifier to generate an errant prediction on a recurring hint, but usually only if that hint contains one or more continuous attributes.

Finally, transduction is the most space efficient of the three, is capable of dealing with new hints as well as making novel predictions, and is especially relevant for Web agent plans because of its applicability at predicting URLs. The only disadvantage to transduction is that it is not always relevant for all speculative opportunities (i.e., some predictions are *associated* with hints, not *computed* based on hints). Table 4.3 compares all techniques along the categories specified earlier including space efficiency.

Strategy	Predicts past values from past hints	Predicts past values from new hints	Predicts novel values from new hints	Space efficiency: growth rate
Caching	Yes	No	No	Linear
Classification	Yes	Yes	No	Sublinear
Transduction	Yes	Yes	Yes	Constant

Table 4.3: Comparing value prediction strategies

4.2 A Unifying Learning Algorithm

In this section, I present a set of algorithms that describe how to combine caching, classification, and transduction in order to generate efficient and accurate predictors. By combining all three strategies, there is an increase in the flexibility for prediction synthesis. For example, with the algorithms I present, it is possible to learn a predictor that synthesizes a new prediction through a combination of caching, classification, and transduction of the hint received.

4.2.1 Value Transducers

My overall approach to value prediction involves inducing a value transducer (VT) that describes how to generate a prediction from a hint, using sub-operations that include classification, transduction, and caching. To learn a VT for the speculative execution of information gathering plans, the following is required:

1. For each attribute of the answer tuple, identify a Static/Dynamic (SD) Template that distinguishes the static parts from dynamic parts of the target string by analyzing the regularity between values of this attribute for all answers.
2. For each static part, add an **Insert** arc to the VT.
3. For each dynamic part, determine if transduction can be used; if so, add a **Transduce** arc to VT.
4. If no transducer can be found, classify the dynamic part based on the relevant attributes of the hint and learn a classifier.
5. If the classifier accuracy is 100%, add a **Classify** arc to the VT.
6. If the classifier accuracy is below 100% (possible when one or more hint features are continuous), build a cache of the data and add a **Cache** arc to the VT.

These steps are implemented in the algorithm LEARN-VALUE-TRANSDUCER, shown in Figure 4.3. The algorithm takes a set of hints, a set of corresponding answers, and returns a VT that fits the data.

```

01 Function LEARN-VALUE-TRANSDUCER returns ValueTransducer
02 Input: set of hints  $H$ , corresponding set of answers  $A$ 
03  $VT \leftarrow \emptyset$ 
04  $tmpl \leftarrow$  LEARN-SD-TEMPLATE ( $A$ );
05 Foreach element  $e$  in  $tmpl$ 
06   If  $e$  is a static element
07     Add Insert ( $e.value$ ) arc to  $VT$ 
08   Else if  $e$  is a dynamic element
09      $DA \leftarrow$  the set of dynamic strings in  $A$  for this  $tmpl$  element
10      $HT \leftarrow$  LEARN-HINT-TRANSDUCER ( $H, DA$ )
11     If  $HT \neq \emptyset$ 
12       Add Transduce ( $HT$ ) arc to  $VT$ 
13     else
14        $CL \leftarrow$  LEARN-CLASSIFIER ( $H, DA$ )
15        $acc =$  TEST-CLASSIFIER ( $CL, H, A$ )
16       If  $acc < 100\%$ 
17          $CH \leftarrow$  BUILD-CACHE ( $H, DA$ )
17         Add Cache ( $CH$ ) arc to  $VT$ 
18       Else
18         Add Classify ( $CL$ ) arc to  $VT$ 
19 Return  $VT$ 
20 End /* LEARN-VT */

```

Figure 4.3: The LEARN-VALUE-TRANSDUCER algorithm

In this algorithm, learning a classifier can be achieved by decision tree induction algorithms such as Id3 (Quinlan 1986). Learning the SD template and the hint transducer, however, require unique algorithms.

4.2.2 Learning templates of string sets

Learning a VT requires first identifying a template for the target value that describes what parts of the target are static and what parts are dynamic. After that, each static part of the template is replaced with **Insert** operations and a each dynamic part becomes a candidate for either transduction, classification, or caching.

To identify a static/dynamic template, I first locate the static parts by comparing the target values to each other. Subsequences of characters that all target values share are considered static parts. The dynamic parts of the template are then simply the varying characters between two static parts (or the start and end of the template). Thus, each SD template will consist of an alternating sequence of static and dynamic parts.

To identify the static parts of a template, I first locate the longest common subsequence (LCS) of a set of target values. To do this, I apply an LCS identification algorithm similar to the one described by (Hirschberg 1975) to the first two target values. If there is an LCS, I then find the next LCS between the current LCS and the next target value. This process continues until the LCS is empty or all target values have been tested and the LCS is not empty. If the LCS is not empty, its *strictly consecutive subsequences* are annotated. Strictly consecutive subsequences are characters of the LCS that appear consecutively (without any intervening characters) in all target strings. For example, while the LCS in the set of strings $\{hello, hall, hill\} = \{hll\}$, this is composed of two strictly consecutive subsequences, $\{h\}$ and $\{ll\}$, as these subsequences always appear one after another in all examples. In contrast, there is at least one example (in fact, all three examples) where a character exists in between the LCS $\{hll\}$.

Once the annotated LCS is identified, we can then iterate through the set of answer values to determine the set of possible SD templates that fit the general form of the answer. Only those templates common to all are kept – from this, one of the set is returned (though all are valid). The algorithm that implements this, LEARN-SD-TEMPLATE, is shown in Figure 4.4.

```

01 Function LEARN-SD-TEMPLATE returns Template
02 Input: set of strings  $S$ 
03  $tpl \leftarrow \emptyset$ 
04  $lcs \leftarrow \text{GET-ANNOTATED-LCS}(S)$ 
05 If  $lcs \neq \emptyset$ 
06  $tplSet \leftarrow \emptyset$ 
07 Foreach string  $s$  in  $S$ 
08  $curTplSet \leftarrow \text{EXTRACT-TEMPLATES}(s, lcs)$ 
09  $tplSet \leftarrow tplSet \cap curTplSet$ 
10 If  $tplSet \neq \emptyset$ 
11  $tpl \leftarrow$  choose any member of  $tplSet$  /* all are valid */
12 Endif
13 Endif
14 Return  $tpl$ 
15 End /* LEARN-SD-TEMPLATE */

```

Figure 4.4: The LEARN-SD-TEMPLATE algorithm

4.2.3 Learning hint transducers

To learn a hint transducer, I also make use of template identification. However, instead of identifying a template that fits all answers, the algorithm I propose identifies templates that *fit all hints*. Based on one of these templates, the algorithm constructs a lower-level hint transducer that accepts the static parts of the hint string and performs character-level transformations (**Accept**, **Copy**, **Replace**, **Upper**, or

Lower) on the dynamic parts. A sketch of the algorithm that implements this, LEARN-HINT-TRANSDUCER, is shown in Figure 4.5.

```

01 Function LEARN-HINT-TRANSDUCER returns HintTransducer
02 Input: the set of hints  $H$ , the set of resulting strings  $S$ 
03 Use LCS to identify static parts between all  $H$ 
04 Foreach  $H, S$  pair  $(h, s)$ 
05    $h' \leftarrow$  extraction of  $h$  replacing static chars with the token 'A'
06    $A \leftarrow$  Align  $(h', s)$  based on string edit distance
07   Annotate  $A$  with character level operations
08 End
09  $RE \leftarrow$  Build a reg expr that fits all annotations (using LCS)
10 If  $RE = \emptyset$ 
11    $ht \leftarrow \emptyset$ 
12 Else
13    $ht \leftarrow$  transducer based on  $RE$  that accepts static subsequences of  $H$  and transduces dynamic subsequences.
14 Endif
15 Return  $ht$ 
16 End /* LEARN-HINT-TRANSDUCER */

```

Figure 4.5: The LEARN-HINT-TRANSDUCER algorithm

For example, suppose prior hints $\{Dr. Joe Smith, Dr. Jane Thomas\}$ had corresponding observed values $\{joe_s, jane_t\}$. The algorithm would first identify the static part of the hints and rewrite the hints using the Accept operation, i.e., $\{AAAAJoe Smith, AAAAJane Thomas\}$ where A refers to the operation Accept. It would then align each hint and value based on string edit distance and annotate with character level operations that reflect the transformation to the observed values, resulting in $\{AAAAALCCRLDDDD, AAAALCCCRLDDDD\}$. Next, it would use the LCS to build the regular expression $\{A^*LC^*RLD^*\}$ fitting these examples and ensure that intermediate operations of indeterminate length (such as the A^* and C^*) shared a common character upon which they stopped. From this, a general predictive transducer can be constructed (partial form shown in Figure 4.6).

For purposes of describing this transducer in text form, we can abbreviate Figure 4.6 as $\{A_{\text{through}}=[], L, C_{\text{upto}}=[], A, L\}$ which means “accept through the first space, lowercase the next character, copy successive characters until the next space, accept the space and then lowercase the next character.”

4.2.4 Detailed example of predictor learning

To better illustrate how a predictor is learned with the LEARN-VALUE-TRANSDUCER algorithm, I describe how the second predictor in the CarInfo plan, which generates the ConsumerGuide summary URL, is learned. In this example, the source value is a

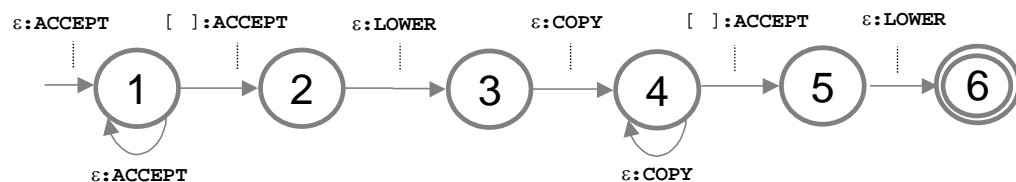


Figure 4.6: Sample hint transducer for the names example

tuple consisting of the make, model, and year of a car (from a list of cars returned by Edmunds). The target value to be predicted is the summary URL that is normally discovered by querying ConsumerGuide.com with the make, model, and year of the car.

It is important to note that the target value also includes the input attribute values - make, model, and year. That is, the target tuple has four attributes. The reason for this is that the Wrapper operator that queries ConsumerGuide.com normally performs a dependent join on the output from the source with the input data. However, this means that the LEARN-VALUE-TRANSDUCER algorithm will be used four times – once for each attribute – so that a hint results in four different value transductions in creating the predicted tuple.

Learning is continuous in the sense that it can be re-applied offline after each run. The reason that the learning is continuous is (1) to allow predictions to be made as soon as possible and (2) to allow the predictors to be refined over time, as more examples have been collected. For purposes of example, let us suppose that the source and target examples shown in Tables 4.4a and 4.4b are observed by the system over successive runs and that learning/re-learning occurs after every run.

Make	Model	Year
Honda	Accord	1999
Honda	Accord	2000
GMC	Sonoma	1997
Acura	NSX	2000

**Table 4.4a: The sequence of source examples
(inputs to the ConsumerGuide search operator)**

Make	Model	Year	Summary URL
Honda	Accord	1999	http://cg.com/summ/2289.html
Honda	Accord	2000	http://cg.com/summ/2289.html
GMC	Sonoma	1997	http://cg.com/summ/2247.html
Acura	NSX	2000	http://cg.com/summ/1997.html

**Table 4.4b: The sequence of target examples
(outputs from the ConsumerGuide search operator)**

I now describe the learning as it would occur tuple by tuple. After the second run of the speculative CarInfo plan, only the first two tuples (*(Honda, Accord, 1999)*, *(Honda, Accord, 1999, <http://cg.com/summ/2289.htm>)*) and (*(Honda, Accord, 2000)*, *(Honda, Accord, 2000, <http://cg.com/summ/2289.htm>)*) would have been observed by the system. LEARN-VALUE-TRANSDUCER was then used to identify a VT for each attribute of the target tuple. As the algorithm specifies, the first step is to define a template and then, based on that template, possibly learn additional transducers or classifiers as necessary. Since two very similar examples are seen initially, the template for the target “make”, “model”, and “summary URL” attributes consists of only a single static element, the template abbreviated here as {**Static**}. As a result,

the resulting VTs for make, model, and summary URL consist of only a single Insert operation.

However, since there is no LCS between the two target year examples, the template for that attribute is **{Dynamic}**. Next, the source tuple attribute values are compared against the target attribute values in order to possibly identify a valid hint transducer. The first target attribute value is the year “1999”. The smallest edit distance between any of the corresponding source attributes (*Honda, Accord, 1999*) and this year value is the source “year” attribute (also “1999”), which has a distance of zero. Next, a case-independent alignment is done between the two strings, the transducer **{CCCC}** is learned, and then the generalized form **Transduce(year: C*)** is retained. This transducer is then verified for the remaining examples: since it correctly produces “2000” from the corresponding source tuple (*Honda, Accord, 2000*) of the remaining example, the transducer is deemed valid and incorporated into the VT for the year attribute. Details about the complete set of VTs after the first run are shown in Table 4.5:

Attribute	Value Transducer
Make	INSERT("HONDA")
Model	INSERT("ACCORD")
Year	TRANSDUCE(year: C*)
Summary URL	INSERT("http://cg.com/summ/2289.htm")

Table 4.5: VTs for the ConsumerGuide search predictor after two examples

After the next run, the system receives a third example: ((*GMC, Sonoma, 1997*), (*GMC, Sonoma, 1997, http://cg.com/summ/2247.htm*)). The predictors are once again re-learned, but this time the target “make”, “model”, and “year” attributes are refined. Because the LCS for the strings (*Honda, Honda, GMC*) = \emptyset , a dynamic template is identified and a VT consisting of **Transduce(make: C*)** is learned. The templates for “model” and “year”, however, are a bit more complicated.

Because the LCS for (*Accord, Accord, Sonoma*) = “o”, the template for the “model” attribute is **{Dynamic, Static, Dynamic}**. Even though we intuitively realize that the correct VT for this attribute should be to simply copy all of the characters of the source “model” attribute, the limited number of examples seen temporarily suggest otherwise. Two hint transducers are learned. The first copies all characters from the source model attribute up to the first ‘o’. Next, an Insert operation inserts an “o” and then a second hint transducer accepts all of the source model characters through the “o” before copying the rest. In short, the fact that an “o” existed in all three examples temporarily made the transducer more complex than it needed to be. The same is somewhat true of the Summary URL attribute – since all examples thus far included a “22”, the system assumed that this substring should be present in all predictions. Table 4.6 shows the state of the VTs after three examples.

Attribute	Value Transducer
Make	TRANSDUCE(make: C*)
Model	TRANSDUCE(model: C _{upto} =[o]), INSERT("o"), TRANSDUCE(model: A _{through} =[o], C*)
Year	TRANSDUCE(year: C*)
Summary URL	INSERT("http://cg.com/summ/22"), CLASSIFY(make, model, year), INSERT(".htm")

Table 4.6: VTs for the ConsumerGuide search predictor after three examples

Finally, the ((*Acura, NSX, 1997*), (*Acura, NSX, 1997*, <http://cg.com/summ/1997.htm>) example irons out the static artifacts that affected both the “model” and “year” attribute and causes the VTs to settle into their final, correct state. Table 4.7 shows the final set of VTs for this predictor.

Attribute	Value Transducer
Make	TRANSDUCE(make: C*)
Model	TRANSDUCE(model: C*)
Year	TRANSDUCE(year: C*)
Summary URL	INSERT("http://cg.com/summ/"), CLASSIFY(make, model, year), INSERT(".htm")

Table 4.7: VTs for the ConsumerGuide search predictor after four examples

As this detailed example has shown, the value predictors learned rely on a hybrid of techniques to predict likely target tuple values. Each predictor consists of VTs that may combine Insert, Transduce, and Classify operations as necessary. Predictors can be learned after only two examples, although as our example predictor has revealed, the final form of the value transducers for a predictor may require a few more examples in order to identify what parts of the predicted value are truly regular (i.e., static) and what parts are not.

4.3 Experimental results

To measure the effectiveness of the approach, I conducted experiments on a set of typical Web agent plans modified for speculative execution. The goal was to compare the benefits of strictly caching versus the benefits of the learning the hybrid predictors I have introduced. Specifically, the goal was to verify that my approach to learning value predictors resulted in:

- **Improved accuracy:** Predictions based on classification and/or transduction make it possible to speculate on recurring as well as new hints, and support the issuing of recurring or novel predictions.
- **Improved space-efficiency:** Since the predictors we learn are more like functions that describe a general process for producing a prediction from a hint, their storage does not necessarily increase linearly as the

number of examples seen increases. In contrast, strictly caching predictors do grow linearly since they capture the association of past source tuples with past target tuples.

- **Faster average agent performance:** Learning hybrid predictors that combine classification, transduction, and caching allow us to obtain faster agent performance, on average, even when dealing with new hints or when needing to issue novel predictions.

I now describe the details of the experimental setup and the results found, using the CarInfo and RepInfo agent plans described in earlier chapters. I also add a new example, the PhoneInfo agent.

The PhoneInfo agent returns demographic information for the geographic location of a particular phone number. The agent takes any phone number and first does a reverse lookup of that number using the Verizon SuperPages (<http://www.superpages.com>) service. The returned state is then used to query a U.S. Census site (<http://quickfacts.census.gov>) in order to obtain demographic data (e.g., population trends, average income) for that location. During the gathering of demographic data, navigation is required from a link on the initial “state summary” page to a subsequent “demographic details” page. The original plan for PhoneInfo is shown in Figure 4.7 and the same plan transformed for speculative execution is shown in Figure 4.8. The PhoneInfo agent is added to the set of plans tested because it demonstrates classification with continuous hint attributes, specifically, the determination of state based on area code.

4.3.1 The learning cycle

After each agent plan was modified for speculative execution, successive runs of the transformed plan predicted data when possible and always gathered more examples so that the predictors learned could be improved. Thus, for the second and future runs, prediction became possible more often, as more examples had been observed and processed by the system.

All learning was done offline. Generally, learning was possible every k runs, where k was customizable by the administrator or user. Prior to each interval of k ,

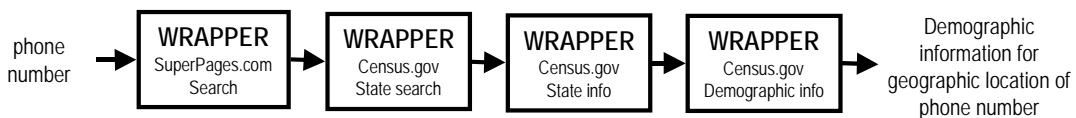


Figure 4.7: The Phone Info agent plan

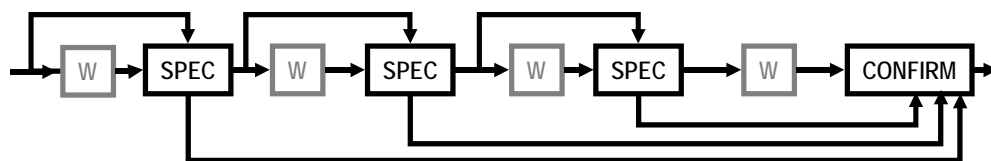


Figure 4.8: Speculative version of PhoneInfo

data would be collected by the system. These represent the set of training examples which would be later fed to the learning algorithm. After every k^{th} run, the system would use the training data to re-learn all of the predictors.

The LEARN-VALUE-TRANSDUCER algorithm was successfully applied to each opportunity in each plan, yielding value transducers that predicted values based on hint transduction or classification. Table 4.8 gives names to each predictor and summarizes the primary technique used in generating predictions from hints:

Predictor	Agent	Hint (source value)	Prediction (target value)	Primary learning method identified by LVT for new attribute
Car-List	CarInfo	User car preferences	List of matching cars from Edmunds.com	<i>Classification</i>
Car-Summary	CarInfo	Car make, model, and year	ConsumerGuide.com summary page	<i>Classification</i>
Car-Full	CarInfo	ConsumerGuide.com summary page	ConsumerGuide.com full review page	<i>Transduction</i>
Rep-List	ReplInfo	User 9-digit zip code	List of federal representatives from Congress.org	<i>Classification</i>
Rep-Cand	ReplInfo	URL to federal representative bio	Representative name and title	<i>Caching</i>
Rep-Summary	ReplInfo	Representative name and title	Open Secrets summary page URL	<i>Caching</i>
Rep-Graph	ReplInfo	Open Secrets summary page URL	Open Secrets funding graph URL	<i>Transduction</i>
Phone-State	PhoneInfo	User phone number	State of origin, as identified by Superpages.com	<i>Classification</i>
Phone-Summary	PhoneInfo	State	Census summary page URL located at QuickFacts.census.gov	<i>Caching</i>
Phone-Detail	PhoneInfo	Census summary page URL	Census demographic details page URL	<i>Transduction</i>

Table 4.8: Summary of predictors learned

Overall, Table 4.8 shows three important things. It shows that the learning algorithms successfully learned a predictor for each speculative opportunity (i.e., there was never a time that the algorithm could not learn a predictor). Second, the table shows that the algorithms resulted in value transducers based on different primary methods of prediction, as a function of past hint/value relationships observed. Third, even when transduction was impossible and classification was not relevant (i.e., hint consisted of only a single, non-continuous feature), caching could still be used. In short, the table shows how my approach to learning value predictors allows either transduction, classification, or caching to be applied to a given

speculative opportunity, based on the nature of relationship between the source and target data.

I now present results having to do with the accuracy and space efficiency of the predictors shown in Table 4.8.

4.3.2 Measurements of predictor accuracy

One of the overall goals is to compare the accuracy of predictors learned via the algorithms presented in section 4 versus predictors that operate strictly by cached data. In doing so, it is important to assess the accuracies with respect to the three prediction scenarios described in Section 3: the cases of the (I) recurring hint / recurring prediction, (II) novel hint / recurring prediction, and (III) novel hint / novel prediction.

Note that not all of these scenarios are relevant to each speculative opportunity. For example, there is no case (II) for the Car_{full} predictor because each unique summary page corresponds to a unique full review URL. Similarly, there is no case (III) for the Car_{summary} predictor because more than one car could correspond with the same summary page. We now describe the accuracy of the predictors in Table 4.8 for each of the prediction scenarios. When learning each predictor, instances were drawn from typical distributions for that domain; for example, instances for RepInfo were drawn from a list of addresses of individuals that contributed to presidential campaigns (obtained from the FEC) – a distribution that closely approximates the U.S. geographic population distribution. Similarly, the phone numbers used in PhoneInfo came from a distribution of numbers for common last names.

Case I: Recurring hints, recurring predictions

In terms of issuing correct predictions for recurring hints, caching generally yields accuracies of 100%. The only cases under which this would not be true would be when the operator/source producing the target values was not deterministic or if there were limits on the maximum cache size (e.g., memory constraints).

Likewise, the algorithms I have presented yield accuracies of 100% for any case where the recurring hint mapped to the same prediction it did when the association was learned. This is because, in each case, the predictor learned was validated on its own training data. If this validation revealed an accuracy of less than 100%, then the next alternative predictor was learned instead (either classification or caching, depending on what failed). Note that it is also possible to have less than 100% accuracy when validating a learned classifier on its training data if any feature of the source (hint) value contained a continuous attribute.

Case II: New hints, recurring predictions

When presented with new hints, simple caches cannot issue predictions, even if they map to recurring hints. This is because caches associate distinct source values with target values and are not designed to infer anything about a new source value.

In contrast, classifiers can handle situations where there is a many-to-one mapping between hints and predictions and thus allow reasonable predictions to be made from a new hint. In Table 4.8, Car_{summary}, Rep_{list}, and Phone_{state} involve

prediction of target values where there is a many-to-one relationship between source and target. I measured the predictive accuracy of each of these predictors on previously unseen hints, as the number of training examples increased. The results were based on averaging a 10-fold cross-validation sample of the data in each case (where no data in the test fold was in any of the training folds). Figure 4.9 shows the results for each of the classifiers $\text{Car}_{\text{summary}}$, Rep_{list} , and $\text{Phone}_{\text{state}}$.

The figure shows that, generally, as the number of training examples increased, the predictive accuracy on unseen examples also increased for each of the predictors. In addition to better accuracy, more examples allow more opportunities for prediction. This is most noticeable in prediction scenarios that do not strictly involve continuous features, such as in the $\text{Car}_{\text{summary}}$ predictor. Figure 4.10 shows the number of times that predictions were not possible by the $\text{Car}_{\text{summary}}$ predictor as a function of the number of examples seen.

Note that the $\text{Phone}_{\text{state}}$ classifier performance improves significantly just after 600 examples. This is due to the fact that the accuracy of the classifier on some of the larger states (like California, Florida, New York, and Texas) is much higher around this point. Since testing instances from larger states also appear more often then file, performance correspondingly improves.

Case III: New hints, novel predictions

The approach described in this paper also allows certain predictors to issue novel predictions for new hints. Such opportunities occur when the cardinality between source and target value is one-to-one and when the target value can be produced

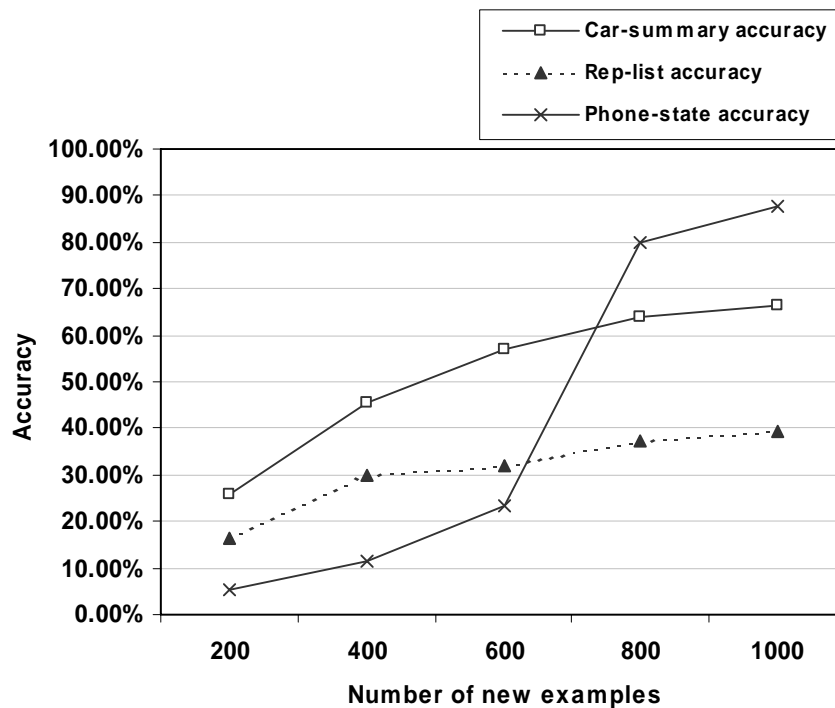


Figure 4.9: Predictive accuracy of $\text{Car}_{\text{summary}}$, Rep_{list} , and $\text{Phone}_{\text{state}}$ classifiers

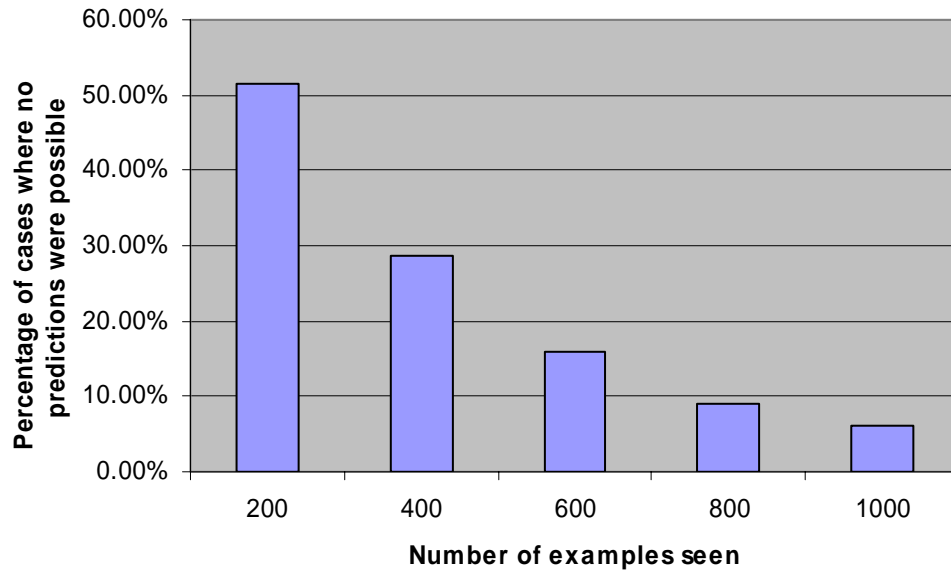


Figure 4.10: Frequency of “no predictions possible” by $Car_{summary}$ based on number of examples seen

through some type of hint attribute value transduction. In Table 4.8, only the Car_{full} , Rep_{graph} , and $Phone_{detail}$ predictors rely purely on transduction of a hint tuple.

Once learned, these transducers have accuracies of 100%. They essentially capture a function and then perform that function on all new hints. The only time when these transducers make mistakes are when too few examples have been seen and LEARN-VALUE-TRANSDUCER identifies an incorrect template. For example, as described earlier, learning that the first three attributes of the $Phone_{state}$ predictor were direct copies of input attribute values (i.e., the definition of a dependent join) required more than two examples for some of the attributes because an LCS “artifact” was caused by learning based on a fewer number of examples.

To understand the difficulty of identifying the correct transducer, we investigated how many examples were required (on average) to learn the transducers Car_{full} , Rep_{graph} , and $Phone_{detail}$. In doing so, we first identified the correct transducer for each case. Then, using 10 different randomized orderings of sample source/target values, we averaged the number of examples required before the correct transducer was learned. Table 4.9 shows these results.

Predictor	Average number of examples required
Car-Full	3
Rep-Graph	8
Phone-Detail	3

Table 4.9: Average number of examples required to learn Car_{full} , Rep_{graph} , and $Phone_{detail}$

4.3.3 Measurements of predictor space-efficiency

In addition to comparing the approach described in this paper to caching in terms of accuracy, I also compared the space efficiency of the two techniques. Specifically, I

measured the space efficiency of three classification-based predictors ($Car_{summary}$, Rep_{list} , and $Phone_{state}$) and three transduction-based predictors (Car_{full} , Rep_{graph} , and $Phone_{detail}$) as well as the space required by strictly caching predictors for the same data. The process involved forming the predictor based on a set of training data and then exporting the structure to the file system for future runs. The space measured was the total number of bytes required by the data structure.

Table 4.10a shows the results for each classification-based predictor, its cache counterpart, and the number of training instances seen by each prior to the exporting of the data structure. In addition to a bytes-to-bytes comparison, the table also shows the resulting space-efficiency “savings” provided. Table 4.10b shows the same information for the transduction-based predictors.

Classifier name	Number of examples seen	Cache size (bytes)	Classifier size (bytes)	Space savings
Car-summary	200	24817	16399	33.92%
Car-summary	400	48577	29675	38.91%
Car-summary	600	72563	42521	41.40%
Car-summary	800	95923	54840	42.83%
Car-summary	1000	119420	67005	43.89%
Rep-list				
Rep-list	200	20791	13725	33.99%
Rep-list	400	40654	25867	36.37%
Rep-list	600	60531	37277	38.42%
Rep-list	800	80312	48272	39.89%
Rep-list	1000	100177	58892	41.21%
Phone-state				
Phone-state	200	21729	13638	37.24%
Phone-state	400	42729	25883	39.43%
Phone-state	600	63729	38088	40.23%
Phone-state	800	84729	52482	38.06%
Phone-state	1000	105729	64939	38.58%

Table 4.10a: Space efficiency of classification-based predictors vs. caches

Transducer name	Number of examples seen	Cache size (bytes)	Transducer size (bytes)	Space savings
Car-full	2	310	58	81.29%
Car-full	10	1550	58	96.26%
Car-full	100	15500	58	99.63%
Rep-graph				
Rep-graph	2	202	58	1.00%
Rep-graph	10	1010	58	94.26%
Rep-graph	100	10100	58	99.43%
Phone-detail				
Phone-detail	2	192	58	69.79%
Phone-detail	10	960	58	93.96%
Phone-detail	100	9600	58	99.40%

Table 4.10b: Space efficiency of transduction-based predictors vs. caches

4.3.4 Effects on average runtime performance

In addition to comparing a hybrid and strict caching approaches in terms of accuracy and space efficiency, I also conducted experiments that demonstrate the resulting performance benefits from a hybrid approach. Specifically, I now describe the results of using a hybrid predictor vs. one based strictly on caching to improve the performance of the CarInfo, RepInfo, and PhoneInfo agents.

For each of the agents tested, I used a smaller subset of the possible inputs that each agent could receive. I did this to limit the number of examples I would need to run to show the resulting effect, and also to avoid disrupting the site with (tens of) thousands of requests. For each agent, I chose well-defined subsets: for example, in the CarInfo agent, I looked only at queries involving compact cars produced in 2000-2002 for various price ranges occurring between \$4000 and \$18000. For the RepInfo and PhoneInfo agents, I looked at randomly ordered lists of valid 9-digit zip codes and valid phone numbers, respectively, in the states of Arizona and Colorado.

The results obtained from CarInfo agent execution are shown in Figure 4.11. The figure is broken up into a set of “performance groups”. Each group contains three bars, each one corresponding to the average time-to-emit the first, average, and last tuple. The “time to emit the average tuple” means the average time at which a tuple was available (different inputs resulted in varying numbers of cars found). For example, if three tuples were produced at the times (3s, 5s, 19s), the time to average tuple would be $(27/3 =)$ 9ms. The first performance group shows the first, average, and last tuple performance for CarInfo with no speculative execution. The groups succeeding to the right show the same information with speculative execution for inputs 1-25, 26-50, and so on. The figure is composed in this manner to show the progressive performance improvement due to learning. For example, one would reasonably expect predictive accuracy to gradually improve for performance groups

to the right, since more examples have been seen to that point. Interpretation of these results is continued in the discussion section (4.3.5) that follows.

The results from the RepInfo agent are shown in Figure 4.12. Recall that these runs describe the performance given a randomly ordered list of valid nine digit U.S. zip codes for the states of Arizona and Colorado. The performance results shown in Figure 4.12 are also broken up into the same set of performance groups as was the CarInfo agent performance in Figure 4.11. The only difference is that the speculative execution runs are grouped for every 20 inputs.

Finally, Figure 4.13 shows the results from the PhoneInfo agent. Similar to the RepInfo agent, these runs were conducted using a randomly ordered list of valid phone numbers for businesses in Arizona and Colorado. One important difference between PhoneInfo and the other two plans is that the former only outputs a single tuple – thus, there is no need to measure the time to output the average tuple or last tuple.

4.3.5 Discussion

The results related to accuracy and space-efficiency generally show that, when possible, the approach I have introduced produces smaller, more intelligent predictors than a predictor based strictly on caching. On one hand, the LEARN-VALUE-TRANSDUCER algorithm makes 100% accuracy possible for recurring hints, identical to what would be obtained from an approach based solely on caching. However, the real value of the approach is shown when it comes to dealing with new hints and making novel predictions. A prediction system based only on caching cannot deal with new hints, even if there is an obvious relationship between hint and prediction, a caching system that has not seen a particular new input cannot manufacture a prediction. In contrast, learning a generalized transducer affords this opportunity. In addition, when there is a many-to-one relationship between source and target values (target values apply to various combinations of source values),

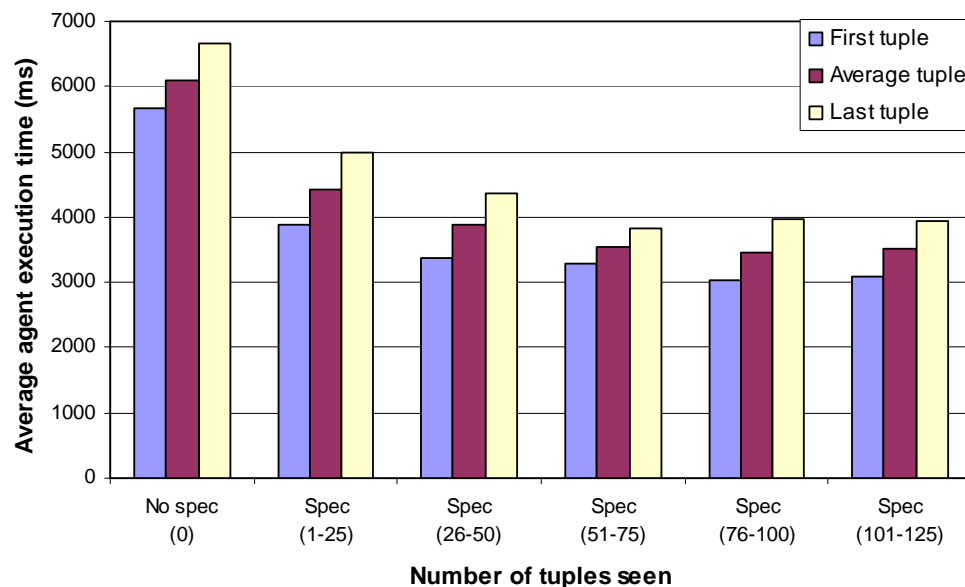


Figure 4.11: Impact of learning on CarInfo agent execution performance

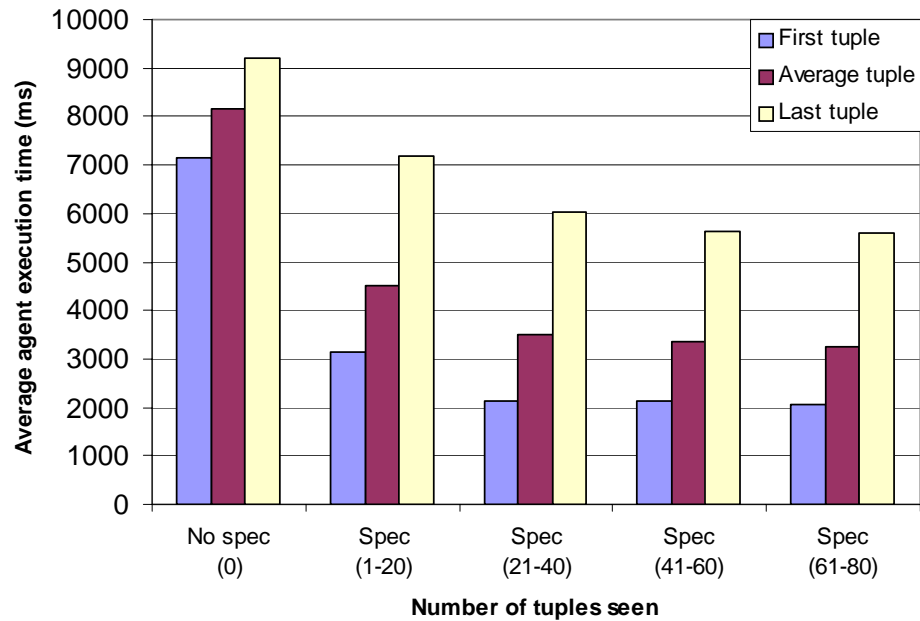


Figure 4.12: Impact of learning on RepInfo agent execution performance

Figure 4.9 shows that classification can be an effective technique for reasoning about certain features of that new hint which can be used to justify a prediction. Further, Figure 4.10 shows that, as more examples are seen, the predictive accuracy of these classifiers continues to improve.

When there is a one-to-one relationship between source and target values, and when target value is simply an manipulated form of one or more source attribute values, the results show that transduction can be an effective solution. By capturing the functional relationship between the source and target, Table 4.9 shows that transducers allow novel predictions to be made on new hints. After only a few

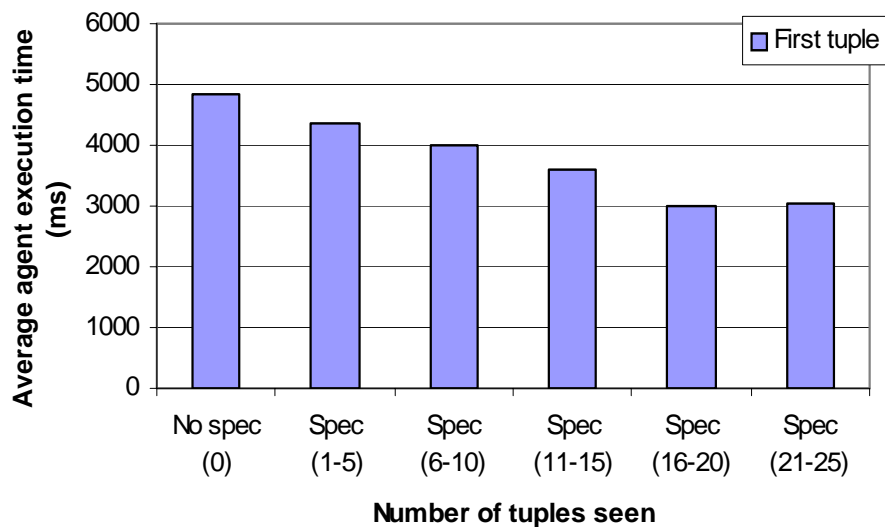


Figure 4.13: Impact of learning on PhoneInfo agent execution performance

examples, transduction accuracy can be 100%. Although it is a technique particularly well-suited to prediction of URL strings, interleaved navigation occurs so frequently in online information gathering that many types of agents can benefit from this type of learning.

The results also show that the predictors learned through the approach I have introduced increase the utility of speculative agent execution. Given a mix of recurring and new hints, prediction is generally more accurate with a hybrid approach that adds classification and transduction. As a result higher average plan speedups are possible.

In addition to being more accurate, the predictors learned through the algorithms described in this paper are more space efficient. Because they encode rules or functions – and not associations of data – these predictors require much less storage than caches for the same set of source/target values. For example, Table 4.10b shows that value transducers that involve **Insert** or hint **Transduce** operations require only a fraction of the space of a cache – more importantly, once learned, its accuracy is 100% and their size thus remains bounded (i.e., it does not continue to increase with the presence of more examples).

Finally, Figures 4.11, 4.12, and 4.13 show that learning predictors that combine classification, transduction, and caching is effective at significantly improving the performance of agents – even when the input to those agents is almost 100% unique. In particular, the benefits of classification (able to predict a past value with a new hint) and transduction (able to predict a new value given a new hint) play an important role in making this possible. Each of Figure 4.11, 4.12, and 4.13 shows a similar trend: an initial performance improvement due to quickly-learned transducers and then gradually better performance as the classifiers involved in each agent see more examples. A good example of this is the RepInfo agent, which shows sharp improvement initially because the senators from each state are relatively easy to learn with only a few examples – thus, the time to first tuple improves dramatically within having seen only a few examples. However, the representatives from each are not quickly learnable, since they vary per zip code. Figure 4.12 shows that over time, however, rules can be learned that allow this prediction to be made even for nine-digit zip codes not previously seen.

4.4 Summary

Profitable speculative execution of agent plans that gather information is fundamentally linked with the ability to make good predictions. The more accurate the predictor, the better the average speedup of execution. While caching is one simple technique that can be used as a basis for prediction, it does not scale well and is not able to handle new hints, even if the new hint corresponds to a prediction that has been previously made.

Classic machine learning techniques, however, can vastly improve the accuracy and space-efficiency of simple cache-like predictors and act as a powerful complement to caching. In particular, I have described how two techniques – classification and transduction – can be combined and applied to the problem. Classification and transduction allow more opportunities for prediction because they

can often respond correctly to new hints and are capable of generating novel predictions. Experimental results confirm this improved accuracy and also show that the space required to store such value predictors is much less than the space required by caches. Overall, this approach represents a successful compromise and hybridization of classification, transduction, and caching that can generate smaller but more intelligent predictors.

Chapter 5

Related Work

In this section, I survey previous work related to the core contributions of this dissertation. First, I consider other approaches to the efficient execution of information gathering plans, including work that spans the AI and database communities. Next, I focus on research specifically related to speculative execution and, more generally, the pre-processing of anticipated operations. Finally, I discuss a variety of other work related to value prediction, including the relationship of value prediction to speedup learning and other approaches to inducing transducers.

5.1 Expressive and efficient plan execution

My approach to efficient agent execution has two key components: a streaming dataflow executor and an expressive language for efficient information gathering. In relating this approach to existing and ongoing research by others, I first address recent work on network query engines as well as more general plan execution systems, to illustrate how the architecture I have introduced leverages aspects of both. Next, I compare the dataflow-style language I have introduced with existing dataflow-style languages used by other scientific computing and signal processing systems.

5.1.1 Network query engines

As discussed earlier, network query engines such as Tukwila (Ives et al. 1999), Telegraph (Hellerstein et al. 2000) and Niagara (Naughton et al. 2001) have focused primarily on efficient and adaptive execution (Avnur and Hellerstein 2000; Ives et al. 2000; Shanmugasundaram et al. 2000; Raman and Hellerstein 2002), the processing of XML data (Ives et al. 2001), and continuous queries (Chen et al. 2000; Chandrasekaran et al. 2003). All of these systems take queries from users, form query plans, and execute those plans on a set of remote data sources or incoming streams. Like the system described here, network query engines rely on streaming dataflow for the efficient, parallel processing of remote data.

The work described here differs from network query engines in two ways. The first, and most important difference, has to do with the plan language. Plans in network query engines consist of relational-style operators and additional operators for adaptive or XML-style processing. For example, Tukwila includes a double pipelined hash join and dynamic collector operators for adaptive execution (Ives et al. 1999), as well as X-scan and Web-join operators for streaming XML data in the form of binding tuples. Telegraph contains the Eddy operator (Avnur and

Hellerstein 2000) for dynamic tuple routing and the SteMs operator to leverage the benefits of competing sources and access methods. Niagara contains the Nest operator for XML processing and other operators for managing partial results (Shanmugasundaram et al. 2000). Outside of these special operators for adaptive execution and XML processing, plans in network query engines look very similar to database style query plans. These plans are typically inaccessible – users can only alter the queries that generate plans, not the plans themselves. This makes any kind of information gathering task beyond basic querying difficult, if not impossible.

In contrast, the plan language I have described is more expressive and the plans are accessible. Like network query engines, the language includes relational-style operators and those for processing XML data. However, it also includes operators that support conditional execution, interaction with local databases, asynchronous notification, and user-defined single-row and aggregate functions. The plan language I have introduced also supports the capability of referencing subplans, a feature that increases opportunities for modularity, re-use, and enables looping-style information gathering tasks to be easily accomplished through recursion. In contrast, languages supported by network query engines do not support such capabilities. As a result, they cannot address the looping and monitoring requirements of plans such as Homeseekers.

In addition, there is the issue of plan accessibility. Although plans in the language I have described can be generated by query processors – just as plans produced by network query engines – they can also be constructed and modified using a text editor. This provides the ability for users to specify more complicated tasks that could not otherwise be expressed in a query. NiagaraCQ (Chen et al. 2000) provides similar accessibility in the sense that it does allow for more flexibility in terms of what kind of processing (action) can be performed per continuous query through use of a stored procedure language. However, it is not necessarily the case that streaming data can cross the boundary into the stored procedure and be executing just as efficiently (streaming dataflow style) as it was in the plan formed by the continuous query. The key advantage of accessibility in the language I have introduced is that the benefits of streaming dataflow are available to all aspects of a plan – the user can use the existing operator set or the apply/aggregate operators to build more functionality into the language and still reap the benefits of streaming dataflow execution between all of the operators. As shown in the experimental results, this can lead to significant performance improvements.

Finally, in this dissertation, I have presented a new thread-pool model for dataflow-style execution of information agent plans. This model relies on communication of work (execution events) via an asynchronous FIFO queue implemented as a circular buffer. Through this architecture, it is possible to realize (within the selected bounds) all of the horizontal and vertical parallelism demanded by a plan at runtime. Throttling the level of parallelism (if necessary) is easy – through an external configuration file, users simply change the number of threads in the thread pool or the size of the work queue. Use of a thread pool also prevents exhaustion of system resources.

5.1.2 General purpose plan execution systems

The language and execution system described here is partially inspired by past work on agent execution systems. In particular, two features commonly found in these systems were influential: support for execution of more generic agent plans, and support for the execution of concurrent actions. Several functioning plan executors exist, including the RAP system (Firby 1994) and PRS-Lite (Myers 1996), as well as other designs (Williamson et al. 1996) that have been proposed but not implemented.

Both RAP and PRS-Lite are well known systems that have placed an emphasis on highly parallel plan execution. The expressivity they support has been influenced by past work in expressive plan languages, such as RPL (McDermott 1991). However, plan executors like the RAP system and PRS-Lite typically do not route information between operators. Instead, the process of execution consists of the operator pre-conditions being fulfilled and their resulting post-conditions being generated. This is an important difference that explains why the execution architecture described here supports some forms of parallelism not found in these other systems. For example, the vertical parallelism of pipelining data between operators has no analog in these general plan executors. In short, a distinction between the system described here and other generic plan executors is that the former is specifically tuned for efficient plan execution when remote, relation-like information is routed between operators, whereas the latter are not.

There are other higher-level differences that should be noted. Unlike generic plan executors, the plan language described here supports the notion of subplans and recursion. However, unlike these generic executors, the system here does not automatically generate plans from a set of initial and goal conditions, nor does it interleave planning and execution. Instead, the plans discussed here are either written by an end-user or are generated from another tool, such as an information mediator, much like a database query processor prepares a plan for execution.

5.1.3 Other dataflow computing languages

There exist a large number of dataflow-related languages that are related to the work described here. Some, such as Id (Arvind et al. 1978) and Val (McGraw 1982) are meant to be compiled for execution on traditional dataflow machines. There are some similarities to the language described here: for example, VAL is single-assignment and Id operates on streams and supports looping as well as re-entrancy. However, there are also several differences, from the type of functionality that the operators provide to the way parallelism is achieved. For example, in the language I have described, parallelism is implicit (i.e., the programmer does not need to worry about denoting parallel code) yet can be translated into a dataflow graph without a special compiler. In contrast, while parallelism is also implicit in VAL and Id, a special compiler is required to translate the serially specified program into a dataflow graph.

In addition to VAL and Id, there are also purely functional languages that are related to dataflow computing only in that they simplify the declaration of data parallel execution. Yet, these languages do not necessarily assume execution occurs on a dataflow machine. Included in this group of languages are SISAL (Feo and

Cann 1990), Haskell (Jones and Hughes 1998), and pH (Nikhil and Arvind 2001). Like the language described here, these other functional languages provide implicit, horizontal parallelism. They are modular and support recursive calls. However, they differ from the language here in that their operators perform lower-level functions on typically scalar values or numerical arrays. Thus, they are not meant to operate on incoming streams of remote data, so they have no need for an iterative style of execution and thus no need for streaming. In contrast, streaming is a useful technique in the system described here and for network query engines because it compensates for the significant latencies of network I/O and the large data sets that can be returned from remote queries.

Finally, embedded systems (Lee 2002) also rely on dataflow-style languages for execution. For example, two popular commercial languages are Verilog HDL (Thomas and Moorby 1998) and VHDL. These languages provide dataflow-style constructs for use on digital signal processing tasks, often on systems that exist within consumer devices such as cell phones and cameras.

To better illustrate the similarity between these languages and the one I have presented, let us consider a simple Verilog example. Verilog consists of modules that have sets of input and output variables. It supports conditional logic, loops, and special constructs (the “initial” and “always” constructs) for indicating what parts of a module require continual execution. For example, Figure 5.1 shows the logic gate design for a basic multiplexer. This multiplexer routes input line *in0* to *out0* when the value of *in0* is zero and otherwise routes *in1*.

Figure 5.2 shows the corresponding Verilog module that can be developed to describe this multiplexer.

There are some important similarities to note about the module above in relation to the language described here. First, both Verilog and the language I have presented support textual representations of dataflow style graphs. Second, both languages are modular, with Verilog modules also consisting of input and output variables. Finally, the logical operations in Verilog consume a set of input variables and produce a set of output variables, just as operators do in the agent language I described.

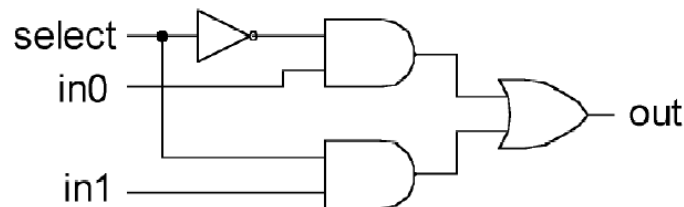


Figure 5.1: Basic multiplexer logic

```

module multiplexer (in0, in1, select, out);
  input in0,in1,select;
  output out;

  wire s0,w0,w1;

  not (s0, select);
  and (w0, s0, in0),
      (w1, select, in1);
  or (out, w0, w1);

endmodule

```

Figure 5.2: Verilog module that represents the multiplexer logic in Figure

In terms of differences, dataflow computing for embedded systems is generally distinct from Web information gathering systems in its focus on more low-level system challenges. For example, power usage and real-time processing are very important issues during the execution of embedded systems – these issues are of less concern with respect to information agents. Also, embedded systems engage in lower-level types of data processing (e.g., signal processing), working with streams of bits, not streams of tuples as do information agents. Nevertheless, embedded systems share the same desire for concurrent execution as information agents, and are frequently used to process streams of data. Thus, they require languages that allow programmers the ability to specify concurrent, stream-oriented processing.

5.2 Speculative execution

Historically, speculative execution is a technique that has been associated with lower level execution. It is a strategy addressed frequently in the context of processor architecture and compiler design. However, far less attention has been given to the use of speculative execution at higher levels of execution, such as at the operating system or database level, although the work on continual computation is certainly relevant and motivating to the work here. In this next subsection, I survey the use of speculative execution at various levels of computation, but focus more on how my approach compares to other strategies for reducing I/O penalties during the execution of information gathering plans, such as prefetching and execution based on approximate or partial results.

5.2.1 Execution based on partial and approximate results

The goals and approach described in this paper are perhaps most closely related to those embodied in current research on the use of partial or approximate results by network query engines. The use of approximation has been shown to be an effective tool for communicating the likely result of queries that involve online aggregation of data-intensive sources (Hellerstein et al. 1997). The general idea is to communicate estimations (and estimation confidences) of otherwise expensive aggregate queries to the user through an interface.

Inspired by this work, some network query engine research has focused on the use of partial results to speed up query plan processing. In Niagara (Naughton et al. 2001), for example, a partial results approach is used to better parallelize the execution of a query plan (Shanmugasundaram et al. 2000) – this is exactly the same as the motivation described in this thesis. The Niagara approach involves communicating approximations of aggregate operators to downstream operators as execution proceeds. Later, upstream operators update their predictions as necessary by routing differentials or re-evaluations to downstream operators. The goal of Niagara’s approach to partial results is to extend approximation techniques to arbitrary blocking operators. For example, while traditional database query languages support blocking operators like Average or Max, newer languages have different types of blocking operators (such as those for nesting XML), motivating the need for a more general strategy in terms of approximation.

There are two major differences between my speculative execution approach described here and Niagara’s partial result strategy. One is that the latter is meant to be applied to operators that block on input tuples, not remote I/O. For example, partial results can be obtained from a Sort or Nest operator, which require all of their inputs before generating output. However, partial results cannot be obtained from a Wrapper operator because it fails to meet the requirements for partial-results capable operators, as listed in (Shanmugasundaram et al. 2000). For example, the “Anytime” output property does not make sense for the Wrapper operator because it is not possible for this operator to produce a partial answer before its remote request is filled. In contrast, the speculative execution approach here can be applied to nearly any operator in a plan (as long as the operator does not affect the external world in unrecoverable ways). Thus, it can be used to optimize plans that suffer from a slow Wrapper operator or a slow aggregate function, like Sort. A second difference between speculative plan execution and use of partial results in the Niagara query engine is that the former advocates only optimizing the most expensive data flow, whereas the latter approach does not specify on which flows partial results optimization should take place. Thus, it could potentially be used to optimize flows that are not part of the main bottleneck in a plan, thus creating unnecessary overhead during execution.

Telegraph (Hellerstein et al. 2000) is another network query engine that uses a partial results strategy to increase the performance of the processing of its queries to online sources. (Raman and Hellerstein 2002) describe an approach that allows partial tuples (tuples with some values “deferred”) to be emitted so that they can be displayed to the user as soon as possible. The idea behind the strategy is to limit the set of deferred information to only those cells of result tuples that remain to be gathered. Overall execution time remains the same with this approach; the key gain is the improved performance for those parts of query answer tuples that have already been computed. Emitting sub-tuples as soon as possible depends to some extent on Telegraph’s use of eddies (Avnur and Hellerstein 2000) which bear some relationship to speculative execution in that operators are allowed process intermediate query results out of order.

The Telegraph approach is different from both speculative execution and Niagara's partial results strategy in that it is targeted, like online aggregation, at returning as many correct results to the caller as soon as possible. There is no approximation in this approach, so there is no chance of suffering from the processing of errant data. At the same time, the approach cannot return entire answers any earlier than normal. In contrast, speculative plan execution can potentially return entirely correct answers much faster than the original plan and is also guaranteed not to return errant answers. While it requires a small degree of overhead, the resulting plan speedups can significantly outweigh these costs.

5.2.2 Executing anticipated actions in advance

Speculative plan execution shares the same motivation as the more general notions of continual computation (Horvitz 2001) and time-critical decision making (Greenwald and Dean 1994) – specifically, the desire to leverage idle computer resources to execute anticipated actions. In the case of time-critical decision making, the challenge is to manage a finite amount of computational cycles in a dynamic planning environment. For example, the work describes the challenge of managing air traffic control for a busy airport where there are busy periods and slow periods. By exploiting the regularity of these periods, on-line deliberation time can be better scheduled. The use of available cycles for online deliberation about future problems is somewhat analogous to the use of idle cycles in our approach to speculative plan execution.

(Horvitz 2001) presents continual computation principles and strategies that apply to speculative plan execution and can be found in this work. For example, calculating the expected value of precomputation and ranking the most productive use of idle time are general proactive computation principles that can be found in the SPEC-REWRITE strategy for identifying the MEP and evaluating costs of various speculative transformations. (Horvitz 2001) also identifies general issues of precomputation that encapsulate some challenges raised in this work. For example, the overhead of speculation discussed here is an example of the cost of “shifting attention” in the landscape of continual computation. Overall, speculative plan execution is best characterized as an example of continual computation.

Finally, past work on predicting user actions in advance is also relevant. (Motoda and Yoshida 1997) and (Davison and Hirsh 1998) describe approaches to predicting the next command a user will issue. In the case of the latter, the work describes an approach that analyzes the regularity in sequences of UNIX commands in order to predict the next command that the user will issue. Predicting user actions can be used for speculative execution, but an important difference is that *user idle time* is being exploited instead of *system idle time*, as is the case in this work. Another subtle difference is the overall goal of command line prediction is to create a more helpful command shell that anticipates what future actions will be needed, a goal similar to that of other intelligent interfaces like Letizia (Lieberman 1995). In contrast, the use of speculative execution here is strictly for improving average performance.

5.2.3 Prefetching data

In a narrow sense, speculative execution can be thought of as a mechanism for prefetching, the gathering of data in advance of its request. There are many uses of prefetching in information systems research, from the construction of materialized views (Chaudhuri et al. 1995; Levy et al. 1995; Ashish 2000) in databases to remote Web site page prefetching (Padmanabhan and Mogul 1996; Horvitz 1998; Jiang and Kleinrock 1998).

As a whole, the purpose of all prefetching systems is to gather data that will likely be needed before it is requested, as a means for reducing the I/O-penalties involved during the execution of the actual request. Prefetching can be viewed as an indirect method of speculation in the sense that it does not involve the pre-execution of inevitable plan operations ahead of schedule, but instead increases the locality of remote (or expensive to access) data *likely* to be requested (but not necessarily requested).

My approach to speculative execution differs from other types of prefetching systems in three major respects. One is that it is a run-time activity which involves pre-executing costly plan operators. While that set of operators may indeed be those used for data retrieval, this is not necessarily the case. That is, speculative execution offers a general means to pre-execute any kind of plan operator in advance, provided that (a) there are sufficient resources, (b) there is potentially a significant profit from the eager execution, and (c) that there is some way to “un-do” speculated actions, should the speculation turn out to have been incorrect. Thus, speculative execution is a more general approach to performance improvement in plans (and programs).

A second difference between prefetching and speculative execution has to do with the quality of data that has been gathered through pre-execution. Throughout the literature in prefetching systems and materialized views there exists the issue of “staleness” – when is pre-fetched data too old to be used? A number of techniques have been suggested regarding how to ensure that the correct version of data is used (Gupta and Mumick 1995), thus avoiding the incorrectness that results from stale data. Still, most of these techniques are useless when a data is updated unexpectedly, even in violation of assumptions that the prefetching has about the freshness of the data. With speculative execution, there is no question of stale data. Pre-execution of information retrieval operators such as Wrapper does not occur until execution begins. Thus, the data cannot be regarded as stale because it is fetched at a time after it was requested.

Related to the staleness issue is the third difference between prefetching and speculative execution: the relevance of gathering data ahead of schedule. In prefetching systems, data is gathered because it is likely to be accessed. The determination of this likelihood is usually done through statistics on past requests (Horvitz 1998). However, there is never a guarantee that pre-fetched data will actually be needed. For example, a materialized view that is updated every day may, for a particular period of time, only be queried after 10 days have passed from the last query. In this sense, the other 9 prefetches could be considered wasteful because their content was never used.

In contrast, it is guaranteed that all correctly predicted values through speculative execution will eventually be used. Since speculation occurs only after execution is initiated it is guaranteed that correct predictions will be needed by downstream plan operators. Arguably, the only case in which data gathered through speculative execution would not be used would be if there existed some other type of conditional operator that re-routed the flow of data based on its content at a given point in time. Even for such cases, an approach such as ours could be modified to speculate only before conditionals or, more interestingly, to extend such an approach to deal with the branch prediction opportunity of conditional dataflow execution.

5.2.4 Speculative execution at the operating system and database level

The upside potential of speculative execution at the software level for this paper was, in part, inspired by (Chang and Gibson 1999) who describe a successful system-level use of speculative execution for informed file prefetching. Their approach automatically modifies existing computer programs with a speculative thread that executes “safe” instructions ahead of their normal schedule in order to generate useful hints to a prefetching file buffer called TIP (Patterson et al. 1995). By executing the safe instructions in advance, their approach provides better hints to TIP about the file access patterns future execution would demand. While there are many differences between the approach of (Chang and Gibson 1999) and our approach as far as the purpose of speculative execution and its actual mechanics, there are also some similarities. For example, both use threads as a vehicle for speculation and both ensure safe execution. In the case of the latter, (Chang and Gibson 1999) designed their system so that the speculative thread added to programs only executed “safe” instructions designed to inform TIP and thus did not execute instructions that affected the external environment in unrecoverable ways.

Prefetching in data-intensive applications was addressed recently by (Hull et al. 2000), in their work on optimizing business logic for e-commerce applications. They characterized application design in terms of “decision flows” – essentially workflow-driven data retrievals. Execution plans for such workflows are essentially dataflow graphs that include synchronization points triggered by workflow states. Speculative execution was used to prefetch relevant data from databases ahead of when it was scheduled. This was essentially a form of control speculation in a dataflow environment that included value-less synchronization dependencies. This research was primarily motivated by the fact that the queries to the databases were often known in advance – it was simply a matter of workflow that determined when/if they were actually necessary. As with many applications of speculative execution, the goal was to improve the ILP of the execution plan and to reduce the latencies caused by database access.

5.2.5 Speculative execution at the hardware level

As discussed earlier, speculative execution is a major topic in computer architecture research. Initially, the motivation was to make use of idle cycles in processor pipelines. The terms “super scalar” and “out-of-order execution” were coined to describe the notion of execution proceeding faster than would be achieved by

linearly executing the instructions under von-Neumann style architecture. Today, speculative execution continues to be a core competency of computer architecture and plays a critical role in optimizing the performance of modern microprocessors.

Compiler research has more recently begun to investigate how speculation can be statically scheduled in generated program code. One focus of interest has been on predicated execution (August et al. 1998), which has been used to reduce code size, improve upon branch prediction, and thus improve execution time. There has also been considerable attention given to the area of thread-level speculation (TLS) (Oplinger et al. 1997; Steffan and Mowry 1998), motivated by similar interests as ours: to use threads as a mechanism for exploiting idle computation resources. Much of the interest by the compiler community in TLS has been generated by research on hardware architectures that have direct support for speculative threads (Roth & Sohi 2000) and by the improvement in scheduling threads even on a single chip, such as is envisioned by SMT research (Tullsen et al. 1995).

Overall, there are understandably many differences between the speculative execution techniques used by processors and compilers for computer programs and the approach described here for information gathering plans. For example, most speculation at the hardware level is about control in the execution of von Neumann style programs; speculative execution for dataflow-style computation has not yet been addressed (Silc and Robic 1999). To a great extent, this is understandable – dataflow computing has traditionally suffered from problems throttling the natural degree of parallelism of a program and was not concerned with trying to increase this level. In contrast to past work on speculative execution at the architecture level, my approach involves speculation during dataflow execution, requires value prediction, and obviously works at a much higher level of execution.

5.3 Value prediction

The contributions of this dissertation in terms of value prediction are (a) the hybridization of caching, classification, and transduction for value prediction, (b) the algorithms for learning two types of transducer, value transducers and hint transducers. Thus, in this section I discuss other techniques for value prediction, at various levels of execution. I also focus specifically on other approaches for learning transducers. However, I start by first considering the broader relationship of value prediction for speculative execution to previous work on speedup learning.

5.3.1 Value prediction as speedup learning

To predict values for speculative execution, I combine machine learning techniques and caching to learn hybrid predictors that are usually more accurate and more space efficient than simply caching alone. The overall goal of my approach to value prediction is to improve the utility of speculative execution. More specifically, better accuracy leads to better speedups.

Thus, to some extent, my approach can be considered a form of *speedup learning*. In speedup learning, the goal is to improve problem solving performance through experience. Past research has focused on a number of areas, including learning “macro operators” for future problem solving (Fikes et al. 1972), learning

heuristics for determining which operators to apply to a given subproblem (Mitchell 1983), and learning control knowledge to aid in choosing what operators to execute next (Minton 1988).

My approach to learning value predictors is similar to much of this past work. For example, the learning of classifiers and hint transducers allows the results of past executions to be leveraged for “new” executions (i.e., previously unseen plan inputs or intermediate data). For example, I described how new “full review” URLs in CarInfo could be accurately predicted based on previously unseen summary review URLs. This kind of function learning is similar to, for example, the application of learned macro-operators to new problems. It should also be noted that strictly caching for value prediction is less related to speedup learning in this sense, because its knowledge cannot be applied to new executions.

The utility problem (Minton 1990) is another interesting point of comparison. In past work on speedup learning, the utility problem describes the case where the matching costs of a concept outweigh its savings when applied. Matching costs generally increase as the number of rules learned increases. While the utility problem is not relevant in my approach with respect to caching⁵ and hint transduction because both have constant matching costs, it can be a factor with respect to classification. For example, as a decision tree grows, the costs to make a prediction may increase (more branches may need to be taken). In turn, this leads to greater speculative overhead and subsequently less applicability of a transformation.

Overall, value prediction for speculative execution can be seen as very similar to, or even a form of speedup learning. While the process of agent plan execution does not involve “problem solving” in the traditional sense, learning can be applied to past executions to improve the performance of future executions.

5.3.2 Other techniques for value prediction

Another related, yet different topic, has to do with how my approach to value prediction differs from other specific techniques proposed in past research. Historically, computer architecture research has largely focused on a type of speculative execution known as branch prediction, which involves predicting *control*, not data. (Hennessey and Patterson 1996) provides a concise summary of the state of the art, describing how instructions are fetched and executed ahead of schedule based on branch prediction results.

There has been recent work on hardware-level value prediction, such as that described by (Gabbay 1996; Lipasti et al. 1996), although the techniques employed are much different than those I have described. For example, both works discuss last value prediction and stride value prediction. These types of predictors are used to determine numeric predictions. For example, stride value prediction involves predicting loop increments. While the types of value prediction done at the hardware level are limited, approaches such as stride prediction are inspiring. I address the potential applicability of stride prediction later, in Chapter 6 in terms future work.

⁵ Assuming caching works by hashing a hint tuple to determine a set of predicted tuples.

At the operating system level, I have earlier described that (Chang and Gibson 1999) proposed an technique which automatically modifies existing computer programs for speculative execution. The technique adds a “speculative thread” to the existing program and allows that thread to execute “safe instructions”, such as those that open files and read data. In this work, the values being predicted in this case required no synthesis: the speculative thread execute the same instructions as the more latent main thread(s), and thus request the same files from the file system.

As I also discussed in the last subsection, there has not been any previous work on value prediction for information gathering systems. (Hull et al. 2000) proposed speculation in a decision flow framework, but one in which only control predictions were necessary. There has also been past work on information gathering with partial results (Hellerstein et al. 1997; Shanmugasundaram et al. 2000), but these systems do not predict data values and instead use approximate values from intermediate aggregate operators in order to obtain approximate final results.

5.3.3 Other approaches to learning transducers

In this subsection, I focus specifically on induction of transducers. As stated earlier, my hybrid approach to value prediction is novel in its design. However, some of the techniques that my approach relies on, such as classification and caching, are already well-understood. Still, much of my approach to value prediction involves learning transducers that can both (a) synthesize predictions and (b) as part of (a), translate the hint string through character level transduction.

Surprisingly, there has been little work on the learning of subsequential transducers. One existing algorithm is OSTIA (Oncina et al. 1993), which is able to induce traditional subsequential transducers capable of, for example, automating translations of decimal to Roman numbers or English word spellings of numbers to their decimal equivalents. For instance, with the proper examples, OSTIA can learn that the Roman “XXII” is equivalent to the Arabic “20”.

My approach differs from OSTIA mainly in that the transducers learned with LEARN-VALUE-TRANSDUCER capture the *general process* of a particular type of string transformation. After learning from only a few examples, the algorithm can achieve a high degree of accuracy on such cases. The algorithm is also well suited to URL prediction, since URLs (and more generally, HTTP GET and POST requests) required to query dependent sources often contain manipulations of structured data extracted from earlier sources (or from plan input). In contrast, while OSTIA can learn more complex types of subsequential transducers, it can require a very large number of examples before it can learn the proper rule (Gildea and Jurafsky 1996).

The transducer learning algorithm suggested by (Hsu and Chang 1999) viewed transduction as a means for information extraction. Our use is similar in that one part of our approach involves extracting dynamic values from hints. However, the type of transducers I have introduced describe go beyond extraction – they transform the source string so that it can be integrated into a predicted value. In doing so, our transduction process is two level: the first level makes use of classification and the second level focuses on the character-level transformations of substrings.

Finally, while the use of classification applies to predicting any type of data value in an information gathering plan, our typical use of transduction is for the prediction of URLs. Other approaches have explored point-based (Zukerman et al. 1999) or path-based (Su et al. 2000) methods of URL prediction, attempting to understand request models based on either time, the order of requests, or the associations between requests. However, unlike our approach, these techniques do not try to understand very general patterns in request content and thus cannot predict previously un-requested URLs

Chapter 6

Conclusion and Future Work

Agent performance can be slow for a number of reasons. For information agents, one of the more frequent reasons involves the aggregation of latencies associated with querying remote data sources. The problem can be especially bad when an agent must query dependent sources – those that are queried based on the answer to an earlier query to a remote source. The sluggish speed of some agents stands in stark contrast to the wealth of resources available on most personal computers today, which contain CPUs that can execute billions of instructions per second.

To address this problem, I have introduced an approach to agent execution that is highly parallel, exceeding the natural dataflow limit of the agent plan. The first element of my approach consists of a streaming, dataflow-style agent plan language and execution system. The language supports the expression of efficient agent plans that engage in traditional information gathering operations, as well as more complex tasks, such as monitoring, integration with local databases, and asynchronous notification. The streaming dataflow nature of the execution model allows a plan to realize the maximum possible degrees of natural horizontal and vertical parallelism possible.

The second part of my approach, speculative plan execution, adds another form of concurrency – speculative parallelism – to plan execution. The process works by leveraging Amdahl's Law: it first detects the most expensive path in a plan and then augments that path with new operators that facilitate speculative execution. The algorithm is applied repeatedly to a plan, until the most expensive path cannot be modified to any greater efficiency. This iterative style of plan refinement gives the algorithm an anytime property. At the same time, the algorithm is also simple, evaluating only the parts of the plan that can possibly improve overall execution time and ignoring those parts that are not relevant. Speculative execution is guaranteed to be safe in that each speculative tuple (and the results born from the operators that consume that tuple) is blocked from leaving the plan or triggering an unrecoverable action, until speculation about that tuple has been confirmed. Speculative execution is also fair: thread prioritization and bandwidth reservation can be used to prevent speculation from subsuming resources needed by normal execution. The resulting speculative execution of a plan yields a degree of concurrency beyond the normal dataflow limit of that plan. Through cascading speculation, resulting speedups can be greater, approaching a degree that is directly proportional to the length of the longest flow in the plan.

Finally, to ensure high average speedups, I have also introduced an approach to value prediction that is both accurate and space efficient. Value predictors are learned by algorithms that first identify the template of a predicted values and then select a computation strategy that fills in the missing parts of those templates as a function of the hint. The strategy, for each dynamic part of the prediction template, may be either caching, classification, or transduction. To produce a prediction based on the template, the value predictors learned can thus require one or more of these techniques. The advantage of classification and transduction is that both methods can respond to new hints, whereas caching cannot. Furthermore, hint transduction allows entirely new predictions to be synthesized from new hints. Both techniques are more space efficient than caching; hint transducers are especially small, since they represent a learned function, not an endlessly growing table of source/target values. In addition, caching remains useful as an alternative to be applied when no learning is possible. It is a simple technique that does well when predicting data it has seen before.

I have demonstrated the validity of my approach through a series of experiments that focused on the streaming dataflow system, speculative execution, and value prediction. Specifically, the experiments have shown that:

- The streaming dataflow language and execution system supports the execution of agent plans not supported by other network query engines and agent executors.
- On simpler, more traditional information integration plans, the system performs just as well or better than a basic network query engine.
- Speculative execution can be applied to a variety of agent plans and result in significant speedups.
- Speculative execution can also be applied to plans generated from queries contained in a well-known database benchmark. When sources from a common schema are assumed to be distributed with varying latencies between them, speculative execution of these plans is faster than traditional execution techniques.
- A value prediction strategy that combines caching, classification, and transduction can make more frequent predictions with greater accuracy than one that relies on only caching. In addition, the resulting value predictors are more space efficient.

6.1 Limitations

While my approach to speculative plan execution can significantly improve the performance of information agent plans, it does have some limitations and disadvantages:

1. *Execution that can be slower than normal.* While this is not the average case, this can occur when average latency of a source suddenly improves. For example, if a source to be speculated about has an average latency of 1s, and on average it is necessary to speculate about 100 tuples per hint received (e.g., a search criteria hint that leads to 100 results), an overhead of less than 10ms per predicted tuple would make this a profitable activity. However, if the

source happened to be faster than 1s on one particular occasion, then the overhead per tuple multiplied by the number of tuples predicted would lead to an execution time worse than having not speculated. For example, if the source latency improved to 500ms and the overhead was 9ms per tuple, predicting 100 tuples would cost 900ms of overhead, 400ms more than the cost of the source without speculation. One way to remedy this limitation would be to periodically re-evaluate the speculative transformation of the original plan and give greater weight to recent latency calculations in determining future transformation.

2. *Execution that may not be fair.* While thread priorities and bandwidth reservation are two well-known ways to partition groups of activities at disparate levels of priority, not all resources can be partitioned so easily. For example, although we might partition connections to a local database into a speculative pool and a normal pool, the database itself does not distinguish between the two types of connections and its own processing will be equally affected by both. Thus, speculative database query processing will compete against database query processing initiated through normal execution. To some extent, this is also true when that resource is a Web site – however, most Web application technology is designed to support fairly high levels of concurrent connections.
3. *Computation of speculative overhead may be inaccurate.* The current design for the system assumes that an administrator choose a reasonable value for speculative overhead. Through experimentation, I have found that this can range from 5ms to 15ms per tuple for the information agent plans and TPC-H queries tested. However, the process of selecting an overhead index is likely more complicated than simply choosing a reasonable value and may require a detailed calculation. Overhead is not merely the sum of the time it takes for Speculate and Confirm to execute their individual instructions on a given tuple. In fact, the number of speculative threads likely has some effect: even through they are prioritized lower, more threads mean more context switching for the CPU. Furthermore, more speculation translates into greater memory and bandwidth crowding – all of this has an impact on the time it takes to process a given tuple. In my discussion of future work, below, I suggest one possible solution that might help in a more precise determination of speculative overhead.

6.2 Future Work

There are several opportunities for future work on speculative plan execution. In this section, I focus on three promising avenues: learning to calculate speculative overhead, classifier compression, and SMT benchmarking.

6.2.1 Learning to choose good values for speculative overhead

As described above, one limitation of the approach I have specified is that a value for speculative overhead must be chosen manually. This is unfortunate for two reasons: (a) it necessitates human intervention in order for speculative execution to be

integrated into an existing agent execution system and (b) it may be hard to manually determine this value because overhead is likely not a constant.

One way to address this problem is to use machine learning techniques to leverage empirical results from past plan executions to identify good choices for overhead in future executions. A relatively easy way to do this would be to determine the true overhead from past speculative executions and associate this with features of the plan. For example, speculative execution of a plan with four speculate operators might be found to have a true overhead of 15ms/tuple whereas speculative execution of a plan with three speculate operators might be found to have an overhead of 10ms/tuple. A variety of machine learning algorithms can be used to learn what overhead to use when considering transformation of a new plan for speculative execution.

6.2.2 Classifier compression / probabilistic classification

One of the claims I make in this thesis is that, for value prediction, classifiers are more space efficient than caches. The difference is most noticeable when hints contain continuous values because classifiers create rules based on a discretization of these values (thus, storing every single value is not necessary). Nevertheless, classifiers are less space efficient than hint transducers, because they do store values and do continue growing as more examples are seen. For hint domains that have a large number of nominal values, classifiers can be quite large. This raises a scalability issue.

To address this problem, one approach is to investigate methods of classifier compression. (Quinlan and Rivest 1989) proposed the idea of minimum encoding as one possible technique. In addition to these methods, an interesting approach would be to explore the integration of Bloom filters into decision trees. Bloom filters are a mechanism that tests set membership: a candidate key is submitted to series of hash functions and then a True or False value is returned as to whether than candidate is in the set. Bloom filters have been successfully used in the Web caching community (Fan et al. 2000) for efficient object lookup.

My intuition is that Bloom filters could be used to test whether a particular feature value matches a choice (branch) from a decision node. Thus, a decision tree would become a tree of filters that would lead to more compressed rules. In short, the overall footprint of a classifier could be substantially reduced – at the cost of occasional false positives. However, since the speculative execution approach I have described handles mispredictions, this is not a major issue. Integrating Bloom filters into decision trees for probabilistic classification may also be an interesting avenue for machine learning research in general.

6.2.3 SMT Benchmarking

One of the claims of my thesis is that speculative parallelism leads to increased usage of local resources, such as CPU. Assigning more threads to speculative work fits neatly into a current trend in microprocessor architecture: the trading of instruction-level parallelism (ILP) for thread-level parallelism (TLP). In recent years, work on simultaneous multithreading (SMT) has spawned a new push towards

the reorganization of processor architectures that trade longer instruction pipelines for multiple instruction pipelines. For example, the most recent CPUs mass-marketed by Intel Corporation contain “hyperthreading” technology, an implementation of SMT. The beneficiary of SMT implementations like hyperthreading is software that performs multiple tasks in parallel using multiple threads.

My intuition is that speculative execution of information agent plans will be more effective on SMT architectures than on non-SMT architectures. This is because the former encourages better TLP. Because of this, speculative overhead is likely to be less on SMT platforms (speculative threads are not taking away all computational resources during their time slice). Past studies of database (Redstone et al. 2000) and Web servers (Lo et al. 1998), application-level software that often involves concurrent I/O-bound threads, have also confirmed that SMT is a more thread-friendly architecture and leads to better performance as well as better overall CPU utilization. I believe that speculative execution of information agent plans will yield similar results and that validating this hypothesis would be a useful contribution.

6.2.4 Integrating additional value prediction techniques

While the hybrid predictors described in Chapter 4 are an evolution beyond strictly caching, I believe that additional techniques could be integrated to improve the utility of speculative execution. Given that value prediction is occurring at a high level of execution, with a significant amount of local resources, there may be more machine learning techniques that can be applied. To some extent, it may be useful to spend more time considering what can be leveraged from past research, for example from the computer architecture literature.

One promising example is stride value prediction. In computer architecture research, stride predictors are generally used to model loop increments. For example, a stride predictor could learn the stride of a loop which started at 0 and incremented by 2 each time (i.e., 0, 2, 4, 6, etc.). The idea of learning this type of sequence is a simple type of function learning that also has relevance for Web agents.

Consider gathering result pages from any source that outputs a list over a series of pages, such as a search engine. As I described earlier in this dissertation, this is a classic example where interleaved navigation and gathering are necessary. Each page must be gathered and processed, including the Next Page link (if any), before the subsequent results page can be gathered.

However, in many cases, the URLs of each result page contain the current results page number or the index of the next set of results. For example, the second page of search engine results in a Google query has a URL that is always similar to *http://www.google.com/search?...&start=25*, while the third page of results has a URL of *http://www.google.com/search?...&start=50* and so on. This was also the case in the Homeseekers example discussed in Chapter 2. If some kind of stride value predictor was learned for this case, it may be possible to learn to issue requests for the second and third page of results concurrently with gathering the first page.

Of course, care must be taken not to generate too many predictions at once (i.e., issue requests for hundreds of result pages before we know that they exist for a particular search). Nevertheless, this appears at first glance to be a promising avenue of future work. More generally, the continued integration of addition value prediction techniques, such as stride value prediction, can enhance the utility of speculative execution and thus increase the average speedups for many types of Web agent plans.

Bibliography

- Abiteboul, Serge, Richard Hull and Victor Vianu. 1995. *Foundations of databases*, Addison-Wesley Publishing.
- Ambite, Jose Luis, Greg Barish, Craig A. Knoblock, Maria Muslea, Jean Oh and Steven Minton. 2002. Getting from here to there: interactive planning and agent execution for optimizing travel. *Proceedings of the 14th Innovative Applications of Artificial Intelligence (IAAI-2002)*. Edmonton, Alberta, Canada.
- Arens, Yigal, Craig A. Knoblock and Wei-Min Shen. 1996. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems - Special Issue on Intelligent Information Integration* 6(2/3): 99-130.
- Arvind, K.P. Gostelow and W. Plouffe. 1978. The ID report: An asynchronous programming language and computing machine. Technical Report 114, University of California at Irvine.
- Arvind and Rishiyur S. Nikhil(90). Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers* 39(3): 300-318.
- Ashish, Naveen. 2000. Optimizing information mediators by selectively materializing data. PhD thesis, Department of Computer Science, University of Southern California.
- August, David I., Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran, et al. 1998. Integrated predicated and speculative execution in the IMPACT EPIC architecture. *Proceedings of the 25th Annual International Symposium on Computer architecture*. Silver Spring, MD, IEEE Computer Society Press.
- Avnur, Ron and Joseph M. Hellerstein. 2000. Eddies: continuously adaptive query processing. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Dallas, TX: 261-272.
- Barish, Greg, Yi-Shin Chen, Craig A. Knoblock, Steven Minton and Cyrus Shahabi. 2000. The TheaterLoc virtual application. *Proceedings of the 12th Innovative Applications of Artificial Intelligence (IAAI-2000)*. Austin, TX.
- Barish, Greg and Craig A. Knoblock. 2002. An efficient and expressive language for information gathering on the web. *Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS 2002) workshop: Is there life after operator sequencing? - Exploring real world planning*. Toulouse, France.
- Barish, Greg and Craig A. Knoblock. 2003. Learning value predictors for the speculative execution of information gathering plans. *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*. Acapulco, Mexico.
- Barish, Greg and Craig A. Knoblock. 2002. Speculative execution for information gathering plans. *Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS 2002)*. Toulouse, France.

- Boag, Scott, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie and Jerome Simeon. 2002. XQuery 1.0: An XML query language. Available from <http://www.w3c.org>
- Chalupsky, Hans, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David Pynadath, Thomas A. Russ and Milind Tambe. 2001. Electric elves: applying agent technology to support human organizations. *Proceedings of the 13th Innovative Applications of Artificial Intelligence (IAAI-2001)*. Seattle, WA.
- Chandrasekaran, Sirish, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, et al. 2003. TelegraphCQ: continuous dataflow processing for an uncertain world. *Proceedings of the First Biennial Conference on Innovative Data Systems Research*. Monterey, CA.
- Chang, Fay W. and Garth A. Gibson. 1999. Automatic I/O Hint Generation Through Speculative Execution. *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. New Orleans, LA: 1-14.
- Chaudhuri, Surajit, Ravi Krishnamurthy, Spyros Potamianos and Kyuseak Shim. 1995. Optimizing queries with materialized views. *Proceedings of the 11th International Conference on Data Engineering*. Los Alamitos, CA, IEEE Computer Society Press: 190-200.
- Chawathe, Sudarshan, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman and Jennifer Widom. 1994. The Tsimmis project: integration of heterogenous information sources. *Proceedings of the 16th Meeting of the Information Processing Society of Japan*. Tokyo, Japan: 7-18.
- Chen, Jianjun, David J. Dewitt, Feng Tian and Yuan Wang. 2000. NiagaraCQ: a scalable continuous query system for internet databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Dallas, TX: 379-390.
- Davison, Brian D. and Haym Hirsh. 1998. Probabilistic online action prediction. *Proceedings of the AAI Spring Symposium on Intelligent Environments*.
- Dennis, Jack B. 1974. First version of a data flow procedure language. *Lecture Notes in Computer Science* 19: 362-376.
- Dewitt, David and Jim Gray. 1992. Parallel database systems: the future of high performance database systems. *Communications of the ACM* 35(6): 85-98.
- Evrpidou, Paraskevas and Jean-Luc Gaudiot. 1991. Input/output operations for hybrid data-flow/control-flow systems. *The Fifth International Parallel Processing Symposium*. Anaheim, California: 318-323.
- Fan, Li, Pei Cao, Jussara Almeida and Andrei Z. Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE-ACM Transactions on Networking* 8(3): 281-293.
- Feo, John T. and David C. Cann. 1990. A report on the SISAL language project. *Journal of Parallel and Distributed Computing* 10: 349-366.
- Fikes, Richard E., Peter E. Hart and Nils J. Nilsson. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3(4): 251-288.

- Firby, R. James. 1994. Task networks for controlling continuous processes. *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*. Chicago, IL: 49-54.
- Friedman, Marc, Alon Y. Levy and Todd D. Millstein. 1999. Navigational plans for data integration. *Proceedings of the 16th National Conference on Artificial Intelligence*. Orlando, FL: 67-73.
- Gabbay, Freddy. 1996. Speculative execution based on value prediction. Technical Report #1080, Electrical Engineering Department, Technion-Israel Institute of Technology.
- Gao, G. R. 1993. An efficient hybrid dataflow architecture model. *International Journal of Parallel and Distributed Computing* 19(4): 293-307.
- Genesereth, Michael R., Arthur M. Keller and Oliver M. Duschka. 1997. Infomaster: an information integration system. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Tuscon, AZ: 539-542.
- Gildea, Daniel and Daniel Jurafsky 1996. Learning bias and phonological-rule induction. *Computational Linguistics* 22(4): 497-530.
- Greenwald, Lloyd and Thomas Dean. 1994. Solving time-critical decision-making problems with predictable computational demands. *Proceedings of the Second International Conference on AI Planning Systems*. Chicago, IL: 25-30.
- Gupta, Ashish and Inderpal Singh Mumick 1995. Maintenance of materialized views: problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering: Special Issue on Materialized Views and Data Warehousing* 18(2): 3-18.
- Gurd, J. R. and D. F. Snelling. 1992. Manchester data-flow: a progress report. *Proceedings of the Sixth International Conference on Supercomputing*. Washington, D.C., United States, ACM Press: 216-225.
- Hellerstein, Joseph M., Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman and Mehul A. Shah. 2000. Adaptive query processing: technology in evolution. *IEEE Data Engineering Bulletin* 23(2): 7-18.
- Hellerstein, Joseph M., Peter J. Haas and Helen J. Wang. 1997. Online aggregation. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. Tuscon, AZ: 171-182.
- Hennessey, John and David Patterson. 1996. *Computer Architecture: A Quantitative Approach, Second Edition*, Morgan Kaufmann Publishers Inc.
- Hirschberg, Daniel S. 1975. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* 18(6): 341-343.
- Hoare, C.A.R. 1978. Communicating sequential processes. *Communications of the ACM* 21(8): 666-677.
- Horvitz, Eric. 1998. Continual computation policies for utility-directed prefetching. *Proceedings of the Seventh ACM Conference on Information and Knowledge Management*: 175-184.
- Horvitz, Eric. 2001. Principles and applications of continual computation. *Artificial Intelligence* 126(1-2): 159-196.

- Hsu, Chu-Nan and Chien-Chi Chang. 1999. Finite-state transducers for semi-structured text mining. *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI) Workshop on Text Mining: Foundations, Techniques, and Applications*.
- Hull, Richard, Francois Lirbat, Bharat Kumar, Gang Zhou, Guozhu Dong and Jianwen Su. 2000. Optimization techniques for data-intensive decision flows. *Proceedings of the 16th International Conference on Data Engineering*. San Diego, CA: 281-292.
- Iannucci, R. A. 1988. Toward a dataflow/von Neumann hybrid architecture. *The 15th Annual International Symposium on Computer Architecture*. Honolulu, Hawaii, IEEE Computer Society Press: 131-140.
- Ives, Zachary G., Daniela Florescu, Marc Friedman, Alon Levy and Daniel S. Weld. 1999. An adaptive query execution system for data integration. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Philadelphia, PA: 299-310.
- Ives, Zachary G., Alon Y. Halevy and Daniel S. Weld. 2002. An XML query engine for network-bound data. *VLDB Journal* 11(4): 380-402.
- Ives, Zachary G., Alon Y. Levy, Daniel S. Weld, Daniela Florescu and Marc Friedman. 2000. Adaptive query processing for internet applications. *IEEE Data Engineering Bulletin* 23(2): 19-26.
- Jiang, Zhimei and Leonard Kleinrock 1998. An adaptive network prefetch scheme. *IEEE Journal on Selected Areas in Communications* 16(3): 358-368.
- Jones, Simon L. Peyton and John Hughes. 1998. A report on the programming language Haskell, a non-strict purely functional language, Technical report DCS/RR-1106, Computer Science Department, Yale University.
- Kahn, Gilles 1974. The semantics of a simple language for parallel programming. *Information Processing Letters* 74: 471-475.
- Karp, Richard. M. and R. E. Miller 1955. Properties of a model for parallel computations: determinacy, termination, queuing. *SIAM Journal on Applied Mathematics* 14: 1390-1411.
- Knoblock, Craig A. 2003. Deploying information agents on the web. *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. Acapulco, Mexico.
- Knoblock, Craig A., Kristina Lerman, Steven Minton and Ion Muslea. 2002. Accurately and reliably extracting data from the web: a machine learning approach. *IEEE Data Engineering Bulletin* 23(4): 33-41.
- Knoblock, Craig A., Steven Minton, Jose Luis Ambite, Naveen Ashish, Ion Muslea, Andrew Philpot and Sheila Tejada. 2001. The Ariadne approach to web-based information integration. *International Journal of Cooperative Information Systems* 10(1-2): 145-169.
- Kushmerick, Nicholas. 2000. Wrapper induction: efficiency and expressiveness. *Artificial Intelligence* 118(1-2): 15-68.
- Lee, Edward A. 2002. Embedded software. In M. Zelkowitz (ed.), *Advances in Computers* 56, Academic Press, London.

- Lee, Edward Ashford and David G. Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers* 36(1): 24-35.
- Levy, Alon Y., Alberto O. Mendelzon, Yehoshua Sagiv and Divesh Srivastava. 1995. Answering queries using views. *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. San Jose, Calif.: 95-104.
- Levy, Alon Y., Anand Rajaraman and Joann J. Ordille. 1996. Querying heterogeneous information sources using source descriptions. *Proceedings of the 22nd International Conference on Very Large Databases*. Bombay, India: 251-262.
- Lieberman, Henry. 1995. Letizia: an agent that assists web browsing. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Chris S. Mellish. Montreal, Quebec, Canada, Morgan Kaufmann Inc: 924-929.
- Lipasti, Mikko H., Christopher B. Wilkerson and John P. Shen. 1996. Value locality and load value prediction. *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA: 138-147.
- Lo, Jack L., Luiz Andr Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy and Sujay S. Parekh. 1998. An analysis of database workload performance on simultaneous multithreaded processors. *Proceedings of the 25th Annual International Symposium on Computer Architecture*. Barcelona, Spain, IEEE Press: 39-50.
- McDermott, Drew. 1991. A reactive plan language. Technical Report CSD-RR-864, Computer Science Department, Yale University.
- McGraw, James R. 1982. The VAL language: description and analysis. *ACM Transactions on Programming Languages and Systems* 4(1): 44-82.
- Minton, Steven. 1988. *Learning search control knowledge*. Boston, MA, Kluwer Academic Publishers.
- Minton, Steven. 1990. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence* 42: 363-392.
- Mohri, Mehryar. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics* 23(2): 269-311.
- Moore, Gordon. 2003. Speech at the International Solid-States Circuits Conference. San Francisco, California.
- Muslea, Ion. 2002. Active learning with multiple views. PhD thesis, Department of Computer Science, University of Southern California.
- Myers, Karen L. 1996. A procedural knowledge approach to task-level control. *Proceedings of the Third International Conference on AI Planning and Scheduling*. Edinburgh, UK: 158-165.
- Naughton, Jeffrey F., David J. Dewitt, David Maier, Ashraf Aboulmaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, et al. 2001. The niagara internet query system. *IEEE Data Engineering Bulletin* 24(2): 27-33.

- Nikhil, Rishiyur and Arvind. 2001. *Implicit parallel programming in pH*, Morgan Kaufmann Publishers Inc.
- Oncina, Jose, Pedro Garcia and Enrique Vidal 1993. Learning subsequential transducers for pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15(5): 448-458.
- Oplinger, Jeffrey, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam and Kunle Olukotun. 1997. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Computer Systems Laboratory, Stanford University.
- Padmanabhan, Venkata N. and Jeffrey C. Mogul. 1996. Using predictive prefetching to improve world-wide web latency. *Proceedings of the ACM SIGCOMM 1996 Conference*. Stanford, CA: 25-35.
- Papadopoulos, Gregory M. and Kenneth R. Traub. 1991. Multithreading: a revisionist view of dataflow architectures. *Proceedings of the 18th International Symposium on Computer Architecture*. New York, NY: 342-351.
- Patterson, R. Hugo, Garth A. Gibson, Eka Ginting, Daniel Stodolsky and Jim Zelenka. 1995. Informed prefetching and caching. In Hai Jin, Toni Cortes and Rajkumar Buyya (ed.), *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. New York, NY, IEEE Computer Society Press and Wiley: 224-244.
- Quinlan, J.R. 1986. Induction of decision trees. *Machine Learning* 1(1): 81-106.
- Quinlan, J.R. and Ronald L. Rivest. 1989. Inferring decision trees using the minimum description length principle. *Information and Computation* 80: 227-248.
- Raman, Vijayshankar. 2002. Personal communication.
- Raman, Vijayshankar and Joseph M. Hellerstein. 2002. Partial results for online query processing. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Madison, Wisconsin, ACM Press: 275-286.
- Redstone, Joshua A., Susan J. Eggers and Henry M. Levy. 2000. An analysis of operating system behavior on a simultaneous multithreaded architecture. *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, Massachusetts, ACM Press: 245-256.
- Schek, H.J. and M.H. Scholl. 1986. The relational model with relation-valued attributes. *Information Systems* 11(2): 137-147.
- Shanmugasundaram, Jayavel, Kristin Tufte, David J. Dewitt, Jeffrey F. Naughton and David Maier. 2000. Architecting a network query engine for producing partial results. *Proceedings of the ACM SIGMOD Third International Workshop on Web and Databases*. Dallas, TX: 17-22.
- Silc, Juirj and Borut Robic. 1999. *Processor architecture: from dataflow to superscalar and beyond*, Springer-Verlag Publishers.
- Steffan, J. Gregory and Todd C. Mowry. 1998. The potential for using thread-level data speculation to facilitate automatic parallelization. *Proceedings of the Fourth International Symposium on High Performance Computer Architecture*: 2-13.

- Su, Zhong, Qiang Yang, Ye Lu and Hong-Jiang Zhang. 2000. WhatNext: a prediction system for web request using n-gram sequence models. *First International Conference on Web Information Systems Engineering*: 214-221.
- Thomas, Donald E. and Philip R. Moorby. 1998. *The Verilog hardware description language (4th edition)*, Kluwer Academic Publishers.
- Thompson, Craig, Tom Bannon, Paul Pazandak and Venu Vasudevan. 1999. Agents for the masses. *Third International Conference on Autonomous Agents, Workshop on Agent-based High-performance Computing: Problem Solving Applications and Practical Deployment*. Seattle, WA.
- Tullsen, Dean M., Susan Eggers and Henry M. Levy. 1995. Simultaneous multithreading: maximizing on-chip parallelism. *Proceedings of the 22nd Annual ACM International Symposium on Computer Architecture*. Santa Magherita Ligure, Italy: 392-403.
- Wiederhold, Gio. 1996. Intelligent integration of information. *Journal of Intelligent Information Systems* 6(2): 281-291.
- Williamson, Mike, Keith Decker and Katia Sycara. 1996. Unified information and control flow in hierarchical task networks. *Theories of Action, Planning, and Robot Control: Bridging the Gap: Proceedings of the 1996 AAI Workshop*. Menlo Park, California, AAAI Press: 142-150.
- Wilschut, Annita N. and Peter M. G. Apers. 1993. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases* 1(1): 103-128.
- Zhang, Lixia, Stephen Deering and Deborah Estrin. 1993. RSVP: a new resource ReSerVation protocol. *IEEE Network* 7(5): 8-18.
- Zukerman, Ingrid, David W. Albrecht and Ann E. Nicholson. 1999. Predicting user's requests on the WWW. *Proceedings of the Seventh International Conference on User Modeling*. Banff, Canada, Springer-Verlag: 275-284.