Learning Efficient Value Predictors for Speculative Plan Execution*

Greg Barish and Craig A. Knoblock
University of Southern California / Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292
{barish, knoblock}@isi.edu

Abstract

Speculative plan execution can be used to significantly improve the performance of information gathering plans. However, its impact is closely tied to the ability to predict data values at runtime. While caching can be used to issue future predictions, such an approach often scales poorly with large data sources and is unable to make intelligent predictions about novel hints, even when there is an obvious relationship between the hint and the predicted value. In this paper, we describe how learning decision trees and transducers can lead to a more efficient value prediction system as well as one capable of making intelligent predictions about new hints. Our initial results validate these claims in the context of the speculative execution of one common type of information gathering plan.

Introduction

Improving the performance of network-bound information gathering plans remains an ongoing research challenge. Recent systems that execute such plans (Ives et al. 1999; Hellerstein et al. 2000; Naughton et al. 2001) employ adaptive query processing techniques to improve execution efficiency. Adaptive query processing is unique to network-bound information gathering; it addresses the unpredictability of source response time and the potentially sizeable results returned by responding to runtime events or by dynamically re-ordering tuples or operators.

Speculative plan execution (Barish and Knoblock 2002) is a new technique that can be used to dramatically improve the efficiency of network-bound information gathering plans. The idea involves using data seen early in plan execution as a basis for issuing predictions about data needed during later parts of execution. This allows sequential data dependency chains to be broken and parallelized, leading to better average performance.

To maximize the impact of speculative execution on plan performance, a good value prediction strategy is required. The basic problem involves being able to use some hint h as the basis for issuing a predicted value v. Of course, a few simple strategies can be applied. Caching is an obvious choice; we can note that particular hint h_x corresponds to a particular value v_y so that future receipt of h_x can lead to a prediction of v_y.

* The research reported here was supported in part by the Air Force Office of Scientific Research under Grant Number F49620-01-1-0053, and in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152. 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.

However, there are two problems with caching. One is space-efficiency: prediction requires the storage of prior hint/value mappings. If the values being predicted are associated with large data sources, the resulting lookup table can become prohibitively large. The second problem is accuracy: caching only allows prediction of previously seen values. Thus, predictions cannot be issued for new hints, even when the former is a simple function of the latter or based on a subset of hint attributes.

In this paper, we propose a new approach that uses machine learning techniques to craft a prediction system that is both efficient and accurate. Our approach uses two learning mechanisms for issuing intelligent predictions: (a) decision trees, which *classify* hints based on their most informative attributes and (b) subsequential transducers, which *translate* hints into predicted values. Specifically, this paper contributes the following:

- An approach using decision trees as a means for classifying a hint into a prediction.
- An approach using subsequential transducers as a means for translating a hint into a prediction.
- An algorithm that unifies the incremental learning of both structures, based on the type of relationship observed between hint and predicted value.
- Initial results of the application of this algorithm to one common type of information gathering plan.

It is important to note that this work focuses on the challenge of *what* to speculate. Although we review it here briefly, prior work (Barish and Knoblock 2002) has addressed the details of one method for *how* to speculate.

The rest of this paper is organized as follows. The next section reviews information gathering and speculative execution. In Section 3, we describe how decision trees and transducers can be used to build efficient and intelligent predictors. Section 4 presents an algorithm that unifies the learning of both types of predictors. Finally, we present the initial results of applying our approach to the execution of a common type of information gathering plan.

Preliminaries

Information gathering plans retrieve, combine, and manipulate data located in remote sources. Such plans consist of a partially-ordered graph of operators $O_1..O_n$ connected in producer/consumer fashion. Each operator O_i consumes a set of inputs $\alpha_1..\alpha_p$, fetches data or performs a computation based on that input, and produces one or more outputs $\beta_1..\beta_q$. The types of operators used in information gathering plans varies, but most either retrieve or perform computations on data.

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 as:

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

Example Plan. To illustrate, consider the plan executed by an information agent called **RepInfo**. This plan, shown in Figure 1, returns information about U.S. congressional officials. Given any U.S. postal address, RepInfo gathers the names, funding charts, and recent news related to U.S. federal officials (members of the Senate or House of Representatives) for that address. RepInfo retrieves this information via the following web data sources:

- *Vote-Smart*, to identify the officials for an address.
- *OpenSecrets*, for funding data about each official.
- *Yahoo News*, for recent news about each official.

In Figure 1, **Wrapper** operators retrieve data from Web sources, a **Select** operator filters federal officials from other types of officials, and a **Join** operator combines the funding and news data in order to return the entire result as a single output. Wrapper-style operators are very common in web information gathering plans. They work by extracting semi-structured data from a set of remote web pages into a relation of structured data.

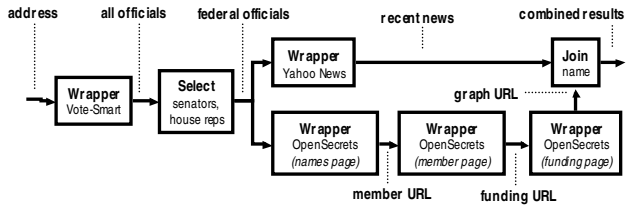


Figure 1: The RepInfo plan

At the start of execution, an input postal address is used to query the Vote-Smart source, returning the set of federal officials for that location. A subsequent Select operator filters Senate and House officials. This subset is then used to query Yahoo News and OpenSecrets. Note that the latter requires additional retrieval steps in order to navigate to the page containing the funding data. The results of both are then joined together, providing the result shown in Figure 2.

The plan shown in Figure 1 is one common type of information gathering plan. Similar plans that combine data from two or more distinct sources can be found throughout prior research (Friedman and Weld 1997; Ives et al. 1999).

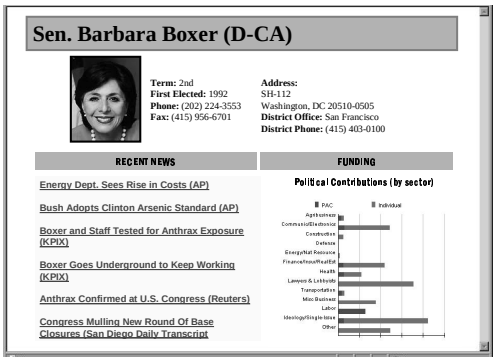


Figure 2: Results from executing RepInfo

Speculative Plan Execution

Normally, execution of an information gathering plan requires that each of its independent data flows be executed serially. For example, in Figure 1, there are two flows. The first one, f_1, includes the Vote-Smart, Select, Yahoo News, and Join operators. The second one, f_2, includes the Vote-Smart, Select, OpenSecrets, and Join operators. Per Amdahl's Law, the time required to execute the plan will be no less than the slowest of these two. Table 1 shows the average execution times for f_1 and f_2.

Flow	Path	Time (ms)
f_1	VoteSmart→Select→Yahoo→Join	3280
f_2	VoteSmart→Select→OpenSecrets→Join	6500

Table 1: Execution time of RepInfo data flows

Thus, execution time is MAX(3280, 6500) = 6500ms.

Speculative execution overcomes Amdahl's Law by parallelizing a sequence of dependent operators through *predictions* about those dependencies. Predictions are made based on *hints*, which occur earlier during the execution of a flow. For example, in Figure 1, we could use the input to Vote-Smart (the postal address) as a hint for predicting the likely set of representatives that will be communicated to the OpenSecrets and Yahoo operators. Thus, the Vote-Smart operator will be executing in parallel with the Open Secrets and Yahoo wrappers, generally leading to faster execution. Of course, we need to ensure correctness by "guarding" against the release of speculative results until earlier predictions are confirmed.

Figure 3 shows the plan in Figure 1 modified for speculative execution. In the figure, two types operators have been added. The first is **Speculate** (labeled SPEC), which receives hints and answers, makes predictions, and issues confirmations. These inputs and outputs have been labeled for the first Speculate operator in the figure. The second new operator is **SpecGuard** (labeled GUARD), which prevents speculative results from exiting the plan until the predictions that led to those results has been confirmed. As the figure shows, speculation can be *cascading* (i.e., based on earlier speculation). Correctness is enforced by SpecGuard – only confirmed speculation can exit the plan (and affect the external world).

In summary, Figure 3 shows three instances of speculation: (a) the prediction of federal representatives based on street address, (b) the prediction of OpenSecrets member page URL based on the name of the official, and (c) the prediction of the OpenSecrets funding page URL based on the member page URL. In (Barish and Knoblock 2002), we demonstrate how executing this plan can result in an optimistic speedup of 3.65.

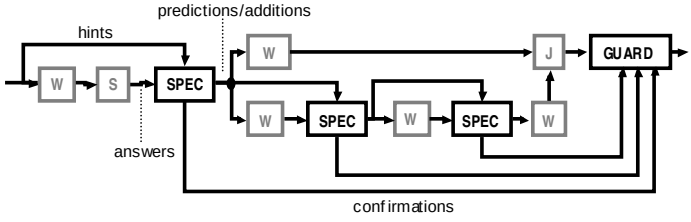


Figure 3: RepInfo modified for speculative execution

Data Value Prediction

While speculative plan execution can lead to substantial speedups, its effectiveness is directly related to the accuracy of the predictions issued. Good predictions improve the average execution time; poor predictions have the opposite effect. Although prediction based on caching prior hint/answer tuples is possible, this method is space-inefficient and unable to make predictions for new hints.

To build a more efficient and intelligent predictor, we show how machine learning techniques can be applied to the problem. In this section, we describe how two such techniques, decision trees and transducers, can be employed. Both are more efficient than caching and both allow predictions to be made about novel hints.

Prediction Using Decision Trees

Decision trees are an effective tool for classifying data. By calculating the information gain from a series of multi-attribute examples, we can use decision trees to identify and rank the attributes most useful when making a prediction. This last feature enables decision trees to issue reasonable predictions about novel hints.

For example, for the modified plan in Figure 3, we can use a subset of address attributes to predict the federal officials that will be fetched. Specifically, we are interested in predicting the two Senate members and single House of Representatives member for that address.

Intuitively, we know that a given senator can be associated with millions of addresses and a given representative can be associated with thousands. Decision trees can be used to help us identify that the state attribute of the address hint is the most informative for senators while city and zip code are the most informative hints when predicting representatives.

As a detailed example, let us consider building decision trees that classify House of Representatives members based on postal addresses. Table 2 shows 10 sample addresses and their corresponding representatives. Using the ID3-based C5.0 classifier based on (Quinlan 1986), the following decision list is constructed:

```
city = Culver City: Jane Harman (0)
city = Marina del Rey: Jane Harman (2)
city = Venice: Jane Harman (3)
city = Santa Monica: Henry Waxman (1)
city = Los Angeles:
:...zip <= 90064: Henry Waxman (1)
    zip > 90064: Diane Watson (2)
```

We can then use this list to issue predictions for recurring and new hints. For example, we can predict Jane Harman for the recurring address 4676 Admiralty Way, Marina del Rey, CA as well as for the new address 4680 Admiralty Way, Marina del Rey, CA.

Note that predictions will not always be correct – for example, the above tree incorrectly predicts Jane Harman (not Diane E. Watson) as the representative for 4065 Glencoe Avenue, Marina del Rey, CA, 90292. However, since these are *predictions* for execution and not final answers, such inaccuracy is not fatal – a speculative execution system that ensures correctness can recover from such errors.

Street	City	State	Zip	Representative
14044 Panay Way	Marina del Rey	CA	90292	Jane Harman
4676 Admiralty Way	Marina del Rey	CA	90292	Jane Harman
101 Washington Blvd	Venice	CA	90292	Jane Harman
1301 Main St	Venice	CA	90291	Jane Harman
1906 Lincoln Blvd	Venice	CA	90291	Jane Harman
2107 Lincoln Blvd	Santa Monica	CA	90405	Henry Waxman
2222 S Centinela Ave	Los Angeles	CA	90064	Henry Waxman
4065 Glencoe Ave	Marina del Rey	CA	90292	Diane Watson
3970 Berryman Ave	Los Angeles	CA	90066	Diane Watson
11461 Washington Blvd	Los Angeles	CA	90066	Diane Watson

Table 2: Sample constituent postal addresses

In summary, decision trees are effective because they enable us to issue intelligent predictions about recurring and new hints, accomplishing the latter by learning which attributes of the hint are the most informative and ranking them accordingly. Note that, in the worst case where the input contains only one attribute or when each hint maps to a unique value, decision-tree based prediction reduces to simple caching. Thus, decision trees represent a means for prediction that trades infrequent inaccuracies for space efficiency and the ability to predict from novel hints.

Prediction Using Finite State Devices

While predictions based on decision trees allow us to predict previously seen data based on new hints, they do not enable us to make *novel predictions* – that is, they do not enable us to predict values that we have never otherwise seen occur during execution. However, if we model the problem of prediction as one of translation, we can issue novel predictions.

Natural language processing research often relies on finite state devices to accomplish translation (Knight and Al-Onaizan 1998). One finite state device of interest is the *subsequential transducer*. This device is a state transition graph that consists of nodes (states) connected by arcs (transitions) labeled *x*/*y* where *x* is the input string and *y* is the output string. The special character ε indicates that no output is generated, the character *?* refers to any symbol other than those appearing on any remaining arcs for that node, and the # character corresponds to the end of the input. Arcs labeled with only output are also possible.

To understand why transducers are useful and how they can be applied, consider again the RepInfo plan in Figure 1. As the figure shows, three separate wrapper calls are required to obtain the funding graph we want. First, a representative name is used to locate the OpenSecrets *member URL*. This result allows us to locate the *funding URL*, the web page that contains the graph we want. Finally, we need to fetch that page in order to identify the URL of the graph image. This costly interleaving of navigation and retrieval is commonplace for information agents, particularly those that query web sources.

However, in looking at the data retrieved, we see that it is surprisingly predictable. As Table 3 shows, although the representative name does not correspond to the member URL, the member URL *does* correspond to the funding URL. Specifically, the latter can be generated from the former by replacing *summary.asp* with *sector.asp* (more precisely, replacing "*ummary*" with "*ector*"). Thus, we could build a transducer that translates a member URL

into a funding URL, enabling us to make novel predictions (i.e., predict funding URLs for new member URLs).

One such transducer that accomplishes this is shown in Figure 4. Following states 0 through 5, we can see how a member URL is translated into a funding URL. For example, the "http://www.opensecrets.org/" part of each member URL causes state 0 to be maintained (although state 1 is briefly visited for each "s", since no "u" follows, control is returned to state 1). Meanwhile, each input symbol is copied to the output. Finally, the "s" of "summary" invites state 1 and the "u" is replaced by "e", leading to state 2. Transition to and at state 3 enable "mma" to be replaced by "cto". Transition to state 4 occurs upon "r" and the other arc at this state removes the "y" from "summary" and the cycle at this state enables the remainder of the URL to be appended.

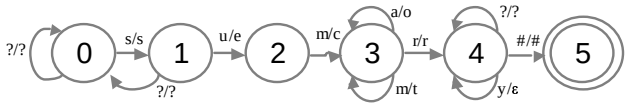


Figure 4: Transducer for member URL to funding URL

At this point, it is tempting to ask: is a transducer simply just a more inefficient means for saying *replace word* w_x *with word* w_y? Although there is no question that transducers can result in word-level replacement, as shown in Figure 4, they permit a far more powerful way of describing – at a letter-level – how replacement occurs. For example, for a source value "Marina del Rey" that is embedded later in an encoded URL as "Marina+Del+Rey", we can learn the transducer shown in Figure 5, which captures the higher-level notion of *replace all spaces with plus signs*. Thus, transducers are a more powerful way to describe how letters from a source value can be used to generate a target value.

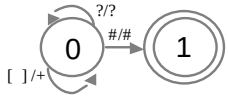


Figure 5: Transducer that replaces spaces with plus signs

A Unifying Learning Algorithm

In this section, we present an incremental learning algorithm called RETROSPECT that incorporates the value prediction methods above.

In general, RETROSPECT uses previous hint and answer tuples to incrementally learn how to construct prediction tuples. Specifically, the algorithm learns a predictor for each attribute of the answer tuple. This predictor is either a decision tree or a *transformation rule*, the latter possibly including a transducer. For example, based on the type of data in Table 2, RETROSPECT learns a decision tree for predicting federal official given a street address. In contrast, based on the data in Table 3, the algorithm learns a transformation rule (that includes a transducer) for predicting Funding URL given a Member URL.

RETROSPECT is applied at the point of speculation (e.g., by the Speculate operator) for each hint and answer tuple pair it receives. The algorithm uses the first pair to initialize all of its predictors as transformation rules. Each rule describes how to use tokens of the hint tuple to derive a value for a particular attribute of the answer tuple. This allows speculation to occur for the very next tuple.

```
FUNCTION Retrospect
  INPUT: old-predictor, hint-tuple, answer-value, predicted-value
  OUTPUT: new-predictor
{
  if old-predictor is NULL
    new-predictor ← create transformation-rule
  else
    new-predictor ← NULL
    if predicted-value <> answer-value
      if old-predictor was a transformation-rule
        new-predictor ← refine transformation-rule hypothesis
        if unable to refine transform-rule hypothesis
          new-predictior ← create decision-tree
          add (hint, answer-value) example to new decision-tree
      else   /* old-predictor is a decision tree */
        add (hint, answer-value) example to existing decision-tree
  return new-predictor
}
```

Future answer tuple values are then first compared with the prediction tuple value issued. If they match (i.e., an accurate prediction), nothing is done – the initial transformation rule was correct. If not, RETROSPECT attempts repair. Repairing a transformation rule involves choosing a new hypothesis that describes the current situation as well as those prior. If no refinement exists, the method of prediction is changed to classification – thus, a decision tree is employed. From then on, all future incorrect predictions are handled via incremental decision tree refinement – for example, using a method similar to that described by (Utgoff 1989).

Transformation rules require further explanation. Each rule is simply a recipe for how to compose a particular value of the prediction tuple from the hint tuple. Rules consist of an ordered set of operations that progressively compose the answer value string. Table 4 shows all possible operations and what they concatenate.

Operation	Concatenates
Append	String literal
Copy	Tokens replicated from hint
Transduce	Tokens transduced from hint

Table 4: Transformation operations

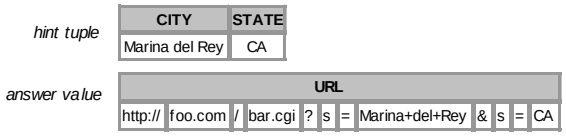
Building a transformation rule for a particular hint tuple and answer value pair then involves three basic steps:

1. Tokenizing the hint tuple and answer values
2. Finding the best alignment between the token sets
3. Using that alignment to build a transformation rule

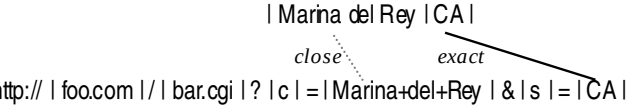
The first step is tokenization. Hint tuples are naturally tokenized by their attribute boundaries. However, certain hint attributes, as well as the answer value, may require further domain-dependent tokenization. With web-based information gathering plans, we have found it necessary to support HTTP URL tokenization. For example, consider tokenization of the following hint and answer tuple pair:

Name	Member URL	Funding URL
Boxer	http://www.opensecrets.org/summary.asp?CID=N00006692&cycle=2002	http://www.opensecrets.org/sector.asp?CID=N00006692&cycle=2002
Feinstein	http://www.opensecrets.org/summary.asp?CID=N00007364&cycle=2002	http://www.opensecrets.org/sector.asp?CID=N00007364&cycle=2002
Harman	http://www.opensecrets.org/summary.asp?CID=N00006750&cycle=2002	http://www.opensecrets.org/sector.asp?CID=N00006750&cycle=2002

Table 3: The relationship between official name and the resulting OpenSecrets member and funding URLs



`The second step of building a transformation rule is to find an alignment between the tokenized hint tuple and answer value. An alignment describes a mapping from hint to answer value tokens, using one of two kinds of edges. *Exact* edges are used to describe a complete replication of a hint token while a *close* edge describes an approximate replication. "Closeness" occurs when some hint and value are not equal, but some threshold percentage ρ of characters from the hint exist in the answer token. For example, one possible alignment of the above hint/answer pair is:



In this example, "Marina del Rey" and "Marina+del+Rey" are close in that the hint token contains all of the characters in the answer value, but the two are not equal.

There may exist several possible alignments and thus several possible transformation rule hypotheses. Any method of choosing one hypothesis over another can be used, since all are capable of generating the target value. However, alternative alignments are not discarded; they may be used later during refinement.

Once an alignment has been chosen, the transformation rule can be built. This simply involves iterating through each token of the answer value and adding one of the operations in Table 4 to the rule, based on the alignment (or lack of) with the tokenized hint. When a value token cannot be aligned to a hint token, **Append** is chosen. When an exact match exists, **Copy** is chosen. Finally, **Transduce** is used for close matches. Each Transduce is associated with a transducer that describes how to translate the hint token into the value token. Learning a transducer can be done via the approach suggested by (Oncina et al. 1993) or by simpler, approximation algorithms.

Hypothesis refinement. When a predicted value does not match the real value and the current method of prediction is transformation, RETROSPECT can potentially repair the old rule through hypothesis refinement. As described earlier, building a transformation rule requires choosing from a set of hypotheses $H = (h_0, h_1,...h_k)$. During refinement, a new set H', is developed for the hint/answer example that violates the existing rule and this new set is used to filter out incompatible hypotheses from the previous set via $H = H \cap H'$. If $H <> \varnothing$, a new hypothesis is chosen from the resulting H, otherwise refinement is deemed impossible.

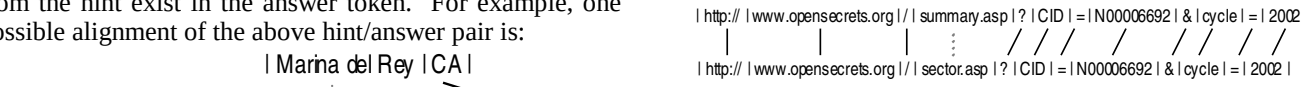
Example: Learning RepInfo Predictors. As another example of RETROSPECT, let us consider how it learns the latter two predictors in Figure 3. That is, we want one that is able to predict member URL given the name of the official and one that predicts the funding URL given member URL. Our discussion will assume that the examples seen during incremental learning are the same as those in Table 3 (Boxer, Feinstein, and Harman).

Consider the first predictor. For the first example, the hint "Boxer" is followed by the member URL. As RETROSPECT specifies, a transform rule is the form of the initial predictor. However, since there is no alignment between hint and answer tuples (i.e., they share no common tokens) the resulting transformation rule is:

1. APPEND ("http://...summary.asp...CID=N006692...2002")

The next example (Feinstein) violates this rule and, since there are no alternatives that explain both, refinement is impossible. Thus, the algorithm reverts to a decision tree to learn the relationship between official name and URL.

Now consider learning the predictor of funding URL based on member URL. As with the previous predictor, RETROSPECT first builds a transformation rule. However, unlike the previous predictor, the alignment:



is possible and results in the (simplified) rule:

1. COPY (hint tokens 0-2)
2. TRANSDUCE (hint token 3, τ_1)
3. COPY (hint tokens 4-11)

In this case, the τ_1 transducer is similar to the one in Figure 4. Future examples (Feinstein, Boxer, etc) validate this rule and no further refinement is necessary.

Thus, after learning the predictors for the data in Table 3, the system can then use the name of a federal official to predict the member URL and subsequently use this speculative member URL to predict the funding URL. Thus, both predictions can be issued in parallel.

Preliminary Results

To begin evaluating the efficiency and accuracy of RETROSPECT, we have initially compared it to a prediction method based on caching for the modified RepInfo plan in Figure 3. Specifically, we conducted three tests – one for each speculative opportunity in Figure 3 – and compared both approaches. These three tests are summarized below:

Test	Hint	Prediction
P1	street address	federal official
P2	federal official	member URL
P3	member URL	funding URL

Each test involved drawing s_{train} training examples and s_{test} testing samples from the same larger pool of size s_{pool}. The P1 pool consisted of 5000 addresses in a distribution roughly equivalent to that of the US population (e.g., there were more addresses from New York and California than Wyoming). For P2, the pool was the set of all US senators and representatives (435 + 100 = 535). The P3 pool consisted of all 535 OpenSecrets member URLs. The predictors learned by RETROSPECT were the same as described earlier: P1 and P2 were decision trees while P3 was a transformation rule. Table 5 shows the accuracy and efficiency of RETROSPECT as compared to caching.

				Caching		Retrospect	
TEST	S_{pool}	S_{train}	S_{test}	Accuracy	Size (bits)	Accuracy	Size (bits)
P1	5000	1000	1000	20.42%	458208	85.48%	113968
P2	535	100	100	17.07%	91840	16.07%	91840
P3	535	100	100	16.78%	117600	99.90%	10944

Table 5: Preliminary results comparing RETROSPECT to caching

For P1 and P3, RETROSPECT was substantially more accurate and space efficient than caching. However, in P2 – where there was only one attribute by which to classify – RETROSPECT degenerated into a tree with one branch. Even under this worst-case classification scenario, however, use of RETROSPECT was roughly equivalent to caching.

Related Work

Learning to speculatively execute programs has been well-studied in computer science. Historically, computer architecture research has largely focused on branch prediction – which involves predicting *control*, not data. Recently, hardware-level value prediction has received some attention in the literature; both (Lipasti et al. 1996) and (Sazeides and Smith 1997) provide good overviews of current approaches. However, in general, the techniques that can be used for hardware-level value prediction tend to be limited by resource constraints.

To the best of our knowledge, this is the first paper that discusses learning value predictors for the speculative execution of information gathering plans. (Hull et al. 2000) describe use of speculative execution in a decision-flow framework, but speculation in that system is control-based and there is no learning involved. There is an interesting relationship between our work and (Shanmugasundaram et al. 2000), which describes how Niagara executes plans based on partial results to combat the performance penalties of blocking operators (like NEST). Both describe plan execution based on unconfirmed results; however, the partial results approach deals with data that has been generated by other operators in the system (no learning involved) and applies to consumers of aggregate operators, whereas the approach here describes how to predict intermediate data and can be applied anywhere within a plan (although it would most commonly occur after a network read). The partial results approach is also similar to past work in optimizing aggregate queries (Hellerstein et al. 1997) and approximate query answering. All of these efforts are related to the notion of speculative execution in that they attempt to increase performance by computing based on unconfirmed results. However, unlike speculative execution, none generate data predictions to accomplish that execution (i.e., they leverage real results, but in partial form) and thus none of them need to synthesize data.

Conclusion

Successful speculative execution of information gathering plans is fundamentally linked with the ability to make good predictions. In this paper, we have described how two simple techniques – decision trees and transducers – can be applied to the problem. Our initial experimental results have shown that such predictors are not only more space efficient than simple caching schemes but that they are also capable of issuing predictions for novel hints. We believe that a bright future exists for data value prediction at the information gathering level, primarily because of the potential speedup enabled by speculative execution and because of the availability of resources (i.e., memory) that exist at higher levels of execution, enabling more sophisticated machine learning techniques to be applied.

References

Barish, Greg and Craig A. Knoblock (2002). Speculative execution for information gathering plans. *Proceedings of the 6th International Conference on AI Planning and Scheduling*. Toulouse, France.

Friedman, Marc and Daniel S. Weld (1997). Efficient execution of information gathering plans. *Proceedings of the 15th International Joint Conference on Artificial Intelligence*. Nagoya, Japan**:** 785--791.

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 Management of Data*. Tuscon, AZ**:** 171--182.

Hull, Richard, Francois Llirbat, 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.

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.

Knight, Kevin and Yaser Al-Onaizan (1998). Translation with finite state devices. *Proceedings of the 3rd Conference of the Association for Machine Translation in the Americas*. Langhorne, PA**:** 421--437.

Lipasti, Mikko H., Christopher B. Wilkerson and John P. Shen (1996). Value locality and load value prediction. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA**:** 138--147.

Naughton, Jeffrey F., David J. DeWitt, David Maier and et al. (2001). "The niagara internet query system." *IEEE Data Engineering Bulletin* **24**(2): 27--33.

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.

Quinlan, J.R. (1986). "Induction of decision trees." *Machine Learning* **1**(1): 81--106.

Sazeides, Yiannakis and James E. Smith (1997). The predictability of data values. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture***:** 248-258.

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 3rd International Workshop on Web and Databases (WebDB)*. Dallas, TX**:** 17-22.

Utgoff, Paul E. (1989). "Incremental induction of decision trees." *Machine Learning* **4**(2): 161--186.