Combining Classification and Transduction for Value Prediction in 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 execution of information gathering plans can dramatically reduce the effect of source I/O latencies on overall performance. However, the utility of speculation is closely tied to how accurately data values are predicted at runtime. While caching is one approach that can be used to issue future predictions, it scales poorly with large data sources and is unable to make intelligent predictions given previously unseen input data, even when there is an obvious general relationship between prior input and resulting output. In this paper, we describe a novel way to combine classification and transduction for a more efficient and accurate value prediction strategy, one that capable of issuing predictions about previously unseen hints. We show how our approach results in significant speedups for plans that query multiple sources or sources that require multi-page navigation.

1 Introduction

The performance of Web information gathering plans can suffer because of I/O latencies associated with the remote sources queried by these plans. A single slow Web source can bottleneck an entire plan and lead to poor execution time. When a plan requires multiple queries (either to the same source or to multiple sources), performance can be even worse, where the overhead is a function of the slowest sequence of sources queried.

When multiple queries are required, speculative plan execution (Barish and Knoblock 2002) can be used to dramatically reduce the impact of aggregate source latencies. The idea involves using data seen early in plan execution as a basis for issuing predictions about data likely to be needed during later parts of execution. This allows data dependency chains within the plan to be broken and parallelized, leading to significant speedups.

To maximize the utility of speculative execution, a good value prediction strategy is necessary. The basic problem involves being able to use some hint h as the basis for issuing a predicted value v. One approach involves caching: 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 prediction of v_y. As a result, a plan that normally queries source S1 with h_x and subsequently source S2 with v_y can be parallelized so that both S1 and S2 are queried in parallel, the latter speculatively. Unfortunately, caching has two major drawbacks. First, it does not scale well when the domain of hints is large. A second drawback is the inability to deal with *novel* (previously unseen) hints, even when an obvious relationship exists between hint and predicted value.

In this paper, we present an alternative to caching that involves automatically learning predictors that combine classification and transduction in order to generate predictions from hints. Our approach succeeds where caching fails: the predictors learned usually consume less space than that demanded by caching and they are capable of making reasonable predictions when presented with novel hints, the latter leading to better speedups. Specifically, this paper contributes the following:

- An algorithm that learns efficient transducers capable of variety of string transformations.
- An algorithm that combines classification and transduction to learn value predictors

The rest of this paper is organized as follows. The next section briefly reviews information gathering and provides a motivating example for speculative execution. In Section 3, we describe how classification and transduction can be used to build efficient and intelligent predictors. Section 4 describes our learning algorithms that combine both techniques. Section 5 describes experimental results of using our approach. Finally, Section 6 details the related work.

2 Preliminaries

Web information gathering plans retrieve, combine, and manipulate data located in remote Web sources. Such plans consist of a partially-ordered graph of operators $O_1..O_n$ connected in a producer/consumer fashion. Each operator O_i consumes a set of inputs $a_1..a_p$, fetches data or performs a computation based on that input, and produces one or more outputs $b_1..b_q$. The types of operators used in information gathering plans vary, but most either retrieve or perform computations on data.

To better illustrate a Web information gathering plan, we consider the example plan *CarInfo*, shown in Figure 1. Given a car type and price willing to be paid, *CarInfo* locates the make and model which has a median price closest to that specified and then retrieves the full review of that car from the Web site ConsumerGuide.com. The plan is simple, consisting of four *Wrapper* operators that retrieve and extract data from various parts of the remote source. Specifically, the plan involves: (a) querying CarsDirect.com for the car make and model having a median price closest to that specified, (b) querying ConsumerGuide.com for the resulting make and model, (c) retrieving the link to the summary page for that car (using the link provided in the search results), and (d) retrieving the full review of that car using the link provided on the summary page.

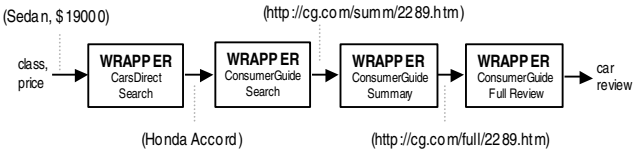


Figure 1: The CarInfo plan

For example, for the input *(Sedan, $19000)*, the car returned is *(Honda Accord)*, the summary URL for this car is *(http://cg.com/summ/2289.htm)* and the full review URL *(http://cg.com/full/2289.htm)*. Once at the full review URL, the review text can be extracted.

Notice that since steps (b), (c), and (d) are dependent on the steps that precede them, the plan must be executed sequentially. As a result, plan performance is the summation of the average time required for each remote query. For example, if the source has an average latency of *2s*, than the average plan execution time is 4*2s = *8s*.

2.1 Speculative Plan Execution

Speculative execution is one technique that can be used to overcome the effects of aggregate latencies in information gathering plans that make queries dependent on value bindings from the answer to a prior query, such as those shown in Figure 1.

As described in (Barish and Knoblock 2002), a plan is transformed into one capable of speculative plan execution by the insertion of two additional operators – **Speculate** and **Confirm** – at various parts the plan, based on a recursive analysis of the most expensive path to execute within that plan. For example, one possible result of transforming the plan in Figure 1 for speculative execution is shown in Figure 2.

As shown, the **Speculate** operator receives copies of data sent to operators executing earlier in the plan. Based on the hints it receives, **Speculate** can generate predicted values for later operators that can be transmitted immediately to those operators. Thus, the earlier and later parts of the plan can be parallelized. After the earlier operators finish executing, **Speculate** can assess whether or not its initial predictions were correct and forward the results onto a **Confirm** operator, which ensures that speculative data does not prematurely exit the plan or cause some other irreversible action to occur. Finally, notice that Figure 2 shows that speculation can be *cascading*: speculation about one operator can drive the speculation of another operator, leading to greater degrees of parallelism and thus arbitrary speedups.

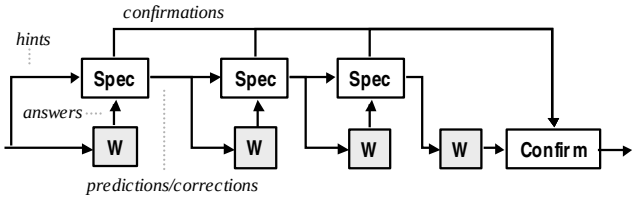


Figure 2: CarInfo transformed for speculative execution

As a result of the transformation shown in Figure 2, execution would then proceed as follows. Input data, such as *(Sedan, $19000)*, would result in the retrieval of the initial search results in parallel with the predicted make and model – which would drive the predictions of summary and full-review URLs so that all four retrievals (three speculative) were executed at once. If all predictions are correct, the resulting execution time can be reduced to only *2s* plus the overhead to speculate, a maximum speedup of about 4. However, the **average** speedup depends on the average accuracy of prediction: *the greater this accuracy, the higher the average speedup*.

3 Value Prediction Strategies

Caching can be used to implement value prediction when speculatively executing plans such as *CarInfo*. Unfortunately, caching does not allow predictions to be issued for unseen hints. As a result, the average accuracy of prediction can be low when the domain of possible hints is large. Further, trying to achieve better accuracy under these conditions can require significant amounts of memory. In this section, we describe how an integrated approach consisting of classification and transduction addresses both drawbacks of caching and results in a more intelligent and space-efficient prediction strategy.

3.1 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 future instances can be classified correctly. Once learned, a classifier can also make reasonable predictions about new instances – a combination of attribute values that had not previously been seen. The ability for classification to accommodate new instances is intriguing for the speculative execution of information gathering plans because, unlike with caching, it is possible to make predictions about novel hints.

For example, consider the prediction of the make and model of a car in the *CarInfo* plan. It turns out that CarsDirect.com returns the same answer (*Honda Accord*) for "*Sedan*" as it does for other types (such as "*All*" and "*Coupe*") in the same price range. The association of the

same make and model to multiple criteria combinations occurs somewhat frequently on CarsDirect.com.

To see why classification is a more effective technique than caching for the prediction of the make and model, consider what conclusions can be made by each technique about the following data:

Type	Price	Car
Sedan	18000	Saturn S Series
Sedan	19000	Honda Accord
Sedan	20000	VW Beetle
Coupe	18000	Saturn S Series
Coupe	19000	Honda Accord
Coupe	20000	VW Beetle

The data above is what a cache would contain. In contrast, a classifier like Id3 (Quinlan 1986) would induce the following decision tree:

```
pri <= 18000 : Saturn S Series (2.0)
pri > 18000 :
  |  pri <= 19000 : Honda Accord (2.0)
  |  pri > 19000 : VW Beetle (2.0)
```

When presented with an instance previously seen, such as *(Sedan, 19000)*, both the cache and the classifier would result in the same prediction: *(Honda Accord)*. However, when presented with a new instance, such as *(Coupe, 18500)*, the cache would be unable to make a prediction. In contrast, the classifier would issue the correct prediction of *(Honda Accord)*. Any errors made by classification would be caught automatically later in execution by the **Confirm** operator.

The decision tree above is also more space efficient than a cache for the same data. The cache requires storing 6*3 = 18 values. The decision tree above requires only storing 5 values (just those shown) plus the information required to describe tree structure and attribute value conditions (i.e., *pri <= 18000*).

In short, classifiers such as decision trees can function as better, more space-efficient predictors. And in the worst case, where each instance corresponds to a unique class, a classifier simply emulates a cache.

3.2 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 $Q \times \Delta$ to Δ^*. Our interest is in a particular type of sequential transducer called a *p*-subsequential transducer that allows at most *p* output symbols to be appended to the output (i.e., exist on the final state transition arc).

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 1999 Honda Accord was *http://cg.com/summ/2289.htm* and the full review was *http://cg.com/full/2289.htm*. Notice that both URLs use the same piece of dynamic information (*2289*), but in different ways. By learning this transduction, we can then predict future full review URLs for corresponding summary URLs we have never previously seen. Transducers can also allow us 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 paper, we 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 describes how to construct 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*, *Classify*, and *Transduce*}. **Insert** constructs the static parts of predicted values. **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, 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 transducers shown in Figure 3, for predicting the full-review URL in the *CarInfo* example. Figure 3 shows the value transducer performs high-level operations – the insertion of substrings and the call to a lower-level transduction. The second transducer (in abbreviated form) uses the **Accept** and **Copy** operations to transform the part of the hint value into its proper point in the predicted value. Thus, the first step builds the "*http://cg.com/full/*" part, the second step copies the "*2289*" part and the third step appends the "*.htm*" part.

In short, 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 compact predictors that capitalize on the regularity of Web queries and source structure.

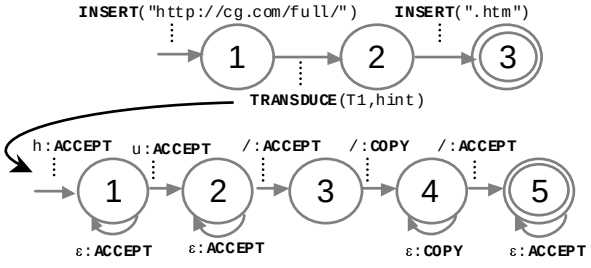


Figure 3: Value transducer for CarInfo

4 A Unifying Learning Algorithm

In this section, we present a set of algorithms that describe how to combine classification and transduction in order to induce value transducers (VTs) for the speculative execution of information gathering plans.

To learn a VT, the general approach consists of:

1. For each attribute of the answer tuple, identify an *SD Template* that distinguishes static 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 add a **Classify** arc to the VT.

We implemented this in the algorithm LEARN-VALUE-TRANSDUCER, shown below. The algorithm takes a set of hints, a set of corresponding answers, and returns a VT that fits the data:

```
1   Function LEARN-VALUE-TRANSDUCER
      returns ValueTransducer
2   Input: the set of hints H, the set of answers A
3     VT ← ∅
4     tmpl ← LEARN-SD-TEMPLATE (A);
5     Foreach element e in tmpl
6       If e is a static element
7         Add Insert (e.value) arc to VT
8       Else if e is a dynamic element
9         DA ← the set of dynamic strings in A for this tmpl element
10        HT ← LEARN-HINT-TRANSDUCER (H, DA)
11        If HT != ∅
12          Add Transduce (HT) arc to VT
13        else
14          C ← LEARN-CLASSIFIER (H, DA)
15          Add Classify (C) arc to VT
16    Return VT
17  End /* LEARN-VALUE-TRANSDUCER */
```

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 transforming transducer, however, require unique algorithms.

4.1 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 or classification.

To identify an SD template, we use an approach based on the *longest common subsequence* (LCS) between a set of values. First, an LCS identification algorithm similar to the one described by (Hirschberg 1975) is applied to the set of answer values. We then iterate through the LCS on each answer value to determine the set of possible static/dynamic templates that fit that 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 below.

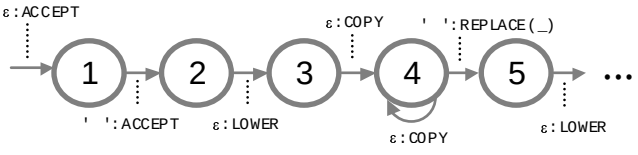
```
1   Function LEARN-SD-TEMPLATE returns Template
2   Input: set of strings S
3     tmpl ← ∅
4     lcs ← GET-LCS(S)
5     If seq != ∅
6       tmplSet ← ∅
7       Foreach string s in S
8         curTmplSet ← EXTRACT-TEMPLATES (s, lcs)
9         tmplSet ← tmplSet ∩ curTmplSet
10      If tmplSet != ∅
11        tmpl ← choose any member of tmplSet   /* all are valid */
12    Return tmpl
13  End /* LEARN-SD-TEMPLATE */
```

4.2 Learning hint transducers

To learn a hint transducer, we also make use of SD-template identification. However, instead of identifying a template that fits all answers, we identify templates that fit all *hints*. Based on this template, we then construct a transducer that accepts the static parts of the hint string and performs the character-level transformation on the dynamic part. A sketch of the algorithm that implements this, LEARN-HINT-TRANSDUCER, is shown below.

```
1   Function LEARN-HINT-TRANSDUCER returns HintTransducer
2   Input: the set of hints H, the set of resulting strings S
3     Use LCS to identify static parts between all H
4   Foreach H,S pair (h, s)
5     h′← extraction of h replacing static chars with the token 'A'
6     A ← Align (h′, s) based on string edit distance
7     Annotate A with character level operations
8   End
9   RE ← Build a reg expr that fits all annotations (using LCS)
10  If RE == ∅
11    Return ∅
12  Else
13    Return general predictive transducer based on RE
        that accepts static sequences of H where necessary
        and transduces dynamic sequences.
14  End /* LEARN-HINT-TRANSDUCER */
```

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., {**AAAA***Joe Smith*, **AAAA***Jane 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**, **AAAAALCCCRLDDDDD**}. Next, it would use the LCS to build the regular expression {**A*LC*RLD***} fitting these examples and, from this, a general predictive transducer (partial form shown):



5 Experimental Results

To evaluate our approach to value prediction, we compared it with caching on three sample plans that can benefit from speculative execution. These plans were chosen because all query multiple sources and/or multiple times within a source in order to retrieve information that is not possible to query directly. These plans normally require sequential execution; however, with speculative execution, significant speedup is possible.

These plans included *CarInfo* (which we have already described), *RepInfo*, and *PhoneInfo*. *RepInfo*, based on the plan described in (Barish and Knoblock 2002), queries the site Vote-Smart.org for the issue positions of U.S. federal representatives for a particular nine-digit zip code. Its plan involves three queries: one for the list of representatives in the desired zip code, navigation to the profile page for each member, and navigation to their corresponding issue positions page. *PhoneInfo* is a similar plan that takes a U.S. phone number, does a reverse lookup of that number (on SuperPages.com) to find the state of origin and then queries the US Census (QuickFacts.census.gov) about demographics for that state. When querying the Census, additional navigation is required to get from the initial summary page about the state to the corresponding demographics details page.

All three were modified for speculative execution, with results similar to that shown in Figure 2. We then learned predictors for each. The *CarInfo* predictors, as described, involved classification (make/model/year to car summary page) and transduction (summary to full review page). *RepInfo* required classification (nine-digit zip to political district page) and transduction (district to issue positions page). Finally, *PhoneInfo* required classification (phone number to state) and transduction (initial state facts page to detailed demographics page).

When learning the predictors, 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.

We implemented speculative execution in Theseus, a streaming dataflow system for Web information gathering. Since our tests involved thousands of requests, we ran our plans using Theseus on local copies of the relevant data and simulated network latencies during retrieval. In doing so, we assumed each source had a latency of 2 seconds (note that the particular latency chosen does not matter –speedups will be the same).

Our results focus on three measurements. The first, shown in Table 1, show the average number of examples required in order to learn the correct transducer required by each of the plans. Notice, that *RepInfo* required more than *PhoneInfo* or *CarInfo* because there was a higher likelihood of the examples sharing some part of their dynamic data in common – and this was extracted by the LCS-based algorithm as a static element.

	CarInfo (full-review URL)	**RepInfo** (issue positions URL)	**PhoneInfo** (demographics URL)
Examples	6	11	7

Table 1: Learning the correct transducer

The second set of results, shown in Figure 4, focuses on the accuracy of classification, as measured over a 10-fold cross-validation sample of the data (where no data in the test fold was in any of the training folds). As expected, domains with larger sets of discrete target classes (such as *RepInfo*) required many more examples than those with smaller numbers of classes (like *PhoneInfo*).

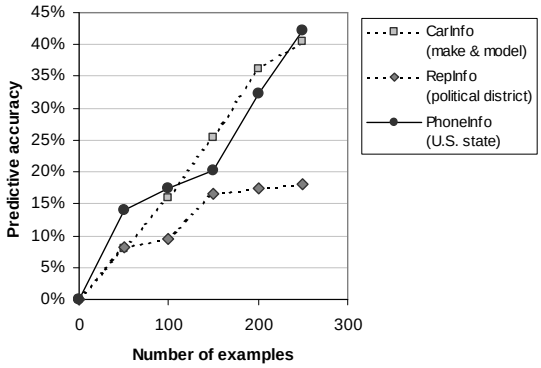


Figure 4: Accuracy of classification

The final set of results show how the learning we have described resulted in better average plan execution speedups than did caching. Due to space constraints, we show results from only one of the plans – *CarInfo* – in Figure 5. This figure shows that, while the benefit of caching degrades significantly as the composition of future requests contains a greater number of unseen examples, the learning allows accurate predictions to be made – even for a mix containing entirely new requests. Furthermore, as the number of examples increases, speedup from learning increases and the accuracy of speculation (even on 100% unseen instances) gradually increases. Though omitted here, these same general trends hold for both *RepInfo* and *PhoneInfo* as well (although to different degrees).

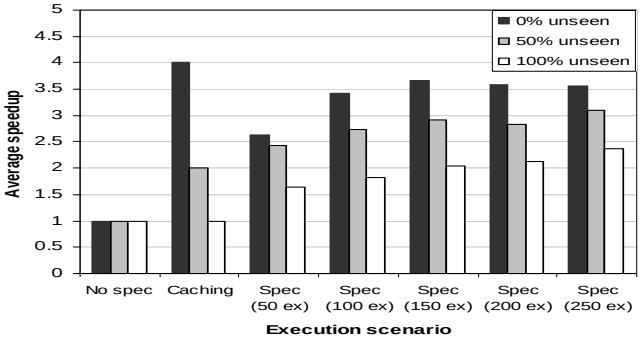


Figure 5: Learning vs. caching in CarInfo

6 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. While hardware-level value prediction is an active area of research, its goals are very different (predicting memory locations or loop iterations) that our goals here.

To the best of our knowledge, there has not been past 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 (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.

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 automating translations of decimal to Roman numbers or English word spellings of numbers to their decimal equivalents. Our work differs from OSTIA mainly in that the transducers we learn capture *the general process* of a regular class of string transformations. After learning from only a few examples, our algorithm can achieve a high-degree of accuracy on such classes. 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 transducers we 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 also makes use of classification.

Finally, while our 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 unrequested URLs.

7 Conclusions

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 – classification and transduction – can be combined and applied to the problem. Our experimental results show that learning such predictors can allow for signficant speedups when gathering information that must be queried indirectly. 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 Sixth International Conference on AI Planning and Scheduling (AIPS 2002)*. Tolouse, France.

Gildea, Daniel and Daniel Jurafsky (1996). "Learning Bias and Phonological-Rule Induction." *Computational Linguistics* **22**(4): 497--530.

Hirschberg, Daniel S. (1975). "A linear space algorithm for computing maximal common subsequences." *Communications of the ACM* **18**(6): 341--343.

Hsu, Chu-Nan and Chien-Chi Chang (1999). Finite-State Transducers for Semi-Structured Text Mining. *Proceedings of IJCAI-99 Workshop on Text Mining: Foundations, Tecchniques, and Applications*.

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.

Mohri, Mehryar (1997). "Finite-State Transducers in Language and Speech Processing." *Computational Linguistics* **23**(2): 269-311.

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.

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.

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.

Zukerman, Ingrid, David W. Albrecht and Ann E. Nicholson (1999). Predicting User's Requests on the WWW. *Proceedings of the 7th International Conference on User Modeling*.