An Expressive and Efficient Language for Information Gathering on the Web

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

While network query engines make it possible to gather and combine data from multiple Web sources, these systems primarily focus on efficient query execution and do not solve some of the more complicated problems of online information gathering. Such problems require alternative types of control flow and better integration with the external world; the unique nature of the Web requires query plans be expressive enough to accommodate these demands. In this paper, we describe an information gathering plan language that is expressive and promotes efficient execution. Through its support for subplans, recursion, and a unique set of operators, the language allows plans that can interactively gather data over a series of pages, monitor remote sources, and asynchronously notify users of updates and results. We also present a execution system that efficiently implements the plan language using a dataflow-style executor capable of pipelining data between operators.

Introduction

Current research on network query engines (Ives et al. 1999, Hellerstein et al. 2000, Naughton et al. 2001) has shown that it is possible to gather and combine data from multiple Web sources using plans similar to those found in traditional database systems. However, such research has focused primarily on the efficiency of plan execution and has tended to ignore the problems associated with more complicated types of Internet information gathering.

The unique nature of the Web is such that certain types of queries require a plan language more expressive than those capable of only basic integration. Consider collecting the results of a search engine query. Nearly all search engines display query results spread across multiple result pages. To collect all of the data, an automated system must be capable of interleaving the collection of partial results with navigation to additional results and must be able to eventually decide when to stop. The control flow required for such a task is not supported by the plan languages of existing network query engines.

Another unique aspect of the Web is that it is highly dynamic and there is considerable interest in being able to monitor sources. However, since the Web has no built-in trigger facility, one has to "discover" updates by querying the Web over a period of time that extends beyond that of a single interactive query. To track an integrated set of data requires a language capable of managing intermediate results and communicating important updates to users asynchronously (i.e., via e-mail) as necessary. Again, most network query engines do not support such capabilities.

While these examples demonstrate that better plan expressivity is desirable, so too is efficient query execution. Gathering data on the Web is an I/O-intensive process that renders CPUs idle for periods of *seconds* during plan execution. Thus, what is needed is a plan language that is not only expressive but also very efficient: specifically, one that supports highly concurrent execution.

A plan language can provide substantial degrees of parallelism in two ways. The first is to support a dataflow representation of plans. The partial ordering of operators enabled by a dataflow representation describes execution in maximally parallel terms – operators are only limited by their own data dependencies. A second language-level strategy is to support operators capable of processing pipelined data (i.e.,. tuple-oriented processing). Pipelining refers to the production and consumption of data as soon as possible – producers emit incremental results to consumers – enabling both to work in parallel on the same relation.

In this paper, we present an information gathering plan language that is both expressive and efficient. The proposed plan language is modular and supports the notion of *subplans* to encourage reusability and facilitate recursion. In addition, the language consists of operators that interact with the external world so that it is possible to monitor sources and asynchronously notify users of important updates. While providing better expressivity, plans in this language are efficient because they consist of a dataflow-style ordering of operators and because those operators support the pipelining of data during execution.

The rest of this paper is organized as follows. Section 2 establishes basic terminology, discusses the details of more difficult Web query tasks, and provides an example that we will use throughout the rest of the paper. In Section 3, we propose an information gathering plan language and describe how it enables us to solve the types of problems shown earlier. In Section 4, we discuss the efficient execution of plans generated in this language. Finally, in Section 5, we discuss the related work, both in terms of network query engines and intelligent agents.

Gathering and Monitoring Web Data

In this section, we describe the problem of gathering and monitoring data on the Web. We first describe basic integration tasks and how existing information gathering plan languages allow these tasks to be completed. Next, we describe more complicated types of information gathering tasks and how they necessitate a more expressive plan language. Finally, we provide an example problem that will be the basis for discussion throughout the paper.

Basic information gathering tasks

Basic Web-based information consists of retrieving data from multiple sources, combining, and then filtering as necessary. For example, the plans described in (Friedman & Weld 1997), (Ives et al. 1999), and (Barish et al. 2000) query distinct Web sites, combine the data found in both (either by unioning or joining that data), and then either filter these results or use them to query other web sources. These plans have simple control flow and involve the same types of operators found in traditional database systems – Retrieve, relational operators like Select, Project, Join, and set-theoretic operators like Union, Minus, and Intersect.

Current technology for querying the Web in this manner exists in two forms. One is that provided by Web-based information mediators (Genesereth et al. 1996, Knoblock et al. 2001). These systems use high-level domain models to describe how logical entities are related Web sources. They utilize Web site *wrappers* to convert semi-structured HTML into structured relations and thus allow web sites to be queried as if they were databases. Mediators have largely focused on enabling multiple heterogeneous data sources to be integrated (through query reformulation). With these systems, it is possible to write queries that are answered through information gathering plans that combine and filter data from multiple sources.

A second, more recent technology for accomplishing these types of tasks comes in the form of network query engines. Although these systems enable the Web to be queried in the same way that mediators do, they have generally focused on the need for efficient execution and on the need to process online XML data. They have been mostly concerned with adaptive execution techniques to overcome the inherent latency of querying remote web sites.

In short, existing mediators and network query engines allow Web data to be queried in a manner similar to that found in traditional database systems. The control flow of the plan and types of operators involved are largely the same. In general, these systems have focused largely on the challenges of interoperability and efficiency.

More complicated tasks

The nature of the Web is such that the expressivity provided by traditional query plans is often insufficient for solving other types of common, yet more complicated online information gathering tasks. In particular, the Web is unique in at least two major respects: (a) it is primarily a visual medium and (b) its highly dynamic nature often invites the need for monitoring. Let us consider how each of these aspects independently impacts online querying.

As a visual medium, data on the Web is often organized in a way that makes sense for visual consumption. For example, querying a web source through an HTTP POST or GET often results in answers to that query being organized across multiple pages. For example, a query to a search engine can result in hundreds of web pages that each contain part of the answer. To collect the complete answer to such queries, it is necessary to navigate to each page, collect the results on that page, find the "next page" link, navigate to the next page, collect the results on that page, and so on. This manner of alternating retrieval with navigation is unique to the Internet does not have an equivalent in traditional database systems.

Secondly, Web sources can be highly dynamic and often need to be monitored. Unfortunately, the Web lacks a database-style trigger facility that notifies users when data has changed and does not provide any automated means for identifying differences between current results and those that existed prior to the update. Instead, updates to its data are only realized through a process of repetitive querying, collection of new results, and then comparison of these new results with prior results to discover the differences. Thus, periodic execution and some sort of mechanism for comparison between queries is necessary.

Example

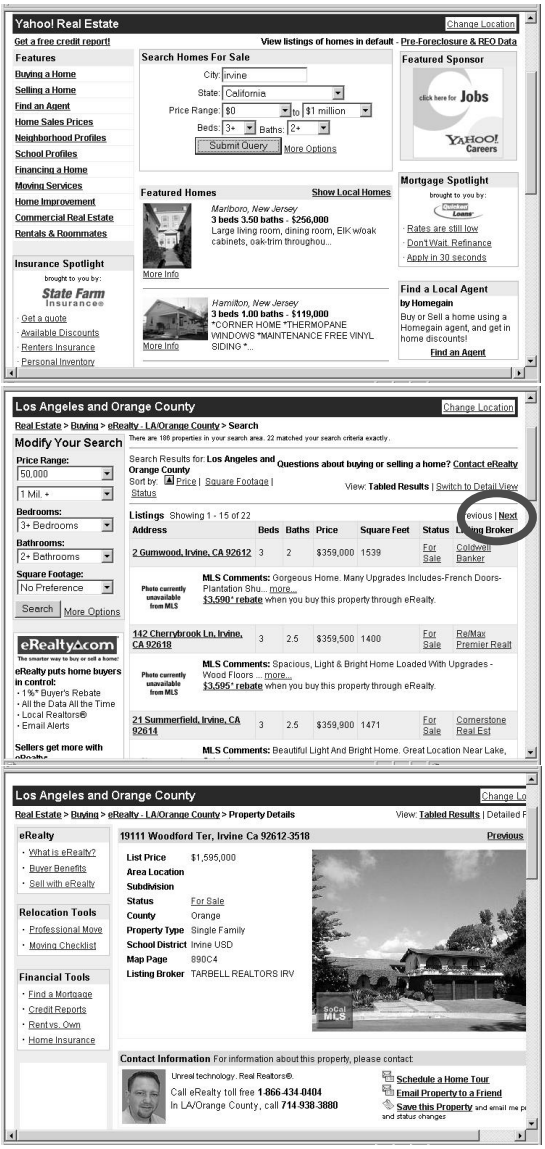
To demonstrate how more complicated types of information gathering problems require more expressive plans, it is useful to describe a detailed example. Consider using the Internet to locate a new home to buy. Suppose we wish to use a site like Yahoo Real Estate to periodically locate houses that meet our search criteria. For example, we wish to find houses that meet a certain set of price, location, and number of rooms constraints.

First, let us discuss how users perform this task manually. Figures 1a, 1b, and 1c show the interface and result pages for Yahoo Real Estate. To query for new homes, users first fill the criteria shown in Figure 1a – specifically, they enter information that includes city, state, maximum price, etc. Once they fill in this form, they submit their query to the site. The initial results are shown in Figure 1b. However, notice that this page only contains results 1 through 15 of 22. To get the remainder of the results, a "Next" link must be followed to the page containing results 16 through 22. Finally, to get the details of each house, users must follow the link associated with each listing. A sample detail screen is shown in Figure 1c.

In practice, performing this task requires manually repeating the above process over a period of days, weeks, or even months. The user must both query the site periodically and somehow keep track of new results. This latter aspect can require a great deal of work – users must note which houses in each result list are new entries.

It is possible to automate part of this process with current data integration technologies. For example, we can use mediators or network query engines to gather and extract data from web pages. But most of these systems do not provide any means for monitoring sources and none provide the ability to gather data spread across multiple pages. To accomplish both tasks, we need plans capable of expressing other types of control flow (such as looping) and operators that facilitate monitoring.

We can consider how such plans generally might look. Figure 2 shows an abstract plan for monitoring Yahoo Real Estate. As the figure shows, search criteria is used to generate houses from Yahoo Real Estate. Houses are separated from their "next page" link and compared against houses that already existed in a local database. Then, the resulting set of new houses are queried for their details and the results are e-mailed to the user. These new results are also appended to the database so that future queries can distinguish new results. Meanwhile, the "next page" link is followed and the resulting new houses go through the same process. Next page links are followed until no more pages are found (i.e., no more next link).



Figures 1a, 1b, & 1c: Querying Yahoo Real Estate

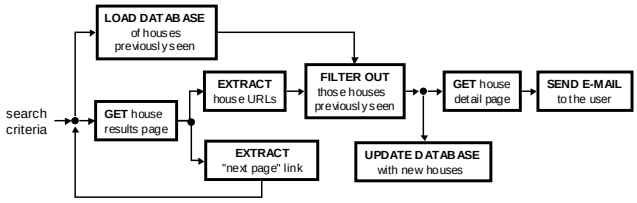


Figure 2: Abstract plan for Yahoo Real Estate

Expressive & Efficient Information Gathering

In this section, we describe an information gathering plan language that makes it possible to construct plans that can accomplish more complicated information gathering tasks, such as the type shown in the abstract plan Figure 2. There are several basic aspects of this plan language to consider – the dataflow representation of plans, the logical pipelining of data during execution, the typing and manipulations on data, the set of operators that are provided, support for modular design through the notion of subplans, and support for information gathering tasks that require looping through use of recursion.

Dataflow representation

All plans in the language we propose consist of a name, a set of input and output variables, and a set of unordered operators that represent the dataflow graph of the plan. A dataflow representation of a plan is desirable from an efficiency standpoint because it describes the maximally parallel mode of execution (Dennis 1974). In contrast to von Neumann models, which rely on an instruction counter to sequentially execute a list of instructions (or operators), a dataflow model allows execution to occur on any operator, whenever its data dependencies are fulfilled. This makes execution fully decentralized, independent for each operator. Thus, execution can be highly concurrent.

Figure 3 shows an abstract plan. As shown, a header part communicates the name of the plan (P1 in this example) and the list of input variables (a and b), and output variables (g). The body section of the plan contains the set of operators. The example below shows four operators – Op1, Op2, Op3, and Op4. Each operator instance consumes one or more inputs and produces zero or more outputs. As shown below, 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.

Figure 4 illustrates how edges in the dataflow graph of operators are communicated through variable names. For example, as described by Figure 3, operator Op1 produces

```
PLAN P1 {
   INPUT: a, b
   OUTPUT: g

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

Figure 3: Sample plan

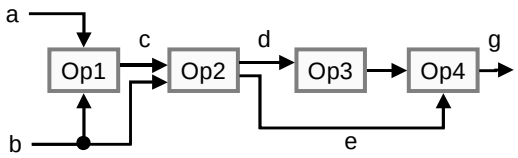


Figure 4: Plan represented as a dataflow graph

c, which is consumed by Op2, and Op2 produces d and e, which are consumed by Op3 and Op4, respectively. Figure 4 shows the corresponding edges that between the operators.

Although the body part of the plan language lists operators in a linear order, this ordering does not affect when they are actually executed. Per the dataflow model of processing, operators fire whenever their individual data dependencies are fulfilled. For example, Op2 can execute when any of its individual inputs b or c are present. Thus, Op2 executes once at the start of the plan (because b is available) and then shortly later on, when c becomes available. In summary, the only ordering of execution that exists at the plan level is that which is communicated by the producer/consumer relationship between operators

Logical data pipelining

Each of the variables in the plan above are logically relational data streams. A stream is a set of tuples in a relation, followed by an *end of stream* (EOS) marker. Operators in the plan logically execute when they receive a tuple for any of their input streams. The conditions that describe when operators can execute is also known in dataflow literature as the firing rule.

For example, a set-theoretic Union operator would take two input streams – *lhs* and *rhs* – and output a stream called *unioned_result* that consists of the unique set of tuples defined by the intersection of lhs and rhs This operator can fire whenever a lhs or rhs tuple is present and emit a unioned_result tuple for each firing. In part, this is due to the nature of the operator. A Minus operator, in contrast, would take two inputs named lhs and rhs and emit a *minus_result* stream that was based on the subtraction of rhs from lhs. However, even though the Minus operator can fire upon receiving a tuple, it cannot emit a *minus_result* tuple until the rhs stream EOS has been received. Both the Union and Minus operator must logically maintain state between invocations (both must not emit duplicates and Minus must keep all of the rhs in memory so that it can be applied to later lhs tuples).

Data types and common manipulations

Data in the system is communicated logically as relations and physically as tuples (i.e., through pipelining). Each tuple consists of a set of attribute/value pairs. Each attribute can be one of five types: char, number, date, relation (embedded), or document (i.e., a DOM object). Embedded relations are supported because they reduce the amount of data communicated during execution.

XML data is supported by the system and is associated with the Document attribute type. The language contains specific operators that allow XML to be converted to relations, for relations to converted to XML documents, and for attributes that are XML documents to be queried in their native form using XQuery. Since XML documents are encapsulated by tuples, they can be pipelined between operators; when/if it is desirable to pipeline the data contained in an XML document, the document is first converted to a relation, is streamed through the system, and can be put back together again as XML later, if desired.

In terms of common manipulations, operators vary on how they output their results with respect to the incoming data. In particular, there are two modes of interaction that merit discussion: the performing of dependent joins and the packing/unpacking of relations.

In data integration plans, it is common to use data collected from one source as a basis for querying additional sources. However, it is tedious (and sometimes impossible because of ordering constraints) to manually join the data input to an operator onto the output data it produces. Instead, many of the operators in this language perform a dependent join of input tuples onto the output tuples that they produce. For example, if the language supported an operator called Round that rounded a floating point value in a column to its nearest whole integer value, and 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)). Dependent joins simplify plans and solve problems related to the joining data when no unique key on the input relation exists.

A related mode of interaction to discuss involves the packing and unpacking of relations. Packing relations is useful when you want to associate a relation with an aggregate function, such as count. 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, if the language supported an aggregate operator called Average, then the result of processing the input described earlier would be (((Jack, 89.73), (Jill, 98.21)), 93.97). Unpacking a relation is necessary to get at the original data. Packing and unpacking is a common activity when a conditional operator needs to evaluate an aggregate measure of a relation and then route it to the proper set of consumer operators which then unpack the data.

Operators

To accomplish more complicated types of information gathering tasks, three basic types of operators are necessary; those that:

- **Gather and manipulate data:** These include the traditional relational operations as well as those capable of processing XML data.
- **Facilitate monitoring:** To effectively monitor data sources, the language includes operators that can access local databases, so that intermediate results can

be stored and then compared against later. In addition, other monitoring operators enable results to be communicated asynchronously – for example, through e-mail, fax, or cell phone.

- **Promote extensibility:** Generally, operating on data either involves operating on individual tuples (single-row functions) or operating on sets of tuples (aggregate functions). The language includes operators that allow users to extend the existing plan language to meet any kind of single-row or aggregate data manipulation necessary.

We now describe the operators in more detail and focus on those not found in other types of network query engines.

Operators for gathering and manipulating data. The language supports basic operators for gathering data from the Web and manipulating it (filtering, combining, etc.). These operators are shown in Table 1.

Of these, the **Wrapper** operator is the most interesting. Its purpose is to use values from an input relation as the basis for querying a specified Web source. In general, wrappers are mechanisms for querying a remote semi-structured Web site as if it were a local relational database. Calling a wrapper involves providing input constraints (if any), executing the wrapper, and then collecting its results. Correspondingly, our Wrapper operator uses values from each tuple of an input relation as the input constraints and queries the remote site accordingly. Results generated by each input tuple are combined with the input that generated them – this is referred to as a *dependent join*. The language also includes operators for manipulating XML data, including XQuery for querying XML and Xml2Rel and Rel2Xml for converting XML data to relational and vice versa. The bulk of the remaining manipulation operators are familiar and can be found in current network query engines and mediators.

Monitoring operators. There are two aspects to the monitoring process – the ability to keep track of past results and the ability to asynchronously notify users of updates. To accomplish these tasks, the plan language we propose supports a set of monitoring-related operators. These are shown in Table 2.

The DbQuery, DbAppend, DbExport, and DbUpdate operators allow plans to interact with local databases. This

Name	Purpose
wrapper	Extracts web page data as relation
xml2rel	Converts XML document into a relation
rel2xml	Converts a relation to an XML document
xquery	Manipulates attributes that are XML documents
select	Filters relation based on specified criteria
project	Extracts specified attributes from relation
join	Combines relations based on specified criteria
union	Performs set union of two relations
minus	Performs set minus of two relations
intersect	Performs set intersect of two relations
pack	Embeds relation in single attribute tuple
unpack	Expands embedded relation from single attribute tuple

Table1: Data manipulation operators

Name	Purpose
dbquery	Fetches relation from DB based on query
dbappend	Append to existing relation in DB
dbexport	Export relation to DB
dbupdate	Processes an update query (no results returned)
email	Emails data to specified e-mail address
fax	Faxes data to specified fax number
phone	Sends text message to specified cell phone number
null	Conditionally routes stream based on if another is empty

Table 2: Monitoring operators

makes it possible to robustly monitor data sources for long periods of time. The Email, Fax, and Phone operators allow data to be accumulated and sent to recipients asynchronously. By its very nature, monitoring is a non-interactive process between user and agent and thus some form of offline propagation of updates is needed.

Extensibility operators. To increase the expressive power of the language, two additional operators – Apply and Aggregate – are included. Both are shown in Table 3. 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 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.

Name	Purpose
apply	Apply single row function to each relation tuple
aggregate	Apply multi-row function to relation

Table 3: Extensibility operators

Subplans

To promote language supports references to subplans reusability and to facilitate recursion (described later), the. Executing a subplan simply refers to the calling of one plan from another.

Recall that all plans are named and consist of a set of input and output streams. Thus, plans present the same interface as operators. It is thus a simple matter to refer to a plan as if it were an operator. For example, consider the example plan P1 introduced earlier. Figure 5 shows how another plan P2 can reference P1as a subplan.

Subplans encourage modularity and re-use. Once written, a plan can be used as an operator in any number of

```
PLAN P2 {
   INPUT: w, x
   OUTPUT: z

   BODY {
      Op5 (w : y)
      P1 (x, y : z)
   }
}
```

Figure 5: Calling a subplan

future plans. This effectively allows users to build whatever operators they need by combining the set of existing operators as necessary. At the same time, subplans can be easily scheduled as part of a dataflow-style plan and can benefit from data pipelining - just like any other typical plan operator does.

For example, one could develop a simple subplan called *Persistent_Diff*, shown in Figure 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. Such a subplan could be as an operator in many types of other plans. Note that executing a subplan does not force us to sacrifice the efficiency of dataflow execution and data pipelining: the Null and DbAppend operators execute at the same time that result is returned to the higher level plan; they also execute on data as soon as it becomes available from the Minus operator.

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 control flow.

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, we might use a *Do...While* loop to manage this type of information gathering need. However, under a dataflow-model of execution, such an approach in practice requires a fair of synchronization and additional operators.

Instead, this problem can be solved quite elegantly 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 basis for the exit condition.

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 7, posts the initial query to the search engine and retrieves the initial results. It then processes the results with a subplan called *Gather_and_Follow*. The search results themselves go to a Union operator and the next link is eventually used to call *Gather_and_Follow* recursively. The results of this recursive call are combined at the Union operator with the first flow.

There are a few notable aspects to the plans shown in

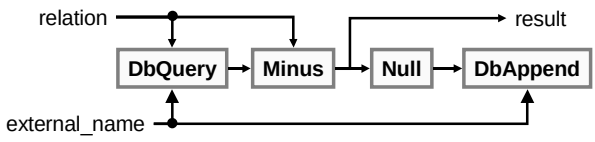


Figure 6: The *Persistent_Diff* subplan

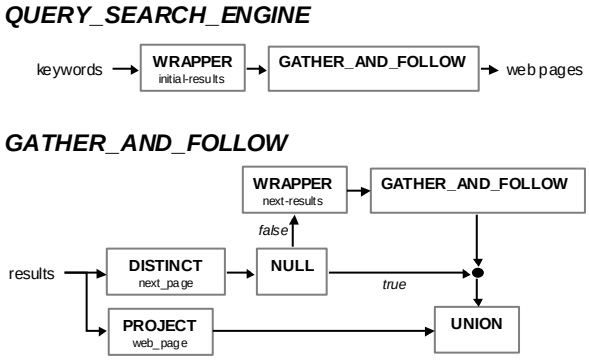


Figure 7: Example of recursion

Figure 7. First, a recursive approach requires very few operators: through the subplan facility, we are able to re-use the body of the gathering-and-following task. Second, data pipelining is exploited: even though recursive execution might go quite deep, results from higher levels are streamed out, back to the higher level *Query_Search_Engine* plan as soon as possible via the pipelined Union operator. Third, notice that we continue to merely require one type of conditional – the Null operator. When the last page is reached, Null routes the EOS to Union (and not to Wrapper, as it normally does). This ends the Union at the lowest level of recursion and this EOS trickles all the way back to the top of the plan, per standard tail-recursive execution.

Revisiting the example

Let us now revisit the earlier house search example and see how such a plan would be written with the proposed plan language. Figure 8 shows one of the two plans, *Get_Houses*, required to implement the abstract real estate plan in Figure 2. *Get_Houses* calls the subplan *Get_Urls*; this plan is nearly identical to the recursive subplan *Gather_And_Follow* in Figure 7, so it is omitted for the sake of brevity. The rest of *Get_Houses* works as follows:

a. 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.
b. A Minus operator determines which houses are distinct from those previously seen; new houses are appended to the persistent store.
c. Another Wrapper operator investigates the detail link for each house so that the full set of criteria (including picture) can be returned.
d. Using these details, a Select operator filters out those that meet the specified search criteria.
e. The result is aggregated and e-mailed to the user.

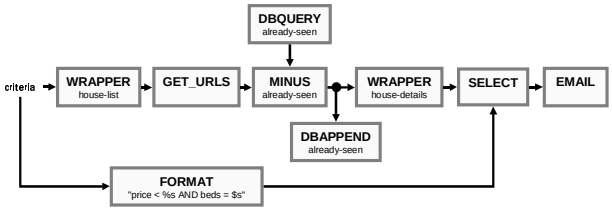


Figure 8: The *Get_Houses* plan

An Efficient Plan Execution Architecture

In this section, we describe an architecture that can efficiently execute the types of plans described in the last section. This architecture is composed of two parts: the language described in the previous section and a dataflow-style executor that efficiently processes these plans.

High-level design

The high-level design of the architecture is shown in Figure 9. The figure shows that the input to the executor is a plan; in addition, a schedule for execution (once, daily, hourly, etc) is input. During each execution, the plan may interface with a local database (e.g. to store tracking information). The figure also shows that it is possible for the plan to communicate updates through a variety of asynchronous communication mechanisms.

Once an input plan is received by the executor, it constructs an internal dataflow graph based on plan operators. At this time, any subplan and recursive relationships are resolved, merging in operators from those plans as appropriate. The system then feeds in input data and execution commences. The input data triggers a subset of plan operators to start firing; their execution and subsequent production trigger other operators that consume their output and so on. If the plan is interactive, output data will be immediately returned to the user as it is produced. Otherwise, it is assumed that the method of user notification (such as email) is already encoded in the plan.

Thus, in the Yahoo Real Estate example, a user can submit the main part of the plan, a set of input data shown, and the schedule of "daily" to the system. The plan will then be executed once (immediately) and an initial set of house search results will be e-mailed to the user. The plan will then be automatically run the next day.

Parallelism during execution

The executor uses threads to service operator execution, and thus functions similar to a threaded dataflow machine (Papadapoulous & Traub 1991). When a tuple becomes available (either via input or through operator production), a thread is assigned to execute a method on the consuming operator with that data. Threads are drawn from a fixed pool, to throttle excessive parallelism and prevent machine resources from being swamped.

The first time that input arrives for a particular operator, an initialization method for that operator is called. During this time, stateful data structures are initialized. All future firings may use this state data structure as is appropriate for that operator. For example, the Union operator uses the state to save all tuples that it has previously output so that it does not output duplicates. The Minus operator keeps this information as well as the entire set of rhs tuples in its state – thus, when new lhs tuples arrive, they are first compared to the rhs set and, if not in this set, output only if they have not been previously output. When EOS markers have been received on each of an operators' inputs, all accumulated state is deleted. State is maintained per level of iteration; thus re-entrant, recursive execution is guaranteed to be correct.

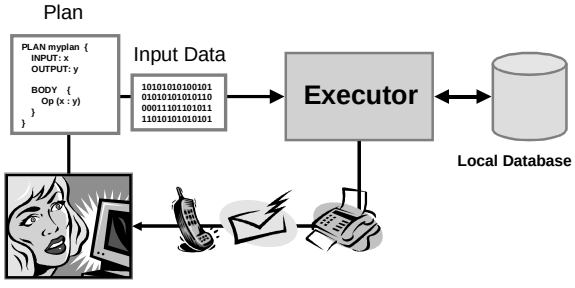


Figure 9: Executor design and interface

In summary, the executor we describe functions as a virtual threaded dataflow machine. By using threads to service operator firings, operator execution can be as horizontally parallel as the number of threads in the fixed pool. Furthermore, it is possible for a producer and consumer operator to fire concurrently on the same logical relation (the consumer operating on an earlier tuple while the producer operates on a later tuple in the stream) thus implementing a form of pipelined, or vertical parallelism.

Data coloring for re-entrancy. Recursion implies that plans are re-entrant and thus introduces an additional complexity – distinguishing data between recursive levels. To address this, the system assigns a color to all data at a particular logical level of execution. For example, during the execution of *Get_Houses*, the input data and any data produced at the same level as a result is assigned the same color. Whenever a subplan like *Get_Urls* is called (including when recursive calls are made), the tuples routed to that subplan are assigned a new color. This allows tuples at multiple levels of execution to be correctly managed by operators in the recursive subplan.

Related Work

In this section, we discuss two areas of related work: that of efficient Internet querying by network query engines and the set of more existing, more general, agent executors.

Network query engines. Recently, network query engines (Ives et al. 1999, Hellerstein et al. 2000, Naughton et al. 2001), have been proposed as means for efficiently gathering information on the Internet. These systems are mostly concerned with the efficiency of query execution. and have proposed adaptive execution strategies to reduce I/O latencies. Like the work described here, these systems represent plans as dataflow graphs and pipeline data between operators. The major difference between existing network query engines and the work described here is in terms plan expressivity. While network query engine research has proposed new operators related to XML processing and adaptive execution, they do not support operators that facilitate monitoring. These systems do not support conditional execution and it is not possible to loop through query results spread across multiple Web pages. In contrast, the plan language here supports conditional

execution, allows plans themselves to be operators, and supports recursion as means for looping during execution.

General plan executors. It is also useful to compare the work here to existing and more general plan execution systems. These systems have proposed highly concurrent execution models similar in spirit to dataflow machines. For example, RAPS (Firby 1994) described execution as a set of concurrent processes while PRS-Lite (Myers 1996) supported concurrent task execution as well as more complex synchronization and control flow. Both projects focused on specifying an event-driven mechanism for the parallel execution of partially-ordered plans – similar to execution of a dataflow graph. The work described here differs from these more generic architectures by focusing, like network query engines, specifically on plans that not only require the enablement of operators, but the routing of information between them. as well. Thus, our work is more closer in spirit to the unified approach of (Williamson et al. 1996), yet it extends that work by specifying an actual plan language, adding support for recursion and subplan execution, and by proposing a dataflow execution architecture.

Conclusion and Future Work

In this paper, we have described an information gathering plan language that promotes better expressivity while retaining the efficiency of traditional plan representation. Support for subplans and recursive execution allow plans to loop through query results that are spread across multiple Web pages. Operators that are extensible and are better integrated with the external world facilitate plans that are capable of monitoring an integrated set of remote sources for an extended period of time. Though expressive, the plan language is dataflow in terms of representation and its operators support the pipelining of data during execution. Thus, such plans can be efficiently executed.

We are currently investigating a method for speculative execution for information gathering plans (Barish & Knoblock 2002) that uses machine learning techniques to analyze data occurring early during execution so that predictions can be made about data that will be needed later in execution. The result is a new form of dynamic execution parallelism that can lead to significant speedups. We are also currently working on an Agent Wizard, which allows the user to define agents for monitoring tasks simply by answering a set of questions about the task. The Wizard will work similar to the Microsoft Excel Chart Wizard, which builds sophisticated charts by asking the user a set of simple questions. The Wizard will generate information gathering plans using the language described in this paper and schedule them for periodic execution.

Acknowledgements

The research here was supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contract/agreement numbers F30602-01-C-0197, F30602-00-1-0504, F30602-98-2-0109, 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 number EEC-9529152. 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.

References

Barish, Greg; Chen, Yi-Shin; Knoblock, Craig A.; Minton, Steven; and Shahabi, Cyrus. The TheaterLoc Virtual Application. *Innovative Applications in Artificial Intelligence (IAAI)*. 2000

Barish, Greg and Knoblock, Craig A. Speculative Execution for Information Gathering Plans. To appear, *AIPS-2002*.

Dennis, Jack B. First version of a data-flow procedure language, *Lecture Notes in Computer Science 19, pp362-376*. 1974.

Firby, R.J. Task Networks for Controlling Continuous Processes. *AIPS-1994*.

Friedman, Marc and Weld, Daniel S. Efficient execution of information gathering plans. *IJCAI-1997*.

Genesereth, Michael R.; Keller, Arthur M.; and Duschka, Oliver M. Infomaster: An information integration system. *SIGMOD-97*.

Hellerstein, Joseph M.; Franklin, Michael J.; Chandrasekaran, Sirish; Deshpande, Amol; Hildrum, Kris; Madden, Sam; Raman, Vijayshankar; and Shah, Mehul A. Adaptive query processing: technology in evolution. *IEEE Data Eng Bulletin 23(2)*. 2000.

Ives, Zachary G.; Florescu, Daniela; Friedman, Marc; Levy, Alon Y.; and Weld, Daniel S. An adaptive query execution system for data integration. *SIGMOD-1999*.

Knoblock, Craig A.; Minton, Steven; Ambite, Jose Luis ; Ashish, Naveen; Muslea, Ion; Philpot, Andrew G.; and Tejada, Sheila. The Ariadne Approach to Web-based Information Integration. *International Journal on Cooperative Information Systems (IJCIS) Special Issue on Intelligent Information Agents: Theory and Applications. Vol 10 (1-2): 145-169*. 2001.

Myers, Karen. A procedural knowledge approach to task-level control. *AIPS-1996*.

Naughton, Jeffrey F.; DeWitt, David J.; Maier, David.; et al. The Niagara Internet query system. *IEEE Data Engineering Bulletin, 24(2)*. 2001

Papadopoulos, Gregory M. and Traub, Kenneth R. Multithreading: A revisionist view of dataflow architectures. In *Proc of the 18th Intl Symposium on Computer Architecture*. 1991

Williamson, Mike; Decker, Keith; and Sycara, Katia. Unified Information and Control Flow in Hierarchical Task Networks. *AAAI Workshop: Theories of Action, Planning, and Ctrl*. 1996.