Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

STREAM DATA ANALYSIS IN PROLOG

D. Stott Parker

January 1989
CSD-890004

Stream Data Analysis in Prolog

D. Stott Parker

Computer Science Department
University of California
Los Angeles, CA 90024-1596

ABSTRACT

Today many applications routinely generate large quantities of data. The data often takes the form of a time series, or more generally just a *stream* – an ordered sequence of records. Analysis of this data requires stream processing techniques, which differ in significant ways from what current database query languages and statistical analysis tools support today. There is a real need for better stream data analysis systems.

Stream analysis, like most data analysis, is best done in a way that permits interactive exploration. It must support 'ad hoc' queries by a user, and these queries should be easy to formulate and run. It seems then that stream data analysis is best done in some kind of powerful programming environment.

A natural approach here is to analyze data with the stream processing paradigm of transducers (functional transformations) on streams. Data analyzers can be composed from collections of functional operators (transducers) that transform input data streams to output streams. A modular, extensible, easy-to-use library of transducers can be combined in arbitrary ways to answer stream data analysis queries of interest.

Prolog offers an excellent start for an interactive data analysis programming environment. However most Prolog systems have limitations that make development of real stream data analysis applications challenging.

We describe an approach for doing stream data analysis that has been taken in the Tangram project at UCLA. Transducers are implemented not directly in Prolog, but in a functional language called Log(F) that can be translated to Prolog. With Log(F), stream processing programs are straightforward to develop. A by-product of this approach is a practical way to interface Prolog and database systems.

This work done under the Tangram project, supported by DARPA contract F29601-87-C-0072.

Table of Contents

1. Stream Data Analysis 1

2. Basics of Stream Data Analysis 3
2.1. Basic Stream Processing 3
2.2. Kinds of Transducers 5
2.3. Composing Transducers 8
2.4. How Stream Processing Differs from List Processing 10
2.5. A Brief Survey of Stream Processing 11
2.6. The Stream Data Analysis Paradigm 13

3. Limitations of Using Prolog for Stream Data Analysis 15
3.1. Prolog Requires Lots of Memory 15
3.2. Prolog Implementations of Atoms are Inadequate 16
3.3. Stream Processing Requires Stack Maintenance 17
3.4. Prolog's I/O Mechanisms are Inadequate 19
3.5. Prolog's Control Mechanisms are Inadequate 22
3.6. Prolog has an Impedance Mismatch with Database Systems 23
3.7. Summary: Problems and Solutions 24

4. Log(F): A Rewrite Rule Language in Prolog 26
4.1. Overview of F* and Log(F) 26
4.2. F* – A Rewrite Rule Language 27
4.3. Log(F): Integrating F* with Prolog 29
4.4. Stream Processing Aspects of Log(F) 31

5. Stream Processing in Prolog 33
5.1. Implementing Transducers in Log(F) 33
5.2. The General Single-Input, Single-Output Transducer 34
5.3. Basic Statistical Time Series Analysis 36
5.4. Aggregate Computations 37
5.5. Computing Medians and Quantiles 39
5.6. Pattern Matching against Streams 42

6. Connecting Prolog with Databases 44
6.1. Stream Processing as a Common Model for Prolog and DBMS 44

6.2. Coupling Prolog and DBMS through Stream Processing 44
6.3. Implementation of Prolog-DBMS Coupling 45

7. Final Remarks 49

Stream Data Analysis in Prolog

D. Stott Parker

Computer Science Department
University of California
Los Angeles, CA 90024-1596

1. Stream Data Analysis

Most people spend a significant amount of their waking existence reflecting about what has happened to them. The human mind is very effective at recalling, replaying, and editing past events. It seems that hindsight is a natural part of survival, and it always will be. It would be nice if computers could somehow be enlisted to help shoulder some of this burden of reflection, or at least help process it more efficiently! In the following pages we will investigate how Prolog can help us.

Consider the following scenario. Suppose that we have recently acquired a great deal of cash, and wish to invest it in the stock market. Wary of stockbrokers, since they do not really seem to have our best interests at heart, we seek some kind of objective advice about which stocks really offer the potential for high returns. Of course, many kinds of publicly-available statistics exist about companies. However, an important summary of how a company is doing is the history of its stock price quotations. One approach to investment, known as the 'technical' approach, is to analyze this history in an attempt to gain some insight about the future behavior of the stock. Variations on this approach have made at least some people wealthy [25].

There are many ways to go about this analysis, but one important field to know about is *time series analysis*, which is concerned with mathematical techniques for investigating sequences of data [30]. Time series analysis provides methods for discerning different components of behavior in sequence data, including:

- real long-term trends
- seasonal variations
- non-seasonal cycles
- random fluctuations.

As stock investors we are basically interested in detecting long-term upward trends in prices. We certainly do not want to be distracted by random (short-term) fluctuations, and do not want to be fooled into mistaking cyclic behavior for long-term trends.

How can we discern trends in a history? A popular strategy is to compute *moving averages* of the history. The m-th moving average of a sequence of numbers S is just the sequence whose i-th element is the average of elements i, $i+1$, ..., $i+m-1$ in S. Moving averages smooth out fluctuations, and thereby (hopefully) expose real trends. They are simple, and are used very widely. For example, when its weekly closing price is above the 40-week moving average, the stock market is called a *bull* market, and otherwise a *bear* market.

Therefore, to decide if we want to invest in AT&T we might want to ask a query like

show the 5-day moving averages for AT&T stock in 1989

and have the answer displayed quickly in a graphic format. If there seemed to be an upward trend, we could follow up with other queries.

Unfortunately, the stock market does not stay in indefinite upward trends, but follows very pronounced cyclic patterns. Between 1929 and 1977 there were nine major bull markets, and nine major bear markets, so the average bull-bear market cycle is 5.3 years. Bear markets tend to last about two years, and bull markets about three years, so the probability being in a bull market has been about 60% for the past fifty years [27]. Cyclic components of behavior require more sophisticated techniques than moving averages, and require fairly powerful tools for effective analysis.

The stock investment scenario discussed here is an instance of the following general problem: we have an ordered sequence of data records that we wish to analyze. We call this ordered sequence a *stream* for the moment. At the very least, we want to get more "intuition" about the structure of the data. Generally we want to do much more, however, such as compare it with other streams or check its agreement with a model we have developed. We call this general problem *stream data analysis*.

Amazingly, today there is not nearly enough in the way of software to deal with this problem. Database management systems (DBMS) permit certain kinds of analysis of data. In modern DBMS users can use a query language (typically SQL) to ask specific questions. DBMS cannot yet handle stream data analysis, however.

We do not mean to imply that there is *no* software today that supports stream data analysis. We are now entering a period where powerful data analysis tools are being combined with external data sources. For example, DBMS are being combined with statistical analysis packages. This combination should support 'exploration' of the data in the way that exploratory data analysis systems do, like the *S* statistical analysis system of AT&T [6]. Also, time series analysis packages have grown in importance recently, as more applications of event data (historical data, temporal data) have developed [57].

Given all these developments, it is natural to ask about Prolog's potential for addressing this problem. Below, once we are armed with some clever techniques, we will see that it is possible to use Prolog very effectively for stream data analysis.

2. Basics of Stream Data Analysis

Stream processing is a popular, well-established AI programming paradigm. It has often been used in languages such as APL and Lisp, although it has certainly been applied in other languages. In this section we review basic stream processing concepts, and how they may be implemented naively in Prolog. This is enough to introduce the stream data analysis paradigm.

To keep our investigation simple, we begin with list processing. Although list processing is not the same as stream processing, it is similar, and a good place to start. Stream processing can be grasped with only a few extensions to list processing concepts.

2.1. Basic Stream Processing

A *stream* is an ordered sequence of data items. As Prolog users, we can initially conceptualize a stream as a list of terms. For example, the list

```
[
quote( 'F',   'Nov 04 1988',  50+5/8,   49+5/8,   49+5/8 ),
quote( 'IBM', 'Nov 04 1988', 121+3/8, 120+1/4, 120+1/4 ),
quote( 'TUA', 'Nov 04 1988',  29,      28+3/8,   28+5/8 ),
quote( 'X',   'Nov 04 1988',  28+1/8,   27+7/8,   27+7/8 )
]
```

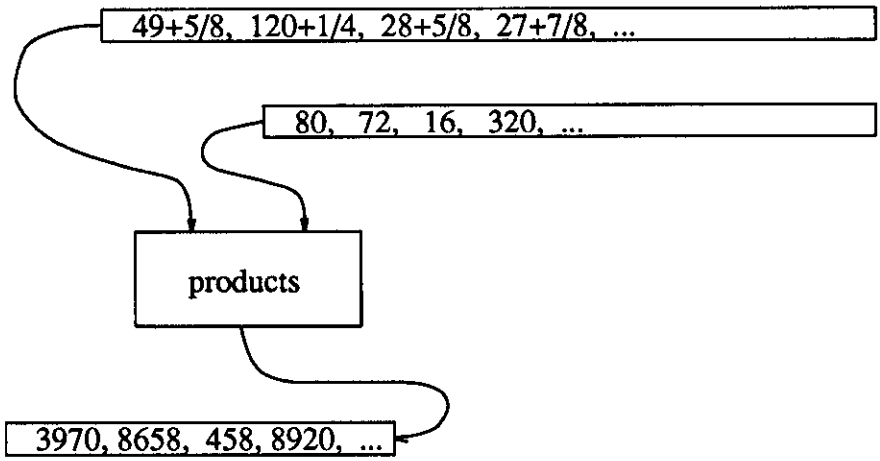
represents a stream with four items. Later on, it will be important that the stream contain only *ground* terms (terms with no variables), so to keep things simple let us assume from now on that all streams contain only ground terms.

A *stream transducer* is a function from one or more input streams to one or more output streams. For example, a transducer could map the stream above to the stream

```
[ 49+5/8, 120+1/4, 28+5/8, 27+7/8 ]
```

by selecting the final value from each quotation. This definition includes transducers that implement translations from streams to streams, aggregate computations, and other functions.

The figure below shows a simple `products` transducer that takes two streams of numbers as input, and produces a stream of products as output:



As transducers are functions, they may be composed in arbitrary ways to form new transducers. For example, the definition

```
portfolio_value => sum( products(closings(quotes), quantities(holdings)) ).
```

uses an expression involving the composition of several transducers. (When single-input, single-output transducers are combined in sequence, we get *pipelining*.) Stream processing is then just the paradigm in which compositions of transducers are used to both define and to manipulate streams.

Although we present them simply as functions, transducers actually relate closely to a number of other important programming paradigms, including:

- automata
- objects (as in object-oriented programming)
- actors
- parsers.

Automata, objects, and actors all accept a sequence of inputs which they use to change their internal state, while possibly issuing a sequence of outputs. Parsers take a sequence of symbols as input and produce a summary describing patterns they have recognized. These paradigms are important precisely because we use them frequently.

While transducers have features of each of the paradigms just mentioned, they are really more general. For example, since transducers can take parameters, and need not have only a finite number of states, it is not accurate to think of them as just automata. Perhaps a better way to look at transducers is as a *generalization* of automata, objects, actors, and parsers.

For the moment, then, let us define *stream processing* to be the programming paradigm in which transducers are composed to define or manipulate streams.

2.2. Kinds of Transducers

Certain kinds of transducers appear frequently, and are worth studying. Abelson and Sussman [1] point out four kinds of transducer: enumerators, maps, filters, and accumulators.

Enumerators

Enumerators (or generators) produce a stream of values. In Prolog, an enumerator could generally look as follows:

```
enumerate(Stream) :-
            initial_state(State),
            enumerate(State,Stream).

enumerate(S,[X|Xs]) :-
            next_state_and_value(S,NS,X),
            !,
            enumerate(NS,Xs).
enumerate(_,[]).
```

Here the `State` variable can be viewed as a parameter, or collection of parameters. For example, the following enumerator generates all the integers in a given range:

```
% intfrom(M,N,Stream) :-  Stream is the list of integers [M,...,M+N].

intfrom(_,0,[]) :- !.
intfrom(M,N,[M|L]) :-
            M1 is M+1,
            N1 is N-1,
            intfrom(M1,N1,L).
```

The parameters `M` and `N` can be thought of as state variables.

Maps

Maps transform an input stream to an output stream by applying a specific function to each element. We can write maps in Prolog directly as follows:

```
map_f([X|Xs],[Y|Ys]) :-  f(X,Y),  map_f(Xs,Ys).
map_f([],[]).
```

A generic version, which applies to any mapping, is as follows:

```
maplist([],F,[]).
maplist([X|Xs],F,[Y|Ys]) :-
            apply(F,[X,Y],FXY),
            call(FXY),
            maplist(Xs,F,Ys).

apply(F,Vs,FVs) :-
            F =.. FL,
            append(FL,Vs,FVsL),
            FVs =.. FVsL.
```

This implementation, however, is expensive since `=../2`, `append/3`, and `call/1` are expensive primitives. Prolog does not encourage this kind of coding, but instead encourages with transducers that do explicit tasks like the following:

```
% squares(L,NL) :-  NL is the result of squaring each member of L

squares([X|Xs],[Y|Ys]) :- Y is X*X, squares(Xs,Ys).
squares([],[]).


% project(I,L,NL) :-  NL is the stream of I-th arguments of terms in L

project(I,[X|Xs],[Y|Ys]) :- arg(I,X,Y), project(I,Xs,Ys).
project(_,[],[]).
```

Map transducers arise constantly in practice, since people tend to want to view any given data in many different ways. For example, note that the result of the Prolog goal

```
?-  L = [
quote( 'F',   'Nov 04 1988',  50+5/8,  49+5/8,  49+5/8 ),
quote( 'IBM', 'Nov 04 1988', 121+3/8, 120+1/4, 120+1/4 ),
quote( 'TUA', 'Nov 04 1988',  29,      28+3/8,  28+5/8 ),
quote( 'X',   'Nov 04 1988',  28+1/8,  27+7/8,  27+7/8 )
],
project(5,L,NL).
```

is the binding

```
NL = [ 49+5/8, 120+1/4, 28+5/8, 27+7/8 ]
```

giving the stream that we wanted earlier.

Filters

Filters transmit to the output stream only those elements from the input stream that meet some selection criterion. Filters are easy to write in Prolog:

```
filter([X|Xs],Ys) :-  inadmissible(X),  !,  filter(Xs,Ys).
filter([X|Xs],[X|Ys]) :-  filter(Xs,Ys).
filter([],[]).
```

A specific example of a filter is a transducer that, given a fixed integer `Q`, lets only non-multiples of `Q` pass to the output stream:

```
% non_multiples(L,Q,NL) :-  NL is the sublist of L of non-multiples of Q.

non_multiples([X|Xs],Q,NL) :- multiple(X,Q), !, non_multiples(Xs,Q,NL).
non_multiples([X|Xs],Q,[X|NL]) :- non_multiples(Xs,Q,NL).
non_multiples([],_,[]).

multiple(A,B) :- (A mod B) =:= 0.
```

We can also develop a generic filter called `select/4` that is something like `findall/3` in Prolog, but which works on streams. Its output consists of those elements in the input stream that simultaneously match a given template pattern and satisfy a given condition:

```
% select(L,Template,Condition,NL) :-  NL is the sublist of L of all
%           terms matching Template and also satisfying Condition.

select([X|Xs],T,C,Ys) :- \+ (X=T, call(C)), !, select(Xs,T,C,Ys).
select([X|Xs],T,C,[X|Ys]) :- select(Xs,T,C,Ys).
select([],_,_,[]).
```

For example, with this transducer the goal

```
?-  L = [
quote( 'F',   'Nov 04 1988',  50+5/8,  49+5/8,  49+5/8 ),
quote( 'IBM', 'Nov 04 1988', 121+3/8, 120+1/4, 120+1/4 ),
quote( 'TUA', 'Nov 04 1988',  29,      28+3/8,  28+5/8 ),
quote( 'X',   'Nov 04 1988',  28+1/8,  27+7/8,  27+7/8 )
],
select(L, quote(Symbol,Date,High,Low,Close), ((High-Low) > 1), NL).
```

yields the binding

```
NL = [quote( 'IBM', 'Nov 04 1988', 121+3/8, 120+1/4, 120+1/4 )]
```

of all quotes whose `High` and `Low` values differ by at least one point.

Accumulators

Accumulators compute 'aggregate' functions of the input stream. That is, they accumulate or aggregate all the elements in the stream into a single value. Generically, they might be written in Prolog as follows:

```
accumulate(List,Value) :-
            initial_state(State),
            accumulate(List,State,Value).

accumulate([X|Xs],S,Value) :-
            next_state(X,S,NS),
            accumulate(Xs,NS,Value).
accumulate([],S,Value) :-
            final_state_value(S,Value).
```

Perhaps the simplest example of an accumulator is a sum. Using the `State` variable to be the partial sum, we get the program below:

```
% sum(List,Sum) :-  Sum is the result of summing the elements in List

sum(List,Sum) :- sum(List,0,Sum).


sum([],V,V) :- !.
sum([X|Xs],OldV,V) :- NewV is X+OldV, sum(Xs,NewV,V).
```

Other Kinds of Transducers

It is important to realize that the four kinds of transducers above give only an important class, and do not make up all possible transducers. For example, the moving averages transducer below is not an enumerator, map, filter, or accumulator:

```
% moving_avg(M,L,NL) :-  NL is the list of Mth-moving averages of L
%   i.e., item [i] in NL is the average of items [i],...,[i+M-1] in L.

moving_avg(M,[],[]).
moving_avg(M,[X|Xs],[Y|Ys]) :-
            prefix_avg(M,[X|Xs],Y),
            moving_avg(M,Xs,Ys).

prefix_avg(M,L,A) :-
            prefix_sum(M,L,0,S),
            A is S/M.
prefix_sum(M,_,S,S) :- M =< 0, !.
prefix_sum(_,[],S,S) :- !.
prefix_sum(M,[X|Xs],S0,S) :-
            S1 is X+S0,
            M1 is M-1,
            prefix_sum(M1,Xs,S1,S).
```

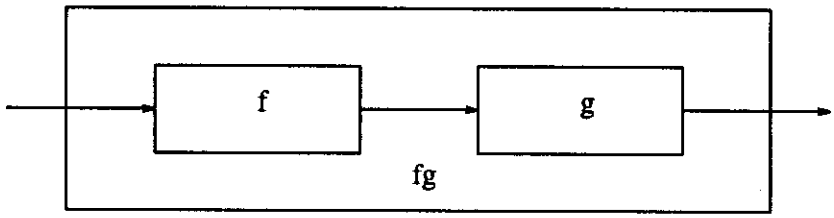
There are many different kinds of transducers, just as there are many different paradigms in programming.

2.3. Composing Transducers

Transducers can be combined with logical variables in Prolog. Given two transducers *f/2* and *g/2*, we can easily form their pipeline composition *fg/2* with a single rule:*

```
fg(S0,S) :- f(S0,S1), g(S1,S).
```

Displaying these compositions pictorially can often be suggestive. If we use boxes to represent transducers, then *fg* can be displayed as follows:



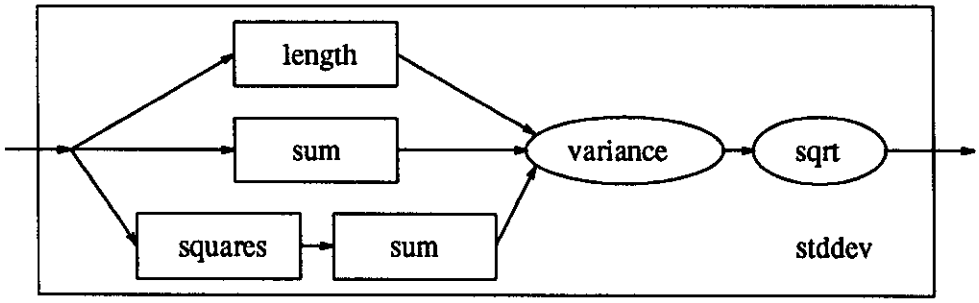
The arrows in these diagrams represent either streams or single parameters. Sometimes these displays are called dataflow diagrams, or 'Henderson diagrams' [1].

*Notice that this rule is the result of expanding the Definite Clause Grammar rule `fg --> f, g.` This is not a wild coincidence. We mentioned earlier that there is a close relationship between transducers and parsers. Much of the transducer code given in the previous section can be rewritten as DCG rules.

This much is really pretty obvious to anyone that has programmed in Prolog. What is perhaps not so obvious to the Prolog programmer is the power of combining transducers from a library, or 'kit'. Through composition, we can build up large numbers of useful transducers. This kit concept is one of the main advantages of the stream processing paradigm: sophisticated transducers can be built up from simpler existing ones. For example, the transducer below computes standard deviations using the `sum/2` and `squares/2` transducers we developed earlier.

```
% stddev(List,Stddev) :-  Stddev is the standard deviation of List

stddev(List,Stddev) :-
            length(List,N),
            sum(List,Sum),
            squares(List,SqList),
            sum(SqList,SumSquares),
            variance(N,Sum,SumSquares,Variance),
            sqrt(Variance,Stddev).

variance(N,_,_,0) :- N =< 1, !.
variance(N,Sum,SumSq,Variance) :-
            Variance is (SumSq - (Sum*Sum/N)) / (N-1).
```

Diagrams can display the composition of transducers nicely. The figure below shows the composition of transducers in `stddev`:



This is not the fastest transducer to form the standard deviation. A faster way would be to write a single transducer to accumulate the length, sum, and sum of squares of the list simultaneously:

```
faster_stddev(List,Stddev) :-
            count_sum_sumsq(List,0,0,0,N,Sum,SumSquares),
            variance(N,Sum,SumSquares,Variance),
            sqrt(Variance,Stddev).

count_sum_sumsq([],N,S,Q,N,S,Q).
count_sum_sumsq([X|Xs],N0,S0,Q0,N,S,Q) :-
            N1 is N0+1,
            S1 is S0+X,
            Q1 is Q0+X*X,
            count_sum_sumsq(Xs,N1,S1,Q1,N,S,Q).
```

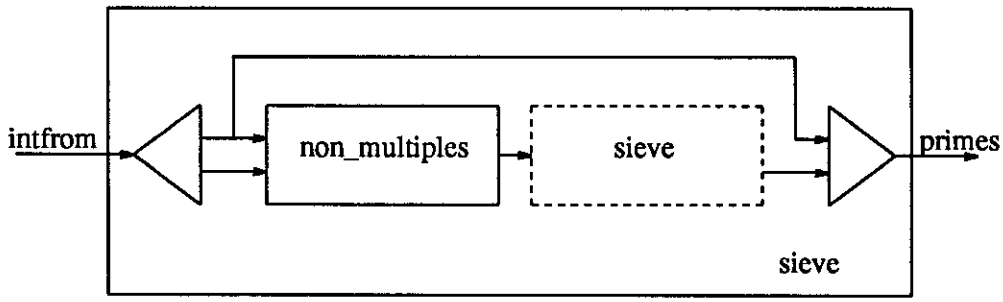
The point, however, is that new transducers can be constructed quickly by composing

transducers developed already.

Transducers can also be combined *recursively*. A classic example of recursive composition is in the computation of primes via Eratosthenes' method of sieves:

```
% primes(M,List) :-  List is the list of primes not larger than M.

primes(M,Primes) :- N is M-1, intfrom(2,N,Ints), sieve(Ints,Primes).

sieve([X|Xs],[X|R]) :- non_multiples(Xs,X,L), sieve(L,R).
sieve([],[]).

non_multiples([Y|Ys],X,NL) :- multiple(Y,X), !, non_multiples(Ys,X,NL).
non_multiples([Y|Ys],X,[Y|NL]) :- non_multiples(Ys,X,NL).
non_multiples([],_,[]).

multiple(A,B) :- (A mod B) =:= 0.
```

This defines the enumerator `primes/2` to be a combination of an enumerator `intfrom/3` and a recursive filter `sieve/2`. Displaying `sieve` diagrammatically requires some innovation. The diagram below of Henderson [38] represents the recursive use of `sieve` with a dashed box, and the breakdown of the stream into its head and tail (through use of the 'cons' functor `[_|_]`) with a triangle.



2.4. How Stream Processing Differs from List Processing

We have treated streams like lists above to introduce the ideas of transducers and stream processing. Two important things differentiate stream processing from list processing.

(1) *Infinite streams.*

A stream may represent an infinitely long (non-terminating) sequence of values. For example, the following sequences are infinite streams:

```
[1,1,1,1,1,...]   % the stream of ones
[1,2,3,4,5,...]   % the stream of positive integers
[2,3,5,7,11,...]  % the stream of prime numbers
```

In Prolog, some infinite streams can be constructed as infinite lists, by exploiting features of Prolog's unification. For example, executing

```
?- Ones = [1|Ones].
```

quickly creates a structure representing infinite list of ones.* However, this is not a general

*Using unification to create such 'circular terms' can be dangerous, unfortunately, so we have to discourage it. First, printing the results of executing this goal (its bindings) will never terminate in

way to create infinite lists, since it cannot be used to produce the stream of prime numbers, for example.

(2) *Lazy evaluation.*
Stream processing includes the use of *lazy evaluation,* a 'demand driven' mode of computation in which elements of a stream are produced by a transducer only on demand. Although programs are written as though the stream is completely available, actually the elements of the stream are produced only incrementally. (This permits us to use infinite streams.) Thus we integrate a view of the stream as a single object with a view of it as a sequence of objects. Lazy evaluation is a computation scheme in which goals are evaluated only on demand.

By contrast, *eager evaluation* is a mode of computation in which elements of a stream are produced immediately. Most programming languages, including Prolog, normally act in an eager way. That is, normal behavior is to evaluate a goal to completion when it is given to the Prolog interpreter.

Lazy evaluation permits us to apply some algorithms that are difficult to use otherwise. For example, lazy evaluation can be used with the algorithm above to compute the infinite list of all prime numbers. Each time a new prime number is demanded of `sieve`, it produces a new call to itself and a call to the filter `non_multiples`, which eliminates all multiples of the head of the stream that was passed to its `sieve`. All computation can be done on demand.

The list processing programs above cannot handle infinite streams, and do not implement lazy evaluation. For example, if we tried to compute the stream of all prime numbers with the Prolog program above, we would never get past the `intfrom/3` subgoal! Instead, we would spend our lives developing some prefix of the infinite list of integers.

Although list processing can be accomplished easily in Prolog, it is not so obvious how stream processing can be accomplished in Prolog. We will come back to this problem shortly.

2.5. A Brief Survey of Stream Processing

Stream processing is a technique that has traditionally been used in languages such as APL and Lisp. It is a powerful paradigm that can be viewed in several different ways. The reader reviewing the literature is at first struck by the number of approaches that have been taken towards it:

(1) Streams are lists, whose successive elements are evaluated lazily (i.e., evaluated only on demand).

---

many Prolog systems. Second, unification without the occur check can lead to unsound inferences. For example, with the program

```
less_than(X,1+X).
surprise :- less_than(1+N,N).
```

the goal `?- surprise` will succeed quietly, giving us the impression that there exists a number `N` such that `1+N` is less than `N`. Third, and most seriously, in goals such as

```
?- X = [1|X], Y = [1|Y], X = Y.
```

the third unification will loop without halting in many Prolog implementations. We need an approach for representing infinite streams that avoids these problems.

(2) Streams are functions yielding a pair, whose first element is a value and whose second element is a closure (a functional expression to be evaluated, and a context or environment in which to evaluate it).

(3) Streams are just sequences of values. Functions that operate on streams iteratively can be composed and optimized using existing compiler optimization techniques.

(4) Streams are special instances of infinite objects. These infinite objects require special computation strategies, including lazy evaluation.

(5) Stream processing is a kind of coroutining, allowing different parts of a computation to be suspended and resumed as necessary.

(6) Stream processing is a kind of dataflow processing, in which the objects being processed are sequences.

(7) Stream processing is a kind of 'normal-order reduction', in which computation proceeds by repeatedly evaluating the outermost, leftmost subexpression of a larger expression.

These points of view are equivalent, but they have different emphases that can lead to very different implementations.

One of the first presentations of stream processing was explored by Burge [16], an elegant tour of examples showing the potential of the paradigm, which appeared in more expanded form in [17]. Burge viewed streams as functions, using the second approach above. He drew directly on the presentation of Landin in [43] (cf. in particular p.96), and Burge credits Landin as being the first to formalize the notion of a stream in 1962.

In 1976 the concept of lazy evaluation was popularized as an alternative evaluation scheme for functional languages [29, 37]. In chapter 8 of his book on functional programming [38], Henderson summarized basic material on lazy evaluation known at the time, and showed how it corresponds directly to coroutining among function evaluations. This has the benefit that certain computations in networks of processes [41] can be shown to be evaluated neatly with lazy evaluation. An extended presentation of the material in Henderson's book, emphasizing stream processing programming techniques, is in section 3.4 of [1]. A recent summary of work on laziness and stream processing can be found in Chapters 11 and 23 of Peyton-Jones' book [60], which among other things discusses the implementation of Miranda, a lazy functional language. Today lazy evaluation is part of a number of functional programming languages.

A rather different view of stream processing is offered by Goldberg and Paige [33]. The authors take the point of view that 'stream processing is a technique that seeks to avoid intermediate storage and to minimize sequential search'. From this perspective, stream processing amounts basically to *loop fusion*, an optimization technique presented by Allen and Cocke for combining consecutive Fortran do loops into a single loop in [2]. Goldberg and Paige give a history of stream processing from this perspective, including applications that have been proposed for it in file processing, data restructuring, and above all database query processing. This point of view is echoed by Freytag for database query processing in [28].

Independently, stream processing in databases has been proposed a number of times in the past. First, streams naturally arise in functional data models, and a number of important functional

models have been developed in the past decade [47, 62, 63]. Buneman's functional query language FQL, reminiscent of Backus' FP language, included lazy evaluation [13, 15]. Recent proposals for including lazy evaluation in databases include [35, 36].

2.6. The Stream Data Analysis Paradigm

In stream data analysis, we put all of our data in the form of streams, and all of our analysis tools in the form of transducers. Just as in a functional programming environment, these transducers should be viewed as basic building blocks that can be accessed whenever necessary by data analysts.

A sample list of transducers is presented below, using functional syntax. Although they seem insignificant individually, as a group they become quite powerful.

Expression	Result
agg(Op,S)	result of applying infix (binary) operator Op to stream S Op can be +, *, ∧, ∨, min, max, avg, sum, count, etc.
append(S1,S2)	concatenation of S1 and S2
avg(S)	average of terms in stream S
comparison(Rel,S1,S2)	stream with ith value 1 if (S1[i] Rel S2[i]), otherwise 0 Rel can be =, <, >, =<, >=, ==, \=, =\=, \==, etc.
constant_stream(C)	infinite stream of constant C
constant_stream(C,N)	stream of constant C of length N
length(S)	number of terms in stream S
difference(S1,S2)	difference of S1 and S2
distribution(D)	stream of values following distribution description D
first(N,S)	stream of the first N elements of stream S
intersect(S1,S2)	intersection of S1 and S2
intfrom(N)	infinite stream [N, N+1, N+2, ...]
intfrom(N,M)	finite stream [N, N+1, N+2, ..., M]
keysort(S)	result of keysorting stream S
lag(S,N)	stream with constant 0 repeated N times, followed by S
maplist(F,S)	stream of results F(X) for each term X in stream S
max(S)	maximum value of terms in stream S
merge(S1,S2)	interleaving of S1 and S2
min(S)	minimum value of terms in stream S
moving_avg(M,S)	M-th moving average of stream S
naturaljoin(S1,S2)	join of S1 and S2
project(N,S)	stream of Nth arguments of each term in stream S
repeat(S,N)	stream with every element of S repeated N times
reverse(S)	reversal of stream S
select(T,C,S)	stream of terms in S that match T and satisfy C
sort(S)	result of sorting stream S
stddev(S)	standard deviation of terms in stream S
sum(S)	sum of terms in stream S
sumsq(S)	sum of squares of terms in stream S
union(S1,S2)	union of S1 and S2

This list resembles that of the *S* data analysis system [6], which extends it only with a few statistical operators. Collections like this one can be useful in many surprising contexts. Waters [68] showed that sixty percent of the programs in the Fortran Scientific Subroutine Package can be viewed as single-input, single-output stream transducers of the kinds we discussed earlier.

It is important to mention that any ordered sequences of data items can be treated as a stream. For example, arrays of data can be treated like streams. Thus many matrix algebra operators

also can fit in the stream data analysis paradigm. This is sometimes cited as a grounds for success of the APL programming language, NIAL [46], and the Nested Array model of data upon which both are based [48, 49]. These languages include not only matrix algebra operators and the list of operators above, but also the ability to define higher-order operators on streams, such as aggregate operators (*min*, *max*, *sum*, etc.), APL's reduction operator, *maplist*, etc.

The stream data analysis paradigm has proven itself in both statistical and scientific areas. It is now migrating to other fields, as increasingly better understanding of data is required by all applications in science and business. Thus, it is important to be able to capture this paradigm in Prolog.

3. Limitations of Using Prolog for Stream Data Analysis

At first glance, Prolog seems to be an excellent starting point for stream data analysis. There are at least two reasons for optimism:

(1) Today, Prolog is arguably the best existing candidate for a language combining data processing with 'intelligent' analysis functions. It integrates relational database functionality with complex structures in data, and its logical foundations provide many features (unification and pattern matching, logical derivation and intensional query processing, backtracking and search, and more generally declarativeness) important in high-level analysis. It naturally supports 'expert system' techniques for interpretation of data.

(2) Prolog is flexible. It is an outstanding vehicle for rapid prototyping, and permits access to systems that perform computationally intensive tasks better than it. Much of data analysis is of a 'rapid prototyping' nature – one wants to 'get a feel' for the data, without spending a great deal of time doing it.

Nevertheless, Prolog has real limitations for large-scale applications such as stream data analysis. A number of complaints have been leveled against today's Prolog systems for stream processing (and other applications), including:

- Prolog requires lots of memory
- Prolog implementations of atoms are inadequate
- Prolog requires stack maintenance
- Prolog's I/O mechanisms are inadequate
- Prolog's control mechanisms are inadequate
- Prolog has an impedance mismatch with database systems.

We will see how most of these problems can be circumvented. Some of the problems may be only transitory, and will disappear with better Prolog implementations, cheaper memory, and new operating systems. Also, some of these problems are not Prolog-specific at all, but rather problems that arise in most high-level languages. In any case, programmers that are interested in stream processing applications should be aware that the problems exist, and they are very serious problems in some systems today.

3.1. Prolog Requires Lots of Memory

Prolog systems today consume a relatively large address space. Every Prolog image is built with the following areas:

(1) *The Control Stack* (sometimes called the 'local stack')
Like most programming languages, Prolog uses a stack to manage calls to predicates. When a predicate call occurs, a frame is pushed on the stack to record the call and the arguments passed. The control stack is also commonly used to hold 'choice point' frames to implement backtracking.

(2) *The Heap* (sometimes called the 'global stack')
Prolog goals will create structures as a result of unification against program clauses. These structures are created by allocating cells incrementally from the top of the heap. Since the structures can be freed as soon as the goal that created them fails, the heap can be implemented as a stack that is popped on failure.

(3) *The Trail*
All bindings that are made in the process of executing a goal must be undone if the goal fails. The unification routine uses the trail as a stack to hold information that tells how to undo these bindings when failure occurs.

(3) *The Atom Table* (sometimes called the 'name table')
The names (character string representations) of all atoms in Prolog are stored in a large table.

(4) *The Database* (sometimes called the 'code space')
All asserted clauses are stored as data structures in an area called the database.

These areas can each grow to be quite large. As a consequence, moderately large Prolog programs run as processes with a memory image of several megabytes. Even though Prolog programs do show locality in their memory references [66], machines without substantial amounts of memory may produce poor Prolog performance due to swapping overhead. A related problem is that since most Prolog implementations do not share text among processes, it can be very difficult to run several large Prolog processes on one machine. Parallel Prolog systems may eventually eliminate this problem.

3.2. Prolog Implementations of Atoms are Inadequate

Prolog systems typically operate without a string datatype. Instead, atoms are used to represent strings in most situations. An atom is essentially a pointer to a string in the atom table. As new atoms are introduced into the Prolog system, they are *interned,* i.e., entered into the atom table and replaced by a pointer. The main benefit of this implementation is speed: checking whether two atoms unify can be accomplished very quickly with just a comparison of pointers (one machine instruction), while checking whether two strings are equal requires calling a subroutine on most machines. This implementation of atoms is basic to most Prolog implementations today, and cannot really be changed.

Usually, when string manipulations *must* be performed in Prolog, lists of integers (representing ASCII codes) are used to hold strings. The `name/2` predicate is used for conversion between atom and list representations. This approach to string manipulation is, of course, quite elegant. String processing is accomplished with only minimal extension of the language, resting on the existing atom mechanism and list processing capability. However, one problem with this approach is that it is extremely inefficient. A more serious problem is that it makes Prolog unable to cope with large-scale data processing, since Prolog systems can store only a modest amount of string data in the atom table.

If we connect a stream of data with Prolog, the strings in the stream will not be in the Prolog atom table when they are encountered. The question is: *should strings in the input stream be converted to Prolog atoms as they are encountered?* If the answer is no, we will pay a price in unifying Prolog atoms with these strings, and at the very least will have to extend the internal unification primitives to handle strings. If the answer is yes, we run the risk of overflowing the Prolog atom table. (And: when the stream is large, this risk is not just a risk, but a certainty.)

To appreciate this problem, consider the following very simple program:

```
% print_stream(Filename) :-  all terms in file Filename are printed

print_stream(Filename) :-
            open(Filename,read,Descriptor),
            read(Descriptor,Term),
            print_stream_term(Term,Descriptor).

print_stream_term(end_of_file,D) :-
            !,
            close(D).
print_stream_term(T,D) :-
            print(T),
            nl,
            read(D,NT),
            print_stream_term(NT,D).
```

This innocuous-looking program will not work for most large files. Whenever the terms it reads contain enough atoms to overflow the atom table, it will abort in failure.

Many approaches have been proposed to answer the question above. They include adding a string datatype to Prolog, and adding periodic garbage collection of the atom table. These approaches require significant amounts of work, unfortunately, and are unavailable in most Prolog systems [55].

3.3. Stream Processing Requires Stack Maintenance

Stream processing programs are basically iterative: the program repeatedly accepts a new input from a stream, does something with it, and incorporates the input into its current 'state' and/or produces some output stream element based on the input and its state.

Unfortunately, it turns out that iteration in Prolog (and Lisp, for that matter) can cause stacks to overflow. Specifically, when iterative programs read streams that are arbitrarily long, it is likely that Prolog stack overflows will occur before the program finishes. To guarantee a program will work for arbitrarily long streams requires some understanding of stack maintenance.

There are basically two ways to write iterative programs in Prolog:

- recursion
- repeat-fail loops.

Recursion is the natural way to perform iteration. The following simple program sums a list recursively:

```
% sum(List,Sum) :-  Sum is the result of summing the elements in List

sum([],0) :- !.
sum([X|Xs],Sum) :- sum(Xs,XsSum), Sum is X+XsSum.
```

Unfortunately this program has a serious problem: when the input is a very long list, the program can overflow the control stack. Each time a list member is inspected by the `sum` predicate, its recursive call allocates a new control stack frame. This recursion will lead to control stack overflows when the input list is very long.

Happily it turns out that we can avoid control stack overflows by using *tail recursion.* A tail recursive version of the program above looks as follows:

```
% sum(List,Sum) :-  Sum is the result of summing the elements in List

sum(List,Sum) :- sum(List,0,Sum).

sum([],V,V) :- !.
sum([X|Xs],OldV,V) :- NewV is X+OldV, sum(Xs,NewV,V).
```

This program is 'tail recursive' in `sum/3`. In other words, the last goal in the last clause of `sum/3` is a recursive call to `sum/3`. The importance of tail recursion is that the control stack frame for a tail recursive predicate is re-used by the recursive subgoal. With the program above, this means that we can sum an abitrarily long list with only one control stack frame for `sum/2`, one for `sum/3`, and whatever is needed by `is/2`. This reclaiming of control stack frames is called *Tail Recursion Optimization (TRO).** TRO thus solves some problems of control stack usage.

Generally speaking, it is a good idea to write iterative programs as tail recursive programs. Unfortunately these programs can still have storage use problems. Tail recursive programs can allocate cells on the Prolog heap for data structures constructed at each step of the recursion. When the stream being processed by a tail recursive program is very long, we run up against the problem of heap overflows.

For example, the program

```
squares([X|Xs],[Y|Ys]) :- Y is X*X, squares(Xs,Ys).
squares([],[]).
```

creates a list of squares of length as long as the first. But both lists remain in their entirety on the heap until the goal that called `squares/2` fails.

To avoid heap overflows, we must use a Prolog system with *Garbage Collection (GC).* Where TRO permits re-use of control stack frames by iterative programs, GC permits storage structures allocated by previous iterations to be reclaimed. Basically, it looks through the heap for data structures that are no longer actively pointed to by variables, marks their cells for reclamation, then compresses the heap by reclaiming all marked cells.

Not all Prolog systems provide complete GC and TRO. (In fact, Prolog *interpreters* can defeat TRO and GC even though the underlying system supports them for compiled code.) But without them, stream processing cannot be done with recursive programs.

---

*More accurately, it could be called *Last Call Optimization,* since the control stack frame for a goal can always be reclaimed when calling the last subgoal in the last clause for the goal. For example, with the program

```
p :- q, r.
p :- s, t, u.
p :- v, w.
```

if in executing the goal `?- p` we get to the subgoal `w`, then the control stack frame allocated for `p` can be re-used as the frame for `w`. At that point, execution of `p` is complete except for executing `w`, so `p`'s control stack frame can be used by `w`.

There is an alternative to recursion that can be used successfully when we are reading the stream from a file: *repeat-fail loops*. The sum program above can be written with a loop that repeatedly reads a number and adds it to the current sum.

```
% sum(Sum) :-  Sum is the result of summing all numbers read.

sum(Sum) :-
            asserta('%%sum'(0)),
            repeat,
                        read(X),
                        add(X),
                        X == end_of_file,
            retract('%%sum'(Sum)),
            !.

add(X) :-
            number(X),
            retract('%%sum'(OldSum)),
            NewSum is X+OldSum,
            asserta('%%sum'(NewSum)),
            !.
add(_).
```

Repeat-fail loops are inelegant, to say the least. Worse than this, they are usually awkward to write, and require the programmer to use `assert` and `retract` (or other primitives with side-effects) to save information obtained in the loop. Both `assert` and `retract` are extremely slow, requiring milliseconds to complete in most Prolog systems. Repeat-fail loops do have one important feature, however: they will not overflow the Prolog stacks. The only storage danger is that a program may overflow the database area provided for `asserted` clauses.

So, for doing iteration on streams we have a choice. If we wish to write our programs as recursive list manipulators, then they must be written tail recursively, and we must use a Prolog system with Tail Recursion Optimization and Garbage Collection. Alternatively we can write programs as repeat-fail loops around either I/O calls, `asserts` and `retracts` (or some other primitives with side effects). These choices are not attractive.

3.4. Prolog's I/O Mechanisms are Inadequate

Real data streams often come from files and external devices, sometimes available over computer networks. Therefore, real stream processing requires a comprehensive set of I/O primitives. No Prolog standard currently exists for these, and many Prolog systems have little in the way of I/O support. This means that, first of all, I/O primitives must be added as needed to the Prolog system as 'foreign functions'.

For example, if we wanted to perform database I/O, at minimum we would need to add something like the following primitives:

```
open_relation(Relation/Arity,Cursor)
```

Opens the database relation `Relation/Arity` for sequential retrieval, returning the descriptor `Cursor` for subsequent accesses.

`retrieve(Cursor,Term)`

Retrieves a tuple from the database relation indicated by `Cursor`. The tuple is returned in `Term`, and takes the value `end_of_file` if no further tuples remain. This predicate does not backtrack. Although `Cursor` does not change after the call, subsequent retrievals will obtain the next tuple in the relation.

`close_relation(Cursor)`

Terminates retrieval from the relation indicated by `Cursor`.

An interface like the one provided by these predicates accomplishes basically what we need for database I/O, but hardly resolves the problem.

An unfortunate problem is that I/O managers expect client programs to manage their file descriptors and cursors, while Prolog's control model permits it to forget about them completely. Specifically, a Prolog goal's control frame can be popped off the stack before all choices of records in the table have been exhausted, leaving the cursor or file descriptor 'dangling'. Prolog's stack-oriented implementation makes it difficult to figure out when to close a cursor or file descriptor that has been opened.

For example, consider the following program.

```
my_portfolio_value(Date,Value) :-
            quote('F',Date,_,_,FPrice),
            quote('IBM',Date,_,_,IBMPrice),
            quote('TUA',Date,_,_,TUAPrice),
            quote('X',Date,_,_,XPrice),
            Value is FPrice*80 + IBMPrice*72 + TUAPrice*16 + XPrice*320,
            !.

%  transparent interface to relational database from Prolog
quote(Symbol,Date,High,Low,Close) :-
            open_relation(quote/5,Cursor),
            repeat,
                 retrieve(Cursor,Term),
                 (
                      Term = quote(Symbol,Date,High,Low,Close)
                 ;
                      Term = end_of_file,
                      close_relation(Cursor),
                      !
                 ).
```

Although this program runs correctly, every time we use `portfolio_value/2` we create four cursors and leave them open!* After computing a several portfolio values we can use up all the

*This concern over cursors might seem unimportant, but it can be deadly serious. In shared database systems, the act of opening a relation for read access will obtain a *read lock* for the relation. Forgetting to close the cursor is then the same as forgetting to unlock the relation, which will

cursors provided by the database interface.

Improved Prolog technology has provided a solution to this problem. When the SICStus Prolog goal `undo(G)` is executed, `G` will be executed whenever the goal's control frame is popped off [18]. Thus if we change the definition of `quote/5` above to the following program, cursors will be reclaimed as soon as the goal that initially called `quote/5` fails:

```
quote(Symbol,Date,High,Low,Close) :-
            open_relation(quote/5,Cursor),
            undo( close_relation(Cursor) ),
            %  this close_relation goal is executed when the
            %  current quote/5 goal is popped off the stack.
            repeat,
                 retrieve(Cursor,Term),
                 (
                      Term = quote(Symbol,Date,High,Low,Close)
                 ;
                      Term = end_of_file,
                      !
                 ).
```

The `undo/1` primitive is very useful, and will find its way into more Prolog systems in the future.

Another serious problem is that I/O primitives like `read` typically cannot be used directly by nondeterministic programs, because they have side effects. Each read from a cursor modifies the cursor. There is usually no way to 'unread' what one has read, so backtracking programs such as nondeterministic parsers will not work with primitives like `read`.

The problems just mentioned can be summarized in a more general way: *the concept of cursors has not yet been cleanly integrated into Prolog*. Only recently have `open-read-write-close` primitives been added to the language, supplementing the simple `see-read-seen` and `tell-write-told` primitives that were available. However, this extension is incomplete. For example, in Prolog there is still no cursor-oriented interface to the Prolog database [56]. The only direct interface to the database is through the `clause` primitives. Since this interface works only by backtracking, it is not possible to implement predicates like `bagof/3` and `findall/3` without using primitives like `assert` that have side effects. These problems could be avoided if we had an interface like the one above for external database relations that could be used for predicates:

```
open_predicate(Predicate/Arity,Cursor)
retrieve(Cursor,Clause)
close(Cursor)
```

We are not arguing that the interface should look precisely like this; we are pointing out only that no such interface exists, and that Prolog should be extended with one.

---

prevent anyone from modifying the relation, and probably cause deadlocks.

Most Prolog systems lack other important I/O features. Specifically, they have no interrupt handling mechanisms or asynchronous I/O primitives. These features are especially important in stream applications where high performance is critical. Fortunately, Prolog systems may permit these extensions without serious changes.

3.5. Prolog's Control Mechanisms are Inadequate

General stream processing programs must selectively read data items from multiple input streams, process these items, and then selectively write multiple output streams. To do this requires the ability to *coroutine* among stream processing goals. That is, the execution of producers of streams must be interleaved with the execution of their consumers.

It is not obvious how to achieve coroutining in Prolog. In ordinary Prolog systems, backtracking is the only control mechanism for clause selection, and this prevents interleaved execution of goals.

An example will make this point clear. Suppose that `stock/3` and `quote/5` are relational database predicates whose clauses are stored in sorted order by their stock symbol, and we wish to find all results of the Prolog goal

```
?- stock(Symbol,Name,Address),
   quote(Symbol,'Oct 19 1987',Price,_,Price).
```

This will give us the stocks whose closing price matched its high price on Black Monday. Finding all results with backtracking will require looking through all of the clauses of the `quote` predicate for each clause in the `stock` predicate. This takes time proportional to the *product* of the number of clauses of the predicates.

A much better way to perform this query is by performing what in the database field is called a *merge join*. A merge join of two predicates with clauses in sorted order works by interleaving (i.e., coroutining) a sequential scan through the clauses of the two predicates, in precisely the same way that a merge sort interleaves scans through two sorted files. It takes time proportional to the *sum* of the number of clauses of the predicates. For the goal above, a merge join would work by repeatedly:

(1) retrieving the next clause `stock(Symbol,Name,Address)` from the `stock` predicate;

(2) given the `Symbol` value obtained in step (1), retrieving (scanning for) the next clause matching `quote(Symbol,'Oct 19 1987',Price,_,Price)` and yielding it as a result;

(3) on backtracking, resuming the scan for `quote` clauses as long as further results are found;

(4) as soon as the `quote` scan fails to match, however, resuming the scan for the next `stock` clause in step (1) above.

The specific problem we are pointing out is that Prolog is currently not capable of performing merge joins for predicates in its database. Fortunately, Prolog can be extended to do so by adding a cursor-like interface to the database [56], like the one described in the preceding section. The more general problem we are concerned with is that coroutining is not an easy thing to do in Prolog. Shortly we will see how coroutining can be implemented by developing a stream

processing meta-interpreter with better control mechanisms.

Prolog systems supporting coroutining as a basic feature have been proposed [24], but these systems are complex and are not generally available. Also, many stream processing extensions of Prolog have been proposed in the past few years. For example, many parallel logic programming systems have been developed essentially as stream processing systems. Typically, these systems fall into one of several camps:

(1) They resemble PARLOG [23] and the other 'committed choice' parallel programming systems (Concurrent Prolog, GHC, etc.).

(2) They introduce '*parallel and*' or '*parallel or*' operators into ordinary Prolog [44].

(3) They are extended Prolog systems that introduce streams by adding functional programming constructs [26, 42, 45, 50, 65]. The thrust of this introduction is to make Prolog more like either Lisp or Smalltalk or both.

Our approach is like that of the third camp, but is more conservative in that no real change or extension is made to Prolog. We present this approach in section 4.

3.6. Prolog has an Impedance Mismatch with Database Systems

It is difficult to interface Prolog systems with DBMS because they seem inherently mismatched. The control model of Prolog is one of finding a single 'proof', selecting one clause at a time and backtracking when necessary to consider alternatives. The DBMS control model, by contrast, is one of batch processing: large queries are run at one time, and all alternatives are considered simultaneously. The fundamental question is how one can interface a 'set-oriented' system like a DBMS with a 'single-clause-oriented' system like Prolog.

The query shown above illustrates some of the issues here. Let us assume, for example, `quote/5` is indexed on its first argument and date together and has 100,000 clauses, but `stock/3` is not indexed and has 1000 clauses. Then the query above

```
?- stock(Symbol,Name,Address),
   quote(Symbol,'Oct 19 1987',Price,_,Price).
```

will take enormously less time than the 'equivalent' query

```
?- quote(Symbol,'Oct 19 1987',Price,_,Price),
   stock(Symbol,Name,Address).
```

If the first query were to take one minute to find all solutions, the second query would take more like 70 days (100,000 minutes).

A DBMS may run this query query efficiently. If we issue a query to the DBMS as

```
?- sql_query('select *
              from   stock, quote
              where  stock.symbol = quote.symbol
              and    quote.date = "Oct 19 1987"
              and    quote.high = quote.close').
```

then the DBMS can, and will, optimize the query in the most advantageous way possible, and return the results much more quickly than the Prolog approach. (The results obtained by this

goal are understood to be `asserted` in the Prolog database.)

Many attempts have been documented over the past few years in connecting Prolog with Relational DBMS, but for the most part they amount to 'glue jobs' (in the all too accurate words of Mike Stonebraker). These connections are inefficient and not practical for large-scale stream processing applications. Later, we will show how stream processing provides a way to integrate Prolog and DBMS in an elegant way.

3.7. Summary: Problems and Solutions

We have shown that, without changes, Prolog has many limitations that make its use in stream processing a challenge. We have also shown that these limitations can be circumvented with modest improvements in Prolog technology or adroit programming techniques:

Prolog implementations of atoms are inadequate

The atom table used in almost all Prolog implementations has the serious flaw that it can overflow. Introduction of a string datatype will mostly eliminate this problem, although overflow will then still be a possibility with poorly-written programs.

Currently, processing of very large streams can be done by restricting the data analyzed to be numeric, with perhaps a bounded amount of string data (e.g., all stock symbols). This numeric restriction is annoying, but permits a great deal of useful analysis.

Prolog requires stack maintenance

Restricting stream transducers to be deterministic, tail recursive predicates avoids most problems in processing of large streams in Prolog systems with Tail Recursion Optimization (TRO) and Garbage Collection (GC). TRO eliminates the danger of stack overflows, and GC also avoids the problem of heap overflows that transducers can encounter in transforming large streams.

Prolog's I/O mechanisms are inadequate

Prolog should be augmented with new primitives, including more comprehensive I/O interfaces, the `undo/1` primitive of SICStus Prolog, and some cursor-like mechanism for accessing the Prolog database [56]. Also, it is desirable to develop (expensive) versions of I/O primitives like `read` that are side-effect-free, i.e., that can back up in their streams, so that nondeterministic programs can be applied to those streams that are accessible only through I/O (streams that will not fit entirely in the Prolog heap, for example).

Prolog's control mechanisms are inadequate

Prolog's poor control seems at first to be the biggest impediment to its use in stream processing. Some problems are avoidable by adding new primitives. For example, the cursor-like mechanism for accessing the Prolog database just mentioned can be used to permit Prolog implementation of merge joins in particular, and coroutining in general.

With a little cleverness, we can overcome most control problems with no change to Prolog. In section 4, especially 4.4, we will see how Prolog can be extended with a meta-interpreter that permits general stream processing.

Prolog has an impedance mismatch with database systems

Prolog and DBMS work with different models of data processing. Where Prolog operates on single sets of bindings at a time, DBMS work on whole predicates at a time. In section 6, we will see that both of these models can be integrated under the model of stream processing.

Thus most problems can be solved without sweeping changes, but some problems cannot. In particular, there is no simple solution to the problem that Prolog requires lots of memory. Fortunately, this does not seem to be a long-term problem, but rather a consequence of current memory prices, so it may be a problem that advances in technology will permit us to ignore.

4. Log(F): A Rewrite Rule Language in Prolog

We pause here to study Log(F), the language we will use shortly to write stream transducers that run in Prolog. Log(F) is a combination of Prolog and a functional language called F*, developed by Sanjai Narain at UCLA [52, 53]. Log(F) is the integration with Prolog of a functional language in which one programs using rewrite rules. This section reviews the major aspects of Log(F), and describes its advantages for stream processing.

4.1. Overview of F* and Log(F)

F* is a rewrite rule language. In F*, all statements are rules of the form

```
LHS => RHS
```

where *LHS* and *RHS* are structures (actually Prolog terms) satisfying certain modest restrictions summarized below.

A single example shows the power and flexibility of F*. Consider the following two rules, defining how lists may be appended:

```
append([],W) => W.
append([U|V],W) => [U|append(V,W)].
```

Like the Prolog rules for appending lists, this concise description provides all that is necessary. The two F* rules are essentially equivalent to the usual Prolog definition

```
append([],W,W).
append([U|V],W,[U|L]) :- append(V,W,L).
```

Log(F) is the integration of F* with Prolog. In Log(F), F* rules are compiled to Prolog clauses. The compilation process is straightforward. For example, the two rules above are translated into something functionally equivalent to the following Prolog code:

```
reduce(append(A,B),C) :- reduce(A,[]), reduce(B,C).
reduce(append(A,B),C) :- reduce(A,[D|E]), reduce([D|append(E,B)],C).
reduce( [], [] ).
reduce( [X|Y], [X|Y] ).
```

Unlike the rules in many rewriting systems, the `reduce` rules here can operate non-deterministically, just like their Prolog counterparts. Many ad hoc function- or rewrite rule-based systems have been proposed to incorporate Prolog's backtracking, but the simple implementation of F* in Prolog shown here provides nondeterminism as a natural and immediate feature.

An important feature of F* and Log(F) is the capability for *lazy evaluation*. With the rules above, the goal

```
?- reduce( append([1,2,3],[4,5,6]), X ).
```

yields the result

```
X = [1|append([2,3],[4,5,6])].
```

That is, in one **reduce** step, only the head of the resulting appended list is computed. (Try it!) The tail, **append([2,3],[4,5,6])**, can then be further reduced if this is necessary.

In order for the **reduce** rules above to work properly, we need the two rules for **[]** and **[_|_]**:

```
reduce( [], [] ).
reduce( [X|Y], [X|Y] ).
```

In F*, the function symbols (functors) like **[]** and **[_|_]** of terms that reduce to themselves are called *constructor symbols*. Below we will call any term whose functor is a constructor symbol a *simplified* term. Simplified terms reduce to themselves. Constructors are the things in F* and Log(F) that implement lazy evaluation, since they terminate reduction.

Where F* computations are naturally lazy because of their implementation with reduction rules, Log(F) permits some *eager computation* as well. Essentially, eager computations are invocations of Prolog predicates. Thus, in the Log(F) rule

```
intfrom(N) => [N|intfrom(N+1)].
```

the subterm **N+1** is recognized by the Log(F) system as being eager, and the resulting code produced is equivalent to

```
reduce( intfrom(N), X ) :- N1 is N+1, reduce( [N|intfrom(N1)], X ).
```

Programmers may declare their own predicates to be eager. By judicious combination of eager and lazy computation, programmers obtain programming power not available from Prolog or F* alone. For a number of useful examples of this combination, see [53].

4.2. F* – A Rewrite Rule Language

In this section we present F* a bit more carefully. F* rules have the form *LHS* => *RHS* where *LHS* and *RHS* are terms. These terms are made up of variables and function symbols, which may or may not be constructor symbols. Certain restrictions are made on *LHS* and *RHS*. After defining terminology, we list these restrictions below.

Constructor symbols are special function symbols that are not reducible by F*. Examples of predefined constructor symbols are **true**, **false**, **[]**, **[_|_]**. *Rewrite rules* give reduction relationships between terms. Examples of Log(F) rules are as follows:

```
if(true,X,Y) => X.
if(false,X,Y) => Y.

and(X,Y) => if(X,Y,false).
or(X,Y) => if(X,true,Y).
not(X) => if(X,false,true).
```

F* rules *LHS* => *RHS* satisfy the following restrictions [52]:

(a) *LHS* is not a variable.

(b) *LHS* is not of the form $c(T_1, \ldots, T_n)$ where c is an n-ary constructor symbol, $n \geq 0$.

(c) If *LHS* is $f(T_1, \ldots, T_n)$, $n \geq 0$, then each T_i is either a variable, or a term of the form $c(X_1, \ldots, X_m)$ where c is an m-ary constructor symbol, $m \geq 0$, and each X_j is a variable.

(d) There is at most one occurrence of any variable in *LHS*.

(e) All variables of *RHS* appear in *LHS*.

An *F* program* is a collection of F* rules. Below is an example of an F* program, provided that `s/1` is a constructor symbol:

```
equal(0,0) => true.
equal(s(X),s(Y)) => equal(X,Y).

lessEq(0,X) => true.
lessEq(s(X),s(Y)) => lessEq(X,Y).

sum(0,X) => X.
sum(s(X),Y) => s(sum(X,Y)).
```

A *reduction* using an F* program begins with a ground term G produces a sequence of rewrites, or reductions, of G, and terminates in a term whose function symbol is a constructor. That is, given a term G, a reduction is a sequence of ground terms $G_0, G_1, ..., G_n$ such that

(1) $G = G_0$;

(2) For each i, $0 \leq i \leq n-1$, there exists a rule $f(T_1, \ldots, T_m) \Rightarrow T$ such that

  (a) $G_i = f(S_1, \ldots, S_m)$;

  (b) For each j, $1 \leq j \leq m$, S_j is either a variable or recursively has a reduction to T_j. If S_j is a variable, we construct the binding $\theta_j = \{ S_j \leftarrow T_j \}$. Otherwise we let θ_j be the bindings obtained recursively in the reduction of S_j to T_j;

  (c) If we let θ be the accumulated bindings $\theta_1 \cdots \theta_m$, then $G_{i+1} = T\theta$, where T is again the right hand side of the rule.

(3) The function symbol of G_n is a constructor, i.e., G_n is simplified.

For example, with the program

```
equal(0,0) => true.
equal(s(X),s(Y)) => equal(X,Y).

sum(0,X) => X.
sum(s(X),Y) => s(sum(X,Y)).
```

the term `equal(sum(s(s(0)),s(0)),s(s(s(0))))` has the reduction

```
equal(sum(s(s(0)),s(0)),s(s(s(0))))
equal(s(sum(s(0),s(0))),s(s(s(0))))
equal(sum(s(0),s(0)),s(s(0)))
equal(s(sum(0,s(0))),s(s(0)))
equal(sum(0,s(0)),s(0))
equal(s(0),s(0))
equal(0,0)
true.
```

In this reduction we alternated applications of the fourth and second rules in the first 6 steps, and used the first rule in the last step.

The restrictions above are carefully designed to be sufficient for proving soundness and completeness properties of F* reductions [53]. However, they also have intuitive practical justifications:

1. F* programs are understood to be used only in rewriting *ground terms*, terms that do not contain any variables. That is, if we try to reduce a term T using an F* program, then T should contain no variables. Restriction (e) then guarantees that whatever T is rewritten to will also contain no variables.
2. Restrictions (d) and (e) above mean that F* programs use only *pattern matching* (matching of terms with no duplicated variables against ground terms), and not the full power of unification. Restriction (d) is sometimes called a 'linearity' restriction, and avoids unification. This is actually an advantage! It leads to fast implementations, and in many situations causes no loss in power. Note that the program above defines equality of terms without using unification.
3. Restriction (c) is the main restriction on F* rules. It requires that the head of a rule be of the form

   $$f(T_1, \ldots, T_n)$$

   where each T_i is either a variable, or a term whose functor is a constructor symbol. This restriction guarantees efficient implementation. Rather than requiring a general equality theory for pattern matching of arguments, all that is needed is binding to variables, or reduction to simplified terms.

These restrictions are really very natural, and are easily grasped once one has written a few F* programs.

4.3. Log(F): Integrating F* with Prolog

Because constructor symbols in F* terminate reduction, we call F* a *lazy rewriting language*. Constructors terminate reduction (evaluation) of a term; for further evaluation the constructors must be removed. Since Prolog has no delayed computation *per se*, we tend to think of Prolog computations as *eager* by contrast with F*.

Log(F) is the integration of F* with Prolog. It therefore combines both lazy F* computations with eager Prolog computations. This combination has many practical applications. For example, in the Log(F) code

```
count([X|S],N) => count(S,N+1).
```

the subterm `N+1` is recognized by the Log(F) compiler as being eager, and the resulting code produced is something equivalent to

```
reduce(count(A,N),Z) :- reduce(A,[X|S]), M is N+1, reduce(count(S,M),Z).
```

Arbitrary Prolog predicates can be declared to be eager. Among other predicates, we can introduce a general eager Prolog interface called `success/1` that yields the value `true` if its argument succeeds when called, and the value `false` otherwise:

```
success(G,true) :- call(G), !.
success(G,false).
```

Another eager predicate that can be useful in writing Log(F) rules is `reduce` itself. With it we can write rules like

```
list_values([H|T]) => [reduce(H)|list_values(T)].
```

that force eager evaluation of Log(F) terms when necessary.

With this interface to 'eager' Prolog predicates we can develop significant programs with compact sets of rewrite rules. For example, the following is an executable Log(F) program for computing primes:

```
primes => sieve(intfrom(2)).

intfrom(N) => [N|intfrom(N+1)].

sieve([U|V]) => [U|sieve(filter(U,V))].

filter(A,[U|V]) => if( success(U mod A =:= 0), filter(A,V), [U|filter(A,V)] ).
```

The `intfrom` rule generates an infinite stream of integers. The rule for `filter` uses the eager Prolog interface `success`.

Compilation of Log(F) rules to Prolog is easy to implement in principle. Following the definition of reductions above, the F* rule

```
f(T1,...,Tn) => T
```

can be compiled to the Prolog `reduce` clause

```
reduce( f(A1,...,An), Z ) :-
            reduce(A1,T1),
            ...,
            reduce(An,Tn),
            reduce(T,Z).
```

provided that each of *T1, ..., Tn* and *T* is a nonvariable term. If any of *T1, ..., Tn* or *T* is a variable, the reduction for it in the body of this clause is eliminated. The compilation of rules with eager primitives, like + and `success`, is only mildly more complex.

Using this compilation algorithm extended for eager predicate calls, the Log(F) `primes` program above would be compiled to the following Prolog rules:

```
reduce(primes,Z) :-
            reduce(sieve(intfrom(2)),Z).
reduce(intfrom(N),Z) :-
            N1 is N+1,
            reduce([N|intfrom(N1)],Z).
reduce(sieve(A),Z) :-
            reduce(A,[U|V]),
            reduce([U|sieve(filter(U,V))],Z).
reduce(filter(A,B),Z) :-
            reduce(B,[U|V]),
            success((U mod A =:= 0),S),
            reduce(if(S,filter(A,V),[U|filter(A,V)]),Z).
reduce(if(A,X,Y),Z) :-
            reduce(A,true),
            reduce(X,Z).
reduce(if(A,X,Y),Z) :-
            reduce(A,false),
            reduce(Y,Z).

reduce([U|V],[U|V]).
reduce([],[]).
reduce(true,true).
reduce(false,false).

success(G,true) :- call(G), !.
success(G,false).
```

As an example of execution, if we define the predicate

```
reducePrint(X) :- reduce(X,[H|T]), write(H - T), nl, reducePrint(T).
```

then the goal

```
?- reducePrint(primes).
```

produces the following (non-terminating) output:

```
2 - sieve(filter(2,intfrom(3)))
3 - sieve(filter(3,filter(2,intfrom(4))))
5 - sieve(filter(5,filter(3,filter(2,intfrom(6)))))
7 - sieve(filter(7,filter(5,filter(3,filter(2,intfrom(8))))))
    ...
```

4.4. Stream Processing Aspects of Log(F)

The example above shows that Log(F) naturally provides *lazy evaluation*. Functional programs on lists can produce terms in an incremental way, and incremental or "call by need" evaluation is an elegant mechanism for controlling query processing.

Furthermore, Log(F) is a superior formalism for stream processing, and thus apparently for stream data analysis. From the examples above, it is clear that the rules have a functional flavor. Stream operators are easily expressed using recursive functional programs.

Log(F) also has a formal foundation that captures important aspects of stream processing:

1. Determinate (non-backtracking) code is easily detected through syntactic tests only. A benefit of the F* restrictions is that deterministic computations are easily detected. If the heads of rules for a symbol do not unify, and only ground terms are reduced, then the reduction will be deterministic. For example, with the rules

```
sum(0,X) => X.
sum(s(X),Y) => s(sum(X,Y)).
```

   in reducing terms like `sum(s(s(0)),s(0))` only one rule can be chosen at any point. Determinate code avoids the overhead of "distributed backtracking" incurred by some parallel logic programming systems.

2. Log(F) takes as a basic assumption that stream values are *ground terms,* i.e., Prolog terms without variables. This avoids problems encountered by parallel Prolog systems which must attempt to provide consistency of bindings to variables used by processes on opposing ends of streams.

These features of Log(F) make it a nicely-limited sublanguage in which to write high-powered programs for stream processing and other performance-critical tasks. Special-purpose compilers can be developed for this sublanguage to produce highly-optimized code. Log(F) compilers can be much more sophisticated than the compiler described above. Among other things, they can ascertain the determinacy of Log(F) rules and prevent multiple reductions of common subexpressions.

In section 3 we grappled with the problem that it is not so obvious how to perform stream processing in Prolog. Specifically, it is not obvious how to implement control strategies like coroutining.

Log(F) offers a solution to this problem, since lazy evaluation gives us a method to implement coroutining. Recall that coroutining basically requires programs to suspend their execution temporarily while the executions of other programs are resumed. The result of lazy evaluation can be, for example, a term

```
[partial_result | computation_to_resume_later]
```

whose tail is a kind of 'closure', representing an unfinished computation. Designing transducers to yield this kind of result is precisely what we need to implement coroutining. In fact, as we will illustrate with further examples in the next section, Log(F) naturally provides enough to implement (even recursively-defined) networks of coroutined transducers, and thus demand-driven stream processing.

5. Stream Processing in Prolog

The Tangram Stream Processor is an extensible stream processing system that uses Log(F) to implement stream processing in Prolog. This section develops techniques for stream processing in Log(F) through a sequence of examples.

5.1. Implementing Transducers in Log(F)

Let us first review how the four basic kinds of single-input, single-output transducers can be implemented in Log(F). It is remarkable how much simpler they are than their Prolog counterparts.

Enumerators

Enumerators in Log(F) are typically very compact:

```
enumerate => enumerate(initial_state).

enumerate(S) => [next_value(S)|enumerate(next_state(S))].
enumerate(_) => [].
```

The enumerator of integers is easy to develop, and the infinite stream version is even simpler:

```
% intfrom(M,N) =>  the list of integers [M,...,M+N].

intfrom(M,N) => if(N=<0, [], [M|intfrom(M+1,N-1)] ).

% intfrom(M) =>  the list of integers [M,...].

intfrom(M) => [M|intfrom(M+1)].
```

Maps

The generic map transducers for the function f can look as follows in Log(F):

```
mapstream([X|Xs]) => [f(X)|mapstream(Xs)].
mapstream([]) => [].
```

It is easy to develop the higher-order version of **mapstream** which takes a function as an argument.

```
mapstream(F,[]) => [].
mapstream(F,[X|L]) => [apply(F,X)|mapstream(F,L)].
```

The Prolog examples given earlier can be adapted to Log(F) as follows:

```
% squares(L) =>  the result of squaring each member of L

squares([X|Xs]) => [X*X|squares(Xs)].
squares([]) => [].

% project(I,L) =>  the list of I-th arguments of terms in L

project(I,[X|Xs]) => [arg(I,X) | project(I,Xs)].
project(_,[]) => [].
```

Filters

Filters in Log(F) are like maps, but involve an if-then-else construct.

```
filter([X|Xs]) => if( inadmissible(X), filter(Xs), [X|filter(Xs)] ).
filter([]) => [].
```

A good example of a filter is provided by the generic selection transducer.

```
% select(S,T,C) =>  the stream of terms in S matching T and satisfying C

select([X|Xs],T,C) =>
       if( success(\+(X=T,C)), select(Xs,T,C), [X|select(Xs,T,C)]).
select([],_,_) => [].
```

Accumulators

Thanks to the functional notation again, accumulators are also easy to develop in Log(F):

```
accumulate(List) => accumulate(List,initial_state).

accumulate([X|Xs],S) => accumulate(Xs,next_state(X,S,NewS)).
accumulate([],S) => final_state_value(S).
```

A simple example is the sum transducer.

```
% sum(Stream,Sum) =>  Sum is the result of summing the elements in Stream

sum(Stream) => sum(Stream,0).

sum([X|Xs],OldSum) => sum(Xs,X+OldSum).
sum([],Sum) => Sum.
```

5.2. The General Single-Input, Single-Output Transducer

The four kinds of transducers above are all special cases of a general single-input, single-output transducer. An advantage of using Log(F) is that the functional nature of the transducer — a sequential mapping between input stream items and output stream subsequences — comes out. Consequently, we can generalize nicely upon the four kinds. A general transducer can be defined by an initial state and three functions:

A single-input, single-output stream transducer T is a 4-tuple

```
(initial_state, output, next_state, final_output),
```

where:

(1) *initial_state* is the state of the transducer when it is invoked;

(2) *output* maps the current state and current stream input(s) to new stream output(s). Stream inputs can be ignored. A stream output can be `[]`, specifying that output stream is not to be changed;

(3) *next_state* maps the current state and current stream input(s) to the next state;

(4) *final_output* specifies the final output(s) to be written on streams when no input is left.

```
transduce(Stream) => transduce(Stream, initial_state).

transduce([],State) => final_output(State).
transduce([Input|Stream],State) =>
                append(
                        output(Input,State),
                        transduce(Stream,next_state(Input,State))
                ).
```

Note that although the third rule uses **append** for the sake of generality, in many cases it is possible to use only `[_|_]`.

To grasp how a single-input, single-output transducer can be written with this generalization in mind, let us consider one example in detail, which we take from temporal database query processing. Consider the following temporal database:

```
holdings(   'IBM',    250,     9/81    )
holdings(   'TUA',    230,     9/85    )
holdings(   'IBM',    330,     12/86   )
holdings(   'F',      250,     9/87    )
holdings(   'IBM',    440,     11/87   )
holdings(   'TUA',    0,       12/87   )
holdings(   'F',      400,     12/88   )
```

From this database a cumulative holdings history can be derived with a stream transducer:

```
holdingsHistory( Holdings ) => holdingsHistory( Holdings, [] ).

holdingsHistory( [],                    CurrentHoldings ) => CurrentHoldings.
holdingsHistory( [holdings(S,Q,D)|H], CurrentHoldings ) =>
      append(
            holdingsRecord(S,Q,D,CurrentHoldings),
            holdingsHistory(H, reduce(newHoldings(CurrentHoldings,S,Q,D)))
      ).
```

This transducer is defined by the 4-tuple

```
([], holdingsRecord, holdingsHistory, identityMapping)
```

where **identityMapping** is the identity mapping:

```
identityMapping(X) => X.
```

To complete the definition of the transducer, we must give the transduction mappings `holdingsRecord` and `holdingsHistory`. These can be defined as follows:

```
holdingsRecord( S,Q,D, [] ) => [].
holdingsRecord( S,Q,D, [holdings(OldS,OldQ,OldD,_)|Holdings] ) =>
        if( OldS==S,
                [holdings(S,OldQ,OldD,D)],
                holdingsRecord(S,Q,D,Holdings)
        ).

newHoldings( [], S,Q,D ) => [holdings(S,Q,D,_)].
newHoldings( [holdings(S1,Q1,D1,_)|Holdings], S,Q,D ) =>
        if( S1==S,
                [holdings(S,Q,D,_) | Holdings],
                [holdings(S1,Q1,D1,_) | newHoldings(Holdings,S,Q,D)]
        ).
```

Let `holdings` be a Log(F) term that yields a stream of the tuples from the 3-column relation `holdings`. The output stream obtained by reducing `holdingsHistory(holdings)` is as follows:

```
holdings(   'IBM',    250,    9/81,     12/86   )
holdings(   'IBM',    330,    12/86,    11/87   )
holdings(   'TUA',    230,    9/85,     12/87   )
holdings(   'F',      250,    9/87,     12/88   )
holdings(   'IBM',    440,    11/87,    _       )
holdings(   'TUA',    0,      12/87,    _       )
holdings(   'F',      400,    12/88,    _       )
```

5.3. Basic Statistical Time Series Analysis

Many basic time series analysis procedures can be formalized now as stream transducers. First, the standard deviation predicate we wrote in Prolog earlier can also be written in Log(F) as follows:

```
stddev(Stream) => sqrt(variance(count_sum_sumsq(Stream,0,0,0))).

variance([N,S,Q]) => if( N =< 1, 0, (Q-(S*S/N))/(N-1) ).

count_sum_sumsq([],N,S,Q) => [N,S,Q].
count_sum_sumsq([X|Xs],N,S,Q) => count_sum_sumsq(Xs,N+1,S+X,Q+X*X).
```

This program assumes the existence of an eager square root function.

A transducer for moving averages is also straightforward to develop:

```
moving_avg(M,[]) => 0.
moving_avg(M,[H|T]) => [prefix_avg(M,[H|T]) | moving_avg(M,T)].

prefix_avg(M,S) => prefix_avg(M,M,0,S).

prefix_avg(M,_,V,[]) => V/M.
prefix_avg(M,N,V,[H|T]) => if( N=<0, V/M, prefix_avg(M,N-1,V+H,T) ).
```

Simple linear regression gives a good final example. Given two streams of values, **X** and **Y**, we can find the simple linear regression coefficients b_0 and b_1 that minimize the mean square error of the approximation $\mathbf{Y} = b_0 + b_1\mathbf{X}$. Since the mean square error is $\sum_{i=1}^{n} (Y_i - (b_0 + b_1 X_i))^2 / n$, where n is the number of elements in each of the streams, the coefficients minimizing this turn out to be

$$b_1 = \frac{(\sum_{i=1}^{n} X_i Y_i) - (\sum_{i=1}^{n} X_i)(\sum_{i=1}^{n} Y_i)/n}{(\sum_{i=1}^{n} X_i^2) - (\sum_{i=1}^{n} X_i)^2/n}$$

$$b_0 = (\sum_{i=1}^{n} Y_i)/n - b_1(\sum_{i=1}^{n} X_i)/n$$

These can be computed in essentially the same way we computed standard deviations.

Log(F) permits us to expand upon conventional statistical time series analysis as we wish. With the power of Prolog at our disposal, we can build 'rule-based' analysis tools. Rather than blindly performing a simple linear regression, for example, we can write a transducer that decides first how *best* to fit **X** against **Y**, Such a system is described by Gale in [31].

5.4. Aggregate Computations

Aggregate operators can be of several kinds. Aggregate reductions, which apply an associative operator to among elements of a stream, are very easy to define:

```
count(S) => count(S,0).
count([],N) => N.
count([_|S],N) => count(S,N+1).

sum(S) => sum(S,0).
sum([],T) => T.
sum([X|S],T) => sum(S,X+T).

avg([]) => 0.
avg([X|S]) => sum([X|S]) / count([X|S]).
```

Aggregate operators may also act as stream transducers, placing partial aggregates in the output stream as each input item is tallied. Snodgrass and Gomez define many interesting stream

aggregation operators for TQuel [64]. Here we investigate the operators for forming counts. The input stream may be taken as containing items of one of two forms:

```
insert(Identifier, Value, Time)
delete(Identifier, Value, Time).
```

The history of these insertions or deletions of Identifier-Value pairs, with corresponding timestamps, can be queried with four stream aggregate operators. The operators transform the input stream to an stream containing the following information:

Operator	*Output stream items*
`count`	total number of Identifier-Value pairs inserted, but not deleted, up to the present
`countC`	total number of Identifier-Value pairs inserted up to the present
`countU`	total number of unique Identifier-Value pairs inserted, but not deleted, up to the present
`countUC`	total number of unique Values inserted, but not deleted, up to the present

These count aggregation operators can be conveniently defined. We first define some constructor symbols: `insert(I,V,T)`, `delete(I,V,T)`, `count`, `countC`, `countU`, and `countUC`. We then define the TQuel stream count aggregation operators:

```
tquel_count(Type, S) => tquel_count(Type, S, [0]).
tquel_count(Type, [], [Count|List]) => [Count].
tquel_count(Type, [Input|S], [Count|List]) =>
      [Count | tquel_count(Type,S, newCounter(Type,Input,[Count|List]))].

newCounter(C, insert(I,V,T), State) => insert(C,I,V,T,State).
newCounter(C, delete(I,V,T), State) => delete(C,I,V,T,State).

insert( count,   I,V,T, [Count|List]) => [Count+1 | [(I,V)|List] ].
delete( count,   I,V,T, [Count|List]) => [Count-1 | diff(List,[(I,V)]) ].
insert( countC,  I,V,T, [Count]) => [Count+1].
delete( countC,  I,V,T, [Count]) => [Count].
insert( countU,  I,V,T, [_|List]) => lengthList( insertCount([(I,V)],List) ).
delete( countU,  I,V,T, [_|List]) => lengthList( deleteCount([(I,V)],List) ).
insert( countUC, I,V,T, [_|List]) => lengthList( insertCount([V],List) ).
delete( countUC, I,V,T, [_|List]) => lengthList( deleteCount([V],List) ).

lengthList(L) => [reduce(listLength(L)) | reduce(L)].

listLength([]) => 0.
listLength([X|L]) => 1 + listLength(L).
```

```
insertCount(X,[]) => [(X,1)].
insertCount(X,[(Y,N)|L]) => if( Y == X,
                                [(X,N+1)|L],
                                [(Y,N)|insertCount(X,L)]
                            ).
deleteCount(X,[(Y,N)|L]) => if( Y == X,
                                if( N==1, L, [(X,N-1)|L] ),
                                [(Y,N)|deleteCount(X,L)]
                            ).

diff([],Y) => Y.
diff([X|L],Y) => if( X==Y, L, [X|diff(L,Y)] ).
```

This definition is instructive; from it one sees immediately that `countC` is fast and easy to compute, requiring only the current count for its state, while `countU` is quite expensive to compute, potentially needing to store the entire input stream in its state.

In this situation there seems to be no way to avoid this expense: if we really want all the information the counts provide here, we must pay for it. The following section shows, however, that cleverness in writing stream transducers can pay off enormously in performance.

5.5. Computing Medians and Quantiles

A useful way to capture the distribution of the values in a stream is to compute certain *quantiles* of the stream. The q-th quantile, $0 \leq q \leq 1$, of a stream $S = [X_1, X_2, \ldots, X_n]$ is the value X_j such that $\lfloor qn \rfloor$ members of S are less than or equal to X_j. For example, the ½-th quantile of a stream will be its *median*. Also, some students submit to years of training now in order to reach the 0.98 quantile (i.e., the 98-th percentile, or top 2 percent) of college entrance examination results.

Quantiles are sometimes called *order statistics*. They are important, since in many cases they give us more basic information than other measures, such as averages or standard deviations. Quantiles characterize the shape of the distribution of values we have, without requiring assumptions about its shape or basic nature.

There is one unfortunate problem: quantiles can be expensive to compute. To see this, consider the following approach for computing all `M`-th quantiles (so when `M`=2 we get the median and the largest value, when `M`=4 we get all quartiles, etc.), where `sort` and `length` are eager primitives defined elsewhere:

```
quantiles(S,M) => everyNth( sort(S), length(S)/M ).

everyNth(S,N) => if( N=<1, S, everyNth(S,[],1,N,N) ).

everyNth([X|Xs],_,I,Limit,N) =>
            if( I>=Limit,
                        [X|everyNth(Xs,[],I+1,Limit+N,N)],
                        everyNth(Xs,[X],I+1,Limit,N)
            ).
everyNth([],Last,_,_,_) => Last.
```

The problem with this approach is that it is very expensive. The sorting will generally take time $O(n \log n)$, where n is the length of the stream `s`, and in addition we need to know the length of the stream before we compute any actual quantiles. Better algorithms for computing quantiles have been developed [9], and the time bound above can be reduced to $O(n)$, but at the cost of requiring all n members of the stream to be accessible in memory for comparisons. When the input stream is large, this approach will not be reasonable.

Fortunately, efficient approaches are now known for *estimating* quantiles. We present a surprising and simple technique developed by Pearl [59], who noticed that quantiles of a random sequence are estimated by the values of specific minimax (or maximin) trees of the sequence. Specifically, if d_1 and d_2 are positive integers and q is the unique positive root of the polynomial $(1-(1-(1-x)^{d_1})^{d_2})$, then it turns out that the value of a maximin tree whose max nodes have branching factor d_1, whose min nodes have branching factor d_2, and whose leaves are the members of a stream `s`, is an estimate of the q-th quantile of `s`.

To implement Pearl's approach we can first develop some stream operators that produce minimax values of a stream, assuming that **`minimum/2`** and **`maximum/2`** are eager primitives defined in Prolog:

```
min([X|Xs],N) => min(Xs,1,N,X).
min([],N) => [].

min([X|Xs],I,N,B) => if( I<N, min(Xs,I+1,N,minimum(B,X)), [B|min([X|Xs],N)] ).
min([],I,N,B) => [B].

max([X|Xs],N) => max(Xs,1,N,X).
max([],N) => [].

max([X|Xs],I,N,B) => if( I<N, max(Xs,I+1,N,maximum(B,X)), [B|max([X|Xs],N)] ).
max([],I,N,B) => [B].

maximin(S,D1,D2) => max( min(S,D1), D2 ).

minimax(S,D1,D2) => min( max(S,D1), D2 ).
```

With these, an estimate of medians is easily obtained. When $d_1 = 5$ and $d_2 = 3$ we have $q = 0.511$, which is fairly close to ½. Thus we can produce a median estimate from a maximin

tree obtained by recursively taking maximins with this d_1, d_2:

```
medianEstimates(S) => maximin(S,5,3).

median([X|Xs]) => if( Xs==[], X, median(medianEstimates([X|Xs])) ).
median([]) => [].
```

This estimate is simple and cheap. Pearl points out that better estimates can be obtained by using $d_1 = 11$, $d_2 = 4$ (which gives $q = 0.495575$) or $d_1 = 44$, $d_2 = 6$ (which gives $q = 0.500034$).

A more general stream transducer to find the q-th quantile could be implemented with a similar transducer that first begins operation with some kind of search for the right parameters d_1, d_2 to obtain q. The transducer below does just this; the eager Prolog predicate `quantileParameters/2` performs a search through its available table of parameters looking for the best match to a request for a quantile value. Once the best d_1, d_2 values have been found, a simple maximin recursion can be used again.

```
quantile(S,Quantile) => approxQuantile(S,quantileParameters(Quantile)).

approxQuantile([X|Xs],[D1|D2]) =>
      if( Xs==[],
            X,
            approxQuantile(quantileEstimates([X|Xs],[D1|D2]),[D1|D2])
      ).
approxQuantile([],_) => [].

quantileEstimates(S,[D1|D2]) => maximin(S,D1,D2).
```

This transducer needs the following eager quantile parameter selection code:

```
quantileParameters(Quantile,[D1|D2]) :-
      quantileTable(Table),
      bestMatch(Table,Quantile,1.0,2,2,D1,D2).

bestMatch([],_,_,D1,D2,D1,D2).
bestMatch([q(D1,D2,Q)|T],Quantile,BestDiff,BestD1,BestD2,FinalD1,FinalD2) :-
      absDiff(Q,Quantile,Diff),
      (Diff >= BestDiff ->
            bestMatch(T,Quantile,BestDiff,BestD1,BestD2,FinalD1,FinalD2)
      ;
            bestMatch(T,Quantile,    Diff,    D1,    D2,FinalD1,FinalD2)
      ).

absDiff(A,B,X) :- D is (A-B), ((D > 0) -> (X is D) ; (X is -D)).

% The table q(D1,D2,Q) below, for positive integers D1 & D2,
% gives values of Q -- the unique positive root of the polynomial
%       (1 - (1 - (1 - x)^D1 )^D2 ).

quantileTable([
q( 2, 2, 0.618),  q( 2, 3, 0.848),  q( 2, 4, 0.920),  q( 2, 5, 0.951),
q( 3, 2, 0.389),  q( 3, 3, 0.683),  q( 3, 4, 0.805),  q( 3, 5, 0.866),
q( 4, 2, 0.282),  q( 4, 3, 0.579),  q( 4, 4, 0.725),  q( 4, 5, 0.803),
q( 5, 2, 0.220),  q( 5, 3, 0.511),  q( 5, 4, 0.665),  q( 5, 5, 0.749),
q( 6, 2, 0.180),  q( 6, 3, 0.461),  q( 6, 4, 0.622),  q( 6, 5, 0.717),
q( 7, 2, 0.153),  q( 7, 3, 0.412),  q( 7, 4, 0.580),  q( 7, 5, 0.687),
q( 8, 2, 0.133),  q( 8, 3, 0.392),  q( 8, 4, 0.558),  q( 8, 5, 0.661),
q( 9, 2, 0.117),  q( 9, 3, 0.367),  q( 9, 4, 0.533),  q( 9, 5, 0.640),
q(10, 2, 0.105),  q(10, 3, 0.347),  q(10, 4, 0.533),  q(10, 5, 0.621),
q(11, 2, 0.095),  q(11, 3, 0.329),  q(11, 4, 0.496),  q(11, 5, 0.605)
]).
```

Of course, this program could be extended to find roots of the polynomial to any desired precision.

5.6. Pattern Matching against Streams

So far we have illustrated how Log(F) makes a fine language for expressing transductions of streams. In this section we show how, when extended slightly, it also makes a fine language for pattern analysis against streams.

It is not difficult to write transducers that detect patterns. For example, the transducer

```
bump(S) => t1(S).
t1([up|S]) => t2(S).
t2([up|S]) => t2(S).
t2(S) => t3(S).
t3([down|S]) => t4(S).
t4([down|S]) => t4(S).
t4(S) => S.
```

successfully recognizes all streams containing sequences of one or more copies of `up` followed by one or more copies of `down`. In short, the `bump` transducer recognizes the *regular expression*

```
([up]+, [down]+)
```

where + is the postfix pattern operator defining the Kleene plus, and , defines pattern concatenation.

An interesting feature of our approach is that we can write transducers to implement *parsers* by specifiying patterns with *grammars*. For example, we can specify regular expressions and, more generally, path expressions, with grammar rules something like the following:

```
(X+) => X.
(X+) => (X,(X+)).
(X*) => [].
(X*) => (X,(X*)).
(X;Y) => X.
(X;Y) => Y.
([],X) => X.
([X|Y],Z) => [X|(Y,Z)].
```

These rules behave just like the context free grammar rules they resemble.

Pattern matching can be enforced with a `match` transducer, which takes its first argument a *pattern* describing the starting symbol(s) of some grammar used for the match, and as its second argument a Log(F) term that produces a stream. This transducer is defined as follows:

```
match([],S) => S.
match([X|L],[X|S]) => match(L,S).
```

With this definition, for example, the following definition for `bump` is equivalent to the one given earlier:

```
bump_pattern => ([up]+,[down]+).

bump(S) => match( bump_pattern, S ).
```

These grammars have many promising uses. For further examples in stream pattern analysis, see [21,22].

6. Connecting Prolog with Databases

We mentioned earlier the widely-held belief that Prolog has an impedance mismatch with database management systems (DBMS). In this section, we show how stream processing can remove this mismatch.

6.1. Stream Processing as a Common Model for Prolog and DBMS

In Prolog systems, an 'inference engine' naturally works on single clauses at a time, with a particular control strategy (depth-first-search with backtracking). In DBMS, by contrast, an entire query is processed at once, by a query evaluator that selects among a variety of sophisticated algorithms. Since both systems go about work differently, it is difficult to connect the two systems in an efficient or easy-to-use way.

The key to the mismatch is that Prolog and DBMS follow different models of data processing:

(1) Where the Prolog system can be said to be 'search oriented', seeking a single proof, the DBMS is viewed as 'set oriented', computing all proofs at once.

(2) A DBMS provides a limited model of computation that it guarantees to handle well, while Prolog strives to provide a general model of computation or inference with neither real limitations nor blanket guarantees of performance.

In the past, attempts in coupling Prolog and DBMS have adopted one model of data processing or the other [12, 54]. Tuple-at-a-time solutions (particularly Prolog-DBMS interfaces) [51, 61, 69] follow the Prolog model. Query-at-a-time solutions that store the results of the query in the Prolog system workspace [20, 32, 40] follow the DBMS model. It is well known that tuple-at-a-time solutions are inefficient, and query-at-a-time solutions can overwhelm the Prolog system with data. A variety of combinations of these strategies have therefore been proposed [19, 39]. The EDUCE system, for example, provides complete tuple-at-a-time and query-at-a-time capabilities [10, 11].

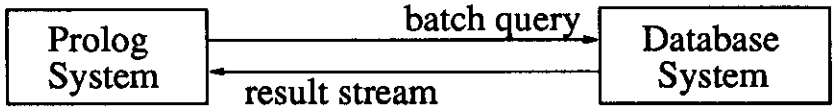
We can reduce or eliminate the mismatch if we can find a common model that will fit both systems. There is such a model: *stream processing*. We have now shown that we can introduce stream processing into Prolog in a clean and natural way. The main issue might appear to be in combining stream processing with DBMS models. In fact, this is not as difficult as it might seem, since databases naturally produce streams of results to queries [58]. Below we will sketch how a stream-based connection can be implemented.

6.2. Coupling Prolog and DBMS through Stream Processing

The Tangram project at UCLA has integrated Prolog and relational DBMS. Although the integration can be used for 'tight coupling' of Prolog and DBMS, it can also be used for 'loose coupling'. By a *loose coupling* we mean a combination in which each system keeps its own identity, and both communicate through a well-defined interface. We see loose coupling not only as necessary because of economic, political, and other forces, but also as *a desirable division of labor* in many situations. One system does bulk data processing well, while the other performs arbitrary analyses on the results.

The stream processing approach for coupling a DBMS with Prolog is basically to have the

DBMS yield a result stream in response to a query, and have Prolog applications analyze the stream using the stream processing paradigm:



Roughly, execution under the approach is as follows:

(1) A DML (Data Manipulation Language) request can be sent from the Prolog system to the DBMS. Naturally, many extensions suggest themselves here, such as piggybacking of requests, translation from nice syntax to DML, semantic query optimization, and so forth. These requests can (and in our impression, should) be made with full knowledge of what is available in the DBMS, such as indices or other access paths.

(2) The DBMS produces the result of the DML request.

(3) The Prolog system consumes the result tuples incrementally (i.e., lazily), as it needs them. Important extensions here include eagerly fetching more than 1 tuple at a time performing unification or other pattern matching at the interface level, selectively retrieving only the needed fields from tuples, etc.

The diagram above shows the Prolog system and DBMS coupled directly, but in fact an interface between the two can improve performance. Such an interface can implement buffering and flow control, and sorting of results when this is not available in the DML, since stream processing often requires sorted streams.

This general approach avoids many problems in traditional couplings between Prolog and DBMS. Couplings that offer *only* access method tuple-at-a-time retrieval from Prolog sacrifice the bulk query processing power of the DBMS, and make very heavy use of the Prolog/DBMS interface. High-level-query-at-a-time couplings that `assert` the entire query result in the Prolog database are slow since `assert` is very slow, and potentially dangerous since they can overflow Prolog data areas. A stream processing approach permits us to take advantage of the best performance aspects of both systems, tune the granularity of data blocks transferred from the DBMS to Prolog, and give the Prolog system access to the data without requiring it to be stored in the Prolog database.

6.3. Implementation of Prolog-DBMS Coupling

To couple Prolog with a DBMS for high-level queries, we can use something like the code below:

```
reduce( sql_query(Query), Stream ) :-
        start_query(Query,Cursor),
            reduce(query_stream(Cursor),Stream).
reduce( query_stream(Cursor), Stream ) :-
        next_query_result(Cursor,Term),
        query_stream(Term,Cursor,Stream).

query_stream(end_of_file,Cursor,[]) :- end_query(Cursor), !.
query_stream(Term,Cursor,[Term|query_stream(Cursor)]).
```

The interface predicates that we must provide are **start_query/2**, **next_query_result/2**, and **end_query/2**. The first sends a full SQL query to the DBMS, the second retrieves result tuples from the DBMS, and the third terminates processing of the query.

Consider the following scenario. In a relational DBMS we have both a relation of stocks, giving stock names and stock symbols and other relevant data, and a relation of daily high/low/close prices for stocks over the past few years. With the **moving_avg** transducer defined above, and a **print_stream** transducer for displaying data, we can quickly implement the query

show the 5-day moving averages for AT&T stock in 1989

by evaluating the following:

```
print_stream(
     moving_avg( 5,
          sql_query(
                    'select quote.close
                     from   stock, quote
                     where  stock.name = "AT&T"
                     and    stock.symbol = quote.symbol
                     and    quote.date >= 89/01/01
                     and    quote.date =< 89/12/31'
          )
     )
)
```

The **sql_query** expression yields a stream, just like any other expression. This stream is then averaged and displayed.

The point here is not the syntax, since we can certainly translate from whatever-you-please to this representation. The point is that *the DBMS and the Prolog system both work on the same model of data.*

Some high points of this example:

(1) We have done a brute force join query solely with the DBMS. This join might have been more expensive in Prolog.

(2) We have performed some intelligent digestion of the result with a straightforward, easy-to-write transducer. (This would have been impossible in ordinary SQL.) The transducer takes definite advantage of the fact that the data is ordered.

(3) Even display primitives can fit the stream processing paradigm.

One of the beautiful things of coupling via stream processing is that it permits us to use an low-level access method interface to obtain a stream from the DBMS as well as a high-level query interface. Access methods provide the sequential retrieval we need to implement streams. For example, we can define a tuple-at-a-time connection to the relations `stock` and `quote` with the following code:

```
reduce( stock, Stream ) :-
            open_relation(stock/3,Cursor),
            reduce(tuple_stream(Cursor),Stream).
reduce( quote, Stream ) :-
            open_relation(quote/5,Cursor),
            reduce(tuple_stream(Cursor),Stream).
reduce( tuple_stream(Cursor), Stream ) :-
        retrieve(Cursor,Tuple),
        tuple_stream(Tuple,Cursor,Stream).

tuple_stream(end_of_file,Cursor,[]) :- close_relation(Cursor), !.
tuple_stream(Term,Cursor,[Term|tuple_stream(Cursor)]).
```

Here the predicate `open_relation/2` sets up a cursor on a relation from which we can retrieve tuples by using `retrieve/2`. With these definitions we can implement the same query handled above with something like the following code:

```
print_stream(
   moving_avg( 5,
      project( 9,
         join(
            select( stock, stock(_,"AT&T",_,_), true),
            select( quote, quote(_,_,_,_,Date),
                 (notLater(89/01/01,Date), notLater(Date,89/12/31)) )
         )
      )
   )
)
```

The point to see here is that we can allocate processing responsibility as we please to Prolog and the DBMS. Stream processing allows us to couple either via a high-level query-at-a-time interface, or a low-level tuple-at-a-time interface.

There is one final remark worth making here. Just as we have used streams to get data from databases, we can use streams for getting results from files, or any other source of data. The following code implements streams from files:

```
reduce( file_terms(Filename), Stream ) :-
        open(Filename,read,Descriptor),
            reduce(file_stream(Descriptor),Stream).
reduce( file_stream(Descriptor), Stream ) :-
        read(Descriptor,Term),
        file_stream(Term,Descriptor,Stream).

file_stream(end_of_file,Descriptor,[]) :- close(Descriptor), !.
file_stream(Term,Descriptor,[Term|file_stream(Descriptor)]).
```

With similar code we could retrieve data across a network, or from some other specialized source. In other words, stream processing gives a simple technique for implementing *transparency among data sources*. Transparency will always be a very powerful capability for data analysis.

7. Final Remarks

We have seen how to implement stream data analysis in Prolog elegantly, even though an elegant implementation is not obvious at first. In fact, we have shown how many problems in using Prolog for stream processing can be overcome.

This raises a point: Prolog can be what you want. People may say Prolog 'cannot' do something, while in fact with a little ingenuity it is not only 'possible' to do what you want, it is natural. Prolog is a very powerful assembler for building larger systems.

After some experience with the tuple-at-a-time and whole-query-at-a-time Prolog/DBMS interfaces that have been developed to date, we feel the best way to integrate Prolog and databases is through streams. Minor extensions to Prolog are sufficient to provide efficient stream processing [56]. Stream processing is also a natural approach for applications like data analysis. It is peculiar that stream processing has not been emphasized more heavily for temporal query processing, as well as for basic relational query processing.

The goal of the Tangram project is to provide a powerful environment for modeling. Probably the most challenging aspect of this goal is in supporting exploratory data analysis in a way that has not been accomplished before. Not only is it necessary to support an increase in the quantity of data that can be effectively analyzed, but also to support analysis of symbolic and structured data. Besides the influence of the Log(F) system described at length earlier, The Tangram system has drawn on the designs of a number of previous systems which have included stream concepts. These include FAD [3], FQL [13, 14, 15], various dataflow database systems [4, 5, 8, 34], and LDL [7, 67].

In the past, query languages have been limited to the scope and flexibility foreseen by their designers. We have now reached a period where applications such as modeling and temporal data processing demand flexibility and expressiveness above all else. At the same time, it is important that a query language introduce some structure, or paradigm, that helps it maintain coherence in the user's mind. Stream processing with transducers is one possible paradigm. Clearly there is much more work to be done here, but the approach we have sketched leads to extensible, expressive data analysis systems.

Acknowledgement

The author is grateful to Cliff Leung, Brian Livezey, Dick Muntz for improvements they suggested on the manuscript.

References

1. Abelson, H. and G. Sussman, *The Structure and Analysis of Computer Programs*, pp. 242-292, MIT Press, Boston, MA, 1985.
2. Allen, F.E. and J. Cocke, "A Catalogue of Optimizing Transformations," in *Design and Optimization of Compilers*, ed. R. Rustin, pp. 1-30, Prentice-Hall, 1971.
3. Bancilhon, F., T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, a Powerful and Simple Database Language," *Proc. Thirteenth Intnl. Conf. on Very Large Data Bases*, Brighton, England, 1987.
4. Batory, D.S., "A Molecular Database Systems Technology," Tech. Report TR-87-23, Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712, 1987.
5. Batory, D.S., T.Y. Leung, and T. Wise, "Implementation Concepts For an Extensible Data Model and Data Language," *ACM Trans. Database Systems*, vol. 13, no. 3, pp. 231-262, Sept. 1988.
6. Becker, R.A. and J.M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth, Inc., Belmont, CA, 1984.
7. Beeri, C., S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur, "Sets and Negation in a Logic Database Language (LDL1)," *Proc. Sixth ACM Symp. on Principles of Database Systems*, pp. 21-37, San Diego, March 1987.
8. Bic, L. and R.L. Hartmann, "AGM: A Dataflow Database Machine," Technical Report, Dept. of Information and Computer Science, Univ. of California at Irvine, February 1987.
9. Blum, M., R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan, "Time Bounds for Selection," *J. Comput. System Sci.*, vol. 7, pp. 448-461, 1972.
10. Bocca, J., "EDUCE – A Marriage of Convenience: Prolog and a Relational DBMS," *Proc. 1986 Symposium on Logic Programming*, pp. 36-45, Salt Lake City, UT, September 1986.
11. Bocca, J., "On the Evaluation Strategy of EDUCE," *Proc. ACM SIGMOD Intnl. Conf. on Management of Data*, pp. 368-378, Washington, D.C., May 1986. Appeared as *ACM SIGMOD Record* 15:2, June 1986.
12. Brodie, M. and M. Jarke, "On Integrating Logic Programming and Databases," in *Expert Database Systems: Proceedings from the First Intnl. Conference*, ed. L. Kerschberg, pp. 191-208, Benjamin/Cummings, 1986.
13. Buneman, P. and R.E. Frankel, "FQL – A Functional Query Language," *Proc ACM SIGMOD Intnl. Conf. on Management of Data*, pp. 52-57, Boston, MA, May-June, 1979.
14. Buneman, P., R. Nikhil, and R. Frankel, "A Practical Functional Programming System for Databases," *Proc. ACM Conf. on Functional Languages and Computer Architecture*, pp. 195-201, 1981.
15. Buneman, P., R.E. Frankel, and Rishiyur Nikhil, "An Implementation Technique for Database Query Languages," *ACM Trans. Database Systems*, vol. 7, no. 2, pp. 164-186, June 1982.
16. Burge, W.H., "Stream Processing Functions," *IBM J. Res. Develop.*, vol. 19, no. 1, pp. 12-25, 1975.

17. Burge, W.H., *Recursive Programming Techniques,* Addison-Wesley, Reading, MA, 1975.

18. Carlsson, M. and J. Widen, "SICStus Prolog User's Manual," Research Report SICS R88007, Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Kista, SWEDEN, February 1988.

19. Ceri, S., G. Gottlob, and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," in *Expert Database Systems: Proceedings from the First Intnl. Conference*, ed. L. Kerschberg, pp. 207-223, Benjamin/Cummings, 1987.

20. Chang, C.L. and A. Walker, "PROSQL: A Prolog Interface with SQL/DS," in *Expert Database Systems: Proceedings from the First Intnl. Conference*, ed. L. Kerschberg, pp. 233-246, Benjamin/Cummings, 1986.

21. Chau, L. and D.S. Parker, "Executable Temporal Specifications with Functional Grammars," Technical Report CSD-880046, UCLA Computer Science Dept., June 1988.

22. Chau, L. and D.S. Parker, "Functional Logic Grammar: A New Scheme for Language Analysis," Technical Report CSD-880097, UCLA Computer Science Dept., December 1988.

23. Clark, K. and S. Gregory, "Notes on the Implementation of PARLOG," *J. Logic Programming*, vol. 2, no. 1, pp. 17-42, 1985.

24. Clark, K.L. and F.G. McCabe, "Control facilities of IC-PROLOG," in *Expert systems in the microelectronic age*, ed. D. Michie, Edinburgh U. Press, 1979.

25. Darvas, N., *How I Made 2,000,000 Dollars in the Stock Market,* Lyle Stuart, Inc., 120 Enterprise Ave., Secaucus, NJ 07094, 1986.

26. DeGroot, D. and G. Lindstrom, *Logic Programming: Functions, Relations, and Equations,* Prentice-Hall, 1986.

27. Fisher, K.L., *The Wall Street Waltz,* Contemporary Books, Inc., Chicago, 1987.

28. Freytag, J.C., "Translating Relational Queries Into Iterative Programs," LNCS 261, Springer-Verlag, New York, 1987.

29. Friedman, D.P. and D.S. Wise, "CONS Should Not Evaluate Its Arguments," in *Automata, Languages and Programming*, ed. S. Michaelson and R. Milner, eds., Edinburgh University Press, Edinburgh, 1976.

30. Fuller, W.A., *Introduction to Statistical Time Series,* John Wiley & Sons, 1976.

31. Gale, W.A., "REX Review," in *Artificial Intelligence & Statistics*, ed. W.A. Gale, Addison-Wesley, 1986.

32. Ghosh, S., C.C. Lin, and T. Sellis, "Implementation of a Prolog-Ingres Interface," *SIGMOD Record*, vol. 17, no. 2, June 1988.

33. Goldberg, A. and R. Paige, "Stream Processing," *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pp. 53-62, Austin, TX, August 1984.

34. Golshani, F., "The Basis of a Dataflow Model for Query Processing," *Proc. Eighteenth HICSS*, Honolulu, January 1985.

35. Gray, P.M.D., *Logic, Algebra and Databases,* Ellis Horwood, Ltd./Halsted Press/John Wiley & Sons, 1984.

36. Hall, P.A.V., "Relational Algebras, Logic, and Functional Programming," *Proc. 1984 ACM SIGMOD Intnl. Conf. on Management of Data*, pp. 326-333, Boston, MA, June 1984.

Appeared as *ACM SIGMOD Record* 14:2, 1984.

37. Henderson, P. and J.H. Morris, Jr., "A Lazy Evaluator," *Proc. Third ACM Symposium on Principles of Programming Languages*, pp. 95-103, 1976.
38. Henderson, P., *Functional Programming: Application and Implementation,* Prentice/Hall International, 1980.
39. Ioannidis, Y.E., J. Chen, M.A. Friedman, and M.M. Tsangaris, "BERMUDA – An Architectural Perspective on Interfacing Prolog to a Database Machine," in *Expert Database Systems: Proceedings from the Second Intnl. Conference*, ed. L. Kerschberg, pp. 229-256, Benjamin/Cummings, 1989.
40. Jarke, M., J. Clifford, and Y. Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System," *Proc. 1984 ACM SIGMOD Intnl. Conf. on Management of Data*, pp. 296-306, Boston, MA, June 1984. Appeared as *ACM SIGMOD Record* 14:2, 1984.
41. Kahn, G. and D. McQueen, "Coroutines and Networks of Parallel Processes," *IFIP 77*, North-Holland, Amsterdam, 1977.
42. Kahn, K., "A Primitive for the Control of Logic Programs," *Proc. Symp. on Logic Programming*, pp. 242-251, IEEE Computer Society, Atlantic City, 1984.
43. Landin, P.J., "A Correspondence Between Algol 60 and Church's Lambda-Notation, Parts I and II," *Communications of the ACM*, vol. 8, no. 2 and 3, pp. 89-101, 158-165, 1965.
44. Li, P-Y.P. and A.J. Martin, "The Sync Model: A Parallel Execution Method for Logic Programming," *Proc. Symp. on Logic Programming*, pp. 223-234, IEEE Computer Society, Salt Lake City, 1986.
45. Lindstrom, G. and P. Panangaden, "Stream-Based Execution of Logic Programs," *Proc. Symp. on Logic Programming*, pp. 168-176, IEEE Computer Society, Atlantic City, 1984.
46. McCrosky, C.D., J.J. Glasgow, and M.A. Jenkins, "Nial: A Candidate Language for Fifth Generation Computer Systems," *Proc. ACM'84 Annual Conference*, pp. 157-166, San Francisco, October 1984.
47. McLeod, D., "A Semantic Data Base Model and its Associated Structured User Interface," Ph.D. Dissertation, Dept. EE&CS, MIT, Cambridge, MA, 1978.
48. More, T., "Axioms and Theorems for a Theory of Arrays," *IBM J. Res. Develop*, vol. 17, no. 2, pp. 135-175, 1973.
49. More, T., "The Nested Rectangular Array as a Model of Data," *Proc. APL79*, pp. 55-73, May 1979.
50. Naish, L., "All Solutions Predicates in Prolog," *Proc. Symp. on Logic Programming*, pp. 73-77, IEEE Computer Society, Boston, 1985.
51. Napheys, B. and D. Herkimer, "A Look at Loosely-Coupled Prolog/Database Systems," in *Expert Database Systems: Proceedings from the Second Intnl. Conference*, ed. L. Kerschberg, pp. 257-272, Benjamin/Cummings, 1989.
52. Narain, S., "LOG(F): A New Scheme for Integrating Rewrite Rules, Logic Programming and Lazy Evaluation," Technical Report CSD-870027, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.
53. Narain, S., "LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1988.

54. Nussbaum, M., "Combining Top-Down and Bottom-Up Computation in Knowledge Based Systems," in *Expert Database Systems: Proceedings from the Second Intnl. Conference*, ed. L. Kerschberg, pp. 273-310, Benjamin/Cummings, 1989.

55. Page, T.W., "Prolog Basis for A Data-Intensive Modeling Environment," Dissertation Prospectus, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

56. Parker, D.S., T. Page, and R.R. Muntz, "Improving Clause Access in Prolog," Technical Report CSD-880024, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

57. Parker, D.S., R.R. Muntz, and L. Chau, "The Tangram Stream Query Processing System," *Proc. Fifth Intnl. Conf. on Data Engineering*, pp. 556-563, Los Angeles, CA, February 1989.

58. Parker, D.S., "Stream Processing: An Effective Way to Integrate AI and DBMS," Technical Report CSD-890005, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, January 1989.

59. Pearl, J., "A Space-Efficient On-Line Method of Computing Quantile Estimates," *J. Algorithms*, vol. 2, no. 2, pp. 164-177, 1981.

60. Peyton-Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice/Hall International, Englewood Cliffs, NJ, 1987.

61. Sciore, E. and D.S. Warren, "Towards an Integrated Database-Prolog System," in *Expert Database Systems: Proceedings From the First Intnl. Workshop*, ed. L. Kerschberg, pp. 293-305, Benjamin/Cummings, Menlo Park, CA, 1986.

62. Shipman, D.W., "The Functional Data Model and the Data Language DAPLEX," *ACM Trans. Database Systems*, vol. 6, no. 1, pp. 140-173, March 1981.

63. Sibley, E.H. and L. Kerschberg, "Data architecture and data model considerations," *Proc. AFIPS National Computer Conf.*, pp. 85-96, June 1977.

64. Snodgrass, R. and S. Gomez, "Aggregates in the Temporal Query Language TQuel," Tech. Rep. TR86-009, Computer Science Dept., Univ. of North Carolina, Chapel Hill, March 1986.

65. Subrahmanyam, P.A. and J-H. You, "Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming," *Proc. Symp. on Logic Programming*, pp. 144-153, IEEE Computer Society, Atlantic City, 1984.

66. Tick, E., *Memory Performance of Prolog Architectures*, Kluwer Academic Publishers, Norwell, MA, 1988.

67. Tsur, S. and C. Zaniolo, "LDL: A Logic-Based Data Language," *Proc. Twelfth Intnl. Conf. on Very Large Data Bases*, pp. 33-41, Kyoto, Japan, 1986.

68. Waters, R., "A Method for Analyzing Loop Programs," *IEEE Trans. Software Engineering*, vol. 5, no. 3, pp. 237-247, 1979.

69. Zaniolo, C., "Prolog — A Database Query Language for All Seasons," in *Expert Database Systems: Proceedings From the First Intnl. Workshop*, ed. L. Kerschberg, pp. 219-232, Benjamin/Cummings, 1986.