Modeling applications routinely generate large quantities of simulation data, and analysis of this data requires a system that differs in significant ways from existing database systems. The data often takes the form of time series, and query processing requires both stream processing techniques and heavy numerical computations.

In this paper we describe the Tangram query processing system, the Tangram Stream Processor (TSP). It is an extensible system based on a functional sublanguage of Prolog that provides a programmable stream processing capability with a number of interesting characteristics.

1. Introduction

The relational data model is founded on set theory: all relations are viewed as sets of tuples. Many developments have encouraged generalization of this model to one of ordered sets. For example, ordered data can be processed much more efficiently than unordered data can be. In fact, many standard query evaluation techniques are described in terms of operators (actors, filters, mappings) acting on ordered sequences of tuples. Moreover, temporal query processing seems to necessitate some kind of ordering if important kinds of queries are to be efficiently answerable. Also, non-first-normal-form data frequently requires some kind of list structuring, essentially implementing ordered sets. Finally, of course, the order of the tuples in relations is important in presentation of the relations to users.

In many important situations, then, it is advantageous to generalize the set foundation of the relational data model to an ordered set model. We call ordered sets streams. Stream-oriented processing is certainly not a new subject, although it has only recently come into its own right as a programming paradigm.

In this paper we describe the Tangram query processing system, the Tangram Stream Processor (TSP). It is an extensible system based on a functional sublanguage of Prolog that provides a programmable stream processing capability with a number of interesting characteristics.

1.1. Stream Processing

Concurrent, object-oriented, functional, and logic programming paradigms all intersect elegantly in the abstraction of streams. Many stream processing systems 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 [13] and the other ‘committed choice’ parallel programming systems (Concurrent Prolog, GHC, etc.).

2) They introduce ‘parallel and’ or ‘parallel or’ operators into ordinary Prolog [19].

3) They are extended Prolog systems that introduce streams by adding functional programming constructs [14, 16, 20, 25, 36]. The thrust of this approach is to make Prolog more like either Lisp or Smalltalk or both.

TSP has drawn on the designs of a number of previous systems which have included stream concepts. These include FAD [4], various dataflow database systems [5, 6, 7, 10, 15], and LDL [9, 37].

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 a better way to integrate Prolog and databases is through streams. Only minor extensions to Prolog are sufficient to provide fairly efficient stream processing [29]. A stream interface offers an effective medium between these two alternatives, uniformly integrating bulk operations at the DBMS end with incremental evaluation at the Prolog end. Prolog stream processing avoids backtracking through a database, using efficient iterative (tail recursive) processing instead. It is a natural approach for applications like analysis of modeling data.

This work done under the Tangram project, supported by DARPA contract F29601-87-C-0072.
1.2. Streams and Temporal Query Processing

One area where streams are very important for query processing is for temporal data, i.e., data with explicit or implicit time ordering. The analysis of streams has been done for many years as "time series analysis". Recently, the subject of time in databases has gotten increasing attention as more applications requiring temporal reasoning have been uncovered, and many interesting systems handling temporal queries in novel ways have been developed.

Previous research has concentrated either on database processing, or on representational issues and generality of modeling. Two important database systems include:

1. TQuel [34], a relational query language with embedded time primitives, is an extension of Quel, both syntactically and semantically. TQuel is essentially a relational query language, resting on the relational model.

2. The Time Sequence approach of Shoshani [31,32] characterizes properties of temporal data and temporal operators without restriction to the relational model. Data are organized into Time Sequence Collections (TSCs), which can take both relational and stream-like representations. Five basic operators provide an algebra working on TSCs.

These systems emphasize performance and complete handling of a well-defined set of query operators. Other researchers in temporal query processing have worked on more complex modeling, combining work on temporal logic and existing representational systems to define new approaches. Sadri [30] reviews three general recent approaches to temporal reasoning:

1. The "event calculus" of Kowalski [17] is an approach for reasoning about events and time within a logic programming framework.

2. Allen's approach [2,3] is similar to the event calculus, defining a set of binary predicates giving basic relationships among time intervals (whether they overlap, one precedes the other, etc.).

3. Lee, Coelho and Cotta [18] present a temporal system for representing and reasoning about time-dependent information and events, specifically for business database applications.

In these approaches it is peculiar that stream processing has not been emphasized more heavily for temporal query processing, as well as for basic relational query processing. Tangram's stream processing approach permits it to handle queries definable under each of the systems listed here.

1.3. The Tangram Stream Processor

Below we describe the Tangram Stream Processor (TSP), a system founded on the abstraction of stream transducers. A transducer is a mapping from some number of input streams to one or more output streams. Thus, a transducer may be viewed as an automaton. However, a transducer can take parameters, and as such need not have only a finite number of states. Thus, it is better to view transducers as mappings instead. Transducers are the basic building blocks of TSP, and are maintained in an (extensible) library. Since arbitrary transducers are permitted, the expressive power of TSP is equivalent to that of any general programming language. Consequently, the stream-based transducer model is more general than many previous approaches: it is capable of handling traditional database queries and non-traditional queries that reason about time in event databases.

TSP has several further unique aspects:

1. TSP permits operation on general stream structures, including for example both lists and array models of data. It supports definition of and parallel evaluation of operators on these stream structures, including the operator families of the APL programming language, NIAL [21], and the Nested Array model of data upon which both are based [22,23]. This includes the ability to define higher-order operators on streams, such as aggregate operators (min, max, sum, etc.), APL's reduction operator, LISP's maplist, etc. In addition, it permits us to define many useful statistical operators on streams, as in the S data analysis system [8].

2. TSP permits operation on infinite streams. A stream may represent a non-terminating sequence of values. This is not permitted, for example, by APL.

3. TSP permits both lazy and eager evaluation of streams. Lazy evaluation permits efficient evaluation of some kinds of queries.

4. TSP transducers are naturally implemented as concurrent processes. These transducers permit easy specification of process boundaries, a feature not enjoyed by some parallel Prolog systems.

The resulting system may be used for 'database-flow' processing, a combination of 'dataflow' and database processing, as well as general feature extraction and data reduction operations that fit in a pipeline structure.

Execution of queries in TSP is quite efficient, in the common situation that the input streams are sorted properly. Also, TSP can handle kinds of queries that are not easily handled by relational query processing systems, including the following:

1. Sliding window queries [31]
2. Event calculus queries [17]
3. Pattern analysis queries [11, 12]
4. Abstractive state information from event data
5. Reasoning about time.

As an example of a query that reasons about time, consider asking about what investment strategy would have been optimal over a given period of stock market history. This requires innovative accumulation of dividends, interest rates and rules for compounding interest, days which are holidays, and many other important details. These 'hindsight queries' illustrate the potential of stream processing in database analysis.
1.4. Organization of this Paper
Section 2 gives the formal definition of TSP, and provides an example library of transducer operations and their Prolog implementation. We henceforth assume that the reader is familiar with Prolog. A good introduction to Prolog can be found, for example, in [35].

Section 3 describes the TSP stream transduction mechanism in detail. Finally, performance of stream-based temporal query processing system is briefly discussed, along with reflections on improvements to TSP and avenues for future work.

2. Log(F)
The Tangram Stream Processor rests on Log(F), a combination of Prolog and a functional language called F*, developed by Sanjai Narain at UCLA [27, 28]. 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 [24].

2.1. Overview of F* and Log(F)
F* is a rewrite rule language. In F*, all statements are rules of the form

\[ \text{LHS} \Rightarrow \text{RHS} \]

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

Consider the following two rules, defining how lists may be appended:

\begin{align*}
\text{append}(\{\}, W) & \Rightarrow W. \\
\text{append}(\{U|V\}, W) & \Rightarrow \{U\} \text{append}(V, W). 
\end{align*}

Like the Prolog rules for appending lists, this concise description provides all that is necessary.

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:

\begin{align*}
\text{reduce} \left( \text{append}(A, B), C \right) & \leftarrow \text{reduce}(A, \{\}), \\
& \text{reduce}(B, C). \\
\text{reduce} \left( \text{append}(A, B), C \right) & \leftarrow \text{reduce}(A, \{D|E\}), \\
& \text{reduce}(D \text{append}(B, E), C). 
\end{align*}

The two F* rules are as concise as, and essentially equivalent to, the Prolog definition

\begin{align*}
\text{append}(\{\}, W, W). \\
\text{append}(\{U|V\}, W, U|L) & \leftarrow \text{append}(V, W, L). 
\end{align*}

Unlike 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 above provides this capability 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

\[ \text{?- reduce} \left( \text{append}(\{1,2,3\}, \{4,5,6\}), X \right). \]

yields the result

\[ X = \{1|\text{append}(\{2,3\}, \{4,5,6\})\}. \]

That is, in one reduce step, only the head of the resulting appended list is computed. The tail, \text{append}(\{2,3\}, \{4,5,6\}), can then be further reduced if this is necessary. Demand-driven computation like this is referred to as lazy evaluation or delayed evaluation, and is basic to stream processing [1].

The astute reader will have noticed that, in order for the reduce rules above to work as we are claiming, we will have to add two definitions:

\begin{align*}
\text{reduce} \left( \{\}, \{\} \right). \\
\text{reduce} \left( \{X|Y\}, \{X|Y\} \right). 
\end{align*}

In F*, functors like \{\} and \{\_\|\_\} with this property are called constructor symbols. Terms whose functors are constructor symbols are said to be simplified; they cannot be reduced further.

The main restriction on Log(F) rules \text{LHS} \Rightarrow \text{RHS} is that \text{LHS} must be of the form

\[ f(t_1, \ldots, t_n) \Rightarrow \text{RHS} \]

where \( n \geq 0 \), and each of the \( t_i \) is either a variable or a term whose functor is a constructor symbol. This restriction guarantees efficient implementation. In order to guarantee soundness and completeness properties, restrictions on variables are also made: first, no variable may appear twice in \text{LHS} (the ‘linearity’ restriction), and second, every variable in \text{RHS} must also appear in \text{LHS} [27, 28].

There is one more important point about the integration of F* with Prolog. Where F* computations are naturally lazy because of their implementation with reduction rules, Log(F) permits some eager computation as well. Essentially, eager computations invoke routines outside F*. For example, in the Log(F) code

\[ \text{count}(\{X|S\}, N) \Rightarrow \text{count}(S, N+1). \]

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

\[ \text{reduce(count}(A, N), Z) \leftarrow \text{reduce}(A, \{X|S\}), \\
N \text{ is } N+1, \text{reduce(count}(S, M), Z). \]

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.

It is easy to develop programs with compact sets of rewrite rules. For example, the following is an executable Log(F) program for computing primes via the sieve of Eratosthenes:

\[ 558 \]
primes => sieve(intfrom(2)).
intfrom(N) => [N|intfrom(N+1)].
sieve([U|V]) => [U|sieve(filter(U,V))].
:- eager multiple/2.
multiple(U,A,true) :- 0 is U mod A, !.
multiple(_,_,false).
filter(A, [U|V]) => if (multiple(U,A), filter(A,V), [U|filter(A,V)]).

The intfrom rule generates an infinite stream of integers. The rule for filter uses the eager Prolog predicate multiple. 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))))))
...

For other useful examples of the combination of lazy and eager evaluation, see [26].

2.2. Advantages of Log(F)

Log(F) is a superior formalism for stream processing, and thus for database query processing. From the examples above it is clear that the rules have a functional flavor. Stream operators are easily expressed using recursive functional programs. The syntax is convenient, and can be considered a useful query language in its own right.

Furthermore, 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.

It turns out furthermore that Log(F) has a formal foundation that captures important aspects of stream processing:

(1) Determinate (non-backtracking) code is easily detected through syntactic tests only. This 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. Again this avoids problems encountered by other 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 that produce highly-optimized code.

We stress that Log(F) is an addition to Prolog without changing Prolog's fundamental primitives such as its unification algorithm or control strategy. All Log(F) code shown in this paper actually runs as shown in a standard Prolog environment.

3. TSP: Stream Processing

The Tangram Stream Processor (TSP) is an extensible stream processing system based directly on the Log(F) system described above. A prototype implementation using SICS Prolog has run for eight months, and has gradually evolved to the state described here. This section discusses the basic functionality of TSP.

The syntax of Log(F) is quite compact. Readers not accustomed to Prolog programming may be daunted by it at first. We hasten to point out that a graphical interface can permit users to specify queries as compositions of stream transducers once an appropriate library of transducers is defined, and avoid the details of syntax.

3.1. Transducers

With Prolog and Log(F) in mind, we follow the syntactic convention identifying a stream with a list of ground terms. For example, the list

```prolog
[signal(128.97.28.26, transmission_failure, Oct 19 08:46:08.171 PST 1987),
]
```

can be a stream. This convention is observed by many parallel logic programming systems.

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

```prolog
[ net_failure, net_failure, cpu_failure ].
```

In addition to recognizing specific patterns, a transducer is capable of recording and summarizing properties of any input stream as it is scanned.

Formally, we define a TSP transducer to be any collection of Log(F) rules, relaxing the restrictions on variables required by Log(F) for completeness. This definition is easily verified to cover mappings from streams to streams, aggregate computations, parsers, and a number of other general kinds of mappings.
In spite of this generality, certain kinds of transducers appear frequently in the context of database query processing. One of the most common, which we call a **simple stream transducer**, defines a sequential mapping between input stream items and output stream subsequences. This kind of transducer can be defined by an initial state and three mappings:

A simple stream transducer $T$ is a 4-tuple $(\text{InitialState}, \text{Transduction}, \text{NewState}, \text{FinalTransduction})$, where:

- **InitialState** is the state of the transducer when it begins;
- **Transduction** maps the current state and current stream input(s) to new stream output(s). Stream inputs can be ignored. A stream output can be $\top$, specifying that output stream is not to be changed;
- **NewState** maps the current state and current stream input(s) to the next state;
- **FinalTransduction** specifies the final output(s) to be written on streams when no input is left.

Simple stream transducers can be expressed concisely with TSP. For concreteness, assume that a transducer takes one stream as input, does not ignore its input in any state, and produces one stream as output. It can then be expressed in TSP as:

$$\text{TransducerName}(\text{Stream}) \Rightarrow \text{TransducerName}(\text{Stream}, \text{InitialState}).$$

$$\text{TransducerName}(\top, \text{State}) \Rightarrow \text{FinalTransduction}(\text{State}).$$

$$\text{TransducerName}(\text{Input}, \text{State}) \Rightarrow \text{TransducerName}(\text{NewState}(\text{Input}, \text{State})) \Rightarrow \text{FinalTransduction}(\text{State}).$$

The following simple example of a simple stream transducer in TSP converts system signal messages into single values:

$$\text{tr}(\text{Stream}) = \text{tr}(\text{Stream}, 0).$$

$$\text{tr}([\text{signal}(-, \text{transmission} \text{-} \text{failure}, -)], \text{State}) \Rightarrow \text{net} \text{-} \text{failure}(\text{State}).$$

$$\text{tr}([\text{signal}(-, \text{site} \text{-} \text{not} \text{-} \text{responding}, -)], \text{State}) \Rightarrow \text{cpu} \text{-} \text{failure}(\text{State}).$$

The number of cpu failures encountered so far is the state of the transducer. When the input stream is exhausted, the transducer outputs the final translation $\text{net} \text{-} \text{failure}(\text{State})$. Meanwhile it translates the input stream into the output stream item by item. Note that the third rule for $\text{tr}$ uses $\text{net} \text{-} \text{failure}(\text{State})$ instead of the equivalent $\text{append}(\text{net} \text{-} \text{failure}, [\text{tr}(\text{State})])$ since it is more efficient.

Since transducers are mappings, they may be combined into expressions. **Stream expressions** can thus be defined inductively:

1. A stream $S$ is a stream expression.
2. If $S_1, \ldots, S_n$ are stream expressions, and $f$ is transducer, then $f(S_1, \ldots, S_n)$ is a stream expression.

Stream expressions determine the algebra of transducers generated by whatever initial library of transducers we choose.

Transducers overlap with a number of important paradigmatic concepts, including automata, objects, actors, grammars, parsers, relations, and functions. A query processing system based on transducers has features of each of these concepts, and should not be labeled as one or the other.

For the rest of this section we show how important queries can be expressed as transducers, including:

1. basic temporal mapping, and
2. database query processing.

A formalism for pattern analysis in Log(F) is investigated in [11, 12] but is beyond the scope of this paper.

### 3.2. Basic Temporal Mapping

The discussion above shows how simple stream transducers can be developed in TSP. In principle, a user need only supply the four items

$$\text{(InitialState}, \text{Transduction}, \text{NewState}, \text{FinalTransduction})$$

to define a simple stream transducer, and in fact compile-time ‘macro expansion’ (or ‘partial evaluation’) can be used to fill in the simple stream transducer template above with these items.

Let us consider one example in more detail to cement this understanding, and relate it to temporal query processing. In [33], Snodgrass and Gomez use the following temporal database:

- position(jane, assistant, 25000, 9/71)
- position(tom, assistant, 23000, 9/75)
- position(jane, associate, 33000, 12/76)
- position(merrie, assistant, 25000, 9/77)
- position(jane, full, 44000, 11/80)
- position(tom, fired, 0, 12/80)
- position(merrie, associate, 40000, 12/82)
- submitted(merrie, cam, 9/78)
- submitted(merrie, tods, 5/79)
- submitted(jane, cam, 11/79)
- submitted(merrie, jacy, 8/82)

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

- position(jane, assistant, 25000, 9/71)
- position(tom, assistant, 23000, 9/75)
- position(jane, associate, 33000, 12/76)
- position(merrie, assistant, 25000, 9/77)
- position(jane, full, 44000, 11/80)
- position(tom, fired, 0, 12/80)
- position(merrie, associate, 40000, 12/82)
- submitted(merrie, cam, 9/78)
- submitted(merrie, tods, 5/79)
- submitted(jane, cam, 11/79)
- submitted(merrie, jacy, 8/82)
facultyHistory(Positions) =>
  facultyHistory(Positions, []).  

facultyHistory([], CurrentFaculty) =>
  CurrentFaculty.  

facultyHistory([position(F,P,W,D) | S], CurrentFaculty) =>
  append(facultyRecord(F,P,W,D, CurrentFaculty),
  reduce (newFaculty (CurrentFaculty, F, P, W, D))
  facultyHistory(S, reduce (newFaculty (CurrentFaculty, F, P, W, D)))
).  

This transducer is defined by the 4-tuple ([], facultyRecord, facultyHistory, identityMapping) where identityMapping is the identity mapping:

identityMapping(X) => X.  

to complete the definition of the transducer, we must give the transduction mappings facultyRecord and facultyHistory. These can be defined as follows:

facultyRecord( F,P,W,D, []) =>
  if (OldF == F,
    [faculty(F, OldP, OldW, OldD, _) | Faculty],
    facultyRecord(F, P, W, D, Faculty))
).

newFaculty( [], F,P,W,D ) =>
  [faculty(F,P,W,D,_)].

newFaculty([faculty(Fl,P1,W1,D1,_) | Faculty],
  F,P,W,D ) =>
  if (Fl == F,
    [faculty(F,P,W,D,_) | Faculty],
    [faculty(Fl,P1,W1,D1,_) | newFaculty(Faculty, F,P,W,D)])
).

The output stream obtained by reducing facultyHistory(tuples(position,4)) is as follows:

faculty(jane, assistant, 25000, 9/71, 12/76).
faculty(jane, associate, 33000, 12/76, 11/80).
faculty(tom, assistant, 23000, 9/75, 12/80).
faculty(merrie, assistant, 25000, 9/77, 12/82).
faculty(jane, full, 44000, 11/80, _).
faculty(tom, fired, 0, 12/80, _).
faculty(merrie, associate, 40000, 12/82, _).

Here tuples(position,4) is a term that yields a stream of the tuples from the 4-column relation position.

3.3. Database Query Processing

It is easy to express simple database queries with TSP. Standard relational algebra operators all can be used as transducers. TSP defines the relational selection transducer

select( Stream, Template, Condition )

which selects from Stream all items that match Template and, in addition, satisfy Condition. It is a stream analogue of findall in Prolog systems.

TSP provides evaluation of stream expressions through Log(F) reduction:

reduce( Expression, Stream )
reduce_eagerly( Expression, Stream )

where Expression is a stream expression that is to be reduced to Stream. These differ in that reduce performs lazy reduction, yielding only one member of the stream at a time, while reduce_eagerly eagerly evaluates the stream expression, not halting until the output stream is complete.

Consider the query: "Find the papers Jane wrote while she was an associate professor." This query involves a relational join and selection. The TSP goal

?- reduce_eagerly( janeQuery, Papers ).

obtains the desired papers if we define two transducers:

janeQuery =>

janeQuery( select( facultyHistory(tuples(position,4)),
  faculty(jane, associate, _, _, _),
  true )).

facultyPapers([faculty(Name, _, _, StartDate,EndDate) | _]) =>
  select( tuples(submitted,3),
  submitted(Name, Journal, Date),
  notLater(StartDate, Date),
  notLater(Date, EndDate) )
).

The join is essentially performed as a nested loops join with two selections, but here the first selection produces only one tuple.

We can implement any operations on time values that are needed. For example, notLater can be implemented as a Prolog predicate as follows:

notLater(Month1/Year1, Date)

notLater(Month1/Year1, Month2/Year2)

+: var(Date).

notLater(Month1/Year1, Month2/Year2)

notLater(Month1/Year1, Month2/Year2)

:- Year1 =< Year2.

notLater(Month1/Year1, Month2/Year2)

notLater(Month1/Year1, Month2/Year2)

:- Month1 =< Month2.

Variables are used to indicate both indefinite and perpetual dates.

Other queries can be composed using relational operators. Having presorted input streams is one of the more important assumptions for many stream processing operations, and therefore, these transducers are implicitly preceded by a sort operation that guarantees the terms in the input streams are in the appropriate order for processing.
4. Conclusions
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.

This paper has introduced TSP, the Tangram Stream Processor. TSP resists on Log(F), an elegant rewrite-rule extension of Prolog that provides functional programming and lazy evaluation. Stream processing is straightforward to implement in functional systems with lazy evaluation.

As illustrated with a sequence of examples, TSP provides a simple yet effective way to integrate the symbolic processing power of Prolog and the raw data management power of databases through streams. A stream connection between these two highly evolved systems is a natural approach for applications like analysis of modeling data.

A great deal of work remains in exploring the performance options in this stream connection. A major concern is query optimization, i.e., determination of an execution plan for a query that is computationally more efficient if one exists. When expressed as a problem of optimizing a combination of transducers on streams, the optimization problem has very interesting structure. It seems likely that program transformation techniques can be applied in both a theoretically challenging and computationally useful way.

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 Tangram Stream Processor is a step in the direction of extensible, more expressive query processing systems.

References


