THE TANGRAM STREAM QUERY PROCESSING SYSTEM

D. Stott Parker
Richard R. Muntz
Lewis Chau

March 1988
CSD-880025

The Tangram Stream Query Processing System

D. Stott Parker

Richard R. Muntz

Lewis Chau

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

ABSTRACT

Tangram is an environment for modeling. It supports development and management of models, simulation of models, analysis of simulation output and analysis of models in general. Its current focus is on computer system performance modeling.

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 therefore query processing requires both stream processing techniques and heavy numerical computations (e.g., basic statistical and time series analysis) beyond ordinary aggregates.

One of the driving concepts behind Tangram has therefore been the combination of large-scale data access and data reduction with a powerful programming environment. The Tangram environment is based on Prolog, extending it with a number of features, including process management, distributed database access, and generalized stream processing.

This paper describes the *Tangram Stream Processor (TSP),* the part of the Tangram environment performing query processing on large streams of data. The paradigm of transducers on streams is used throughout this system, providing a '*database-flow*' (database dataflow) computation capability.

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

The Tangram Stream Query Processing System

D. Stott Parker
Richard R. Muntz
Lewis Chau
Department of Computer Science
University of California
Los Angeles, CA 90024-1596

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. The Tangram Modeling Environment

Tangram is an environment under development at UCLA for modeling. It is implemented as an extension of Prolog that includes integration with the Unix environment and database managers, and provides distributed processing constructs. The main goal of Tangram is to provide a high-powered interactive modeling environment. It therefore incorporates the following capabilities:

Model Management

Just as DBMS are managers of data, Tangram is a manager of models. Model management includes the storage and retrieval of 'data dictionary' knowledge about available models, workloads (load generators, benchmarks), experiments, experiment output, and tools. Where multiple models are used to describe a single system from different viewpoints or levels of abstraction, model management also provides information on how these models relate, how they can be solved by existing modeling tools, and so forth.

Measurement Data Management

Modeling experiments generate enormous quantities of data. Tangram is concerned with capturing data from different tools, translating it to a common format, storing it, and supporting arbitrary queries with parallel processing. This presents research challenges in that the data is frequently both structured and has a temporal flavor not supported by current DBMS. Also, current DBMS do not support 'exploration' of the data in the way that exploratory data analysis systems do, like the *S* system of AT&T [9]. It is important to be able to support exploration of a model, encouraging a modeler to expand his intuition of its behavior. The modeler should be able to view his model actually 'running' with various kinds of graphical displays, for example. Parallelism is important to make interactive real-time modeling possible.

Advanced Modeling Tools

Current modeling tools typically force the modeler into expressing his model in a limited framework, and investigating the model's behavior with a limited set of query facilities. These tools are also frequently not extensible, i.e., do not permit addition of new features. Tangram is directed at tools permitting declarative specification of 'deep' models – involving complex structuring of knowledge and complex querying of the model behavior. An object-oriented environment for developing these tools will be used, supporting knowledge base management and arbitrary query. The environment will be extensible in permitting the addition of new object classes for models, new families of models, and so forth. Currently a prototype has been built giving a methodology for designing and implementing Markov modeling tools for analytic, statistical, simulation as well as expert-system-like 'conceptual' modeling of computer systems [12].

Tangram is implemented primarily in *C* and Prolog (currently SICStus Prolog). Prolog is an excellent starting point for this development for at least two reasons:

(1) Prolog is unarguably the best existing candidate for a database/knowledge base language. 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 modeling. It naturally supports increases in structure of models and 'expert system' techniques for interpretation of these models.

(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.

To be effective, however, an environment based on Prolog must offer the following features:

Industrial Strength

Increases in the quantity of modeling data and in the complexity of interpretation operations necessitate heavy computation. Parallelism is needed to deal with the increased volume of information. Interactive display of model behavior is essential for effective modeling. An environment like Tangram incorporating these techniques will be successful only if it provides 'industrial strength' performance. Important performance factors come through optimization, advanced data management technology, support at the operating systems level, and hardware parallelism.

Integration

A modeling environment must combine different systems of different types effectively and efficiently. Tools and testbeds for developing models that have taken man-centuries to develop should be accessible conveniently from a workstation. Prolog is excellent for representing and making inferences how these tools and testbeds should be accessed, but efficient access prohibits the use of *'glue job'** connections between them and Prolog. Database systems, for example, require stream access rather than the tuple-at-a-time access adopted by existing Prolog-DBMS connections. Integration comes through modularization, general ability to connect with diverse programs, general knowledge representation of program functions, and support for translation tools.

Support for Evolution and Multiple Models

There are many ways to represent the same information. As models are refined over time they become more detailed, and focus on specific aspects of systems. Also, different models using different abstractions are necessary for representing complex systems, to keep focus and to reduce data processing requirements. Qualitative reasoning about a system, for example, is done on an abstracted model of the system.

Both evolution of models and different views of complex systems require different kinds of models, and hence different languages or *paradigms* for capturing different aspects of the real world. Bobrow, for example, has criticized Prolog on the grounds that there are many programming paradigms other than logic programming, and existing Prolog environments should, but do not yet, support them [14]. For modeling in particular, this criticism is of the essence. Paradigms can be low-level in nature, such as with parallel/distributed processing, object-oriented programming, or constraint satisfaction. They can also be more high-level or 'semantic' in nature, such as with extended queueing networks.

Tangram provides multiple paradigms as sublanguages which can be compiled to Prolog, or to some other fast low-level implementation. This paper describes the functional/stream paradigm of Tangram. It is an extension of Log(F), developed by Sanjai Narain at UCLA [35, 36].

1.2. 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 [16] and the other 'committed choice' parallel programming systems (Concurrent Prolog, GHC, etc.).

(2) They introduce *'parallel and'* or *'parallel or'* operators into ordinary Prolog [28].

(3) They are extended Prolog systems that introduce streams by adding functional programming constructs [19, 23, 27, 33, 45]. The thrust of this introduction is to make Prolog more like either Lisp or Smalltalk or both.

* terminology borrowed from Mike Stonebraker.

TSP has drawn on the designs of a number of previous systems which have included stream concepts. These include FAD [5], various dataflow database systems [6,7,8,13,20], and LDL [10,46].

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 [37]. 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.

1.3. Streams and Temporal Query Processing

One area where streams are very important for query processing is for temporal data, 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 [3,4,11,15,17], and many interesting systems handling temporal queries in novel ways have been developed [2,18,24,25,26,29,39,42,44].

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

(1) TQuel [42], 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 [39,40] 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 [38] reviews three general recent approaches to temporal reasoning:

(1) The ''event calculus'' of Kowalski [24] is an approach for reasoning about events and time within a logic programming framework.

(2) Allen's approach [3,4] 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 [26] 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.

1.4. 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 [30], and the Nested Array model of data upon which both are based [31,32]. 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 [9].

(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. In fact, TSP query processing can be considerably more efficient than that in relational DBMS. For many TSP queries a single scan of the input streams is sufficient, requiring linear time and constant space, while relational DBMS approaches require significantly more resources. Also, TSP can handle kinds of queries that are not easily handled by relational query processing systems, including the following:

1. Sliding window queries [39]
2. Event calculus queries [24]
3. Pattern matching queries (see Section 4 below)
4. Abstracting 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.

1.5. 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 [43].

Section 3 describes the TSP stream transduction mechanism in detail. Section 4 then goes on to discuss pattern matching against streams, and how it relates to stream transduction. 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 [35,36]. 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.

2.1. 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.

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).
```

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

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

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

```
?- 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. The tail, `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:

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

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 *LHS* => *RHS* is that *LHS* must be of the form

$$f(t_1, \ldots, t_n)$$

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 *LHS* (the 'linearity' restriction), and second, every variable in *RHS* must also appear in *LHS*.

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

```
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 equivalent to

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

```
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 [34].

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 must stress strongly that Log(F) is an *extension* of Prolog. All Log(F) code shown in this paper actually runs as shown. The issues here are not so much language design issues as in developing compilers for Log(F) which exploit its restrictions for speed. Where speed is not critical, the full power of Prolog is available to its users now.

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 six months, and has gradually evolved to the state described here. This section discusses the basic functionality of TSP. Later sections describe how stream pattern matching may be specified neatly.

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

```
[
signal('128.97.28.26',transmission_failure,'Oct 19 08:46:05.823 PST 1987'),
signal('128.97.28.26',transmission_failure,'Oct 19 08:46:08.171 PST 1987'),
signal('128.97.28.26',site_not_responding, 'Oct 19 08:46:12.452 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

```
[ net_failure, net_failure, cpu_crash ].
```

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 an 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

(*InitialState,Transduction,NewState,FinalTransduction*),

where:

(1) *InitialState* is the state of the transducer when it begins;

(2) *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 `[]`, specifying that output stream is not to be changed;

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

(4) *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:

```
TransducerName(Stream)   =>  TransducerName(Stream, InitialState).

TransducerName([],                State) =>  FinalTransduction(State).
TransducerName([Input|Stream], State) =>
            append(
                  Transduction(Input, State),
                  TransducerName(Stream, NewState(Input,State))
            ).
```

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

```
tr(Stream) => tr(Stream,0).
tr([],N) => [total_number_of_cpu_failures(N)].
tr([signal(_,transmission_failure,_)|S],N) => [net_failure|tr(S,N)].
tr([signal(_,site_not_responding,_) |S],N) => [cpu_failure|tr(S,N+1)].
```

Here `N` is the number of cpu failures encountered so far in the input stream, and defines the state of the transducer. When the input stream is exhausted, the transducer outputs the final translation `[total_number_of_cpu_failures(N)]`. Meanwhile it translates the input stream into the output stream item by item. Note that the third rule for `tr` uses `[net_failure|tr(S,N)]` instead of the equivalent `append([net_failure],[tr(S,N)])` 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 $S_1,...,S_n$ are stream expressions, and f is transducer, then $f(S_1,...,S_n)$ 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,

(2) aggregate computations, and

(3) measurement data analysis.

The next section continues in showing how pattern matching can be accomplished.

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

(InitialState,Transduction,NewState,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 [41], 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,   cacm,       9/78     ).
submitted(  merrie,   tods,       5/79     ).
submitted(  jane,     cacm,       11/79    ).
submitted(  merrie,   jacm,       8/82     ).
```

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

```
facultyHistory( Positions ) => facultyHistory( Positions, [] ).

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

newFaculty( [], F,P,W,D ) => [faculty(F,P,W,D,_)].
newFaculty( [faculty(F1,P1,W1,D1,_)|Faculty], F,P,W,D ) =>
        if( F1==F,
                [faculty(F,P,W,D,_) | Faculty],
                [faculty(F1,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 => facultyPapers(
                    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)   :- var(Date).
notLater(Month1/Year1,Month2/Year2)   :- Year1 =< Year2.
notLater(Month1/Year1,Month2/Year2)   :- Year1 = Year2, Month1 =< Month2.
```

Variables are used to indicate both indefinite and perpetual dates.

Other queries can be composed using relational operators. The table below lists the conventional database transducers on streams defined in the TSP library. 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. While there exists an implied order on the terms in a stream, the operators union, join, intersect, and difference follow the same basic definitions as in a relational query language.

Expression	Result
`sort(S)`	The reordering of terms in `S` in increasing order, using the standard Prolog term lexicographic order.
`keysort(S)`	The reordering of terms in `S` in increasing order by key. Terms are required to be of the form `K - T` where `K` is a key value used in performing the sort.
`union(S1,S2)`	Stream union of `S1, S2`.
`intersect(S1,S2)`	Stream intersection of `S1, S2`.
`difference(S1,S2)`	Stream difference of `S1, S2`.
`select(S,T,C)`	The substream of terms in `S` that match (unify with) the template `T`, and in addition satisfy the logical condition `C`.
`project(Cols,S)`	`Cols` is a list of the column indices $[i_1, \ldots, i_n]$ on which the projection is to be made.
`join(S1,S2)`	(merge-)join of `S1, S2`. The streams are assumed to be keysorted, so their items are of the form `KeyValue - Term`.
`tuples(R,A)`	Yields the stream of all tuples in `R`, a relation with `A` columns.
`clauses(P,A)`	The stream of clauses in `P`, a predicate of arity `A`.
`file_terms(F)`	The stream of terms stored in the file `F`.

3.4. Aggregate Computations

Aggregate operators can be of several kinds. Aggregate reductions, which apply an associative operator cumulatively 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 [41]. 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).
```

With these forms, the operators perform the following operations:

Operator	*Function*
`count`	total number of non-deleted identifiers and values
`countC`	total number of inserted identifiers and values
`countU`	total number of unique identifiers and values
`countUC`	total number of unique values

For streams that have been transformed to this structure, the count aggregation operators can be conveniently defined with TSP: We first define some constructor symbols: `insert(I,V,T)`, `delete(I,V,T)`, `count`, `countC`, `countU`, and `countUC`. We 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 what `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.

Finally, another class of predefined TSP stream operators perform pair-wise operations on two streams of numbers. For example, the following stream addition succeeds:

```
?- reduce_eagerly( [1,3,5,7] + [2,4,6,8],   [3,7,11,15] ).
```

In general `Op(S1,S2)` performs the element-wise numerical `Op (max, min, +, -,` etc.) of the streams `S1, S2.`

3.5. Stream Processing of Measurement Data

One emphasis of TSP for Tangram is to support data analysis. Let us now explore several examples applying stream processing to analyze measurement data.

Suppose that we are executing a distributed algorithm on a set of processors and the implementation is expected to yield balanced utilization of the processors. Through instrumentation we have captured a stream of measurement data with the schema `util(Time,Utilizations)`.

Each term in the stream gives the time at which the measurements were taken and a list of the utilizations observed on each processor. An example instance of such a stream is:

```
[util(1, [0.5, 0.8, 0.5]),
 util(2, [0.7, 0.7, 0.7]),
 util(3, [0.8, 0.6, 0.6]),
 util(4, [0.6, 0.7, 0.5]),
 util(5, [0.7, 0.9, 0.8])
]
```

Let us suppose that this stream is stored in the file `experiment2.output`.

A useful summary of how well our distributed algorithm actually balanced the processor utilizations can be determined by finding, for each measurement interval, the variation in processor utilizations. We can do this by finding the maximum deviation by any processor from the average processor utilization during that time period. The TSP Query to accomplish this could be

```
?- reduce_eagerly( summary(file_terms('experiment2.output')) ).
```

if we first make the following additional definitions:

```
summary([]) => [].
summary([util(T,S)|L]) => [dev(T,reduce(maxdeviation(S)))|summary(L)].

maxdeviation(S) => max(deviation(avg(S),S)).

deviation(_,[]) => [].
deviation(N,[L|Ls]) => [L-N|deviation(N,Ls)].
```

Here `avg` and `max` are aggregate operators, as described earlier.

Now let us consider an application to queueing systems. Suppose we wish to gather statistics on the time customers spend at specific servers. For simplicity, assume that the type of server is either FCFS or LCFS, and is specified in a binary predicate:

```
type(server1) => [fcfs].
type(server2) => [lcfs].
```

Arrivals and departures of customers to a specific server are captured in a stream whose items have one of the two formats:

```
a(Time)
d(Time)
```

Transducers can be developed that map this stream to a stream of 'elapsed time' values, or to a stream of 'service time' values. The transducers are defined as follows:

```
fcfs_e([],_) => [].
fcfs_e([a(T)|L],State) => fcfs_e(L,append(State,[T])).
fcfs_e([d(T)|L],[T0|S]) => [T-T0|fcfs_e(L,S)].

lcfs_e([],_) => [].
lcfs_e([a(T)|L],State) => lcfs_e(L,[T|State]).
lcfs_e([d(T)|L],[T0|S]) => [T-T0|lcfs_e(L,S)].

fcfs_s([],_) => [].
fcfs_s([a(T)|L],(N,T0)) => if(N==0, [fcfs_s(L,(1,T))],
                                    [fcfs_s(L,(N+1,T0))]
                              ).
fcfs_s([d(T)|L],(N,T0)) => [T-T0|fcfs_s(L,(N-1,T))].

lcfs_s([],_) => [].
lcfs_s([a(T)|L],[]) => lcfs_s(L,[(T,0)]).
lcfs_s([a(T)|L],[(T0,T1)|S]) => lcfs_s(L,[(T,0),(_,T-T0+T1)|S]).
lcfs_s([d(T)|L],[(T0,T1),(T2,T3)|S]) => [T-T0+T1|lcfs_s(L,[(T,T3)|S])].
lcfs_s([d(T)|L],[(T0,T1)]) => [T-T0+T1|lcfs_s(L,[])].
```

These transducers may appear a little forbidding. We can however make these available in a simpler and more natural form by introducing 'higher level' transducers, much like defining *views* in relational databases:

```
elapsed_times(Server,S) => policy_elapsed_times(type(Server),S).
service_times(Server,S) => policy_service_times(type(Server),S).
policy_elapsed_times( [fcfs], S) => fcfs_e(S,[]).
policy_elapsed_times( [lcfs], S) => lcfs_e(S,[]).
policy_service_times( [fcfs], S) => fcfs_s(S,(0,_)).
policy_service_times( [lcfs], S) => lcfs_s(S,[]).
```

Interesting statistics (e.g. mean, standard deviation, etc) can then be calculated from this output stream by applying further aggregate operators. For example, the average elapsed times and maximum service times at Server 1 can be obtained with:

```
avg( elapsed_times(server1, file_terms('server1.trace')) )
max( service_times(server1, file_terms('server1.trace')) ).
```

4. Pattern Matching against Streams

In the section above we 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 matching against streams.

4.1. Detecting Patterns With Transducers

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

```
t1([net_failure|S]) => t2(S).
t2([net_failure|S]) => t2(S).
t2(S) => t3(S).
t3([cpu_failure|S]) => S.
```

successfully recognizes all streams containing sequences of one or more copies of `net_failure` followed by a `cpu_failure`. In other words, the `t1` transducer implements the *regular expression*

```
( [net_failure]+, [cpu_failure] )
```

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

More generally, transducers can implement *parsers*. Pattern matching of arbitrary computational power is possible even with deterministic (non-backtracking) transducers if they are not restricted in the state information they can maintain.

4.2. Specifying Patterns With Functional Grammars

Unfortunately, writing transducers to recognize such patterns is clumsy. The approach taken in Tangram is to let users specify patterns with *grammars*, which are compiled into transducers like the one above. Moreover, users can express their patterns using a library of grammars. For example, regular expressions and, more generally, path expressions, can be easily defined with grammar rules something like the following:

```
(X+) => X.
(X+) => (X,(X+)).
(X*) => [].
(X*) => (X,(X*)).
(X;Y) => X.
(X;Y) => Y.
(X,Y) => append(X,Y).
skipto(X) => X.
skipto(X) => ([_],skipto(X)).
```

These Log(F) rules behave just like the context free grammar rules they resemble.

In TSP, pattern matching is signalled explicitly with the `match` transducer, which takes its first argument a *functional grammar term* 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. For example,

```
match(
      ( [net_failure] +, [cpu_failure] ),
      tr(file_terms('net.traffic.87.10.19'))
)
```

matches the pattern we just defined to the result of transducing the file `'net.traffic.87.10.19'` into a stream of event types, as done in section 3.1.

The rules for pattern matching are very simple. The entire definition is as follows:

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

With this definition, we can immediately define grammars using rewrite rules. We call these resulting rules a *functional grammar*.

The `match` transducer can be thought of as applying a grammar to a stream, in an attempt to find a prefix of the stream that the grammar can generate. There is a certain elegance to this; the rules of the grammar by themselves act as pattern generators, but when applied with the `match` transducer they act like a parser. This acceptance/generation duality is familiar to users of definite clause grammars, and the ability to use grammars both as acceptors and as generators has a number of uses. One interesting use is in communications protocol testing [21].

Functional grammars in TSP relax the Log(F) variable restrictions to obtain the full power of Prolog's backtracking and unification. That is, they include arguments to rewrite rules which are used not strictly as inputs, but as grammar outputs. This technique is widely practiced in writing *Definite Clause Grammars (DCGs)* [43] in Prolog systems.

4.3. Sample Functional Grammars

Suppose we wish to count the number of times one or more network failures were followed by a cpu failure. That is, we want to count the occurrences of a specific pattern in the input stream. We can use the pattern:

```
number( ( [net_failure] +, [cpu_failure] ), Total)
```

where we include the following grammar for `number`:

```
number(Pattern,Total) => number(Pattern,Total,0).
number(Pattern,Total,Total) => [end_of_file].
number(Pattern,Total,Count) => skipto(Pattern), {Count1 is Count+1},
                               number(Pattern,Total,Count1).
```

Here the variable `Total` is used to return output values, and the pattern `{Count1 is Count+1}` defines Prolog code to be executed immediately without disturbing the input stream. The `number` transducer counts all occurrences of `Pattern` that it finds, possibly skipping over parts of the input stream in its search.

Functional grammars differ from DCGs in several important ways.

(1) Functional grammars are *higher-order*, i.e., patterns to be matched can be passed as arguments. For example, in the grammar above `Pattern` is an argument and can be instantiated as anything.

(2) Pattern matching can proceed lazily.

(3) DCGs are defined by a translation to Prolog that serves as an implementation. Functional grammars can be compiled into lower-level implementations, possibly not in Prolog. Simple patterns should execute very quickly against streams. We discuss implementation below.

Just as it is possible to define aggregate computations with the stream transduction part of TSP, it is also possible to do aggregate computation in pattern matching. For example:

```
count(Result) => count(Result,0).
count(Result,Result) => [end_of_file].
count(Result,Count) => [_], {NewCount is Count+1}, count(Result,NewCount).

sum(Result) => sum(Result,0).
sum(Result,Result) => [end_of_file].
sum(Result,Current) => [Value], {Sum is Current+Value}, sum(Result,Sum).

avg(Result) => avg(Result,0,0).
avg(Result,N,S) => [end_of_file], {S = 0 -> Result is 0 ; Result is S/N}.
avg(Result,N,S) => [V], {NewS is S+V,NewN is N+1}, avg(Result,NewN,NewS).
```

These aggregates could be used in larger patterns that match certain values (counts, totals, averages, etc.) derived from an input stream with grammar parameter values.

In addition to the above, we can do much more sophisticated parsing of the input. We believe that grammars have an important role to fill in translation among output formats of different tools. For example, below is part of a grammar which parses version 2.0 PAWS output files. These files, which normally must be gone through by hand, a time-consuming and error-prone task, can be translated automatically to a format amenable to stream analysis.

```
output(H,NR,QS) =>
            header(H),
            nodeResults(NR),
            queueingStatistics(QS).

header(run(NEvents,BatchNumber,BatchDuration,BatchStartTime,CurrentTime)) =>
            "NUMBER OF EVENTS:", numeric(NEvents),
            "BATCH NUMBER:", numeric(BatchNumber),
            "BATCH DURATION:", numeric(BatchDuration),
            "BATCH STARTED AT:", numeric(BatchStartTime),
            "CURRENT TIME:", numeric(CurrentTime).

nodeResults([Node|OtherNodes]) => nodeResult(Node), nodeResults(OtherNodes).
nodeResults([]) => [].

nodeResult(node(Name,Cat1,Cat2)) =>
            "NODE:", name(Name), category(Cat1), category(Cat2).

category(cat(Category,ThruputRate,ThruputCount,InputRate,InputCount)) =>
            "CAT:", name(Category), numeric(ThruputRate), numeric(ThruputCount),
                              "*", numeric(InputRate),   numeric(InputCount).

queueingStatistics( [QueueingStat|OtherStats] ) =>
      queueingStatistic(QueueingStat),
      queueingStatistics(OtherStats).
queueingStatistics( [] ) => [].

queueingStatistic( stats(Name,
           queuelength(LMean,LSecondMoment,LVariance,LStddev),
           queueingtime(TMean,TSecondMoment,TVariance,TStddev)
          )
      ) =>
      "QUEUE AT NODE:", name(Name),
      "QUEUE-LENGTH",    summary(LMean,LSecondMoment,LVariance,LStddev),
      "QUEUEING-TIME",   summary(TMean,TSecondMoment,TVariance,TStddev).

summary(Mean,SecondMoment,Variance,Stddev) =>
      "SUMMARY",
      "MEAN:", numeric(Mean), "2ND MOMENT:", numeric(SecondMoment),
      "VAR:", numeric(Variance), "STNDRD DEV:", numeric(Stddev).
```

Here we are taking advantage of the fact strings like `"NUMBER OF EVENTS:"` are represented as lists in Prolog, and hence the definition of `match` above is sufficient for this grammar to work.

4.4. Implementation

The main issue of implementation here is finding a way to compile functional grammars and `match` into *efficient* transducers. This subject goes beyond the scope of this paper, but it is important to notice that a translation like that of DCGs is often possible. For example, the definition

```
netcrash => ( [net_failure]+, [cpu_failure] ).
```

can be compiled to

```
match(netcrash,S) => t1(S).
t1([net_failure|S]) => t2(S).
t2([net_failure|S]) => t2(S).
t2(S) => t3(S).
t3([cpu_failure|S]) => S.
```

The point is not so much that functional grammars execute correctly (although they do), but that they can be compiled to something that runs as a transducer.

Characterizing the efficiency of these transducers is important. Not surprisingly, deterministic tail-recursive grammar rules which do not construct large structures for their state can be compiled to efficient transducers. These grammars are much like classical right linear regular grammars, and like DCGs that are actually written in practice.

5. 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 rests 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.

For example, for some transducers it is easy to establish basic algebraic properties (associativity, commutativity, distributivity) [47] and the less-widely used property of *order preservation*. If S_1 and S_2 are stably sorted, we say f is *order preserving* iff

$$\mathrm{sort}(f(S_1,S_2)) \equiv f(\mathrm{sort}(S_1),\mathrm{sort}(S_2)).$$

Order preservation is extremely important because most stream operations require input streams to be sorted for efficient processing. If we know in advance that some operations are order preserving, we can eliminate some redundant sort operations. Consider the stream expression

```
sort( (sort(S1) ∪ sort(S2)) ∪ sort(S3)).
```

Since ∪ has the order preservation property, an equivalent expression is

```
(sort(S1) ∪ sort(S2)) ∪ sort(S3).
```

The savings will be significant when the input streams are large.

This query can be optimized still further when we know more about the size of the streams. By associativity and commutativity, we can rearrange the query to perform union operation in ascending order of the length of the input streams. A well known theorem on optimal merge patterns [22] states that this will be most efficient.

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

1. Abelson, H. and G. Sussman, *The Structure and Analysis of Computer Programs,* pp. 242-292, MIT Press, Boston, MA, 1985.
2. Adiba, M. and N.B. Quang, ''Historical Multimedia Databases,'' *Proc. Twelfth Intnl. Conf. on Very Large Data Bases*, pp. 63-70, Kyoto, Japan, 1986.
3. Allen, J.F., ''Maintaining Knowledge about Temporal Intervals,'' *CACM*, vol. 26, no. 11, pp. 832-843, November 1983.
4. Allen, J.F., ''Towards a General Theory of Action and Time,'' *Artificial Intelligence*, vol. 23, pp. 123-154, 1984.
5. 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.
6. Batory, D.S. and T.Y. Leung, ''Implementation Concepts for an Extensible Data Model and Data Language,'' Tech. Report TR-86-24, Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712, 1986.
7. 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.
8. Batory, D.S., T.Y. Leung, and T. Wise, ''Implementation Concepts For an Extensible Data Model and Data Language,'' *ACM Trans. Database Systems*, to appear.
9. Becker, R.A. and J.M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics,* Wadsworth, Inc., Belmont, CA, 1984.
10. 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.
11. Ben-Zvi, J., ''The Time Relational Model,'' Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1982.
12. Berson, S., E. de Souza e Silva, and R.R. Muntz, ''An Object-Oriented Methodology for the Specification of Markov Models,'' Technical Report CSD-870030, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, July 1987.
13. 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.
14. Bobrow, D.G., ''If Prolog is the Answer, What is the Question?,'' *Proc. Intnl. Conf. on Fifth Generation Computer Systems*, pp. 138-145, ICOT, Tokyo, November 1984.
15. Bolour, A., T.L. Anderson, L.J. Dekeyser, and H.K.T. Wong, ''The Role of Time in Information Processing,'' *ACM SIGMOD Record*, vol. 12, no. 3, pp. 27-50, 1982.
16. Clark, K. and S. Gregory, ''Notes on the Implementation of PARLOG,'' *J. Logic Programming*, vol. 2, no. 1, pp. 17-42, 1985.

17. Clifford, J. and D.S. Warren, "Formal Semantics for Time in Databases," *ACM Transactions on Database Systems*, vol. 8, no. 2, pp. 214-254, June 1983.

18. Dean, T.L. and D.V. McDermott, "Temporal Data Base Management," *Artificial Intelligence*, vol. 32, pp. 1-55, 1987.

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

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

21. Gorlick, M.D., C. Kesselman, D. Marotta, and D.S. Parker, "Mockingbird: A Logical Methodology for Testing," Technical Report, The Aerospace Corporation, P.O. Box 92957, Los Angeles, CA 90009-2957, May 1987. To appear, *Journal of Logic Programming*, 1988.

22. Horowitz, E. and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD, 1978.

23. Kahn, K., "A Primitive for the Control of Logic Programs," *Proc. Symp. on Logic Programming*, pp. 242-251, IEEE Computer Society, Atlantic City, 1984.

24. Kowalski, R.A., "Database Updates in the Event Calculus," Research Report 86/12, Department of Computing, Imperial College, London, 1986.

25. LeDoux, C.H., "A Knowledge-Based System for Debugging Concurrent Software," Technical Report CSD-860060 (Ph.D. Dissertation), UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1986.

26. Lee, R.M., H. Coelho, and J.C. Cotta, "Temporal Inferencing On Administrative Databases," *Information Systems*, vol. 10, no. 2, pp. 197-206, 1985.

27. 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.

28. 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.

29. Lum, V., P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill, "Designing DBMS Support for the Temporal Dimension," *Proc. ACM SIGMOD Conference on Management of Data*, pp. 115-130, June 1984.

30. 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.

31. More, T., "Axioms and Theorems for a Theory of Arrays," *IBM J. Res. Develop*, vol. 17, no. 2, pp. 135-175, 1973.

32. More, T., "The Nested Rectangular Array as a Model of Data," *Proc. APL79*, pp. 55-73, May 1979.

33. Naish, L., "All Solutions Predicates in Prolog," *Proc. Symp. on Logic Programming*, pp. 73-77, IEEE Computer Society, Boston, 1985.

34. Narain, S., "A Technique for Doing Lazy Evaluation in Logic," *J. Logic Programming*, vol. 3, no. 3, pp. 259-276, October 1986.

35. 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.

36. 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.

37. Parker, D.S., T. Page, and R.R. Muntz, ''Improving Clause Access in Prolog,'' Technical Report, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, January 1988.

38. Sadri, F., ''Three Recent Approaches to Temporal Reasoning,'' Research Report 86/23, Department of Computing, Imperial College, London, Nov. 1986.

39. Segev, A. and A. Shoshani, ''Logical Modelling of Temporal Data,'' Tech. Rep. LBL-22636, Computer Science Research Department, Lawrence Berkeley Laboratory, Mar. 1987.

40. Shoshani, A. and K. Kawagoe, ''Temporal Data Management,'' *Proc. Twelfth Intnl. Conf. on Very Large Databases*, pp. 79-88, Kyoto, Japan, August 1986.

41. 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.

42. Snodgrass, R., ''The Temporal Query Language TQuel,'' *ACM Transactions on Database Systems*, vol. 12, no. 2, pp. 247-298, June 1987.

43. Sterling, L. and E. Shapiro, *The Art of Prolog,* MIT Press, Cambridge, MA, 1986.

44. Studer, R., ''Modeling Time Aspects of Information Systems,'' *Proc. Second Intnl. Conf. on Data Engineering*, Los Angeles, CA, 1986.

45. 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.

46. 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.

47. Ullman, J.D., *Principles of Database Systems, 2nd ed.,* Computer Science Press, Potomac, MD, 1982.