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

STREAM PROCESSING: AN EFFECTIVE WAY TO INTEGRATE AI AND DBMS

D. Stott Parker

January 1989
CSD-890005

Stream Processing: An Effective Way to Integrate AI and DBMS

D. Stott Parker

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

ABSTRACT

We present a novel approach for integrating AI systems with DBMS. The 'impedance mismatch' that has made this integration a problem is, in essence, a difference in the two systems' models of data processing. Our approach is to avoid the mismatch by forcing both AI systems and DBMS into the common model of *stream processing*.

By a *stream* here we mean an ordered sequence of data items. Stream processing is a well-known AI programming paradigm in which functional operators (which we call 'transducers') are combined to obtain arbitrary mappings from streams to streams. The stream processing paradigm can be, and has been, applied equally well as an AI programming model and as a query processing model in databases.

We argue first that, in practice, the relational model of data is actually the stream model. The pure relational model cannot capture important aspects of relational databases such as column ordering, duplicate tuples, tuple ordering, and access paths, while the stream model does so naturally.

We then describe the approach taken in the *Tangram* project at UCLA, which integrates Prolog with relational DBMS. Prolog is extended to a functional language called Log(F) that facilitates development of stream processing programs. The integration of this system with DBMS is simultaneously elegant, easy to use, and relatively efficient.

January 1989

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

Stream Processing: An Effective Way to Integrate AI and DBMS

D. Stott Parker

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

1. Why Connecting AI Systems with DBMS is Hard

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

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

(1) The formal model of the AI system (e.g., logic or the lambda calculus) apparently differs from the formal model of the DBMS (e.g., the relational model).

(2) Where the AI system can be said to be 'search oriented', seeking a single proof or a solution path in some structure, the DBMS is viewed as 'set oriented', computing all proofs or solution paths at once.

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

In the past, solutions to the coupling problem have adopted one model of data processing or the other [7, 35]. Tuple-at-a-time solutions (particularly Prolog-DBMS and Expert System-DBMS interfaces) [32, 40, 43] follow the AI system model. Query-at-a-time solutions that store the results of the query in the AI system workspace [13, 16, 25] follow the DBMS model. It is well known that tuple-at-a-time solutions are inefficient, and query-at-a-time solutions can overwhelm the AI system with data. A variety of combinations of these strategies have therefore been proposed [12, 24]. The EDUCE system, for example, provides complete tuple-at-a-time and query-at-a-time capabilities [5, 6].

We can reduce or eliminate the mismatch if we can find a common model that will fit both systems. There is such a model: *stream processing*. After summarizing what we mean by it, we describe how both DBMS and AI systems can be cast in the mold of stream processing. We then show how an integration of the two based on stream processing can be developed.

2. The Stream Processing Model

Stream processing is a well-established AI programming paradigm. We give only a brief definition of stream processing, and illustrate how it may be done in Prolog.

2.1. Basic Stream Processing

A *stream* is an ordered sequence of data items. These items can be tuples, or more generally any structure, such as a Prolog term or Lisp list. Being ordered sequences, streams can be initially thought of as lists. For example, we can think of

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

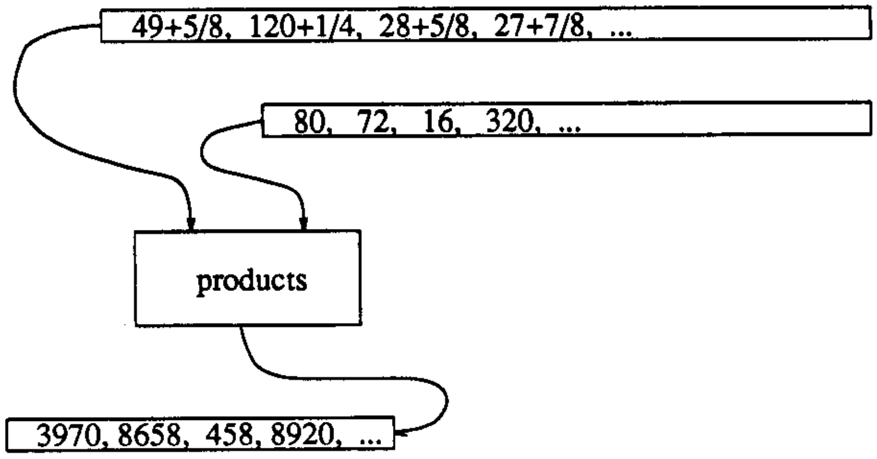
as a stream.

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

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

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

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



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

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

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

There are several differences between stream processing and list processing. First, it is common to work with 'infinite streams' such as the stream

```
[1, 2, 3, 4, 5, ... ]
```

of all positive integers. Generally speaking, time-dependent variables will lead to effectively infinite streams. Infinite streams are not stored in their entirety! Instead, they are usually stored using 'closure'-like structures giving a definition to be evaluated when further elements from the stream are needed. Definitions such as

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

declare the infinite stream `integers_from(N)` of integers beginning at `N` as the list whose head is `N` and whose tail is recursively defined by `integers_from(N+1)`.

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

2.2. A Brief Survey of Stream Processing

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

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

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

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

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

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

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

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

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

One of the first presentations of stream processing was developed by Burge [10], an elegant tour of examples showing the potential of the paradigm, which appeared in more expanded form in [11]. Burge viewed streams as functions, using approach (2) above. He drew directly on the presentation of Landin in [27] (cf. in particular p.96), and Burge credits Landin as being the first to formalize the notion of a stream in 1962.

In 1976 the concept of lazy evaluation was developed as an alternative evaluation scheme for functional languages [15, 22]. In chapter 8 of his book on functional programming [23], Henderson summarizes basic material on lazy evaluation known at the time, and shows how it corresponds directly to coroutining among function evaluations. This has the benefit that certain computations in networks of processes [26] can be shown to be evaluated neatly with lazy evaluation. An extended presentation of the material in Henderson's book, emphasizing stream processing programming techniques, is in section 3.4 of [1]. A recent summary of work on laziness and stream processing can be found in Chapters 11 and 23 of Peyton-Jones' book [39], which among other things discusses the implementation of Miranda, a lazy functional language. Today lazy evaluation is part of a number of functional programming languages.

A rather different view of stream processing is offered by Goldberg and Paige [17]. The authors take the point of view that 'stream processing is a technique that seeks to avoid intermediate storage and to minimize sequential search'. From this perspective, stream processing amounts basically to *loop fusion*, an optimization technique presented by Allen and Cocke for combining consecutive Fortran do loops into a single loop in [2]. Goldberg and Paige give a history of stream processing from this perspective, including applications that have been proposed for it in file processing, data restructuring, and above all database query processing.

Independently, stream processing in databases has been proposed a number of times in the past. First, streams naturally arise in functional data models, and a number of important functional models have been developed in the past decade [42, 29, 41]. Buneman's FQL, based on Backus' FP language, included lazy evaluation [8, 9]. Recent proposals for including lazy evaluation in databases include [19, 21].

2.3. Stream Processing in Prolog

It is not immediately obvious to most people how to implement stream processing in Prolog, particularly such features as infinite streams and lazy evaluation. We just point out here that it is not only possible to combine stream processing with Prolog, but also that it can be done elegantly. A more thorough presentation appears in [37].

In the Tangram project at UCLA, stream processing is implemented with Log(F), developed by Sanjai Narain [33, 34]. Log(F) is a combination of Prolog and a functional language called F*. In F*, all statements are rules of the form

LHS => *RHS*

where *LHS* and *RHS* are structures (actually Prolog terms). The definition for `integers_from` above in fact uses Log(F) rules. Log(F) implements lazy evaluation and permits infinite lists. Its semantics are defined by a direct translation of these rules to Prolog, and programmers using these rules can take advantage of the Prolog programming environment if they wish.

To appreciate the power and flexibility of Log(F) for stream analysis, consider the following brief examples. The transducer above that computes products can be written as follows:

```
products([],[]) => [].
products([H1|T1],[H2|T2]) => [H1*H2|products(T1,T2)].
```

It terminates if the two input streams are empty, and otherwise successively multiplies the heads `H1` and `H2` of the two streams, then deals with the tails `T1` and `T2` in the same manner. The transducer below computes *moving averages*. In general, the *M*-th moving average of a stream *S* is a stream whose *i*-th element is the average of elements *i*, (*i*+*1*), ..., (*i*+*M*−*1*) in *S*. The following rules give a simple definition:

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

prefix_avg(M,S) => prefix_sum(M,S) / M.

prefix_sum(M,[]) => 0.
prefix_sum(M,[H|T]) => if( M=<0, 0, H + prefix_sum(M-1,T) ).
```

For a complete discussion of this approach, including practical implementation issues, see [37]. Further examples and discussion of Tangram stream database applications are offered in [38].

Log(F) is modular and extensible, like all good functional systems, yet it retains the expressive power of logic programming. It offers a strong integration of functional and logic programming that permits stream processing as a natural consequence.

3. In Practice, the Relational Model is Really the Stream Model

The relational model is an elegant formal model of data. Not only is it conceptually pleasing, but its mathematical foundation and connections with both predicate logic and functional computation have facilitated many advances in the database field.

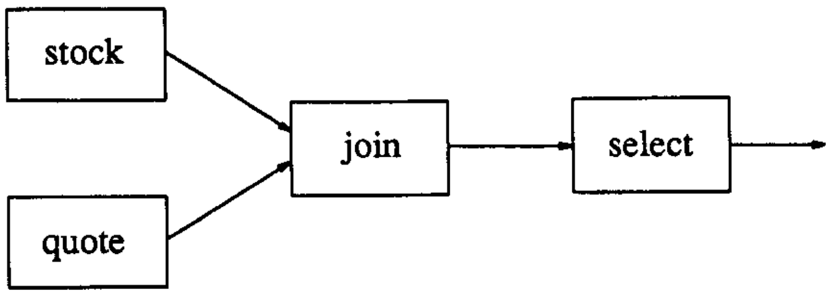
Nevertheless, the stream model is a more accurate model of databases as they are implemented today, and is increasingly becoming more accurate as a semantic model of data in some applications such as temporal data processing.

The relational model is normally presented as 'set oriented'. In theory, relations are defined as unordered sets of tuples, which in turn have an unordered set of columns (attributes). This model is unwieldy without *some* notion of ordering. Consequently, as a first concession, many developments of the relational model take the columns to be ordered.

In practice the model really is not 'set oriented' at all, but 'multiset oriented' at the least. That is, tuples are allowed to appear more than once, although they cannot do so in a set. This divergence between theory and practice has been pointed out by many researchers.

Furthermore, practical relational systems are heavily concerned with the ordering of tuples in a relation. First, this ordering is important for user interpretation of the tables. More critically, the ordering of tuples has direct impact on query processing performance, including the size of indices and the kinds of algorithms that can be used for joins.

Codd's relational algebra operators can, in fact, be viewed as stream transducers. Query 'parse trees' are then just networks of transducers, presented in diagrams like the following:



These diagrams resemble what Abelson and Sussman have termed "Henderson diagrams" in [1], and are integration similar to dataflow diagrams. Treating joins as transducers requires attention to critical aspects of ordering – the sort ordering of relations and their available access paths. When the join here has sorted input streams (as it probably would for this example), it can be implemented with a simple merge transducer. Otherwise sort transducers would be required for the join inputs.

Treating ordering explicitly gives us practical ways to model concepts such as *actions, events,* and *temporal relationships*. These concepts have eluded efficient DBMS modeling in the past. Without ordering to represent the precedence relations among events in a system, we are obliged to adopt relational representations (such as timestamps) that are clumsy and expensive to query.

It is a pity to deny one's self the power afforded by the ordering of tuples, simply because the chosen model of data is unable to capture it. Naturally, moving from the well-understood relational model to the relatively newer model of streams is not a trivial change. But the stream model is also well established in functional and dataflow database systems [3,4,8,9,14,18,19,20,21,41]. Interestingly, More's array theory [30], which forms the theoretical foundation of the APL and NIAL languages as well as a generalization of the relational model, resembles the stream model. For both practical and theoretical reasons in many situations, it is wise to move from the relational to the stream processing model.

4. Loose Coupling of Prolog and DBMS through Stream Processing

In this section, we describe an integration of Prolog and relational DBMS. Although we describe an integration with Prolog, other AI systems (such as Lisp) could be used as well. The essential assumptions we make are that:

(1) the AI system supports stream processing (this is not true of some Expert System shells);

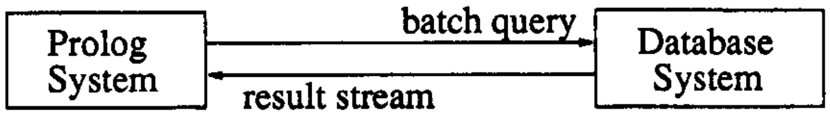
(2) the DBMS (or other data server) favors bulk/batch retrieval operations;

(3) the data being provided by the DBMS can reasonably fit in a stream format (this may not be true of large CAD data objects).

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

4.1. Basic Idea

The Tangram approach for loosely coupling a DBMS with Prolog is basically to have the DBMS yield a result stream in response to a query, and have Prolog applications analyze the stream using the stream processing paradigm:



Basic execution under the approach is as follows:

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

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

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

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

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

The diagram above is not really new; for example one very like it appears in [6]. What is new here is the use of stream processing techniques in general for data manipulation, and the use of stream processing in Prolog in particular, which augments Prolog's only existing control strategy: backtracking.

4.2. Actually Coupling Prolog and a DBMS through Stream Processing

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

show the 5-day moving averages for AT&T stock in October 1987

by evaluating the following:

```
print_stream(
     moving_avg( 5,
          sql_query(
                    'select  quote.close
                      from   stock, quote
                      where  stock.name = "AT&T"
                       and   stock.symbol = quote.symbol
                       and   quote.date >= 87/10/01
                       and   quote.date =< 87/10/31'
          )
     )
)
```

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

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

Some high points of this example:

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

(2) We could just as well have used an access method-level interface to obtain a stream from the DBMS, since access methods typically provide the sequential retrieval needed to implement streams.

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

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

5. Final Remarks

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

The Tangram architecture was originally developed for the application of analyzing large traces generated by modeling tools such as event traces from simulation packages [31,36]. Only recently have we begun to view it in the more general setting described here. Although we have talked about interfaces between Prolog and DBMS, it seems that interfaces to other data sources would work as well. Other developments at UCLA include a Log(F)-to-C compiler, and a parallel version of Log(F) that supports analyses on distributed databases [28].

We believe that a stream processing integration of the AI systems and DBMS of today gives a loose coupling that is easy to develop, supports exploitation of the full functionality of both systems, and can even be efficient. The stream processing model appears to hold great potential as a conceptual data model, and as an architectural framework for the knowledge-based systems of tomorrow.

References

1. Abelson, H. and G. Sussman, *The Structure and Analysis of Computer Programs*, pp. 242-292, MIT Press, Boston, MA, 1985.
2. Allen, F.E. and J. Cocke, "A Catalogue of Optimizing Transformations," in *Design and Optimization of Compilers*, ed. R. Rustin, pp. 1-30, Prentice-Hall, 1971.
3. Bancilhon, F., T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, a Powerful and Simple Database Language," *Proc. Thirteenth Intnl. Conf. on Very Large Data Bases*, Brighton, England, 1987.
4. 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.
5. Bocca, J., "EDUCE – A Marriage of Convenience: Prolog and a Relational DBMS," *Proc. 1986 Symposium on Logic Programming*, pp. 36-45, Salt Lake City, UT, September 1986.
6. Bocca, J., "On the Evaluation Strategy of EDUCE," *Proc. ACM SIGMOD Intnl. Conf. on Management of Data*, pp. 368-378, Washington, D.C., May 1986. Appeared as *ACM SIGMOD Record* **15**:2, June 1986.
7. Brodie, M. and M. Jarke, "On Integrating Logic Programming and Databases," in *Expert Database Systems: Proceedings from the First Intnl. Conference*, ed. L. Kerschberg, pp. 191-208, Benjamin/Cummings, 1986.
8. Buneman, P. and R.E. Frankel, "FQL – A Functional Query Language," *Proc ACM SIGMOD Intnl. Conf. on Management of Data*, pp. 52-57, Boston, MA, May-June, 1979.
9. Buneman, P., R.E. Frankel, and Rishiyur Nikhil, "An Implementation Technique for Database Query Languages," *ACM Trans. Database Systems*, vol. 7, no. 2, pp. 164-186, June 1982.
10. Burge, W.H., "Stream Processing Functions," *IBM J. Res. Develop.*, vol. 19, no. 1, pp. 12-25, 1975.
11. Burge, W.H., *Recursive Programming Techniques*, Addison-Wesley, Reading, MA, 1975.

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

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

14. Danforth, S., S. Khoshafian, and P. Valduriez, "FAD, A Database Programming Language, Rev.2," Technical Report DB-151-85, MCC, Austin, TX, September, 1987.

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

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

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

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

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

20. Gray, P.M.D., "Efficient Prolog Access to Codasyl and FDM Databases," *Proc. ACM SIGMOD Intnl. Conf. on Management of Data*, pp. 437-443, Austin, TX, May 1985. Appeared as *ACM SIGMOD Record* **14**:4, December 1985.

21. Hall, P.A.V., "Relational Algebras, Logic, and Functional Programming," *Proc. 1984 ACM SIGMOD Intnl. Conf. on Management of Data*, pp. 326-333, Boston, MA, June 1984. Appeared as *ACM SIGMOD Record* **14**:2, 1984.

22. Henderson, P. and J.H. Morris, Jr., "A Lazy Evaluator," *Proc. Third ACM Symposium on Principles of Programming Languages*, pp. 95-103, 1976.

23. Henderson, P., *Functional Programming: Application and Implementation*, Prentice/Hall International, 1980.

24. Ioannidis, Y.E., J. Chen, M.A. Friedman, and M.M. Tsangaris, "BERMUDA – An Architectural Perspective on Interfacing Prolog to a Database Machine," in *Expert Database Systems: Proceedings from the Second Intnl. Conference*, ed. L. Kerschberg, pp. 229-256, Benjamin/Cummings, 1989.

25. Jarke, M., J. Clifford, and Y. Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System," *Proc. 1984 ACM SIGMOD Intnl. Conf. on Management of Data*, pp. 296-306, Boston, MA, June 1984. Appeared as *ACM SIGMOD Record* **14**:2, 1984.

26. Kahn, G. and D. McQueen, "Coroutines and Networks of Parallel Processes," *IFIP 77*, North-Holland, Amsterdam, 1977.

27. Landin, P.J., "A Correspondence Between Algol 60 and Church's Lambda-Notation, Parts I and II," *Communications of the ACM*, vol. 8, no. 2 and 3, pp. 89-101, 158-165, 1965.

28. Livezey, B.K., "ASPEN: A Stream Processing Environment," Technical Report CSD-880098, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, December 1988.

29. McLeod, D., "A Semantic Data Base Model and its Associated Structured User Interface," Ph.D. Dissertation, Dept. EE&CS, MIT, Cambridge, MA, 1978.

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

31. Muntz, R.R. and D.S. Parker, "Tangram: Project Overview," Technical Report CSD-880032, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

32. Napheys, B. and D. Herkimer, "A Look at Loosely-Coupled Prolog/Database Systems," in *Expert Database Systems: Proceedings from the Second Intnl. Conference*, ed. L. Kerschberg, pp. 257-272, Benjamin/Cummings, 1989.

33. 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.

34. 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.

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

36. Parker, D.S., R.R. Muntz, and L. Chau, "The Tangram Stream Query Processing System," Technical Report CSD-880025, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, January 1988.

37. Parker, D.S., "Stream Data Analysis in Prolog," Technical Report CSD-890004, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, January 1989.

38. Parker, D.S., R.R. Muntz, and L. Chau, "The Tangram Stream Query Processing System," *Proc. Fifth Intnl. Conference on Data Engineering*, Los Angeles, CA, January 1989.

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

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

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

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

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