A RELATIONAL DATABASE MACHINE BASED ON FUNCTIONAL PROGRAMMING CONCEPTS

Yasushi KIYOKI, Kazuhiko KATO and Takashi MASUDA
Institute of Information Sciences and Electronics
University of Tsukuba
Sakura, Niihari, Ibaraki 305, Japan

ABSTRACT

We present a novel approach to a relational database machine for processing knowledge bases. This approach is based on functional programming concepts in order to manage processor resources and memory resources with the theoretical neatness of functional computation. By using demand-driven evaluation as a driving method of functional computation, parallelism can be exploited in executing relational operations (relational database operations) and inference operations based on unification. Furthermore, these operations can be executed avoiding the complexity of resource management within a restricted resource environment. This approach is implemented under a multiprocessor architecture combined with a demand-driven evaluation mechanism. In this paper, we define the basic primitives which are used to implement demand-driven evaluation and function application. We also present a basic algorithm and a system architecture for executing basic operations for knowledge bases by using a demand-driven evaluation mechanism. To ascertain feasibility of our approach, a relational operation system has been implemented on the basis of the approach.

1. Introduction

The relational model [4] has received much attention as a promising data model for implementing database systems with theoretical basis. Furthermore, it has been recognized that the model is significant for implementing knowledge base systems based on mathematical logic, and it has been studied how relational database concepts can be applied to mathematical logic [6].

It is known that basic operations in such knowledge base systems are relational operations and inference operations based on unification. The important feature of these operations is that they deal with a vast amount of data represented as relations.

We consider the basic operations within the framework of functional programming concepts. Functional programming [2] include many attractive concepts, and its fundamental concepts are summarized as follows:

(1) The value of a function expression depends only on its textual context, not on computational history. This notion is referred to as referential transparency.

(2) Functional computations are free of side effects. The parameter passing mechanisms call-by-value, call-by-name and call-by-need have the same semantics.

(3) Functional programs often contain implicit and easily detected parallelism [5].

In this paper, we present a novel approach to a relational database machine for processing knowledge bases. In this approach, functional programming concepts are applied to both relational and inference operations in order to exploit parallelism in a natural way and to manage processor resources and memory resources with the theoretical neatness of functional computation.

In [16], relational operations have been described in a dataflow language based on data-driven evaluation [1]. Processing of relational operations has been considered as stream processing in the data-driven evaluation including the eager and lazy evaluations. Furthermore, in [17], an advanced stream-oriented algorithm which exploits parallelism inherent relational operations has been presented in detail.

In the approach presented in this paper, demand-driven evaluation is used in computing functions of relational and inference operations. A relational operation, such as the selection operation or the join operation, is defined as a function, and operand relations of the relational operation are manipulated as arguments of the function. The arguments corresponding to operand relations are evaluated as streams of tuples by using the demand-driven evaluation mechanism. By using this evaluation mechanism in functional computation, it becomes possible to execute both relational and inference operations in parallel within restricted computing resources. In this approach, the stream-oriented algorithm [17] can also be realized within the framework of demand-driven evaluation.

In this paper, we also discuss how relational database concepts can combine with inference operations based on unification. In our approach, a set of fact clauses in Horn clauses is represented as a relation, and the fact clauses are manipulated by using functional computation.

The relational database machine proposed in this paper exploits parallelism inherent in knowledge-base processing, and makes it possible to execute basic operations for knowledge bases within a restricted resource environment.

2. Approach to Functional Computation

In our approach, each process of relational and inference operations is described on the basis of functional programming concepts. In order to realize several driving methods of functional computation, basic primitives are defined. The basic primitives sup-
port various kinds of parameter passing mechanisms required in functional computation.

2.1 Demand-driven Evaluation in Functional Computation

The methods of driving functional computation are classified [20],[21] as follows:

1. Demand-Driven Evaluation,
2. Data-Driven Evaluation, and

We employ demand-driven evaluation in executing relational operations and in executing inference operations, in order to execute those operations in parallel within a limited resource environment. Demand-driven evaluation generally introduces a fair amount of overhead in issuing demands. However, the demand-driven evaluation allows better control of parallelism, more selective evaluation, and a natural way of handling a large amount of data within a limited resource environment. The advantage of demand-driven evaluation includes the potential for eliminating a vast amount of computation by evaluating only what is necessary for computing the result. Since relational operations and inference operations generally require to manipulate a large amount of data, the potential inherent in demand-driven evaluation is effectively utilized. In executing these operations, granularity of data which is transferred by a single demand can make large. Therefore, when compared with the total amount of data transferred by a single demand, the overhead caused by the demand transfer is insignificant.

In this approach, the following parallelisms inherent in functional computation are exploited:

1. Parallel evaluation for arguments of a function,
2. Parallel execution between a function which generates its return value as a stream [10] and a function which consumes the stream as an actual argument. That is to say, stream-oriented parallel processing between a function of stream-producer (producer function) and a function of a stream-consumer (consumer function).

To exploit the parallelism of (1), demands are simultaneously issued from a consumer function to producer functions which generate actual arguments of the consumer function. Consequently, the independent operations can be executed in parallel.

The parallelism of (2) is exploited between a producer function and a consumer function. In data-driven evaluation, to exploit the parallelism, it is necessary for the consumer function to begin the computation eagerly before the producer function completes producing all intermediate results [1]. To extract the parallelism in demand-driven evaluation, it is necessary for the producer to begin the computation eagerly before the data arrives from the consumer. In our approach, parallelism is exploited by pre-issuing a demand to the producer function before the consumer function begins computation. When the producer function receives a demand pre-issued from the consumer function, the producer function begins computation. In producing a stream, the producer function does not produce every stream element by a single demand. The producer generates some fixed amount of stream elements by a single demand. After the producer function completes producing the fixed amount of stream elements, it suspends computation and waits for the subsequent demand. As a result, the stream-oriented parallel processing [17] between the producer and consumer functions can be performed within the framework of demand-driven evaluation.

In this approach, each relational operation or each inference operation is defined as a function. Several arguments of the function correspond to operand relations of an operation, and such a argument is evaluated as a stream of tuples in a relation.

To implement demand-driven evaluation, call-by-name or call-by-need is employed as a parameter passing mechanism. That is, an argument of a function is not evaluated until a reference to the argument is encountered in the execution of the function body. In call-by-name, if a formal argument is encountered more than once, the corresponding actual argument is reevaluated each time it is encountered in the function body. In this case, the actual argument may be defined after a reference to it is completed. Therefore, if call-by-name is employed in evaluating the argument corresponding a stream of tuples, relational and inference operations can be performed within limited memory resources. However, when the same argument is encountered more than once in the function body, it must be reevaluated, that is, the function which generates the stream corresponding to the actual argument must be recomputed. We call this parameter passing mechanism "recomputation mechanism."

On the other hand, in call-by-need, a formal argument is evaluated only once when the first reference is encountered. The evaluated actual argument is used in the other references to the argument. In this parameter passing mechanism, the actual argument must be retained until every reference to it completes. If the actual argument is huge like a stream of tuples in a relation, it seems that memory could be swamped. However, recomputation of the same function is unnecessary. This mechanism is referred to as "caching mechanism"[11]. The decision of a parameter passing mechanism is important in functional computation of relational operations or inference operations, because actual arguments of functions, which are relations, are generally very large. If recomputation mechanisms are used in evaluating every argument, computations may increase drastically. On the other hand, if caching mechanisms are used, the memory overflow may cause heavy overhead. In our approach, both recomputation and caching mechanisms are supported and are used together.

2.2 Basic Primitives

Our approach to parallel processing for relational and inference operations is based on the following strategy:

1. Each operation for processing knowledge bases is defined as a function. A query is decomposed into several function applications. The relationship between operations, that is the relationship between a producer function of a stream and a consumer function of the stream, can be decided at compile time for a query.
[2] Referential transparency is ensured among functions. A function is allocated to one of the multiple processors connected to a communication network. As a result, parallelisms based on demand-driven evaluation are exploited among functions. Independent functions are evaluated in parallel. Furthermore, stream-oriented parallelism between a producer function of a stream and a consumer function of the stream is also exploited under demand-driven control. If another function is called in the function during computation of the function body, it can be allocated to another processor and these functions can be executed in parallel within the framework of demand-driven evaluation.

(3) An individual function is compiled so as to maximally extract the ability of the processor architecture to which the function is allocated. If a sequential processor is used to compute the individual function, the function is compiled into the sequential object codes. For example, if data-driven-processor is used to compute the individual function, the function is compiled into single-assignment codes.

In this subsection, the basic primitives for realizing demand-driven evaluation and for realizing function application are presented. The basic primitives are implemented at the architecture level of each processor in the relational database machine shown in Section 4.

channel(type, granularity, parameter_passing_method)

The primitive "channel" specifies a channel between a producer function instance of a stream and a consumer function instance of the stream. This primitive returns the identifier "cid" of the channel as a return value. The channel is used to communicate a stream between two function instances. The channel corresponds to a buffer which stores elements of a stream. As the properties of a channel, "type," "granularity" and "parameter_passing_method" are specified. Here, "type" indicates the data type of an element of a stream, and "granularity" indicates the amount of data transferred by a single demand. The buffer size of the channel is decided according to granularity. And, "parameter_passing_method" indicates "recomputation" or "caching" alternatively as the parameter-passing mechanism for a formal argument corresponding to a stream.

new(f, pid, cid, parameters)

This primitive creates a function instance of a function specified by "f." Here, "pid" is the identifier of the processor to which the function instance is allocated, and "cid" indicates the output channel for the stream returned from the function instance. The formal arguments (arguments of input streams and the other arguments) of the function are specified in "parameters." The function instance does not begin computation until a demand is issued to it from the consumer function of its output stream.

When a query is decomposed into relational operations or inference operations, function instances corresponding to those operations are created by using the primitive "new" and they are connected to channels by using the primitive "channel." When a function instance requires to call another function or the function itself (recursive call) during the execution of the function body, these primitives are specified in the definition of the function body.

pre-demand(cid)

This primitive is used to issue a first demand from a consumer function of a stream to the producer function of the stream. The channel for passing stream elements is indicated as "cid." This primitive is one of the basic primitives which implement demand-driven evaluation. By pre-issuing a first demand, the producer function can begin computation eagerly. As a result, parallelism is exploited between the consumer function and the producer function.

get1(cid)

This primitive accesses an element of a stream in the buffer of the input channel indicated by "cid" and returns the element as the return value. If the buffer is vacant, this primitive issues a demand to the producer function of the stream, then waits until the buffer is refilled with stream elements. Each element is deleted from a buffer once it is accessed by this primitive.

get2(cid)

When this primitive is used, the double buffering mechanism must be supported in the channel indicated by "cid." While the producer function stores stream elements in one area of the buffer, the consumer function can access a stream element stored in the other area by using this primitive. When one area of the buffer is vacant, this primitive pre-issues a demand to the producer function to have the area refilled. Then, it begins to access a stream element in the other area. Each element is deleted from a buffer once it is accessed by this primitive.

put1(d)

This primitive stores a stream element, which is indicated by "d," as a return value of a function in the buffer of the output channel. When the buffer is filled with stream elements, that is, when the amount of data indicated as granularity is generated, the execution of this primitive is suspended until the subsequent demand arrives.

put2(d)

When this primitive is used, the double buffering mechanism must be supported in the output channel. While the consumer function accesses a stream element stored in one area by using the primitive "get2(cid)," the producer function can simultaneously stores a stream element indicated by "d" as a return value in the other area of the buffer by the primitive put2(d). When the area of the buffer is filled with stream elements and when the subsequent demand is received, this primitive begins to store a stream element in the other area of the buffer. Otherwise, it waits until the subsequent demand is received.

The primitives "get1(cid)" and "put1(d)" are used when two function instances which communicate via the channel "cid" are allocated to the same processor. On the other hand, the primitives "get2(cid)" and "put2(d)" are used when two function instances which communicate via the channel "cid" are allocated to different processors. When "get2" and
"put2" are used between two function instances, the stream-oriented parallelism is exploited between these function instances.

If another function is called in the function body, a new function instance is created by using the primitive "new" in the function body. Furthermore, new channels are specified by using the primitive "channel" to pass streams as actual arguments and to receive a stream as a return value. In this case, the stream elements are passed to the new instance via channel "cid" by primitive "send1(cid)" or "send2(cid)." The primitives "send1" and "send2" are used as "put1" and "put2," respectively. However, in "send1" and "send2," the channel identifier ("cid") is explicitly specified to pass stream elements to the new function instance. The output stream of the new function instance, that is the return value, is received by a primitive "receive1(cid)" or "receive2(cid)." The primitives "receive1(cid)" and "receive2(cid)" are used as "get1" and "get2," respectively.

mark_end_of_stream(cid)
This primitive writes "EOS" into the end of an output stream as an identifier indicating the end of the stream.

cHECK_end_of_stream(cid)
This primitive detects the end ("EOS") of a stream. It returns a logical value "TRUE" if the end of the stream is detected. Otherwise, it returns "FALSE."

2.3 Functional Computation

The amount of data propagated by a single demand is referred to as "granularity." As described in 2.2, granularity is indicated as a property of the channel. When a relational operation is described as a function, three kinds of granularity can be specified as follows:

(1) tuple-level granularity
(2) page-level granularity
(3) relation-level granularity

In tuple-level granularity, although the size of the buffer between the producer and the consumer can be minimized, many demands may be issued to the producer function. This may cause heavy communication overheads [3]. In relation-level granularity, the whole intermediate data, that is the complete intermediate relation, is produced by a single demand. In this granularity, the buffer of a channel is required to store the whole intermediate relation. Furthermore, the stream-oriented parallelism between producer and consumer functions is not exploited.

In each granularity, it can be specified whether stream-oriented parallel processing is performed between the producer function and the consumer function. When a formal argument corresponding to a stream is encountered more than once in a function body, one of the parameter passing mechanisms ("recomputation" or "caching") is indicated as the property of the channel.

In the following, several function definitions of relational operations are presented. The following programs abstractly show compiled object codes of functions by using the notation of the C language. In our approach, if a sequential processor is used to execute a function, the function is compiled into sequential object codes including basic primitives. The basic primitives are implemented at the architecture level in the processor.

(1) page-level granularity without stream-oriented parallelism

Stream elements (tuples of a relation) can be transferred via the limited size of buffer. The size of buffer is referred to as "granularity" and it is equal to the "page size." When the program of a function "selection" receives a demand from another function, "get1(cid)" issues a demand to the producer function of a stream, and then waits for a page of stream elements. When the page of stream elements is stored in the input buffer, the "selection" function begins to compute the selection operation and stores the resulting elements in the output buffer of the output channel by the primitive "put1(d)" until the output buffer is filled. In this case, according to granularity, several demands are issued to the function producing the stream elements.

Program 1

```
define_function selection(relation, a)
stream relation;
item a;
{
  tuple tu;
  while (!check_end_of_stream(relation)) {
    tu = get1(relation);
    if (selection_test(tu, a))
      put1(tu);
  }
  mark_end_of_stream();
}
c1 = channel(tuples, INPUT_BUFFER_SIZE,
  RECOMPUTATION or CACHING);
/* specification of channel for output stream */
c2 = channel(tuples, OUTPUT_BUFFER_SIZE,
  RECOMPUTATION or CACHING);
/* specification of channel for input stream */
new(selection, pid, c2, c1, a);
/* creation of function instance */
```

(2) page-level granularity with stream-oriented parallelism

In demand-driven evaluation, a demand is issued to the producer function when the actual argument is encountered. However, in order to perform stream-oriented parallel processing between consumer and the producer functions, the demand must be pre-issued from the consumer function to producer function. It is achieved by using the primitives as shown Program 2. In the function "selection," the primitive "pre-demand" pre-issues a first demand to the producer function of the input stream, and then the execution of the selection operation begins. At the same time, the producer function begins function computation by the pre-issued demand, and stores a resulting page into the output buffer. In function "selection," a demand is pre-issued again to the producer function by the primitive "get2(cid)," and begins computing the selection operation to the page. As the result of the pre-issued demand, the page is stored by the primitive "put2(d)" in the producer function.

In this function, demands are always pre-issued by "pre-demand" or "get2(cid)." This stream-oriented parallelism is exploited by supporting the double buffering mechanism in the buffer of a channel between
the consumer function and the producer function of its actual argument.

Program 2
define-function selection(relation, a)
stream relation;
stream a;
{    tuple tu;
    pre-demand(relation); /* pre-issuing a demand */
    while (!check_end_of_stream(relation)) {
        tu = get2(relation);
        if (selection_test(tu, a))
            put2(tu);
    }
    mark_end_of_stream();
}
c1 = channel(tuple, INPUT~BUFFER~SIZE,
RECOMPUTATION or CACHING);
c2 = channel(tuple, OUTPUT~BUFFER~SIZE,
RECOMPUTATION or CACHING);
new(selection, pid, c2, c1, a);

(3) page-level granularity with stream-oriented parallelism (Several references to same argument are encountered.) In the function "selection," a reference to the input stream is encountered only once. When references to the input stream are encountered more than once in a function body, it is required to reproduce the same stream by recomputation or to retain the whole stream by caching. For example, binary relational operations, such as the join or union operations, require to refer to the same input stream (the stream of the inner relation) more than once [17]. Therefore, recomputation or caching is specified alternatively as the property of a channel. In the case of recomputation, function computation can be performed within the limited memory resource. In the case of caching, the retained stream data may cause memory overflow. The function "join" is shown in Program 3. In this function, tuples in the outer-relation page are sorted on the joining attribute, and then each tuple in the inner-relation page is joined with tuples of the outer-relation page by using binary search algorithm.

Program 3
define-function join(relation_1, relation_2, pagesize)
stream relation_1; /* stream of outer-relation */
stream relation_2; /* stream of inner-relation */
in int pagesize; /* size of outer-relation page */
{
    tuple in1[pagesize], in2; /* tuples of outer-relation page */
    int i;
    pre-demand(relation_1);
    pre-demand(relation_2);
    while (!check_end_of_stream(relation_1)) {
        for (i = 0; (i < pagesize) &&
            !check_end_of_stream(relation_1); i++)
            in1[i] = get2(relation_1);
    }
    sort(in1); /* sorting of outer-relation page */
    while (!check_end_of_stream(relation_2)) {
        in2 = get2(relation_2);
        if (!check_end_of_stream(relation_1) ||
            !check_end_of_stream(relation_2))
            /* pre-issuing a demand to request
            re-reference to the stream of
            inner-relation */
            pre-demand(relation_2);

        if (binary_search(in1, in2))
            put2(concatenate(in1, in2)); /* comparing an inner-relation tuple
            (in2) with outer-relation tuples
            (in1) by binary search algorithm */
            /* concatenating tuples if joining condition is satisfied */
    }
    mark_end_of_stream();
}
c1 = channel(tuple, COMPLETE~REDUCTION, RECOMPUTATION
or CACHING);
c2 = channel(tuple, COMPLETE~REDUCTION, RECOMPUTATION
or CACHING);
c3 = channel(tuple, OUTPUT~BUFFER~SIZE,
RECOMPUTATION or CACHING);
new(join, pid, c3, c1, c2, BUFFER~SIZE~1);

(4) relation-level granularity When the producer function is required to be reduced completely by a single demand, "complete-reduction" is specified as granularity. In this granularity, operand source relations or intermediate relations may overflow the limited size of buffer. Furthermore, stream-oriented parallelism is not exploited between producer and consumer functions. To realize relation-level granularity, the function "selection" shown in Program 2 is used, and "complete-reduction" is specified as granularity in the primitive "channel" as shown:

c1 = channel(tuple, COMPLETE~REDUCTION, RECOMPUTATION
or CACHING);
c2 = channel(tuple, COMPLETE~REDUCTION, RECOMPUTATION
or CACHING);
c3 = channel(tuple, OUTPUT~BUFFER~SIZE,
RECOMPUTATION or CACHING);
new(selection, pid, c2, c1, a);

3. Relational Operations and Inference Operations

Functional programming concepts can be applied to various kinds of applications. In this section, we discuss an approach to processing relational and inference operations in knowledge base systems. A basic algorithm of stream-oriented parallel processing for relational operations is discussed in [17] in detail. In this section, the algorithm for relational operations is briefly reviewed, and the stream-oriented algorithm for inference operations is presented.

3.1 Relational Operations

When a function of a relational operation receives a demand from the consumer function, it begins accessing tuples in its input buffer, then executes the relational operation until it completes the production of one resulting page of tuples in the output buffer. The output buffer is then regarded as the input buffer for the consumer function. An individual function instance of a relational operation is not required to create a whole intermediate relation by a single demand. Each function instance creates only one page of tuples specified as granularity. Individual buffers do not require the capacity to store an entire intermediate relation. In this algorithm, if a producer function and a consumer function are allocated to different processors, it is assumed that the double
buffering mechanism is supported in every buffer. That is, while the consumer function gets tuples in one area of its input buffer, the producer function can store tuples in the other area of the same buffer at the same time.

Just before a consumer function begins accessing tuples in one of two areas of its input buffer, it pre-issues a demand to the producer function to make the other area of the input buffer refilled with the subsequent page.

As a result, stream-oriented parallel processing is performed between the producer and consumer functions. By using demand-driven evaluation, unary relational operations (the selection, restriction and projection operations), and binary relational operations (the join, union, intersection, difference and Cartesian-product operations) can be concurrently executed within a limited memory resource environment. In particular, this algorithm shows attractive advantages in executing the join, union and Cartesian-product operations, which are the most time-consuming and the most memory-consuming operations.

### 3.2 Inference Operations

The combination of logic programming concepts with relational databases often appears as a promising approach to knowledge base processing [6],[18]. In our approach, sets of Horn clauses are combined with relational databases. Inference operations based on unification are executed within the framework of functional programming concepts.

Three ways to combine logic programming concepts with relational databases can be considered as follows:

1. A logic program is used as a high-level query for a relational database system. In this way, a query written in a logic language is translated into a sequence of relational operations, and the sequence of relational operations are executed by a relational database operation system.

2. Relations in a relational database are regarded as sets of Horn clauses. An inference system operates unification for relations. That is, the relational database is regarded as a part of sets of Horn clauses, and inference operations are executed in relational databases. Such a database is generally referred to as a deductive database [6]. A relation is regarded not only as a set of tuples but also as a set of fact clauses. The relational database is regarded as the collection of relations and the collection is defined as the extensional database [6]. Relations are used to store fact clauses, and rule clauses are referred to as the intensional database [6]. That is, fact clauses are distinguished from rule clauses, and they are stored as relations. That is, elementary information is classified into rules and facts. Sets of facts are represented explicitly as a relational database.

3. As in (2), a relational database is also used to represent logic programs. However, not only a fact clause but also a rule clause are represented as a tuple [22]. Inference operations are executed in a relational database with the rule and fact clauses. This way is effective when the rule base is huge. This is because the rule base can be managed within the framework of the relational database management.

We realize two ways of (1) and (2). That of (3) is not supported. Usually, the size of a fact base determines the number of concurrent activities that can be carried out by the parallel machine. Hence, ways (1) and (2) are oriented toward knowledge base management applications.

In way (1), a query described in a logic program is translated into a sequence of relational operations. The sequence is executed by using the parallel processing scheme based on the functional programming concepts as described in Subsection 3.1.

To support way (2), the mechanism for executing unification of a goal clause with fact clauses is also implemented on the basis of functional programming concepts. The resolutions for rule clauses are performed by a rule reduction system. A query represented as a goal is reduced into AND-literals. Each AND-literal is referred to as a subgoal. A goal is reduced into AND-literals in the rule reduction system until each subgoal requires to be unified with fact clauses. A set of fact clauses are represented as a relation. A predicate name of a fact clause corresponds to a relation name. The facts which have the same predicate name are represented as a set of tuples in a relation. AND-literals reduced from the goal clause are unified with fact clauses in relations. AND-literals can be processed independently as long as no variables are shared among literals. As a result, AND-parallelism is exploited. Between AND-literals sharing free variables, binding environments of variable/value are transferred in the form of a stream.

In our approach, each AND-literal inputs a binding environment and returns a new binding environment as presented in [7] and [8]. In [7] and [8], binding environments are propagated within the framework of the data-driven evaluation. In [7], the eager and lazy evaluations are used to control the stream of binding environments. In our approach, demand-driven evaluation is used for processing fact clauses. The process of solving each literal in AND-literals is regarded as a function that returns a stream of binding environments. Each binding environment represents an alternative solution to the goal clause. Like relational operations, inference operations for a relational database are also performed on the basis of functional programming concepts. If AND-literals are sharing free variables, the input and output of binding environments among those AND-literals are performed using the stream-oriented scheme based on demand-driven evaluation.

For example, a goal clause " c < - L1,L2,L3 " is constructed on three function nodes. The output of a function node is served as the input to another. It is clear that the activation of a literal does not have to wait for complete intermediate solutions to be generated by execution of the preceding function node. Therefore, stream-oriented parallel processing can be performed among these functions. In each function node, a literal is unified with fact clauses in a corresponding relation. Each function node is executed on the basis of demand-driven evaluation by using the following algorithm.

1. When an inference function node receives a
demand from the consumer node, it repeatedly executes (2) and (3) until one page of new binding environments is created and stored in the output buffer.

(2) A single page of binding environments is accessed in one area of the input buffer. At this time, the other area of the input buffer becomes available and a demand is pre-issued to the producer node to refill the area with the subsequent page. As a result, stream-oriented parallel processing is performed between this node and the producer node.

(3) The search and unification to fact clauses represented in the operand relation are executed using the binding environments in the page that has just been accessed in (2), and new binding environments are created and stored in the output buffer. If the output buffer is filled with a page of new binding environments, execution is suspended and the node waits for the next demand from the consumer node, otherwise, (2) and (3) are executed repeatedly. If the page being manipulated is the last one of the binding environments served by the producer node, the execution of this inference node is terminated.

4. System Architecture

In this section, a relational database machine architecture based on functional programming concepts is presented. In this architecture, a sequential processor is used for computing an individual function. The overview of the multiprocessor architecture is shown in Fig. 1. Each processor (RKU or QRU) supports the basic primitives defined in Section 2. The processor includes an internal memory, and the processors are connected via a high-speed interconnection network. The relations in the database are distributedly stored in disk devices. The disk devices are connected to the processors via Staging Buffers. The three-level memory hierarchy employed in this architecture. As discussed in [9, 12, 13, 15] or [19], the three-level memory hierarchy is effective in manipulating relational operations.

Although a sequential processor is currently used to compute an individual function, advanced processors as discussed in [14] and [15] can be attached to each processor in future.

The total internal memory of each processor is not large enough to store a whole source relation or a whole intermediate relation in general. In this architecture, an operand source relation is staged up into a staging buffer as a stream of tuples, using the demand-driven evaluation.

The system consists of the following components:

 QRU: QRU decomposes a query into a sequence of relational operations or into a goal clause which consists of AND-literals. QRU is regarded as a rule reduction system. In manipulating a query as a goal clause, the query is reduced into AND-literals until each literal requires to be unified with fact clauses. A query is reduced by unifying with rule clauses in the rule base. QRU allocates each relational or inference operation as a function instance to one of RKU’s, and specifies the channels between function instances.

 RKU: RKU’s carry out relational operations and inference operations on the basis of functional programming concepts. Each RKU supports the basic primitives for realizing demand-driven evaluation and function application, and executes relational or inference operations as functions discussed in Sections 2 and 3. One or more operations can be allocated to each RKU by QRU. If several operations are allocated to a single RKU, they are executed as coroutines within the framework of demand-driven evaluation. This allocation enables the machine to execute a query within the limited number of processors. The execution sequence of the operations allocated to a single RKU is controlled by a scheduler. The scheduler gives control to one of the operations, when the operation has already received a demand and its operand pages have already been prepared in the internal memory. The function node to which control is given executes the operation to the data in the input buffer until the output buffer is filled (put-wait) or the operation to the data stored in the input buffer completes (get-wait). Then, the node suspends execution of function computation and returns control to the scheduler.

Communication Network: Stream data (the sequence of tuples or the sequence of binding environments) and demands are transferred via the Communication Network. Information for allocating function instances and for specifying the channels among function instances is also transferred between QRU and RKU’s. The transfers of stream data and demands between processors are managed by Communication Processing Units connected to every processor.

Data Staging Network: Source relations are transferred from disk devices to Staging Buffers via the Data Staging Network. A source relation is stored to

Fig. 1 System architecture
one of the Staging Buffers which is connected to a single RKU. This network supports the allocation of source operand relations to the staging buffer.

5. Experimental Implementation

For simulation experiments of the relational database machine, we have implemented the basic primitives in software, and have developed a relational operation system as described in Sections 2 and 3 on the Sun-2 workstation [23]. Although this system is currently running on a single processor, it can simulate parallel processing environments based on demand-driven evaluation. Each relational operation is defined as a function and executed within the framework of demand-driven evaluation. In this section, experimental results of query execution are shown. The relational operation system can realize various environments of demand-driven evaluation.

5.1 Environments of the Experiment

The query chosen for presenting several environments of demand-driven evaluation is shown in Fig. 2. The query includes four selection operations, three join operations, and a projection operation. The cardinality (the number of tuples) of each relation, the tuple length, selection selectivity factors and join selectivity factors are set as shown in Table 1. It is assumed that five processors (RKU's) are used to execute the query. Relational operations are allocated to processors as shown in Fig 2.

The following environments are assumed in the experiments.

(1) The transfer of stream elements is exclusively performed between two processors. While two processors are communicating via the Communication Network, no other processors can use it. The transfer of demands can be simultaneously performed among processors. The data transfer rate is set to 16.6 milliseconds for a 2k-byte data.

(2) The elements of a stream corresponding to each source operand relation have been already stored in Staging Buffers. Tuples of a source relation are staged up to the Staging Buffer as a stream. Each operand relation is stored in the Staging Buffer connected to the processor which manipulates the operand relation.

5.2 Experimental Results

The query was executed in four environments shown in Table 2. These environments are realized by using basic primitives and by specifying the properties of channels as discussed in Section 2. Experimental results of Environment-1, Environment-2, Environment-3 and Environment-4 are shown in the time charts in Fig. 2. Fig. 4, Fig. 5, and Fig.6, respectively.

In Environment-1, stream-oriented parallelism is not exploited. On the other hand, in Environment-2, parallelism is exploited and performance of query processing is improved. In Environment-3, recomputation is performed in node-7 of the selection operation. In this environment, although recomputation causes heavy overhead, it enables the relational operation system to perform a query even in a case where the whole relation to be processed is not stored in the internal memory of the processor. In comparing Environment-2 with Environment-4, parallelism in Environment-2 is higher than that in Environment-4. In Environment-4, since the relation-level granularity is employed, the stream-oriented parallelism is not exploited. However, the parallelism is exploited by evaluating formal arguments of a function in parallel.

Response times were measured by varying granularity settings in Environment-2 as shown in Fig. 7. In this experiment, only the sizes of inner relation pages in executing the join operations are varied, and the granularity for streams of outer relations is set to the relation-level granularity. The granularity is set to the same value in every join operation in a query. For small granularity, response time is long. This is because demands are issued many times and the number of communication times of stream data increases in the small granularity case. For large granularity, the response time is long.

![Fig. 2 Query](image)

Table 1 Parameter settings

<table>
<thead>
<tr>
<th>Parameter settings</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Operand attribute (integer value)</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Cardinality (number of tuples)</td>
<td></td>
</tr>
<tr>
<td>of each source relation</td>
<td>10000 tuples</td>
</tr>
<tr>
<td>Cardinality of each intermediate relation</td>
<td>1000 tuples</td>
</tr>
<tr>
<td>Selection selectivity factor (ssf)</td>
<td>0.1</td>
</tr>
<tr>
<td>Join selectivity factor (jsf)</td>
<td>0.001</td>
</tr>
</tbody>
</table>

(cardinality of result relation): ssf*(cardinality of source relation) in selection, jsf*(cardinality of outer relation) in join.
### Table 2: Query processing environments

<table>
<thead>
<tr>
<th>Experimental Environment</th>
<th>Selection Operations</th>
<th>Join Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>node-2</td>
<td>node-4</td>
</tr>
<tr>
<td></td>
<td>buffer</td>
<td>buffer</td>
</tr>
<tr>
<td>Environment-1</td>
<td>page-level</td>
<td>page-level</td>
</tr>
<tr>
<td></td>
<td>(1000)</td>
<td>(1000)</td>
</tr>
<tr>
<td>Environment-2</td>
<td>page-level</td>
<td>page-level</td>
</tr>
<tr>
<td></td>
<td>(1000)</td>
<td>(1000)</td>
</tr>
<tr>
<td>Environment-3</td>
<td>page-level</td>
<td>page-level</td>
</tr>
<tr>
<td></td>
<td>(1000)</td>
<td>(1000)</td>
</tr>
<tr>
<td>Environment-4</td>
<td>page-level</td>
<td>page-level</td>
</tr>
<tr>
<td></td>
<td>(1000)</td>
<td>(1000)</td>
</tr>
</tbody>
</table>

### Figures

- **Fig. 3** Time chart (Environment-1)
- **Fig. 4** Time chart (Environment-2)
- **Fig. 5** Time chart (Environment-3)
- **Fig. 6** Time chart (Environment-4)
This is because stream-oriented parallelism is not exploited. In the case of large granularity, larger memory space is required in order to store a larger page. Optimal granularity is dependent not only on hardware performance, such as communication speed of the Communication Network, the size of available memory space or processing power of each processor, but also on contents of manipulated data. The proper page-level granularity exploits the highly parallelism within a limited memory resource environment.

5. Conclusions
We have presented a novel approach to a relational database machine for processing knowledge bases. The principle purpose of the approach is to concurrently perform both relational operations and inference operations within the limited resource environment in knowledge base systems. This approach is based on functional programming concepts in order to manage computer resources with the theoretical neatness of functional computation. By using demand-driven evaluation as a driving method of the functional computation, parallelism can be exploited in knowledge base processing. In this paper, we have defined the basic primitives which are used to implement demand-driven evaluation and function application. We have also presented a basic algorithm and a system architecture for executing relational and inference operations by using a demand-driven evaluation mechanism.

In this paper, we have also discussed how the relational database concepts are combined with logic programming concepts. For processing a set of fact clauses, we have presented an algorithm based on stream-oriented parallel processing.

We have developed the relational operation system based on the proposed approach. Currently, we are designing the inference operation system for fact clauses, and also designing the architecture in more detail.

Acknowledgements
The authors would like to thank Dr. Makoto Amamiya and Ryohe Hasegawa (NTT Electrical Communication Laboratories) for their valuable discussions and suggestions.

References