TOWARD FUNCTION-BASED DISTRIBUTED DATABASE SYSTEMS[1]

Robert M. Keller
Gary Lindstrom

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

January 1982

ABSTRACT

We discuss the suitability of a function-based (or "applicative") approach to the construction of distributed database systems. Certain aspects of applicative systems are immediately appealing for this purpose (e.g. data oriented toward conceptual objects rather than toward particular representations in memory). However, distributed systems present special requirements (e.g. updating of shared data) that appear to make the applicative approach less well-suited. We discuss techniques whereby the applicative approach can nevertheless profitably be brought to bear. Our methods are illustrated using an existing functional programming language, and a example dealing with a multiuser distributed database system. Some physical aspects of a distributed processing of functional programs are also discussed.

Key phrases: applicative programming, functional programming, distributed databases, primary site model, concurrency, distributed systems

---

[1]Work supported in part by National Science Foundation grants MCS 81-06177 and MCS 78-03832.

1. AN APPLICATIVE APPROACH TO DISTRIBUTED DATABASE SYSTEMS

1.1. Designing Distributed Systems

Distributed computing fundamentally involves the cooperation of two or more processing elements in the pursuit of a shared computational mission. In contrast to CPUs in conventional multiprocessing systems, which typically are dispatched by a centralized scheduler, the processing elements (hereafter, "PEs") in distributed systems generally are subsystems with a significant degree of individual control autonomy. Their cooperation is maintained by communication, rather than by tight-coupling. Although the term "distributed computing" is most commonly applied to geographically dispersed processing elements, it is becoming increasingly appropriate for modern single-location systems as well. Such new instances of distributed computing systems include pools of small computers grouped by local networks, and "object-oriented" architectures [Metcalfe 76, Kahn 81, Zeigler 81].

Given its broadness, it is not surprising that distributed computing is the subject of many contrasting viewpoints on how best it should be exploited in various applications. Most current approaches structure through explicit decomposition into processes, i.e. sequential program modules which constitute the primary "granularity" of the system's parallelism. This assumption in practice leads to designs which we deem excessively specific and complex, primarily as a result of the application programmer's direct concern for the system's process organization. Typically, many available design and communication tools seem to compel the system's process structure to be fixed at the earliest stages of design. This framework establishes a decomposition and logical connectivity that becomes irreversibly woven into the fabric of the

system. This invites the assignment of processes to particular PEs, to assure that the system's process structure is well served by the available physical configuration. As a consequence, the system is burdened throughout its lifetime with an a priori logical and physical structure that can obstruct the smooth evolution of either aspect.

As an alternative, this paper considers the possibility of distributed computing within a much freer logical and physical organization, in which computing load can be shifted transparently from one PE to another. Our approach, which is based on the notion of an applicative (or functional) multiprocessing [Brown 62, Friedman 78, Keller 79], is therefore oriented towards general purpose distributed computing with minimal explicit process structure or PE management.

1.2. Applicative Programming Systems

Computation in applicative systems proceeds by the application of functions (either primitive or programmer-defined) to data structures as abstract objects, rather than as explicitly modifiable representations in memory cells. Hence the notions of assignment, program counters, side-effects, etc. are banished.

The advantages of this approach have often been enumerated [Backus 78, Keller 80a]. These include:

1. Semantic basis: relatively clean uniform semantics, facilitating formal reasoning about programs, and hence their verification;
2. Modularity: adoption of the mathematical notion of function as a basis for program modularity, thereby enhancing incremental understanding and module reusability, and
3. Flexible evaluation: given its independence of any particular machine model, the applicative approach accommodates a wide variety

of evaluation schemes, including parallel and asynchronous methods.

1.3. Problems in Functional Distributed Systems

From even the brief sketch above, it should be evident that certain characteristics of applicative programming are very appealing for distributed system programming. These include:

1. abstraction away from concrete data representations, and from the associated sensitivity to particular memory structures and management policies;

2. a fundamentally non-sequential computational model, which removes one serious obstacle to distributed implementations, and

3. the general attractiveness from a software engineering perspective of the modularity and verification properties of the approach.

Moreover, other less obvious features of modern applicative programming are also relevant to distributed systems. These include:

1. Selective object copying: While new objects are conceptually created ab initio whenever they are produced as a function result, in practice only selected components are created anew, with references to components of previously constructed data objects completing physical representation of the new object. Since assignment side-effects are precluded, this space-efficient sharing is semantically transparent.

2. Processing incomplete objects: Through the use of lenient data constructors, data objects need not be constructed in their entirety before they are functionally operated upon. An important consequence of this technique is that input sequences of unknown or infinite length, called streams, are data objects. Moreover, their use overlapped incrementally with their generation [Burge 75, Friedman 76, Henderson 76].

Nevertheless, a number of questions have persisted concerning the ultimate suitability of the functional approach for adequately treating certain aspects of distributed systems, both physical and logical. These include:

1. Version-based objects: How can the fundamental indeterminacy of shared object updating (e.g. multiuser databases) be modeled without

wholesale compromise of the functional approach?

2. Distributed access control: How can the fundamentally non-sequential control model be reconciled with the need to establish temporal ordering (i.e. "serialization") among conflicting accesses to shared objects? Must this introduce bottlenecks that severely attenuate the distributed control that is possible with the applicative approach?

3. Port and site addition and removal: What functional techniques can be used to represent the allocation and deallocation of logical and physical access paths in the system, as required, for example, by the accommodation of new database users, and the release of terminated ones?

4. Load management: A general solution must be found for the task migration problem, whereby overloaded PEs can export portions of their activity backlog to less exercised neighbors. Clearly, any such solution must be compatible with site additions and removals, and should also permit pragmatic "targeting" of tasks to sites known to be especially appropriate.

In this paper we demonstrate a logical approach to database management based upon functional techniques. This provides one solution for dealing with version-based objects [Reed 78], with the added advantage that it suppresses the need for explicitly representing version numbers. In conjunction with this discussion, we mention how some aspects of the load management problem can be handled in a reasonably transparent manner, and also show how our techniques relate to the representation of site and port management in a distributed system.

More specifically, we develop a functional model of a shared database, with the distributed processing of several transaction streams arising from individual users. The transactions each return a transaction response to the originating user, as well as possibly performing an "update" on the shared database.

In the database context, our goals of programmer freedom from explicit process and processor management correspond to two of four forms of transparency

enumerated by Traiger et al. [Traiger 79] (where "node" means "PE"):

- "Location transparency: Although data is geographically distributed and may move from place to place, the programmer can act as though the data is all in one node.
- Concurrency transparency: Although the system runs many transactions concurrently, it appears to each transaction as though it is the only activity in the system. Alternately, it appears as though there is no concurrency in the system."

It may be observed that these forms of transparency arise naturally in the functional approach to distributed database design. While we will not touch on the other two forms of transparency discussed by Traiger, et al. (replication transparency and failure transparency), we do not consider them to be alien to the functional approach, but rather opportunities for future investigation.

2. A FUNCTIONAL DATABASE MODEL

We now introduce a particular functional programming language, FGL [Keller 80b], and use it to develop a simple model of a multiuser database. Our intent is only to provide sufficient background for the subsequent refinement of our database model, and to motivate our architectural ideas.

2.1. Graphical Foundation

FGL ("Function Graph Language") is based on a graphical notion of program structure, whereby

- Nodes represent individual function instances, and
- Directed arcs indicate data connections among those instances (i.e. from the result of one to an argument of another).

Nodes may be instances of standard operators and constants ("primitives"), or of programmer-defined ("macro") functions. The latter are defined by reduction

rules, which associate consequent graphs with function names ("antecedents") (see fig. 2-1). Recursion is of course permitted, and in general extensively utilized.

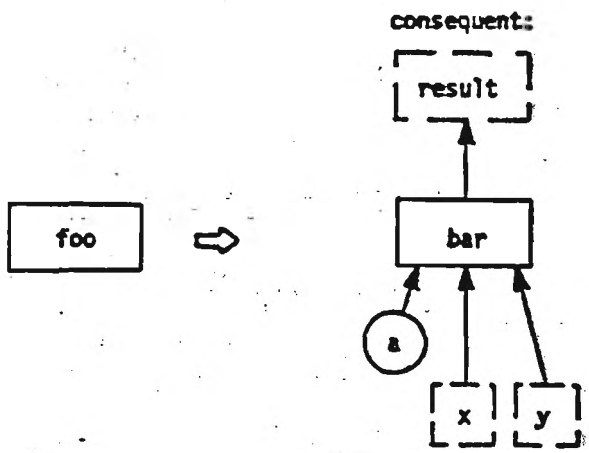

Figure 2-1: Sample macrofunction reduction rule.

"Phantom" nodes, customarily drawn with dashed lines, indicate points where a function communicates with its environment, by receiving data from it or sending data to it. These can represent the I/O channels of an overall system (i.e. for a "top-level" function), or simply a modular interface within a system. Arcs emanating from phantom nodes are termed input arcs; arcs directed toward phantom nodes are termed output arcs. The set of arcs directed toward a node (its "argument arcs") is considered to have a left to right ordering, as does the set of phantom nodes in a consequent. Arcs are permitted to fork (i.e. share a source) but not converge (i.e. arrive at the same argument position of a node).

Evaluation of a function instance proceeds by transformations on its graph, as follows:

> A macrofunction instance node may be "invoked" by its replacement with a copy of the named function's consequent. Argument arcs to the

macrofunction node are reconnected, in the same orientation, to the input arcs of the consequent copy being spliced in. An example of this transformation is shown in fig. 2-2.

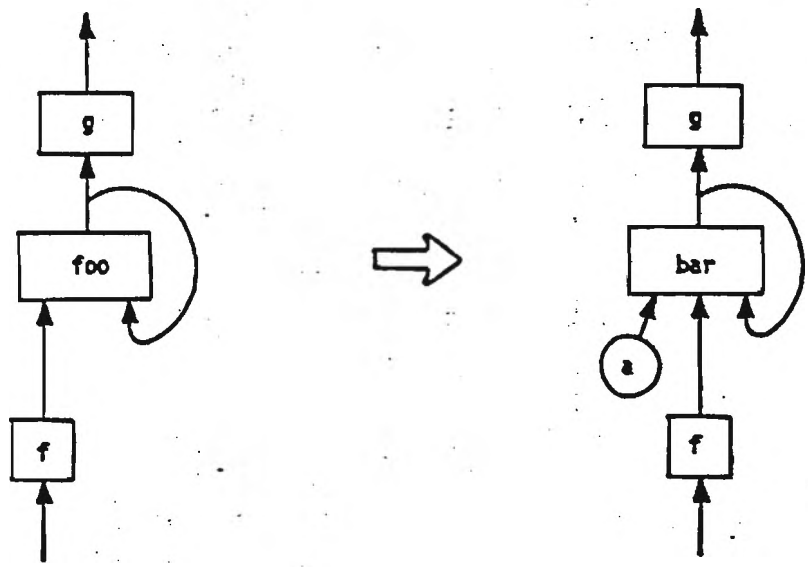

Figure 2-2: Sample macrofunction invocation.

This model has the following important properties:

1. The overall result is determinate, i.e. functionality is preserved by composition in this graphical sense.
2. Each arc has associated with it a single value (possibly an infinite structure) [Davis 82].
3. Arc forking provides a natural form of common subexpression sharing.
4. Cyclic graph structures ("applicative loops") are permitted.

2.2. Data Structures

The fundamental data structure of FGL is the tuple, denoted by $[v_1, \ldots, v_k]$ for values v_i. In addition to being used as a general structuring primitive, tuples are the primary mechanism for passing arguments to macrofunctions. Since the tuple constructor is "lenient" in that none of its components need be evaluated prior to its use, there is no implied synchronization of the order of

argument computation vs. macrofunction expansion [Friedman 76, Henderson 76].

Tuples are also usable idiomatically to construct lists and streams [Burge 75]. This use is signalled by the function fby (for "followed by"), where fby(f, r) constructs a stream with f as the first component, and stream r as the rest of the stream. When the function first is applied to such a stream, the result is f, and when rest is applied, the result is r. The function null tests for the null stream, and nil creates it.

2.3. Textual Notation

The graphical form of FGL has demonstrated utility as a system design tool [Keller 81a] and as a framework for its underlying semantics. However, it is very convenient in practice to have an equivalent textual notation. The simplest way this can be accomplished is by introducing symbolic names for the value associated with each arc, and expressing the functional relationships among them by equations. For example, a function often used is one which "filters" a stream according to a predicate, dividing it into a pair of result streams, the left component of the pair consisting of only those components satisfying the predicate (the "good" components) and the right component of the pair consisting of the others (the "bad" components). This function would be accomplished by testing the first component of the stream with the predicate, and deciding which of the two result components should begin with it. In textual FGL notation, this function could be expressed as follows:

```
FUNCTION select_from_stream(good)(stream)

LET     first_comp BE first(stream),

        [good_rest, bad_rest] BE
                select_from_stream(good)(rest(stream))

RESULT  if null(stream)
                then [nil(), nil()]
                else if good(first_comp)
                        then [fby(first_comp, good_rest),
                                bad_rest]
                        else [good_rest, fby(first_comp,
                                bad_rest)]
```

In this example we have a "two-tiered" or "Curried" form, in that select_from_stream is first applied to the argument "good", yielding a function which is then applied to its argument "stream". The specification above can be thought of as an equation defining the function select_from_stream, which employs an inner "local" equation defining good_rest and bad_rest.

For purposes of illustration, we give an alternate, slightly more verbose, definition which does not employ the two-tiered form, but which uses the notion of "IMPORTS" instead. Here, the scope of the name "selector" is local to the definition of select_from_stream, due to the use of the "WHERE" clause.

```
FUNCTION select_from_stream(good)

RESULT  selector

WHERE
        FUNCTION selector(stream)

        IMPORTS good

        LET     first_comp BE first(stream),

                [good_rest, bad_rest] BE selector(rest(stream))

        RESULT  if null(stream)
                        then [nil(), nil()]
                        else if good(first_comp)
                                then [fby(first_comp, good_rest),
                                        bad_rest]
                                else [good_rest,
                                        fby(first_comp, bad_rest)]
END
```

2.4. Expressing Communication

A language with lenient data constructors (such as fby and [....] presented above), provides a means for representing and implementing communication among distributed functions, which we will exploit heavily. More specifically, the following points may be noted:

- We represent communication through streams connecting functions in producer and consumer relationships. (Each function may, of course, bear several different such relationships with other functions.)

- The producer and consumer in a communication relationship each recurse through the stream, with the producer stimulating the consumer to produce components as demanded.

- However, those components need not be demanded in list order, nor need they be computed serially. Instead, considerable overlap can result through the delivery of "suspensions" [Friedman 76] representing the "promise" by the producer to provide the indicated components when truly needed by the consumer. This effect can lead to very effective pipelining, as we shall see later.

- Lastly, it will be seen that the possibility of cyclic graph structures permits a very clean representation for streams of version-based objects. This capability provides the feature by which we will accomplish a functional form of updating in our multiuser

database model.

A corollary of the use of lenient data constructors is that many potential sites for concurrent execution are introduced, due to there being less enforced synchronization by not waiting for arguments prior to expanding macrofunctions. Despite this potential, actual concurrency realized would be low, if it were not for the occasional introduction of operators which anticpate demand of values, in a semantically transparent fashion. The introduction of such operators is discussed in reference [Keller 81b].

3. FUNCTIONAL DATABASE PROCESSING

The use of a functional programming model for database applications has been only partly explored, the works of Buneman, et al. [Buneman 79, Buneman 82] and Shipman [Shipman 81], being the most notable. However, the first of these does not deal with the question of updating as we do here, while the second is mainly concerned with modeling data, rather than modeling the programs which operate on that data. It is notable nonetheless that the functional data model is being pursued even as an extension to the non-functional language Ada [Smith 81]. An approach similar to ours is discussed in [Friedman 77, Friedman 79]. There "tail-recursion" is used instead of our cyclic program graphs. In the first reference, editing of a file is discussed, while in the second the "database" is a single integer (the number of seats on an airplane). No measurements of concurrency are reported.

3.1. Functional Formulations

It is common to employ a transaction model in the application domain of distributed databases. Briefly, a transaction is a sequence of operations on

the database which must have the effect of uninterrupted execution. An individual user or application program interacts with the database system by submitting a stream of transaction requests, to which there is a stream of corresponding transaction responses. In the case of several users or application programs submitting requests on the same database, there is interaction among them when one transaction modifies a portion of the database which is used by a subsequent transaction. Hence there is a distinct non-functional appearance in the customary formulation of such systems.

Nevertheless, there turns out to be a simple way of specifying the desired behavior in a "pseudo-functional" manner. This entails the use of a merge (or "multiplex") operation, which provides an interface consistent with other functional operators, but is not strictly a function (cf. [Keller 78]). Informally, a merge has as its input several request streams, and its output is an arbitrary interleaving of those streams. We henceforth refer to a specific interleaving as the merged stream of requests. The order of interleaving can be that in which the merge receives the requests. In order to direct the response for each transaction back to its origin, a tag must accompany each request. (The tagging idea was also used in FriedmanWise79.) The discussion below initially assumes that such tags have been stripped off. We will show in Section 3.3 how the information provided by tags is taken into account. The FGL definition of a (2-way) merge is given in [Keller 80c] and will not be repeated here.

We can rephrase the standard criterion (called "serializability", cf. [Ullman 80]) for the processing of concurrent transactions as follows:

Process the merged stream sequentially.

This criterion conveniently decomposes the overall problem into a pseudo-

functional part (the merge) and a purely functional part (the apparently-sequential processing of the merged stream).

We depend on a sufficiently powerful functional implementation which will extract concurrently executable operations from the merged stream automatically. Thus, the apparent bottleneck due to merging is minimized if components of the transactions are sufficiently independent. There is a momentary "locking" effect among transactions as transaction streams are merged; this establishes a definite sequence from which concurrent operations are extracted. This will be demonstrated after presenting a functional approach to the processing of the merged stream. It is further possible to "optimize" the concurrency of transactions for greater component concurrency by judiciously ordering the transactions to be merged, so long as the order of transactions from each individual stream is maintained.

The functional approach requires that we do not directly modify any object. Instead, we only create new objects from existing ones and destroys unneeded objects. Consequently, the first problem to be solved is how to represent the phenomenon normally thought of as database updating. Our approach is that each transaction operates on a database, and functionally produces a new database. In addition, as mentioned previously, each transaction produces some response which is returned to the user to indicate what happened. It is the new database which is to be used for the next transaction to be processed, the database yielded by that transaction for the transaction after it, etc.

With this orientation, we can refine our example by mapping through a series of four initial levels of refinement.

Level 1. Corresponding to the stream of transactions is a stream of "old" databases and a stream of "new" databases. However, these streams are related; the "old" stream begins with the initial version of the database, which is followed by exactly the "new" stream. This may be expressed as an equation (recall fby is the "followed-by" stream building function):

old_databases = fby(initial_database, new_databases).

Here we use an equational representation in place of the more verbose textual FGL form, for brevity.

Another equation expresses a further relationship between new and old database streams, as determined by the stream of transactions:

new_databases = apply_stream(transactions, old_databases).

Here we use apply_stream to denote a function which applies the first transaction to the first old_database (i.e. to the initial database), the second transaction to the second old_database (i.e. to the result of the previous transaction), and so on. In order to include the responses to the requests, we can adjust this equation slightly so that apply_stream actually returns a pair of streams, one of which is the stream of new databases and the other of which is the stream of responses:

[responses, new_databases] = apply_stream(transactions, old_databases)

This, and the "fby" equation above form a system of equations, which can be expressed graphically as shown in fig. 3-1. It should be pointed out that this graph, or equivalently the system of equations, forms the top-level functional program for solving the database problem.

A similar problem is discussed in [Friedman 77]. Their approach is to use "tail recursion" rather than the cyclic program structure used here. In the interest of broading the understanding of the distinction, the tail-recursive

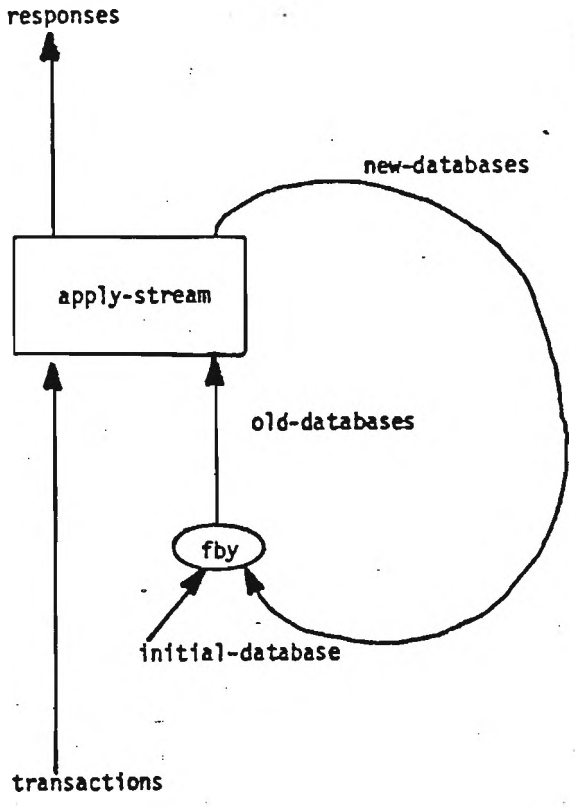

Figure 3-1: Transaction application in graphical form.

expression for the present example would be

```
responses = process(initial_database, transactions)
process(db, transactions) =
        fby(response, process(new_db, rest(transactions)))
                where [response, new_db] = first(transactions)(db)
```

Level 2. Next, we give details of apply_stream. The incoming stream of transaction requests is typically in symbolic form. One of the jobs of apply_stream is thus to interpret the symbols to make functional sense of them. But what is the result of this interpretation? For a given transaction, we wish to produce two functions, one of which is applied to a database to give a

response, and the other of which is applied to get a new database. Let us call this function-producing function map_trans. Thus, the type of map_trans is

[requests -> [databases -> responses x databases]]

The notation here is that A x B is the Cartesian product of A and B, and [A -> B] is the set of functions from A into B.

In order to form apply_stream from map_trans, what must be done is roughly to apply map_trans to each transaction in the sequence, then apply the result to the stream of old_databases. However, that will give us a stream of pairs, rather than a pair of streams. So we must "un-pair" the former, to get the latter. Thus:

```
apply_stream(transactions, old_databases) =
    un_pair(pointwise(toall(map_trans, transactions), old_databases))
```

Here, toall is a function which applies its first argument, a single function, to every component in its second argument stream, to produce a new stream. Furthermore, pointwise applies each component of its first argument, a stream of functions, to the corresponding component in its second argument, a stream of arguments to those functions.

Level 3. Refinement now proceeds to the definition of map_trans. This is the first level at which any details of the form of the transactions must be taken into account. Details of the database representation can be postponed until level 4. Recall the type of map_trans as expressed above. Effectively, map_trans must "decode" a request to produce a function of the desired type. Thus, it provides a sort of case-analysis. Suppose that a transaction t is decomposed into a transaction type (e.g. insert, delete, print, etc.) and a list of parameters which specify, for example, the names of objects to be

inserted, the relations into which they are to be inserted, etc. Then the form of map-trans is

```
FUNCTION map_trans(trans)

RESULT  case(typeof(trans),
                "insert",       insert_in_db(parameters(trans)),
                "delete",       delete_from_db(parameters(trans)),
                "print",        print_fun(parameters(trans)),
                    .
                    .
                    .
                                error_fun(trans))
WHERE
        FUNCTION insert_in_db
                    .
                    .
        FUNCTION delete_from_db
                    .
                    .
        FUNCTION print_fun
                    .
                    .
        FUNCTION error_fun
                    .
                    .
END
```

where error_fun specifies what to do in case of an unidentifiable transaction request. The functions insert_in_db, delete_from_db, etc. are examples of two-tiered functions (as discussed in section 2.3) in that each, when applied to parameters(trans), yields a function which is ultimately applied to the database.

Level 4. Lastly, we specify details for each of the two-tiered functions above, taking into account for the first time the specific representation of the database. We illustrate with an example for insert_in_db. Suppose that for simplicity we use linked lists to represent arbitrary-length sequences of objects. Moreover, let the database be a sequence of "relations", the contents of a relation be a sequence of "sets", and each set consist of a key and a

sequence of "members". This could be considered an implementation of a form of "entity-relationship" model (cf. [Ullman 80]). We also want each relation to have an identifier, so choose to represent a relation as a pair

[relation_id, sequence_of_sets]

To summarize the types involved, let A^* designate a sequence of objects of type A. Then

database = relation*

relation = identifier x set*

set = key x member*

identifier = atom

key = atom

member = atom

where, by atom, we mean some primitive values, which might include strings and integers.

To specify the two-tiered function insert_in_db, we will assume that the parameters for insertion specify the relation name, the key, and the member to be inserted, as in:

```
FUNCTION insert_in_db(parameters)(db)
IMPORTS insert_into_relation, search_list, create_reln
LET     parameters BE [rel_id, key, member],
        insert_fun BE insert_into_relation(key, member),
        [dummy, new_db] BE
                search_list(db,
                            compare_id(rel_id),
                            insert_fun,
                            list(insert_fun(create_reln(rel_id)))
                            )
RESULT  ["done", new_db]
```

This specification can be understood as follows: The function is two-tiered in the sense that it maps the tuple of three parameters [rel_id, key, member] to a function which in turn maps a database into a pair consisting of a response and a new database. We assume that an insertion will always succeed, so that the response "done" is uniformly offered. If the specified relation is not found, then one will be created. The new database is obtained using the utility function search_list, which performs both a search and reconstruction of its first argument list.

The general form of search_list is search_list(x, pred, modifier, not_found). Here x is the list to be searched, pred is a predicate on individual list elements, modifier is a function which maps one list element into another, and not_found is a list. The list is searched sequentially until an element e is found for which pred(e) = true. As the searching is done, a new list is being constructed, which carries over elements for which pred(e) = false identically. Assuming e is found such that pred(e) = true, the new list contains modifier(e), in place of e, and continues with the rest of the old list as is. If no such e is found, then the list not_found is appended to the end of the new list. The value returned by search_list is a pair consisting the first e such that pred(e) (or nil if there is no such e) and a new list. In the case of the use of search_list within insert_in_db, only the second component of the result is important, by the assumption that an insert always succeeds. Fig. 3-2 shows the result of applying search_list with x as a first argument, where pred(e_3), but not pred(e_1) and not pred(e_2).

To complete the explanation, functions insert_into_relation, compare_id, and create_reln are also used within insert_in_db, respectively to insert into a relation, compare a specified id to that of a relation in the database's list,

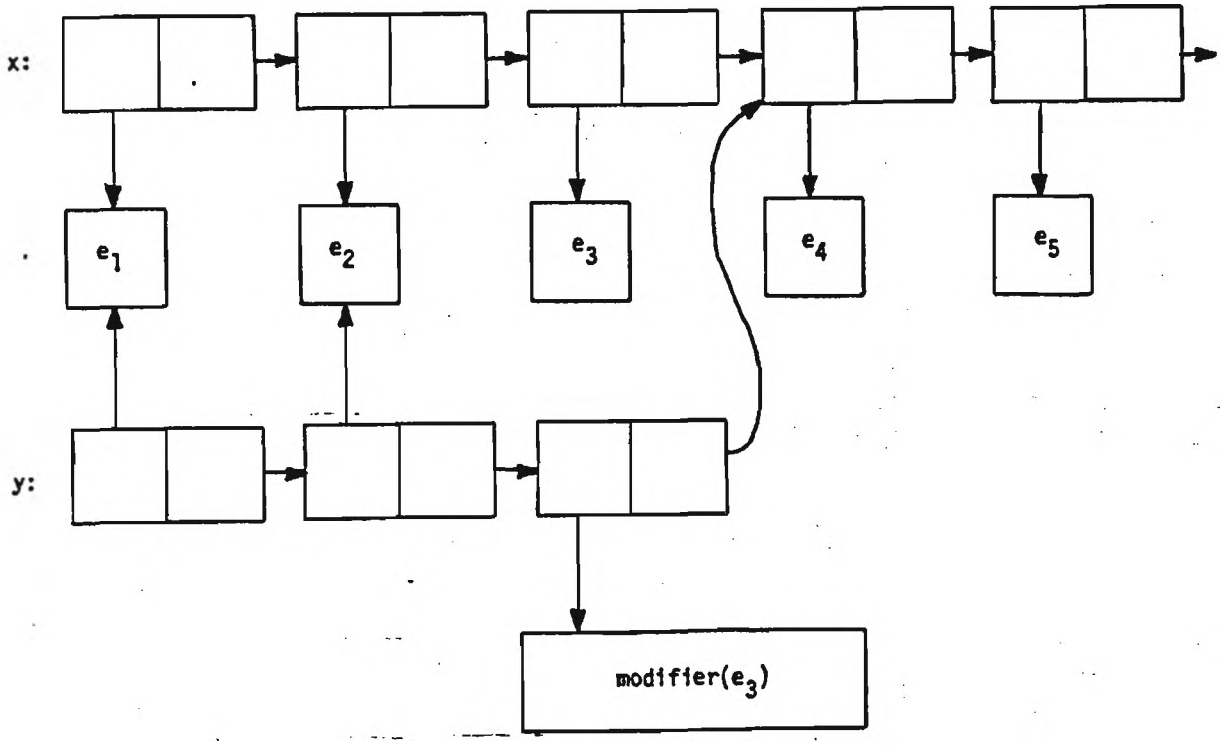

Figure 3-2: Sample search_list application.

and create a new relation with the specified id. We omit the detailed refinement of these functions. The last two are trivial, and the first resembles insert_in_db, using search_list.

3.2. Database Concurrency and Synchronization

The functional approach to updating, as exemplified by the discussion in the previous section, performs all necessary synchronization implicitly. The user is not required to program locks, semaphores, etc., as with other methods (cf. [Ullman 80]). In effect, the linkage mechanism underlying the reduction implementation of our functional language effects the equivalent of a "timestamp order" execution, but without explicit reliance on timestamps.

To see how synchronization is accomplished, consider a database composed of three relations R, S, and T. For simplicity in explanation, we shall assume that the database is represented as a pointer to a 3-tuple, rather than as a linked-list, each component being one of the relations, as shown in fig. 3-3a. Each transaction yields a new database, which is represented by a new 3-tuple. Since no in-place modifications are ever performed, it is perfectly safe to share some of the components between the database before and after a transaction. The result of a pure-read transaction applied to the database of fig. 3-3a, in which all components are thus sharable, is shown in fig. 3-3b.

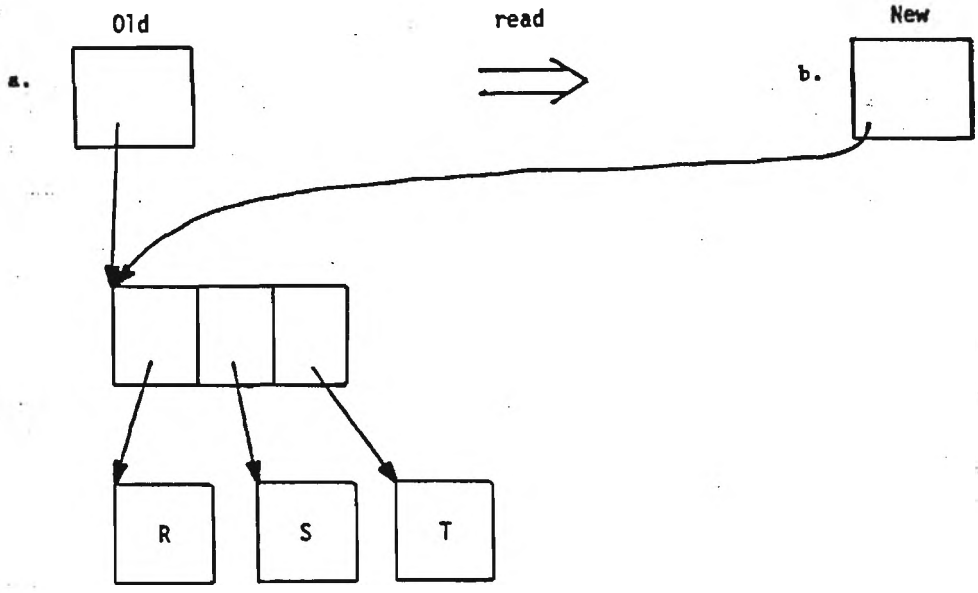

Figure 3-3: Pure-read transaction.

Now suppose that a transaction is to be applied which modifies one of the relations, say S. As always, a new tuple is created, but now a new version of S, call it S', is created, while the unmodified R and T remain shared with the former version, as shown in fig. 3-4. A subsequent transaction sees the

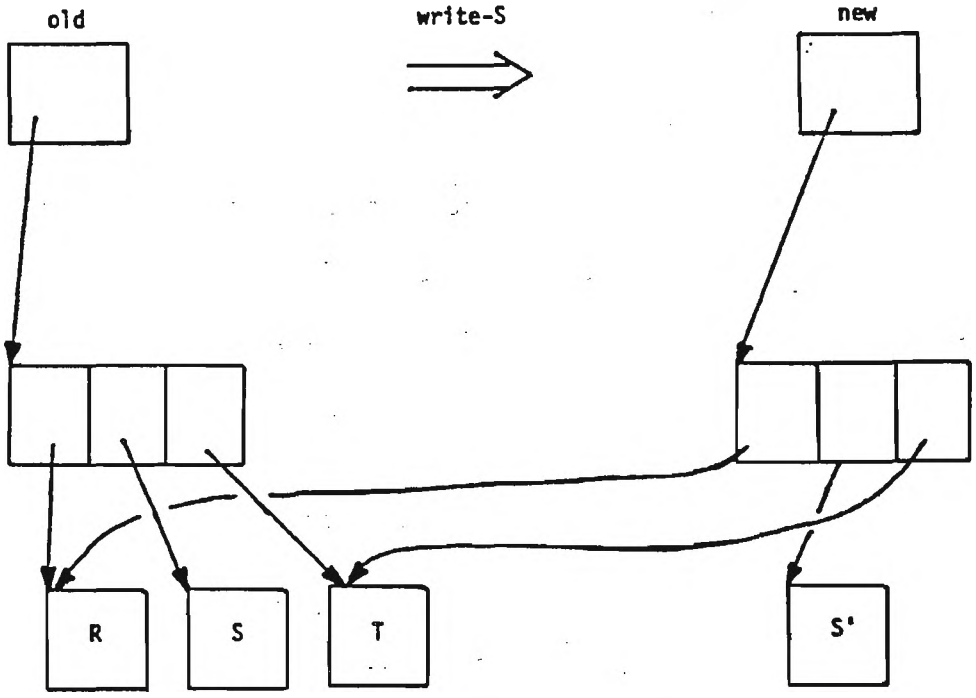

Figure 3-4: Partial update effect.

database as composed of R, S', and T.

An important aspect of the lenient data constructor [....] is that a tuple node typically becomes available before its components are completely formed. Thus, if a transaction following the write transaction of the previous paragraph depends only on the R and T components, it can proceed immediately without waiting for the S' component to be completely established. This means that the database can effectively be pipelined through the transaction stream, in that different transactions can be processing constituent objects concurrently. As with synchronization, this concurrency takes place without explicit directives to set up concurrent processes, etc.

The degree of concurrency actually realized is sensitive to the programming of the functions used to carry out transactions. This is one reason why we suggest combining the search and modification functions, as in search_list of Section 3.1. By reconstructing the list as it searches, the new list is made available for processing by subsequent transactions sooner that if the list were first searched, and then a new list created, in two steps.

The pipelining technique described above need not stop at the level of relations as components. It can be extended to work within relations as well. Suppose, for example, that a relation were represented as a singly-linked list of "sets", as in the preceding section. Then a read-only use of the relation could return the entire relation immediately, whereas a write-transaction could return a partially-constructed replacement relation. If, for example, the write were known to entail a modification to a single set in the relation, based upon the attributes therein, it could search successive sets in the list, forming a new list of the sets which are not modified. This list, even though only partially complete, could be used by a subsequent transaction.

The maximum concurrency due to pipelining a single relation represented as a linked list is on the order of the length of the list. That is, it is possible to have each set being processed by a different transaction at the same time. Of course, this maximum will rarely materialize, due to differences in rates of issuance and progression of transactions. A representation which is likely to be less sensitive to rates of progression uses "bushy" trees, rather than linked lists [Keller 80d]. Particularly attractive are various "balanced" trees, which offer a worst-case logarithmic insertion and search times. A great deal of attention has been devoted to exploiting concurrency in such tree representations using explicit locking [Bayer 77, Kwong 80]. While the degree

of concurrency achieved using the functional approach described here might not be as great, it is reaped with much less sophisticated programs and less susceptibility to error. Although the possibility of applicative updating of B-trees has been verified by members of our group [Hudak 81, Doany 81], we do not present the technical details of this aspect here.

3.3. Multi-user Transactions

The problem of processing transactions independently introduced into the system at multiple sites has been conveniently decomposed into the merging of the stream of transactions, and the processing of the merged stream. Assuming that each transaction request is originally tagged with its source, the merged stream is actually a sequence of objects of the form

[tag, contents]

Our previous exposition has ignored the tags, and simply dealt with processing a merged stream of contents. However, a simple functional technique permits the separation of the tags and rejoining them with the corresponding responses, since the processing of the merged stream is sequential from an external point of view. A functional expression of this processing is

```
FUNCTION process_tagged_stream(stream)
LET      [tags_stream, contents_stream] BE untuple(stream)
RESULT   tuple(tags_stream, process(contents_stream))
```

Here untuple and tuple are two functions which, respectively, convert a stream of tuples into a tuple of streams, and vice-versa.

In order to select its responses from the tagged stream, each "user" (i.e. issuer of transactions) is effectively performing the function choose(my_tag, result_stream), defined in terms of the function select_from_stream defined in

Section 2.3. The functional definition of choose is

```
FUNCTION choose(tag, stream)

LET     [my_stream, their_stream] BE
                select_from_stream(lambda(x)x=tag)(stream)

RESULT  my_stream
```

3.4. Application to Distributed Database Systems

Distributed database systems generally observe either a primary-site model or a primary-copy model (cf. [Bernstein 81]). In the former, at every instant of time, some site plays the role of the primary site, through which all transactions must pass for coordination, regardless of origin. This creates a bottleneck which is temporary, in the sense that once a transaction passes through the site, finer grain actions associated with it may be done concurrently. In the primary-copy model, a transaction simply proceeds without initial coordination, all required coordination being done at a "primary copy" of each database object. (If the database is unredundant, then each object is its own primary copy.)

The technique demonstrated in this paper is applicable to the primary-site model. As we have already discussed, the required coordination can be done in a manner which is almost completely functional. Although functional representations for the primary-copy model also appear possible, they are more complicated, due to the need to retain the ability to abort transactions to resolve deadlock. We leave the handling of such behavior to a future exposition.

3.5. Relevance to the Primary Site Model

For sake of simplicity, assume a non-hierarchical "local" network model, in which the physical connectivity permits each site to send a message to each other. The "Ethernet" model [Metcalfe 76] is a workable example. We immediately note that such a network does not physically provide secure communication. This is an issue which is best dealt with by cryptographic means; these lend themselves naturally to a functional description, but which will not be further discussed here.

An important observation is that the network medium acts as one large merge pseudo-function. The stream of messages which appear on it over time will not be deterministic, but will consist of an interleaving of messages generated at different nodes. Interestingly enough, a functional representation of message handling is possible in a manner analogous to the handling of merged streams in Section 3.3. Instead of transactions, we have arbitrary messages, again with destination tags. A site effectively selects the messages directed to it by performing the choose operation on the entire message stream. Figure 3-5b illustrates the logical view of a network, the physical structure of which is suggested in fig. 3-5a.

It also seems noteworthy that the problem of site addition, deletion, etc. can be viewed as another database problem. That is, a central administration site must maintain a database of authorized sites and respond to requests for site addition, deletion, etc. In turn, the administration site may be responsible for providing physical addresses for various logical sites, encryption keys, etc.

Logically, the site at which database functions are processed is irrelevant.

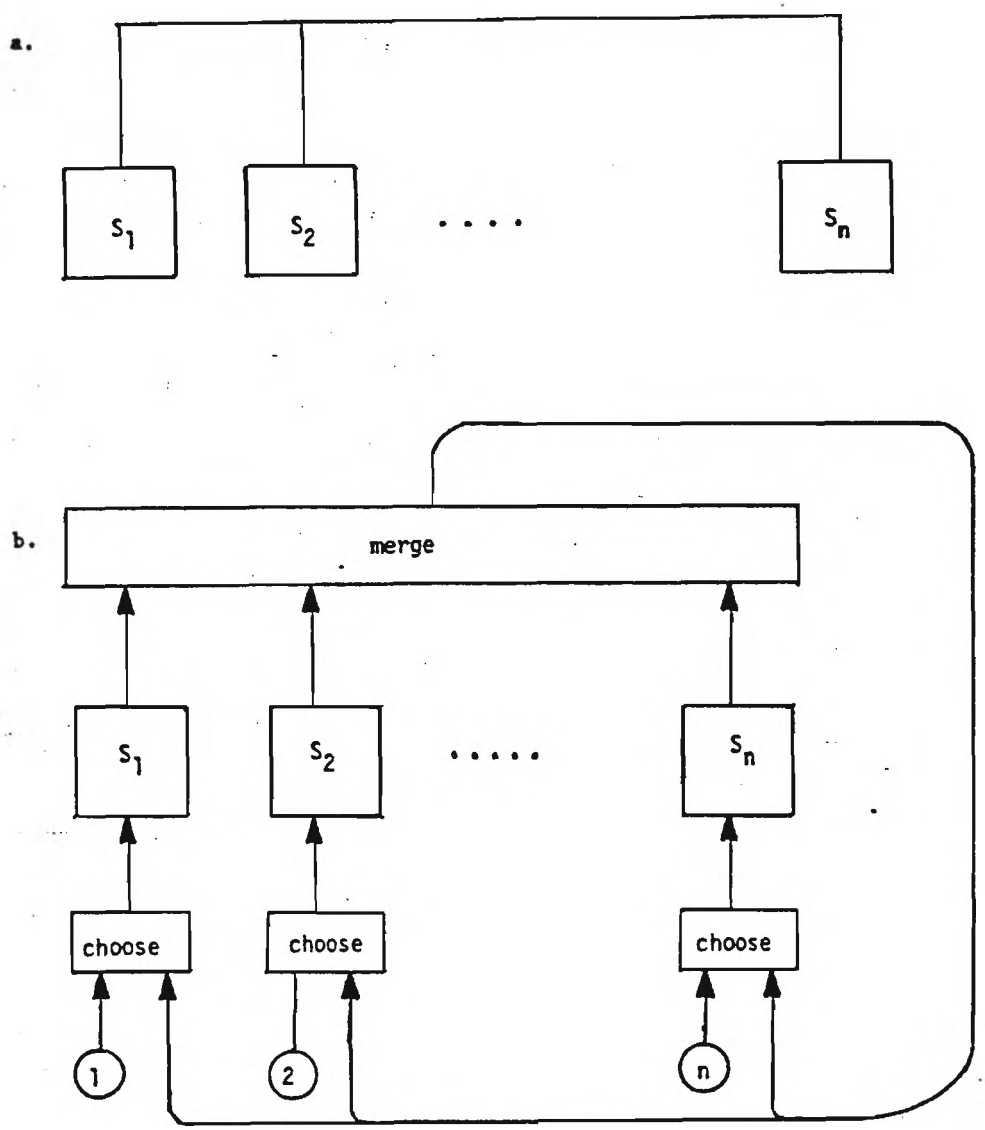

Figure 3-5:
Site-based substream selection; a. Physical network; b. Logical merge/choose.

However, it may be physically more efficient or otherwise important to choose one site over another for the application of a given function. For this reason, we suggest the use of a site pragma as an option to a function. This pragma can take the form of a parameter to the function which gives the address of the preferred site of execution. A tentative form would be

RESULT_ON(functional-expression, site)

which yields the value of the first argument, but requires the outermost function to be computed on the specified site. That function could likewise specify the execution of subsidiary functions on particular sites, or on its own site, which it could obtain by executing the expression

my_site()

To retain functionality, site parameters could be made unavailable for use by any function except RESULT_ON. If a primary-site is used, it could contain necessary site values in the root directory for the overall database.

4. PHYSICAL DESIGN CONSIDERATIONS

In order to better describe how our approach to database processing fits into a distributed architecture, we discuss some aspects of a multiprocessor/network architecture oriented toward the execution of functional programs. The ideas here derive from our work presented in [Keller 79], with emphasis on logical interconnectedness, rather than physical topology. A more extensive form of task migration will also be described. We concentrate on those architectural aspects which relate to the necessary flow of communication traffic from one physical PE to another, as opposed to fine-grain aspects of operator execution. More detail on the latter may be found in reference [Keller 80e].

4.1. Integration of Processors and Memory

The functional implementation described in [Keller 79] avoids the shared-memory bottleneck common to many multiprocessors by integrating memory with processing capability. More specifically, we stipulate that a PE shall have both a processor and a memory which it alone directly accesses. Access by one processor of another processor's memory is logically possible, but physically

occurs by the former processor sending a message to the latter containing the locations to be accessed. After this message makes its way through the interconnection network, it becomes a task for the receiving processor. The latter returns a message containing the contents of the requested locations, which becomes a task to be executed by the processor desiring to use the contents of those locations. The fact that each processor is solely responsible for direct access to its own memory simplifies considerations for implementing mutual exclusion. Each processor effectively becomes a "serializer" of its own local activities.

Despite the allusion to "locations" in the preceding paragraph, we emphasize that the functional programmer does not program directly using locations. Rather, locations are assigned dynamically by the system as a means for referencing objects, such as blocks of words representing the consequent of an FGL macrofunction.

4.2. Transparency of Interconnection Topology

It is possible to factor out considerations of PE interconnection topology from logical communication aspects of the scheme being described. The essential aspect of being able to coordinate distributed execution in our model is the assignability of a unique system-wide address to each object, and the ability for the physical topology to route information to any specified address. Since a PE will likely have a fixed memory size, such addressing could be achieved by concatenating a PE address with the address of a location within a PE. Nodes which route information within the network must, of course, take the physical topology into account. Sufficient information must be programmed into the routing nodes to enable them to make local decisions regarding the direction in

which to forward an arriving message.

4.3. Address Space as a Task Management Medium

The system-wide logical address space plays an important role in the management of the linkages necessary to achieve the graphical expansion of macrofunctions. If a decision is made to allocate the storage for the consequent of a macrofunction in PE B, which is different from the PE A of the antecedent, then it is necessary to first find storage in B. When storage has been found, the address at which it was found is returned to A, which enables it to forward any necessary arguments. Similarly, B knows the location of the antecedent, and can use it to return the result of the consequent's execution. Figure 4-1 suggests this using the graph reduction model.

4.4. Task Pre-Linkage Migration

The mode of linkage described above implies that a task representing the pending expansion of a macrofunction is entirely mobile up until the time storage is allocated for the expansion. It carries with it the address of the code used to initialize the consequent storage block, and the address of the antecedent node of the expansion, but it may be freely moved about the system. This permits a certain degree of load balancing to be accomplished [Keller 79].

By monitoring the amount of memory required to perform macro-expansions for various nodes, it is possible to determine whether one processor is back-logged relative to its neighbors. If so, any tasks corresponding to unexpanded nodes can be moved to a less heavily loaded PE. The migration of such tasks can thus help keep the processing load more evenly distributed, which is desirable to achieve speed-up through parallel processing.

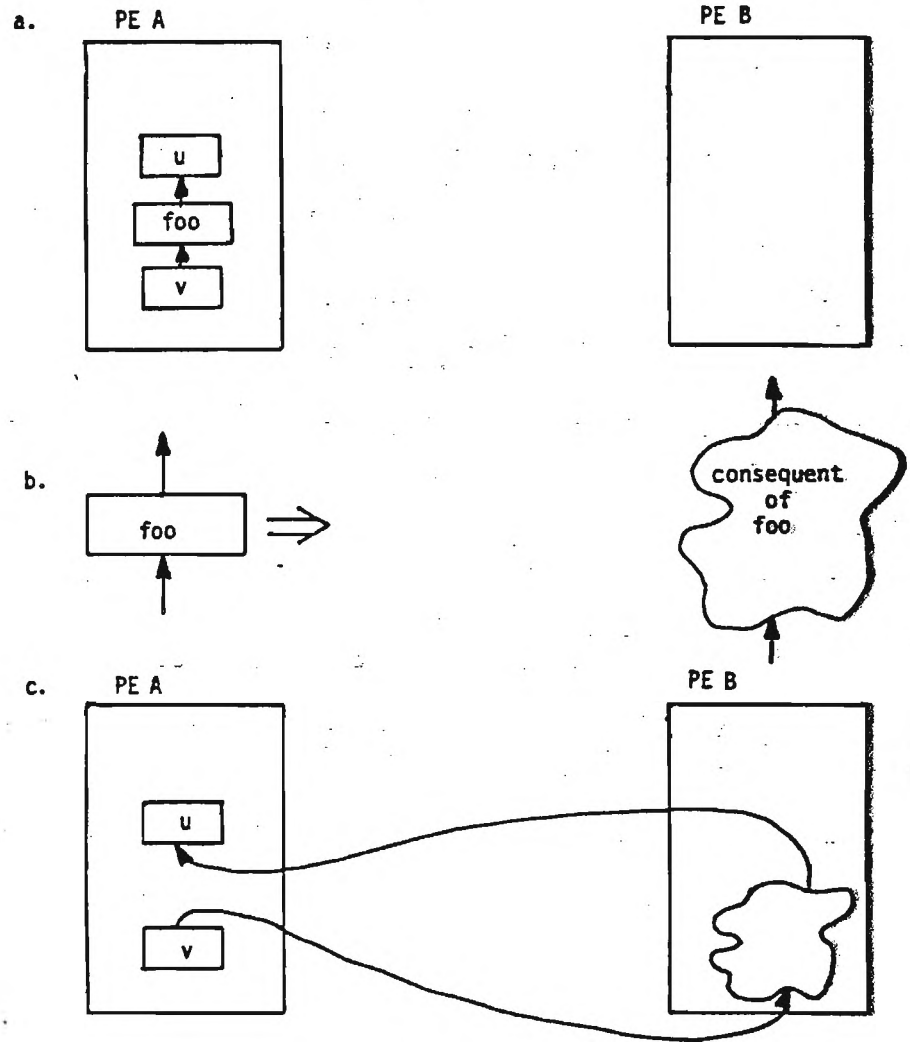

Figure 4-1:
Illustrating the remote application of a macrofunction foo; a. instance of antecedent; b. production; c. expansion

4.5. Task Post-Linkage Migration

We described in the previous section a scheme for permitting the migration of tasks corresponding to a macro-expansion prior to the actual expansion. A more extensive form of load distribution could obviously be obtained if tasks were also able to migrate following expansion. To achieve this effect, it is necessary to be able to relocate the storage of an active block to another PE.

One scenario in which such migration would be useful is that of a "dual-purpose" local network. Typically, PEs in a local network serve as semi-autonomous units, say of the personal-machine level, which communicate on occasion by message passing. However, suppose that an application which initiates at one such unit can exploit the extra processing capabilities at other idle units. The method described in previous sections makes this possible. But there is a conflict if the user of a personal machine wishes to assume control of a machine while it is being used in conjunction with a multiprocessing application. The conflict can be resolved if it is possible to gracefully move the multiprocessor workload from that machine to another within the network.

One method of accomplishing "post-linkage" migration is to provide an additional mapping table. This mapping table is used to translate logical block addresses into physical ones, similar to the segment table translation which accompanies most segmentation schemes [Bensoussan 69]. However, there is one such table for each PE. An entry in the segment table exists for each logical block number referenced within the PE. The entries themselves either point to a physical block within the current PE, or to a segment table entry in some other PE. We call the latter case an indirect segment reference.

In order to move a block, we must do the following:

1. Temporarily suspend reference to it.
2. Find a new physical location for it.
3. Update the segment table of the PE from which it was moved.

One disadvantage of this migration scheme is that long chains of indirection might build up after several moves of the same block. However, we anticipate

that moves will not be sufficiently frequent to seriously impair efficiency. It is also possible to optimize indirection chains by occasionally "updating" indirect pointers so that they refer to some table entry further along in the chain.

To summarize this section, we have sketched a class of system architectures for physically managing the tasks resulting from the availability of concurrently-executable functions, in a manner which achieves transparency with respect to the number of PEs and their physical location. Thus, tasks may be completely managed by the system as a default, with the site pragma of Section 3.5 being usable optionally to enhance efficiency.

5. EXPERIMENTAL RESULTS

The techniques mentioned in this paper have been implemented in the functional language FGL and tested using a simulator which reports the degree of concurrency achieved, among other measurements. An experiment was performed which processed 50 transactions on a database of 1, 3, and 5 relations, having a total of 50 "sets". The transactions were all either simple inserts or finds, and the percentage of inserts was varied through 4, 8, 16, 32, and 64. The simple linked-list implementation described in Section 3.1 was used. Table I shows the maximum and average concurrency achieved for all combinations of the experimental parameters. These results are idealized, in that no communication delay is taken into account.

Table I

	number of relations					
	5		3		1	
0%	18	7.06	20	7.98	15	8.49
4%	20	6.51	19	7.26	20	8.38
inserts 8%	18	6.62	18	7.55	22	8.51
16%	19	6.68	19	7.28	23	7.88
32%	19	6.63	20	6.87	25	6.90
64%	16	5.67	16	5.46	14	5.85

maximum and average concurrency

It is notable that the degree of concurrency seems reasonably high for such a small example. The small variation with respect to the percentage of updates seems remarkable. We conjecture that it is due to the use of the function search_list of Section 3.1, which reconstructs relations incrementally as insertions are being made.

6. CONCLUSION

We have presented a functional approach to distributed database processing processing, indicating the handling of updating and multi-site transaction processing in a manner which is transparent with respect to logical sites and concurrency. Some language and implementation aspects of our approach have been discussed, particularly applicative approaches to updating which enhance concurrency.

7. REFERENCES

[Backus 78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 21(8):613-641, August, 1978.

[Bayer 77] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. Acta Informatica 9:1-21, 1977.

[Bensoussan 69] A. Bensoussan, C.T. Clingen, and R.C. Daley. The Multics virtual memory. In Second ACM Symposium on Operating Systems Principles, pages 30-42. Princeton University, October, 1969.

[Bernstein 81] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. Computing Surveys 13(2):185-222, June, 1981.

[Brown 62] G. Brown. A new concept in programming. Wiley, 1962, . in M. Greenberger (editor), Management and the computer of the future.

[Buneman 79] P. Buneman and R.E. Frankel. FQL - A functional query language. In ACM Sigmod, pages 52-58. May-June, 1979.

[Buneman 82] O.P. Buneman, R.E. Frankel, and R. Nikhil. An implementation technique for database query languages. ACM TODS to appear, 1982.

[Burge 75] W.H. Burge. Recursive programming techniques. Addison-Wesley, 1975.

[Davis 82] A.L. Davis and R.M. Keller. Dataflow program graphs. Computer , February, 1982. (to appear).

[Doany 81] R. Doany. Implementation of a network database using a function graph language. Master's thesis, University of Utah, Dept. of Computer Science, June, 1981.

[Friedman 76] D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. Edinburgh University Press, 1976, pages 257-284. in S. Michaelson and R. Milner (eds.), Automata, Languages, and Programming.

[Friedman 77] D.P. Friedman and D.S. Wise. Aspects of applicative programming for file systems. Sigplan Notices 12(3):41-55, March, 1977.

[Friedman 78] D.P. Friedman and D.S. Wise. The impact of applicative programming on multiprocessing. IEEE Trans. on Computers C-27(4):289-296, Apr, 1978.

[Friedman 79] D.P. Friedman and D.S. Wise. Applicative multiprogramming. Technical Report 72, Computer Science Dept., Indiana University, April, 1979.

[Henderson 76] P. Henderson and J.H. Morris, Jr. A lazy evaluator. In Proc. Third ACM Conference on Principles of Programming Languages, pages 95-103. 1976.

[Hudak 81] P. Hudak. Applicative implementation of b-tree insertion/deletion. 1981.private communication, University of Utah.

[Kahn 81] K.C. Kahn and F. Pollack. An extensible operating system for the Intel 432. In Compcon '81, pages 398-404. IEEE, February, 1981.

[Keller 78] R.M. Keller. Denotational models for parallel programs with indeterminate operators. North-Holland, 1978, pages 337-366. in E.J. Neuhold (editor), Formal description of programming concepts.

[Keller 79] R.M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In AFIPS, pages 613-622. AFIPS, June, 1979.

[Keller 80a] R. M. Keller. Semantics and Applications of Function Graphs. Technical Report UUCS-80-112, University of Utah, Computer Science Department, 1980.

[Keller 80b] R. M. Keller, B. Jayaraman, D. Rose, G. Lindstrom. FGL (Function Graph Language) Programmers' Guide. Technical Report AMPS Technical Memorandum No. 1, University of Utah, Computer Science Department, July, 1980.

[Keller 80c] R.M. Keller, G. Lindstrom, and S. Patil. Data-flow concepts for hardware design. In IEEE Compcon '80, pages 105-111. Feb., 1980.

[Keller 80d] R.M. Keller. Divide and CONCer: Data structuring for applicative multiprocessing. In Proc. 1980 Lisp Conference, pages 196-202. August, 1980.

[Keller 80e] R.M. Keller and G. Lindstrom. Hierarchical analysis of a distributed evaluator. In Proc. International Conference on Parallel Processing, pages 299-310. August, 1980.

[Keller 81a] R.M. Keller and W-C. J. Yen. A graphical approach to software development using function graphs. In Proc. Compcon 81, pages 156-161. IEEE, 1981.

[Keller 81b] R.M. Keller and G. Lindstrom. Applications of feedback in functional programming. In ACM Conference on functional languages and computer architecture, pages 123-130. October, 1981.

[Kwong 80] Y.S. Kwong and D. Wood. Approaches to concurrency in B-trees. In P. Dembinski (editor), Mathematical foundations of computer science, pages 402-413. Springer Verlag, September, 1980. Lecture Notes in Computer Science, No. 88.

[Metcalfe 76] R.M. Metcalfe and D.R. Boggs. Ethernet: distributed packet switching for local computer networks. Commun. ACM 19(7):395-404, Jul, 1976.

[Reed 78] D.P. Reed. Naming and synchronization in a decentralized computer system. PhD thesis, MIT, September, 1978.

[Shipman 81] D.W. Shipman. The functional data model and the data language DAPLEX. ACM TODS 6(1):140-173, March, 1981.

[Smith 81] J.M. Smith, et al. Reference manual for Adaplex. Technical Report CCA-81-02, Computer Corporation of America, January, 1981.

[Traiger 79] I.L. Traiger, J.N. Gray, C.A. Galtieri, B.G. Lindsay. Transactions and consistency in distributed database systems. Technical Report RJ2555, IBM Research Lab., San Jose, CA, 1979.

[Ullman 80] J.D. Ullman. Principles of database systems. Computer Science Press, 1980.

[Zeigler 81] S. Zeigler, et al. The Intel 432 Ada programming environment. In Compcon '81, pages 405-410. IEEE, February, 1981.