5-1-1985

Approaching Distributed Database Implementations Through Functional Programming Concepts

Robert M. Keller
Harvey Mudd College

Gary Lindstrom
University of Utah

Approaching Distributed Database Implementations through Functional Programming Concepts

Robert M. Keller
Gary Lindstrom

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

Abstract: The application of functional programming concepts to the *data representation* and *querying* aspects of databases has been discussed by Shipman and Buneman, et al. respectively. We argue the suitability of a function-based approach to additional aspects of database systems, including *updating, transaction serialization,* and *physical distribution and communication.* It is shown how the "merge" extension of a purely functional model permits serializable concurrent "primary site" distribution control. We also present preliminary experimental results which indicate that a reasonable degree of concurrency is attainable from the functional approach.

Key phrases: *applicative programming, functional programming, lenient data constructors, distributed databases, primary site model, concurrency, distributed systems.*

1. An Applicative Approach to Distributed Computing

We select as our distributed computing context a model presenting flexible, logical and physical organization, in which computing load can be shifted transparently from one PE to another. It is based on the notion of *applicative* (or functional) *multiprocessing* [5, 10].

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. As such, there is no explicit notion of "locking" as is found in typical discussions of concurrent database systems. Thus, functional programs can give rise to substantial advantages, including:

Easily-realized concurrency: Due to the absence of visible side-effects in functional languages, sub-expressions which exhibit no data dependencies may freely be evaluated concurrently without reservation. The typical use of powerful recursion and mapping operators, which conceptually expand into large expressions with many independent sub-expressions, provides numerous tasks which may be concurrently evaluated.

Semantic basis: Functional languages have relatively clean uniform semantics, facilitating formal reasoning about programs, and hence their verification;

Modularity: The adoption of the mathematical notion of function as a basis for programming enhances incremental understanding and module reusability.

In addition, less obvious features of modern applicative programming are relevant to distributed systems, including:

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 achieving a sharing effect. Since assignment side-effects are precluded, this space-efficient sharing is semantically transparent.

Processing incomplete objects: Through the use of *lenient data constructors* (our phrase [10] for the semantic counterpart of the operational notion of "lazy evaluation" [4, 7]), data structures need not be constructed in their entirety before they are used as components in other structures. For example, a lenient *tuple* constructor creates a tuple (1-dimensional vector) which itself is an object, the components of which are made positionally accessible *before* any of the components are necessarily completely computed. Several systems supporting such data constructors have been implemented [2, 13, 22]. An important consequence of this technique is that input sequences of unknown or infinite length, called *streams*, are *bona fide* data objects. Moreover, the possibility of overlapped access and generation of lenient data constructors gives rise to further opportunities to exploit parallel processing capability [5].

This work was supported by a grant from the IBM corporation, and by National Science Foundation grants MCS 81-06177 and MCS 78-03832. A preliminary version appeared as [12].

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. [21]:

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.

We offer a note on types of concurrency that may be helpful in understanding certain points brought out in the remainder of the paper. Concurrent processing can informally be classified into two types:

Pipelining, in which several (logical) processors operate on different items in a sequence, the result produced by one processor being passed on to the next; and

Flooding, in which a set of independent data are operated on concurrently by a set of processors.

As an example from database systems, we can classify concurrency such as would occur in the search of several relations within one transaction as being flooding. On the other hand, concurrent processing of parts of successive transactions would be pipelining, this relationship arising from, e.g., write-read dependencies on particular relations.

Despite the aforementioned advantages of applicative programming, a number of questions have persisted concerning the ultimate suitability of the functional approach for adequately treating certain aspects of database systems, both physical and logical. These include:

Updatable objects: How can shared object updating be modeled without wholesale compromise of the functional approach?

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 constrict the potential distributed evaluation associated with the applicative approach?

Site addressing: What techniques can be used to represent optional explicit physical node accesses in the system, as would be required, for example, for the on-line addition of new database users, and the release of terminated ones?

Load management: In a true multiprocessing setting, a general solution must be found for the task migration problem, whereby overloaded PEs can export portions of their activity backlog to less burdened neighbors.

The first three of these issues are addressed in this paper. The last is discussed in [14]. We first indicate a general method for dealing with updating, indicating how the applicative viewpoint can be retained without requiring reconstruction of the entire database when updating. The sharing and concurrency aspects of this approach are emphasized. We then show how the approach can be incorporated into a concurrently-accessed database, and how concurrency control can be effected. Lastly, we present evidence of the merit of our approach by giving figures for degree of concurrency obtained from some simple experiments.

2. Functional Database Processing

2.1. A Functional Formulation of Transaction Processing

It is common to employ a *transaction model* in the application domain of databases, distributed or otherwise. 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 ("queries"), from which there is generated a stream of corresponding transaction responses.

The functional approach to programming entails representing the entire programming task in terms of a specification of objects created from other objects. The essential contrast with traditional assignment-based approaches is that the functional approach does not directly modify any object. However, unneeded objects may be destroyed (through garbage collection) at the system's convenience. Consequently, one problem to be solved is that of representing the phenomenon normally thought of as database *updating*.

Our viewpoint is that each transaction reads a database, and *conceptually* produces a *new instance* of it. Thus, we describe

transaction: databases --> responses **x** databases

(We adopt the convention that the *plural* (e.g. *databases*) of a name to indicate the *set of* objects of that name (e.g. *database*).) As mentioned previously, each transaction produces some response which is returned to the user. The new database is then used for the next transaction to be processed, the database resulting from that transaction is used for the transaction following, etc.

Of course, if the databases in question are large, it is infeasible to *physically* produce a new database for each transaction. However, this is not likely to be necessary if appropriate structuring techniques are used. There exists a variety of contemporary methods for logically decomposing a database and accessing its data. We indicate how these methods may be used to achieve a complete logical reconstruction of databases through a partial physical reconstruction. We shall also argue that in a paged environment, the time overhead resulting from this viewpoint is negligible.

The techniques to be described can be applied to any of the popular structuring schemes, such as the network, entity-relationship, hierarchical, or relational model [23]. For simplicity, we choose to work with only one of these, although the level at which we present our approach is sufficiently high that the details of the model do not appreciably invade the presentation.

Assume the use of the relational model (*cf.* [23]) for concreteness. For notational simplicity in what follows, we assume that a relational database is a set of relations, along with a mapping

names --> relations

from a set of *relation names* to the relations themselves, for purposes of identification. Each relation is a set of *tuples* of data items.

By a *query* we mean a symbolic description of a *transaction* which, for a given database, will produce a response and a new database. Thus, we assume a function

translate: queries --> transactions

which provides such functions from their symbolic descriptions. Thus, translate must parse the query and produce a function which is the transaction itself. Here is where a language capability for "higher-order" (or function-producing) functions is very useful.

We reemphasize that an incoming query is in symbolic form. To map a stream of such queries (such as might be entered from a terminal) into a stream of transactions, we merely apply *translate* to each query. Thus, we can say (in the language FEL [13])

transactions = translate || queries

where || is the *apply-to-all* operator, which applies the function on the left to each component of the stream on the right.

We have already mentioned the formal nature of a transaction; each transaction maps a database to a new database and a response. Let *apply-stream* be an operator which applies a *stream* of transactions one-by-one to a stream of databases, yielding a pair of streams: the stream of responses to the transactions and the stream of databases resulting from the transactions. By feeding to apply-stream the stream of transactions and a stream of databases consisting of the initial database *followed by* the output databases of apply-stream, as shown in Figure 2-1, we achieve a functional program for the complete processing. This may also be expressed as a set of functional equations

```
old-databases = initial-database ^ new-databases

[responses, new-databases] =

    apply-stream:[transactions, old-databases]
```

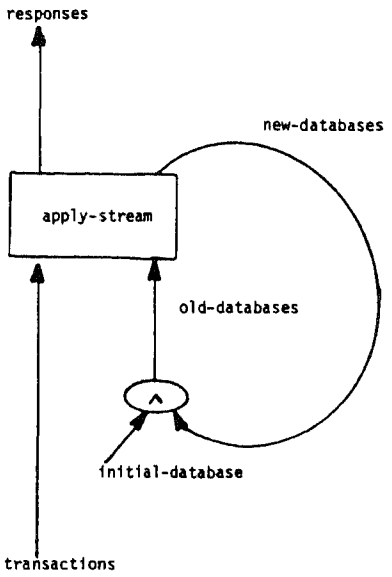

Figure 2-1: Transaction application in graphical form.

Notationally, the colon represents the application of the function on the left to the argument on the right. Brackets represent tupling; a multi-argument function is represented as a function applied to the tuple of its arguments. The symbol ^ is the infix form of the lenient stream-building function "followed-by" which constructs a stream by following the first argument with the second (a stream). It should be pointed out that this graph, or equivalently the system of equations, forms a top-level functional *program* for solving the database update problem.

Similarly, a functional expression for apply-stream may be given as

```
apply-stream:[transactions, databases] =
  if transactions = []
    then [[], []]
    else
      {
      [response, new-database] =
        (first:transactions):(first:databases),

      [more-responses, more-databases] =
        apply-stream[rest:transactions,
```

```
                rest:databases],
    RESULT [response ^ more-responses,
            new-database ^ more-databases]
    }
```

Here *first* and *rest* are the functions which, when applied to a stream, yield the first object in the stream and the rest of the stream respectively, and [] represents the empty stream.

2.2. Functional Database Updating

To demonstrate how our model can achieve full logical updating through partial physical updating, it now becomes necessary to consider typical transactions. Obviously, a transaction tr is *read-only* if it returns the *same* database as its argument:

tr:db = [...some response..., db]

For such transactions, no physical modification is necessary.

Another type of transaction involves insertion of tuples into one or more relations. Usually the specific relations are syntactically derivable from the query, and in a large number of cases, involve only one relation per transaction. We assume this to be the case for the sake of illustration in what follows.

In the same way that we view a transaction as creating a new database, we also view the insertion of a tuple into a relation as the creation of a new relation. Thus, we assume a function

insert-in-relation: relations **x** tuples --> relations

which will be used in the implementation of the function

insert-in-db: databases **x** relation-names **x** tuples --> databases

We now demonstrate how partial reconstruction is accomplished. Suppose that the initial database is

D0 = [R0, S0]

where R0 and S0 are two relations. Consider the transaction sequence:

insert x into R

insert y into S

The database resulting from the first command is

D1 = insert-in-db:[D0, R, x]

while that resulting from the second is

D2 = insert-in-db:[D1, S, y]

= insert-in-db:[insert-in-db:[D0, R, x], S, y]

But

D1 = [R1, S0] where R1 = insert-in-relation:[R0, x]

and

D2 = [R1, S1] where S1 = insert-in-relation:[S0, y]

Thus, we see that D0 and D1 both *share* the relation S0, while D1 and D2 share the relation S1. Thus, a net reconstruction of *two* relations, rather than *four*, has taken place in processing the indicated two transactions.

The principle articulated above generalizes. Clearly, the greater the number of relations, the more sharing possible, and thus the less reconstruction. Furthermore, the same idea can be used *within* the relations themselves. Supposed that a relation is implemented as a set of pages, with each page containing a set of tuples, and that there is a directory page which indexes the other pages. If an insertion or modification affects only a few pages, then all other pages can be shared. A new directory structure is created, the old one being left intact. This is illustrated in Figure 2-2.

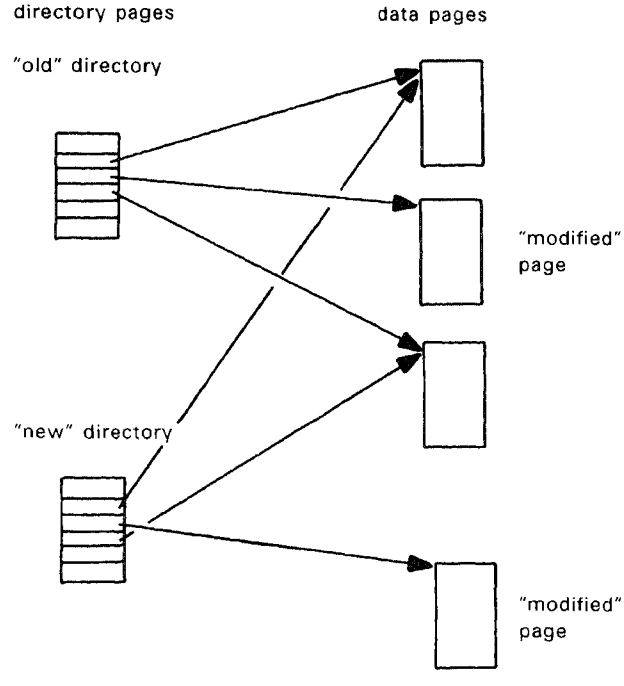

Figure 2-2: Sharing of pages through separate directories

The technique extends with even further sharing possibilities by making the directory structure into a *tree*. In such methods, a path on the order of log n, where n is the number of tuples, is traversed to find a given tuple. Furthermore, insertion or deletion of a tuple requires the replacement of a number of internal node records of the same order. Thus, all but a proportion (log n)/n of a relation can be shared during updating. Moreover, as this

sharing is achievable when the result of the update is expressible as a *function* of the relation prior to updating, the functional approach is indeed attractive for such representations.

2.3. Database Concurrency and Synchronization

Conventional methods for accomplishing concurrent updates to a database required the systems programmer to program locks, semaphores, etc. (*cf.* [23]). In contrast, the functional approach to updating, as exemplified by the discussion in the previous section, performs all necessary synchronization implicitly.

To see how synchronization is accomplished, consider again the database composed of relations R and S. Each transaction yields a new database, which is represented by a new pair. Thus, if a transaction following the insert in S depends only on the R component, it can proceed immediately without waiting for the S component to be completely established. We are here relying on the "lenient" aspect of the tupling constructor, [....], i.e. that we can select and use one component while other components are as yet uncomputed. Regarding the previous example, since the relations R1 and S1 in D2 are clearly derivable *independently* from R0 and S0 respectively, the corresponding insertions can be done in parallel. This means that the database versions can effectively be *pipelined* through the function apply-stream, in that different transactions can be processing constituent objects *concurrently*.

A corollary use of lenient data constructors is that many potential sites for concurrent execution within data structures are available, due to a reduction in the amount of forced synchronization. Although a stream is produced conceptually in sequence, at the top level, many elements of the output sequence are demanded in an anticipatory fashion, to generate as much parallel execution as possible.

The degree of potential concurrency is sensitive to the programming style employed in the transaction functions and the underlying data structures used. For example, a great deal of attention has been devoted to exploiting concurrency in such tree representations using explicit locking [1, 16]. In contrast, the functional approach to tree-updating induces implicit synchronization. While the space overhead in the latter is greater due to the avoidance of in-place modification, the concurrency should in principle be at least as good or better, since only essential data dependencies play a role in synchronization. We also claim that the functional versions are much simpler to program and therefore less susceptible to error.

2.4. Multi-user Transactions and Serialization

Non-functional aspects of distributed database systems cannot be represented within the approach outlined thus far. 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.* [9]).

This merge technique is widely known in the functional programming community, so we treat it only briefly. A semi-functional definition of a (2-way) merge can be found in [11] and will not be repeated here. Informally, a merge has as its input several query 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* indicating that origin must be paired with each request. The function processing the transactions ignores the tag, but keeps it associated with the data so that the response can be routed when desired. (The tagging idea was also used, for example, in [6].) The discussion below ignores such tags for simplicity.

A sufficient condition for the standard criterion of "serializability", (*cf.* [23]) for the processing of concurrent transactions is as follows:

Process the merged stream sequentially.

This condition 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).

It may appear that this approach *loses* concurrency; however this is not the case. We assume an execution mechanism capable of evaluating independent stream components concurrently, such as that described in [10, 14]. Due to the construction of streams and other data objects with lenient data constructors, executable operations will be extracted from the merged stream *as they become available*, rather than in the implied sequence. Then the pipelined processing of transactions can take place, as described earlier. 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. In effect, the linkage mechanism underlying the functional implementation effects the equivalent of a "timestamp order" execution (*cf.* [23]) but without explicit reliance on timestamps. It can thus be seen that the stream structures induce the effect of *version-based objects* [19] on the relations which form the database. It is further possible to "optimize" the transactions for greater concurrency among relational components by judiciously ordering the transactions to be merged, so long as the

order of transactions from each individual stream is maintained. This is a topic for future research.

Figure 2-3 illustrates the merging of two independent transaction streams, and one possible decomposition of the merged stream for concurrent execution. This type of behavior has been experimentally verified. We re-emphasize that no clever compilation or locking techniques need be employed in causing the concurrency to materialize.

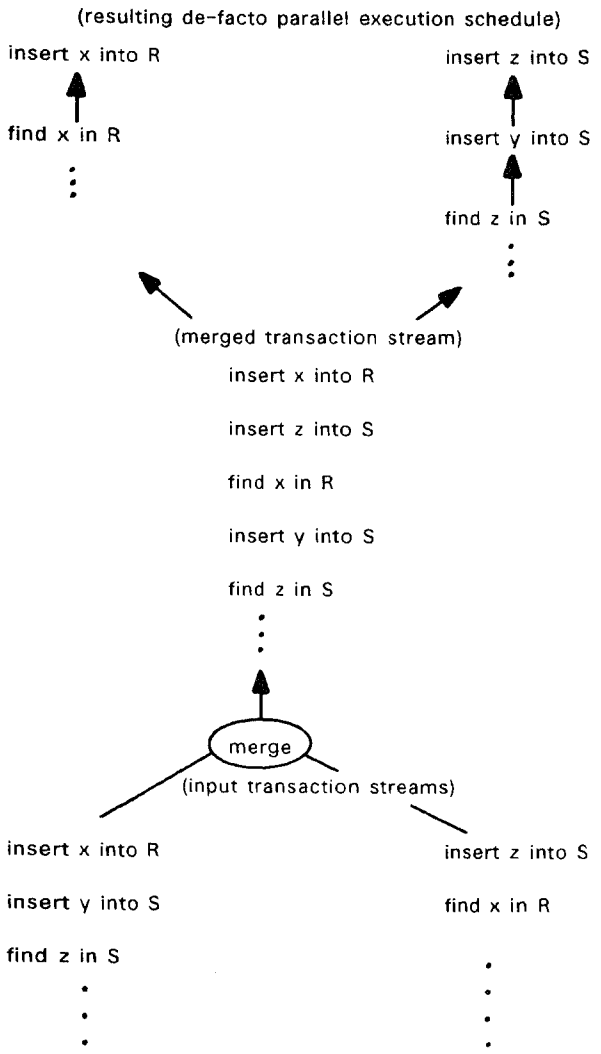

Figure 2-3: Merging and decomposition of transaction streams

3. Application to Physical Aspects of Distributed Database Systems

3.1. Relevance to the Primary Site Model

Two principal models which have been identified for distributed database update control are the *primary-site* model and *primary-copy* model (*cf.* [23]). 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 non-redundant, 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.

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 [17] is a workable example. 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 2.4. Instead of transactions, we have arbitrary messages, again accompanied by destination tags, for ultimate routing of responses. A site effectively selects the messages directed to it by applying a *choose* function to the entire message stream, which selects those messages having a tag which coincides with the site tag. Figure 3-1b illustrates the logical view of a network, the physical structure of which is suggested in Figure 3-1a.

3.2. Site-Selection Pragmas

Logically, the site at which database functions are processed is irrelevant. 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 might 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 evaluating 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 consult the root directory for the overall database to obtain any necessary site values.

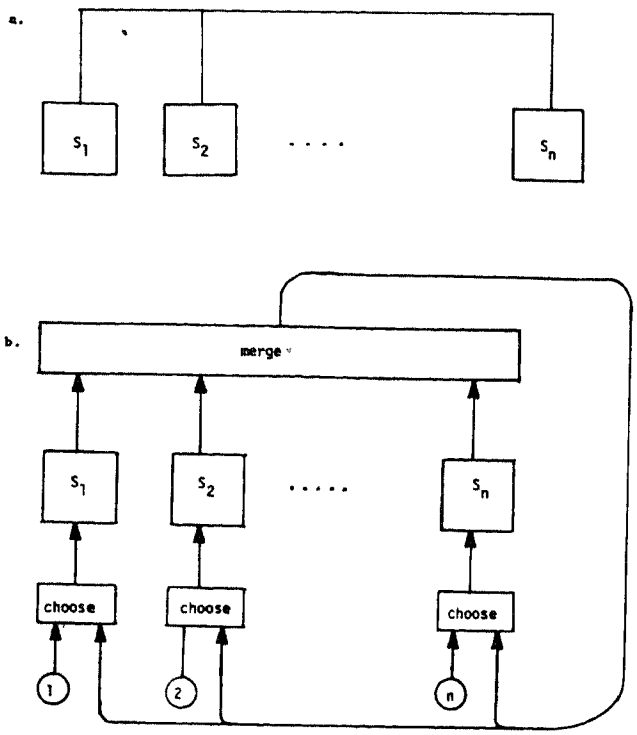

Figure 3-1: Site-based substream selection; a. Physical network; b. Logical merge/choose.

3.3. Secondary Storage Considerations

To further suggest that advantages of functional programming can be obtained in database applications without appreciable extra time costs, the *paging aspect* of physical implementations may be considered. It is common to use a balanced tree strategy in which the size of a tree node is one physical *page*, rather than being based on a specific fan-out. Since the transit time of a page from secondary to main memory is likely to dominate the processing time, the cost of reconstructing the page, as required by applicative updates, is likely to be negligible. Of course, additional parameters and experimental investigation are necessary to substantiate this claim.

Space cost is more of a problem. However, there is reason to believe that some applications will permit "complete archives" to be constructed, using e.g. optical storage. For others, garbage collection must be used to reclaim data, the access to which is dropped.

3.4. Network Topology

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 [10], with emphasis on logical interconnectedness, rather than physical topology. The functional implementation described there avoids the shared-memory bottleneck common to many multiprocessors by integrating memory with processing capability. More specifically, it is stipulated 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 variables 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 variables, which becomes a task to be executed by the processor desiring to use the contents of those variables. 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.

The coordination of distributed execution in our model is simply due to 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.

4. Experimental Results

The techniques mentioned in this paper have been implemented in the functional language FEL [13] and tested using a simulator which simulates the Rediflow evaluation mechanism [14].

An experiment was performed which processed 50 transactions on three versions of a database, with 1, 3, and 5 relations respectively, having a total of 50 tuples among them initially. The transactions were all either single-tuple inserts or finds, and the percentage of inserts was varied through 4, 7, 14, 24, and 38 percent.

For simplicity, a linked-list implementation of both the database and individual relations was used. Intuitively, indications of concurrency for this implementation are apt to be conservative. Tree representations are projected to be even more efficient, since fewer nodes need to be modified on insertion.

The Rediflow simulator [14] has a number of modes available. The first mode assumes an arbitrary degree of parallelism (effectively infinitely-many processors), unit task lengths, and zero communication costs. It is used primarily to estimate the degree of concurrency in the application with given input data. In this mode, the simulator measures maximum and average concurrency in the form of "ply width", where a ply is a maximal set of tasks, all of which can be executed in parallel.

Table I displays the results of execution in this mode. It is notable that the degree of concurrency seems reasonably high for such a small example. It should also be mentioned that the observed concurrency can be classified as the "pipeline" variety, since, by design, no transaction entailed the updating of more than one relation.

Table I: Maximum and Average Degree of Concurrency

percent	number of relations					
updates	5		3		1	
0%	25	14	27	15	39	17
4%	25	14	28	15	45	17
7%	26	14			46	15
14%	26	14	29	13	42	13
24%	24	12	28	11	36	9
38%	24	10	24	9	22	9

A second simulation mode specifies a network topology and a specific number of processors. In this mode, communication delay is taken into account. Table II shows the speedup resulting from the same programs and data using an 8-node binary hypercube, while Table III shows it for a 27-node (Euclidean) cube (3x3x3).

Table II: Speedup, 8-node hypercube

percent	number of relations		
updates	5	3	1
0%	5.6	5.7	6.2
4%	5.6	5.7	6.1
7%	5.6		5.9
14%	5.4	5.5	5.6
24%	5.2	5.0	4.7
38%	4.8	4.6	4.7

Table III: Speedup, 27 node Euclidean cube

percent	number of relations		
updates	5	3	1
0%	7.2	7.6	8.9
4%	7.2	7.6	8.9
7%	7.1		8.9
14%	7.2	7.6	7.8
24%	6.8	6.4	6.1
38%	6.0	6.2	6.0

5. Relation to previous work

The use of a functional programming model for database applications has been only partly explored, notably by Buneman, et al. [2] and Shipman [20]. . However, the first reference does not deal with the question of updating, while the second is mainly concerned with modeling data, rather than modeling the programs which operate on data.

We observed earlier that our functional formulation of updating provides one approach similar to version-based objects [19], but the need for explicit version numbers is suppressed. It remains to be seen whether our formulation can correspondingly be used to simplify the programming of error recovery mechanism.

The conceptual exploitation of concurrency in balanced trees was first discussed by Bayer and Schkolnick [1]. A survey of subsequent work is given by Kwong and Wood [16]. A functional formulation of B-tree insertion and deletion was implemented by Paul Hudak and was incorporated into a simple network (i.e. Codasyl) database in a master's thesis by Rima Doany-Bhakit [3]. Equational code for the special case of 2-3 trees was developed by Hoffman and O'Donnell [8]; Mamdouh Ibrahim has transcribed that code to FEL [13]. A related work is that of Myers [18] which discusses advantages of applicative updating in AVL trees. Kim [15] showed how a functional language with set abstraction can be used as an effective database query language.

6. Conclusion

We have examined the suitability of functional programming concepts for the design and implementation of distributed database applications. We conclude that this approach has considerable merit, especially in its facilitation of transparent concurrency detection and potential exploitation of multiple PEs. Many data representation schemes, such as tree schemes which permit a high degree of sharing, can be expressed in a purely functional way. These offer the possibility of updating without gross modification to the database, as would be suggested by the *mathematical* view of updating in which the entire database is recopied on each update. This view entails a stream of databases (versions), where the first element is the initial database, and the i+1*th* element is the result of applying the i*th* transaction to the i*th* element. The individual transactions then proceed with as much potential concurrency as their data interdependencies permit. In particular, non-update transactions don't lock out each other (once their initial serialization order is determined).

While the timely servicing of multiple transaction streams is fundamentally indeterminate, and hence requires a non-applicative approach, we have shown how this nonfunctionality can be contained in a relatively small portion of the overall system. This is accomplished by a pseudo-functional merge operation combining the various transaction streams, thereby serializing the transaction processing logically, *but not temporally*. Observations were also offered on pragmatic extensions to the

functional approach for control of physical system management issues, such as site selection and network interfacing.

7. Acknowledgment

We thank the referees for their comments which resulted in improvement of the presentation.. We also thank Paul Hudak, Rima Doany-Bakhit, and Mamdouh Ibrahim, Peter Badovinatz, and Frank Lin for coding help in related experimental work.

8. References

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

[2] O.P. Buneman, R.E. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM TODS* 7(2):164-186, June, 1982.

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

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

[5] 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.

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

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

[8] C.M. Hoffman and M.J. O'Donnell. Programming with equations. *TOPLAS* 4(1):83-112, January, 1982.

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

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

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

[12] R.M. Keller and G. Lindstrom. *Approaching Distributed Database Implementations through Functional Programming Concepts.* Technical Report UUCS-82-100, University of Utah, Department of Computer Science, June, 1982.

[13] R.M. Keller. FEL (Function Equation Language) Programmer's guide. 1982. AMPS Technical Memorandum No. 7, University of Utah, Department of Computer Science.

[14] R.M. Keller and F.C.H. Lin. Simulated performance of a reduction-based multiprocessor. *Computer* 17(7):70-82, July, 1984.

[15] J. Kim. Set abstraction and databases in a Function Equation Language. Master's thesis, University of Utah, August, 1983.

[16] 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.

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

[18] E.W. Myers. Efficient applicative data types. In *Proc. Eleventh ACM Conference on Principles of Programming Languages*, pages 66-75. ACM, 1983.

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

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

[21] I.L. Traiger, J.N. Gray, C.A. Galtieri, B.G. Lindsay. Transactions and consistency in distributed database systems. *Transactions on database systems* 7(3):323-342, 1982.

[22] D.A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience* 9:31-49, 1979.

[23] J.D. Ullman. *Principles of database systems, 2nd edition.* Computer Science Press, 1982.