DATA DRIVEN LOOPS

Gregory R. Ruth

Key Words: Automatic Programming
Software design
Very high level languages

This empty page was substituted for a blank page in the original document.

Contents

Part I: *Introduction*

I.1 The HIBOL Language: A Brief Introduction

The notion of the data driven loop arises in connection with our work in the Very High Level Language HIBOL and the automatic programming system (ProtoSystem I) that supports it. Although the concept is of general interest outside of VHLL's and automatic programming, we find it profitable to use HIBOL as a vehicle for our discussion and a means of narrowing the scope of our discussion. Therefore we first present a brief description of the domain which HIBOL treats.

I.1.1 Flows

The HIBOL language concerns a restricted but significant subset of all data processing applications: batch oriented systems involving the repetitive processing of indexed records from data files. It provides a concise and powerful way of dealing with data aggregates. HIBOL has a single data type, the *flow*. This construct is a (possibly named) data aggregate and represents a collection of uniform *records* that are individually and uniquely indexed by a multi-component *index*. The components of a flow's index are called *keys* and the set of an index's keys is called its *key-tuple*.[1] Each record has a single data field (*datum*) in addition to the index information. (Real-world data aggregates, such as files, with more than one datum per logical record are abstracted in HIBOL as separate flows, one for each data field.)

[1] This term is historical. A more expressive term would be "key set", but that has historically been used to indicate the universe from which a key may take its values.

1.1.2 Flow Expressions

Flow expressions can be formed through the application of arithmetic operators such as "+" or "*" to flows. The meaning of such an application to two flows is that the operation is applied to the data of corresponding records (those with matching indices) of the argument flows. The result is a new flow, having a record for each matched pair for which the operation was performed. The index value of such a record is identical to that of the matched pair, and the datum value is the result of the operation performed on the data of the pair. This concept is generalized to an arbitrary number of flow arguments.

Flow expressions can also be constructed using a conditional operator (similar to a "CASE" statement) which evaluates logical expressions in terms of corresponding flow records in order to select and then compute an expression as the individual records of the flows are processed. The logical expressions are constructed using the arithmetic comparison operators ">", "=", and "<". In addition the PRESENT operator may be used to test the presence of a record in a flow for a given value of the index of that flow. These may be composed using the logical connectives "AND", "OR" and "NOT".

Finally, there is a class of reduction operators permitted on flows and flow expressions. The function of such an operator is to reduce a flow with an n-key index to one with an m-key index, where m < n, and the key-tuple of the m-key index is a subset of the key-tuple of the n-key index. All records of the argument flow that correspond to a single record of the result form a set to which a reduction operator (e.g. "maximum", "sum") can be applied to obtain a single value.

I.1.3 Flow Equations

Relationships between flows are are expressed by *flow equations* of the form:

`<flow-name> IS <flow-expression>`

where `<flow-name>` is a named flow and `<flow-expression>` is a flow expression in terms of named flows. The right- and left-hand sides must have identical indices.

I.1.4 Example

Consider a chain of stores whose items are supplied from a central warehouse. The collection of store orders for item restocking on a given day can be thought of as a flow called, say, CURRENTORDER. A record of that flow contains the quantity ordered by a particular store of a particular item. Each record has as its datum the quantity ordered and a 2-component index identifying the store making the order and the item ordered (the keys of the index are a store-id and an item-id). Let BACKORDER be the name of a flow (of similar structure) representing the collection of (quantities of) previous orders that could either not be filled or filled only partially. The HIBOL statement

```
DEMAND IS CURRENTORDER + BACKORDER
```

describes a new flow DEMAND representing the total demand of each item by each store. That is, each record in DEMAND contains a 2-component (item-id, store-id) index identifying its datum which is the sum of the data for the same item and store in the CURRENTORDER and BACKORDER flows.

The HIBOL statement

```
ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID
```

illustrates the use of the reduction operator SUM. It describes a new flow ITEMDEMAND representing the total demand of each item from *all* stores. That is, each of its records has a single-component index (item-id) identifying a particular item; and its datum is the total quantity in demand summed across all stores in the chain.

I.1.5 Additional Information

The computational part of a data processing system can be described by giving a full set of flow equations of the type shown above. To complete the system's description additional data and timing information must be given:

- for each flow, the components of its index, the type of its data value, and the periodicity with which it is computed

- for each key its type

- for each period its time relation to other periods

I.2 Iteration Sets and Explicit HIBOL

A flow expression, as explained above, represents a set of records obtained by the record-by-record application of a formula to the records of the flows that appear as terms in the expression. In this paper we shall be interested in exactly for which index values (and thus records) the indicated formula is applied. The set of these index values is termed the *iteration set*[2]

The HIBOL language is rather informal about specifying iteration sets. It contains abundant provisions (through the use of defaults) for implicit semantics based on the presence or absence of records in the flows appearing in flow expressions. For example, the HIBOL flow expression

```
CURRENTORDER + BACKORDER
```

[2] After Baron [1]

describes a flow that has a record for each index value for which either CURRENTORDER or BACKORDER (or both) has a record:

if both flows have a record for a given index value, the resultant flow has a record with the same index value, whose datum is the sum of those of the corresponding records in the two flows;

if only one flow has a record for a given index value, the resultant flow has a record with the same index value and the same datum value;

otherwise there is no record in the resultant flow.

One way of looking at the semantics of addition in HIBOL, then, is to convene that the operation + is performed if and only if at least one of its operands is present and that each missing operand is treated as if it were the additive identity (0).

Although such conventions are convenient in writing HIBOL, for the sakes of clarity and rigor, we require fully explicit iteration set specifications. Such can be obtained through the thorough use of the HIBOL primitives IF and PRESENT. Thus, the fully explicit form of the above HIBOL flow expression would be:

```
      CURRENTORDER + BACKORDER   IF     CURRENTORDER PRESENT
                                    AND BACKORDER    PRESENT

ELSE  CURRENTORDER               IF     CURRENTORDER PRESENT

ELSE  BACKORDER                  IF     BACKORDER    PRESENT
```

Here the index values for which the flow expression's formula is to be applied have been made explicit by restructuring it as a three-clause conditional expression in terms of three sub-expressions, each of whose iteration sets is specified by an associated condition on the presence of records in the flows involved. This is a legal HIBOL flow expression, although in view of the existing conventions it is overspecified (redundant). For our purposes we will distinguish a language, called Fully Explicit HIBOL (FE-HIBOL) whose legal utterances are the subset of the legal utterances of HIBOL in which the iteration set of each flow expression is fully and explicitly specified.[3]

In general what this means is that no flow expression in FE-HIBOL is legal unless it is (at least) qualified by the condition that all of the flows involved are present. Thus, the following are not legal flow expressions in FE-HIBOL:

(1) A

(2) A/B IF B PRESENT

(3) A+B IF A > 40

Their correct versions would be:

(1) A IF A PRESENT

(2) A/B IF A PRESENT AND B PRESENT

(3) A+B IF A PRESENT AND B PRESENT AND A > 40

Throughout the rest of this paper, unless explicitly stated otherwise, all HIBOL expressions will be written in FE-HIBOL.

1.3 Implementation from a HIBOL Description

The implementation of a HIBOL description of a data processing system involves three basic stages:

Static Analysis:

The HIBOL description must be understood in data processing terms. The HIBOL

---

[3] In ProtoSystem I, in fact, immediately after a HIBOL data processing system is described it is translated into an internal language (DSSL) which has exactly this requirement.

description is *declarative* in nature: it describes the relationships among the flows. An implemented data processing system is *procedural* in nature: it must describe in detail how the flows are computed. The flow equations must be reinterpreted as basic computation steps (with an output flow and one or more flows as inputs) and constraints on the order in which these computations can be performed (the computation producing a flow must be performed before any computations using that flow) must be made explicit.

Design:[4]

The implementation will make use of files of data to be processed by job steps which will in turn create other files. Each file will contain the information represented by one or more flows; each job step will perform the processing to satisfy one or more flow equations. The design of each file (information contained, organization, storage device, record sort order) and of each job step (equations implemented, loop structure, accessing methods used) should be made in such a way as to minimize some overall cost measure (e.g. dollars-and-cents cost, time used, number of secondary storage I/O events) for the execution of the data processing system. Typically this requires dynamic (behavioral) analysis of tentative design configurations.

Code Generation:

The system's design must be coded in a supported high-level language so that it can be executed.

1.4 Data Driven Loops

Each flow equation represents a *computation* whose implementation is essentially iterative in

[4] In ProtoSystem I the design process is performed by the Optimizing Designer module.

nature. That is, the evaluation of the relation described will entail iteration over the records of the flow(s) in the flow expression on the right-hand side, performing the indicated operations in order to generate the records of the flow on the left-hand side. In traditional high-level languages this iteration is effected through the use of a loop statement.[5]

A single loop (possibly with nested loops) is used to implement a flow equation. That is, a loop can be devised that will produce the entire flow appearing on the left-hand side of the flow equation from the flows appearing on the right-hand side. In functional terms, the flows on the right-hand side of the flow equation are called the inputs to the corresponding loop and the flow appearing on the left-hand side is called the output of the loop.

The body of the loop specifies the computation, for a given index value, of a record of the output that is to be generated, if any. For example, consider the (fairly explicit) HIBOL flow equation:

```
S = A + B   IF A PRESENT AND B PRESENT
    ELSE A  IF A PRESENT
```

The body of the corresponding loop will distinguish two cases and corresponding courses of action:

1. Records in both A and B are present for the current value of the loop index, in which case a corresponding record of the flow S is produced whose datum is the sum of those in the records of A and B.

2. Only A contains a record corresponding to the current value of the index, in which case a corresponding record of S is produced that is identical to A's record.

If neither of these cases obtains, no output record is produced.

Clearly, in a correct implementation the body of the loop must be performed for every index value for which records of the input(s) exist that will be used to produce an output record. We call

[5] APL is an exception, and for this reason (among others) it is often considered a VHLL.

any set of values for a particular index an *index set* and we distinguish two special kinds of index sets:[6]

The set of index values for which a flow F contains a record is called the *index set* of F (denoted IS(F)).

The set of index values for which an input flow F_i contains a record that will be used in generating a record of the output flow F the *critical index set of* F_i *with respect to* F (denoted $CIS_F(F_i)$).

These two should not be confused. $CIS_F(F_i)$ for some flow F will often be a proper subset of $IS(F_i)$.[7]

The problem we face is that of finding some way of enumerating the critical index sets of each input so that loop can be properly driven.[8] It is generally impractical to use the set of all possible (legal) index values for which an input might have a record. For one thing this set may be unbounded. Even if it is finite and enumerable, it will often be much larger than the critical index set and thus grossly inefficient. In the DEMAND flow equation example given above, for instance, the critical index set of the input flow CURRENTORDER is likely to be orders of magnitude smaller than its maximum possible size (the case where every store has orders for every item).

A much more efficient way of enumerating a set of index values that is assured to cover the critical index sets of the inputs is to use the union of the index sets of the input flows. This will work because a record of the output can be produced *only* if there is some input flow in which that

[6] Unfortunately, this terminology is at variance with that used by Baron in his thesis [1]. Baron uses the term "critical index set" to mean what we call the "index set".

[7] On no account, of course, can it be other than a proper or improper subset of $IS(F_i)$.

[8] This statement is somewhat oversimplified, but it will suffice for now. A fully precise statement of the problem is given by the Fundamental Driving Constraint in Part IV.

record is present. Moreover, the union of these sets is [illegible] simply by reading the input flows (which have to be read anyway). A loop that is thus driven by index values supplied by its inputs is said to be a *data driven loop* and the flows used to drive the loop are termed its *driving flows* (the set of flows that drive a loop are called its *driving flow set*). The structure and implementation of data driven loops is the subject of this paper.

It will not always be necessary to read all of inputs of a loop to cover their critical index sets. From the flow equation for S above it can be deduced that the index set of A alone is sufficient to cover $CIS_S(A)$ and $CIS_S(B)$. This is so because [illegible] a record of B will be needed only if there is corresponding record in A. Therefore, the loop can be driven by A alone; that is, it is sufficient to perform the body of the loop only for [illegible] for which A has a record. In general, for the sake of efficiency it is of interest to find [illegible] subset of subsets of the inputs that will be sufficient to drive the loop.

Part II: *Structure of Data Driven Loops*

Before a general treatment of data driven loops can be developed it is necessary to examine the structures of the loops encountered in the HIBOL system. We begin by presenting a taxonomy of computation types and their corresponding loop implementations.

II.1 Loop Terminology

Before discussing loop structures it is useful to establish some terminology. By the term *loop* we mean a control construct which somehow enumerates a set of values for a *loop-index* and which performs a fixed sequence of statements (its *body*), once for each value of the loop-index. A loop may contain one or more loops within its body. The inner loops are said to be *nested* within the outer (enclosing) loop and the structure as a whole is called a *nested loop structure*. Each enclosure defines a different *level* of the nested loop structure. The degenerate case of a nested loop structure, where there is no loop in the body of the outer loop, is called a *single-level loop*, since there is only one loop level.

A *totally nested loop* is a nested loop structure whose component loops are totally ordered under enclosure (i.e. for any two loops L_1 and L_2 either L_1 is inside L_2 or L_2 is inside L_1).

II.2 Kinds of Computations and Their Loops

Each run (computation, job step, program) in the implementation produced for a HIBOL description of a data processing system is essentially a loop that iterates over the records of its input files to generate records of its output file(s). The structure of this loop depends on the nature of the computation being performed. We will begin with computations that directly implement single HIBOL flow equations of various types. Then we will consider computations that implement more than one flow equation (aggregated computations) simultaneously.

II.2.1 Simple Computations

A simple computation computes a non-aggregate flow equation involving a single flow. Thus, it takes one flow as input and produces one flow as output. An example of such a computation is described by the HIBOL flow equation

```
PAY IS   HOURS * 3.00                    IF HOURS PRESENT AND
                                            NOT HOURS > 40
         ELSE 120 + (HOURS - 40) * 4.50  IF HOURS PRESENT
```

This assumes that there is a flow HOURS indexed by (employee#) each of whose records contains as its datum the number of hours the indicated employee worked. Each employee is paid $3.00/hour, with time-and-a-half for overtime. PAY is a flow containing the gross pay for each employee who worked (i.e. had a corresponding record in the HOURS flow).

The loop implementing this kind of computation is of the simplest kind: it is a single-level loop having a single input file which is the driver. On each iteration the next sequential record of the input is read and used to provide both a datum and index value for the body, which calculates the datum of the corresponding output record, assembles the record from this datum and the current index value, and outputs it.

In the SEAL language (see Appendix I) this would look like:

```
for each (employee-id) from HOURS

   get HOURS(employee-id)

   PAY(employee-id) =

      if defined[HOURS(employee-id)]
         and not(HOURS(employee-id) > 40)

         then HOURS(employee-id) * 3.0

         else if defined[HOURS(employee-id)]

                 then 120.0 + (HOURS(employee-id) - 40) * 4.5

                 else undefined

   if defined[PAY(employee-id)]
      then write PAY(employee-id)

end
```

The for-end construct represents the basic iteration over values of the index employee-id. It specifies that the values for the index are obtained from the HOURS flow. For each index value, the corresponding record of HOURS is read, the corresponding record of PAY is generated, and (if generation was successful) that record is written out. Notice that the PAY calculation is a direct translation from the HIBOL flow equation.

For reasons of exposition the loop implementation presented here is of the most general form. An actual implementation would incorporate various efficiency enhancing improvements.[9] Nevertheless, we shall continue to use such forms to show explicitly where I/O and testing occur *conceptually*.

[9] For instance, since the for has to read the next record of the driver to get the current index value, the get could be omitted. Furthermore, the defined tests in the PAY calculation could be omitted since they are testing the presence of record which must be present. Finally, in this computation, the check before output could also be omitted.

II.2.2 Matching Computations

A matching computation computes a non-reduction flow expression involving two or more flows. Thus it is similar to a simple computation, but instead of operating on a single record of a single input flow to produce an output record, it operates on a set of corresponding records, one from each input flow. Correspondence is established by common index values. The name "matching computations" derives from the necessity of matching up the records of the inputs by index values before they can be operated on.

Two sub-classes of matching computations can be distinguished depending on whether all of the inputs have indices with identical key-tuples or not.

II.2.2.1 Expressions Involving Flows with a Uniform Index

Consider the a pay calculation similar to that given above, but where employees are paid various hourly rates. Let RATE be a flow, indexed by (employee-id), each of whose records has as its datum the hourly pay rate for the employee indicated by its index value. The pay calculation then becomes

```
PAY IS   HOURS * RATE                 IF      HOURS PRESENT
                                         AND RATE PRESENT
                                         AND NOT HOURS > 40

        ELSE
          RATE * 40 +
           (HOURS - 40) * 1.5 * RATE  IF      HOURS PRESENT
                                         AND RATE PRESENT
```

HOURS and RATE have identical indices, each consisting of the single key "employee-id". The loop that implements such a computation has a single level.

Because a record of the output is generated only if there is a record in the HOURS file, that file alone is sufficient to drive the loop. (Alternatively, by similar reasoning, the RATE file could be used to drive the loop.) This is the simplest case of a matching computation because only one input is needed to drive the loop. (The computation of the flow S above is also of this type.) On each iteration the next record of the HOURS file is read, the corresponding RATE record is fetched, and the computation of gross pay performed.

This loop is represented in the SEAL language thus:

```
for each (employee-id) from HOURS

    get HOURS(employee-id)

    get RATE(employee-id)

    PAY(employee-id) =

        if defined[HOURS(employee-id)]
           and defined[RATE(employee-id)]
           and not(HOURS(employee-id) > 40)

           then HOURS(employee-id) * RATE(employee-id)

           else if defined[HOURS(employee-id)]
                   and defined[RATE(employee-id)]

                   then RATE(employee-id) * 40 +
                        (HOURS(employee) - 40) * RATE(employee-id) * 1.5

                   else undefined

    if defined[PAY(employee-id)]
       then write PAY(employee-id)

end
```

Again, the defined checks on the driver, HOURS, are superfluous. But those on RATE are necessary (to determine whether the corresponding get was successful) and the defined check on PAY is necessary (so that a record is written if and only if a datum was generated).

Now consider the HIBOL flow equation for the DEMAND flow given above:

```
DEMAND IS CURRENTORDER + BACKORDER   IF  CURRENTORDER PRESENT
                                     AND BACKORDER    PRESENT
     ELSE CURRENTORDER               IF  CURRENTORDER PRESENT
     ELSE BACKORDER                  IF  BACKORDER    PRESENT
```

Again, CURRENTORDER and BACKORDER have identical indices, both consisting of the component keys item-id and store-id, so again the implementing loop has a single level. In this case both inputs are necessary to drive the loop.

When there is more than one driver for a computation some of the drivers will have records for a given index while the others do not.[10] If the drivers have their records sorted in the same order (say, alphabetically) by index values, the loop may be performed by making only one pass through the inputs in the following manner:[11]

0. Read the first record of each input.

1. Use the smallest index value among the current driver records as the loop index and perform the loop body, fetching other non-driver records if necessary.

2. Discard all driver records with the index value just used and read the next record of every driver whose record was discarded.

3. Repeat from 1.

If the only sorting constraint on the inputs is that the drivers be sorted in the same order (i.e. no constraints on the order of the records in the other inputs), the records of each non-driving input will have to be fetched by random access (if the organization of the file allows random access) or by search; but if any non-driving input is sorted in the same order as the drivers, its records can be fetched by sequential reading, which is generally more efficient.

---

[10] If this were not the case [illegible] would be superfluous.

[11] If they are not similarly sorted, more inefficient means must be used.

These details are implicit in the SEAL representation of the loop which is simply:

```
for each (item-id, store-id) from CURRENTORDER, BACKORDER

   get CURRENTORDER(item-id, store-id)

   get BACKORDER(item-id, store-id)

   DEMAND(item-id, store-id) = ...

   if defined[DEMAND(item-id, store-id)]
      then write DEMAND(item-id, store-id)

end
```

II.2.2.2 General Discussion of Expressions Involving Flows with Mixed Indices

The treatment of mixed-index flow expressions in this paper will be restricted to those that are legal in HIBOL. The restrictions that HIBOL imposes are made for good reasons. A brief discussion of the various conceivable types of mixed-index flow expressions is presented here in order to show the motivation behind these restrictions.

The various cases where the flows in a flow expression have mixed indices (i.e. their indices have different key-tuples) can be distinguished by the set interrelationships among the key-tuples.

Consider the case where flows have disjoint key-tuples (e.g. (w, x) and (y, z)). Correspondence among records of such flows is meaningless, so we do not allow them to appear in the same flow expression.

Now consider the more general case where there is intersection among index key-tuples, but *the union of their pair-wise intersections is not identical to their (simple) union.* In this case correspondence is always ambiguous. For example, consider the two flows: A with index (x, y) and B with index (y, z). Suppose that there are records in A for the particular index values (x_1, y_1) and

(x_2, y_1) and that there are records on B for index values (y_1, z_1), (y_1, z_2) and (y_1, z_3). Which of A's records correspond to which of B's records?[12]

For correspondence to be meaningful and unambiguous it must be the case that the union of the pair-wise intersections of the key-tuples of the indices involved is identical to their union. This is always the case when there exists an index among the flows involved whose key-tuple is a superset of all the key-tuples of the other flows.

To be sure, there are other ways of satisfying the condition of the preceding paragraph. These involve conjunctions of three or more indices. Consider, for instance, the three flows: A with index (x, y); B with index (y, z); and C with index (x, z). Corresponding triplets are all unique and unambiguous, of the form (x_i, y_j), (y_j, z_k), (x_i, z_k). For the sake of simplicity, however, this case is prohibited in HIBOL.

II.2.2.3 Mixed-Index Flow Expressions Allowed in HIBOL

It is possible in HIBOL to apply operators to two or more flows having different indices as long as each index is a sub-index of the index of some unique flow involved (i.e. as long as the key-tuple of each index is a subset of the key-tuple of the index of the unique flow). Clearly, the index of this unique flow is identical to the index of the flow expression as a whole. HIBOL allows a mixed-index flow expression only if its computation can be driven by the set of those flows involved having indices identical to that of the flow expression.

[12] Of course, we could allow *all* pairs to match (in Cartesian product fashion) so that the expression A + B would represent the six possible combinations of additions for these 5 index values; but this would change (extend) the semantics of HIBOL.

For example, suppose we want to calculate the extended prices[13] of the current store orders (the flow CURRENTORDER) in our store chain example. Let PRICE be a flow indexed by (item-id), each of whose records has as its datum the per-item price associated with the item identified by its index. The flow equation for EXTENDEDPRICE, indexed by (item-id, store-id) would be expressed in HIBOL thus:

```
EXTENDEDPRICE  IS  CURRENTORDER * PRICE  IF     CURRENTORDER PRESENT
                                           AND PRICE        PRESENT
```

The intent here is: for every record in CURRENTORDER find the corresponding record in PRICE and, if the latter is present, multiply their respective data to calculate the datum of a corresponding record in EXTENDEDPRICE. Notice that because PRICE and CURRENTORDER have different indices ((item-id) and (item-id, store-id), respectively) the notion of correspondence must be extended in a natural way from pure identity of index values. We convene that for a particular value of item-id the index (item-id) matches any index (item-id, store-id) with the same value of item-id, regardless of the value of store-id. This augmented definition of correspondence is extended to the general case where the key-tuple of one index is a subset of the key-tuple of another. That is, for given values of $k_1, ..., k_m$ the index $(k_1, ..., k_m)$ is said to match any instance of an index $(k_1, ..., k_m, k_{m+1}, ..., k_n)$ with the same values of $k_1, ..., k_m$, regardless of the values of $k_{m+1}, ..., k_n$.

Since a set of input flows, each with index identical to the flow expression's, can be used to drive a mixed-index matching computation, its implementation is similar to that for a uniform-index matching computation: the sorted drivers are read in such a way as to enumerate the critical index sets of all of the input flows; the resulting index values are used to fetch records from the rest of the inputs (including all those whose indices are sub-indices of the flow expression's index).

[13] The extended price of a quantity ordered is the product of the quantity and the per-item price.

The general form SEAL implementation is a single loop:

```
for each <index> from [illegible]
    get <input>1
    get <input>2
    .
    .
    .
    get <input>n
    output(<index>) ← ...
    if defined[output(<index>)]
    then write output(<index>)
```

As with uniform-index computations, sequential access can be used in fetching the records of the non-drivers if they are sorted in the same order as the drivers are.

Additional I/O economies can be realized in fetching a flow whose index is a sub-index of the flow expression index if the additional constraint is made that all drivers have their records sorted by its sub-index.

Consider the EXTENDEDPRICE computation. It is driven by the flow CURRENTORDER, which has (item-id, store-id) as its index. The index of the other input, PRICE, is just (item-id). If the records of both CURRENTORDER are sorted by item-id, then all of the CURRENTORDER records for a given value of item-id can be processed in sequence with only one fetch from PRICE. On the other hand, if the records of CURRENTORDER are not sorted by item-id, [illegible], the processing of each of its records will require a fresh fetch from PRICE; that is, a particular record of PRICE is liable to be read more than once.

In the case where the records of CURRENTORDER are sorted by item-id, the EXTENDEDPRICE computation is implemented as a nested loop structure with two loops (one inside the other).

Basically, the outer loop chooses a value of the sub-index (item-id) and fetches the corresponding PRICE record. Then it performs the inner loop. Within the inner loop the value of the item-id key is held constant. All corresponding records of CURRENTORDER are read and the computation described in the flow equation is performed using the data of these records together with the datum of the PRICE record fetched in the outer loop. The results are used to build and output the corresponding records of EXTENDEDPRICE. This process is repeated until the flows are exhausted.

In detail the implementation is as follows. Before either loop is entered a record of CURRENTORDER is read. The outer loop uses this record to obtain the first value of the sub-index (item-id) and fetches the corresponding record from PRICE. Then it performs the inner loop. The inner loop uses the current record of CURRENTORDER and continues to read records *sequentially* from CURRENTORDER until the sub-index is observed to change or an end-of-file condition occurs. When either of these conditions occurs, it exits to the outer loop. If an eof has occurred, the outer loop exits. Otherwise it iterates, using the sub-index value of the current CURRENTORDER record as the new value to be held constant in the inner loop, fetching the corresponding PRICE record and performing the inner loop again.

The corresponding SEAL code is:

```
for each (item-id) from CURRENTORDER
   get PRICE(item-id)
   for each (store-id) from CURRENTORDER(item-id)
      get CURRENTORDER(item-id, store-id)
      EXTENDEDPRICE(item-id, store-id) =
         if defined[CURRENTORDER(item-id, store-id)]
            and defined[PRICE(item-id)]
            then CURRENTORDER(item-id, store-id) * PRICE(item-id)
            else undefined
      if defined[EXTENDEDPRICE(item-id, store-id)]
         then write EXTENDEDPRICE(item-id, store-id)
```

Notice that the outer loop is driven by CURRENTORDER (the whole flow), but that the inner loop is driven by CURRENTORDER(item-id) (the sub-flow of CURRENTORDER consisting of just those records whose indices correspond to the value of the sub-index (item-id) fixed by the outer loop). What this means is that for the outer loop the next value of the sub-index (item-id) will be taken from the next record of the CURRENTORDER flow. But for the inner loop the next value for the sub-index (store-id) will be taken from the next record of the sub-flow of CURRENTORDER corresponding to the current value of (item-id); if there are no further records in CURRENTORDER for this fixed value of (item-id) this will be treated just like an end-of-file condition and the iteration of the inner loop will terminate. Thus the inner loop is driven by a succession of sub-flows, one for each iteration of the outer loop.

This nested-loop implementation scheme is easily extended to 3 or more loop levels when appropriate sorting constraints hold among the flows involved. For example, suppose that there are 3 flows involved: A with index (k_1, k_2, k_3); B with index (k_1, k_2); and C with index (k_1). And suppose further that B is sorted by k_1 and that A is sorted first by k_1 and, *within segments corresponding to a fixed value of* k_1, the records of A are further sorted by k_2. Then the flow equation can be implemented using a nested loop structure involving 3 loops (innermost loop, middle loop and outermost loop). The outermost loop chooses a value for the key k_1 to be held constant within the middle loop (and perforce in the innermost loop, which is contained in the middle loop). It also fetches the corresponding record of C for use within the contained loops. Then it executes the middle loop, which, in turn, choose a value for the key k_2 to be held constant within the inner loop. The middle loop also fetches the corresponding record of B for use within the innermost loop. Then it executes the innermost loop. In the innermost loop the values of the keys k_1 and k_2 are held constant. The innermost loop reads all corresponding records of A, using their data and those of the already read records to perform the calculations described in the flow equation and to build and output the records of the output flow. When the innermost loop has read and processed all records of A corresponding to the fixed values of k_1 and k_2, it exits to the middle loop, which chooses a new value for k_2 and iterates. When the middle loop has exhausted all possibilities for the value of k_1 fixed in it, it returns to the outermost loop, which chooses a new value of k_1 and iterates. This loop structure expressed in the SEAL language looks like:

II.23 Simple Reduction Computations

A simple reduction computation computes a reduction flow expression involving a single flow. A reduction flow expression is the application of a reduction operator to an arithmetic flow expression. Often the flow expression to which the reduction operator is applied is simply a single flow itself, as in the previously cited example:

```
ITEMDEMAND   IS   THE SUM OF DEMAND   FOR EACH ITEM-ID
```

where SUM is applied to DEMAND. However, constructions such as

```
ITEMDEMAND   IS   THE SUM OF  CURRENTORDER + BACKORDERS
                  FOR EACH ITEM-ID
```

(written in HIBOL[14]) are also possible. In any event, the argument to the reduction operator is

[14] The standard HIBOL form is used here for clarity and conciseness; the corresponding FE-HIBOL form is rather ponderous.

treated as a single flow.

Conceptually, the argument flow is partitioned into subsets (sub-flows) by an equivalence relation defined on the sub-index (a key or keys) indicated in the FOR EACH clause; then the reduction operator is applied to the members of each subset to generate the value of the datum of the output record corresponding to that subset. For instance, in the first example given above the DEMAND flow is conceptually partitioned into record subsets by item-id. Thus, all records in DEMAND whose index contains the value item-id_1 for the item-id key are in one subset, all records for item-id = item-id_2 are in another, and so forth (empty subsets are ignored). The datum for the record in ITEMDEMAND with index = (item-id_i) is calculated by summing all of the data in the records in the subset corresponding to item-id = item-id_i.

Conceptually, the implementing iteration for a simple reduction expression in a single flow consists of two loops, one nested inside the other. The inner loop implements the application of the indicated reduction operation to a subset of the input's records. Within this loop the value of the sub-index defining the subset is held constant. Returning to the SUM OF DEMAND example, the inner loop implements the summation of the data of the records of each subset of DEMAND. That is, the inner loop is performed for each value of item-id_i for which there are records in DEMAND. *Within* the inner loop the particular value of the key item-id is held constant, all records of DEMAND corresponding to that key value are fetched and their data are summed.

The outer loop performs clerical work. It chooses a value the subsetting sub-index (e.g. a value of item-id), executes the inner loop (which fetches records of the input corresponding to the chosen sub-index and, for example, adds them to the accumulator), and when the inner loop is finished, it uses the resulting value as the datum of the output record corresponding to the chosen sub-index, and writes that record out.

If the records within the input flow are sorted by the sub-setting sub-index, the computation can in fact be implemented using the nested loop structure described above. Before either loop is entered a record of the input is read. The outer loop uses this record to obtain the first value of the sub-index and then performs the inner loop. The inner loop reads this record and fetches records sequentially from the input until the sub-index is observed to change or an end-of-file condition occurs. When either of these conditions occurs, it exits to the outer loop, which builds and writes the output record. If an eof has occurred, the outer loop exits. Otherwise it reinitializes the accumulator and uses the sub-index value of the last record read as the new value to be held constant in the inner loop. In SEAL this implementing loop structure would look like:

```
for each (item-id) from DEMAND
    sum = undefined
    for each (store-id) from DEMAND(item-id)
        get DEMAND(item-id, store-id)
        sum = if defined(sum)
                 and defined(DEMAND(item-id, store-id))
                 then sum + DEMAND(item-id, store-id)
              else if defined(DEMAND(item-id, store-id))
                 then DEMAND(item-id, store-id)
              else undefined
    end
    if defined(sum) then begin
        ITEMDEMAND(item-id) = sum
        write ITEMDEMAND(item-id)
    end
end
```

It may at first seem unnecessarily baroque to initialize the accumulator sum to "undefined" in the outer loop, test it in the inner loop for definedness and then initialize it if undefined. In this simple example we could just initialize it to 0 in the outer loop and not bother with the definedness checks. We have chosen the former course for two reasons. First, we wish to make explicit the conditions under which the sum (and thus a record of the output ITEMDEMAND) is defined for a given value of the key item-id. Second, a little thought will show that for other reduction operations (viz. MAX and MIN) initialization of the accumulator must (at least conceptually) be postponed until the inner loop where the initializing value is obtained by the first get. Moreover, in general, when computations are aggregated (see below) and more than one activity is performed in the inner loop, it is then possible (if some driver besides DEMAND is used) that for some values of item-id no sum is calculated in the inner loop and thus sum is undefined on exit from that loop.

If the input flow is not sorted as above, the computation for a reduction operation becomes somewhat more complex. One possibility is to create and maintain separate accumulators for each value of the sub-index value occurring in the input flow. Since the number of accumulators cannot be known *a priori* (i.e. at compile time), storage for them must be allocated on the fly (during execution of the computation). In PL/I, for example, the following (roughly outlined) scheme might be used:

Declare an accumulator array to have CONTROLLED storage.

Make a pre-pass through the input flow to count the number of different sub-index values occurring.

Execute an ALLOCATE statement to define the size of the array.

Make a second pass over the input flow to perform the accumulation.

Write all accumulated values out to the output flow.

In this scheme there are two separate loops instead of a totally nested loop structure.

Alternatively, a nested loop, multi-pass scheme could be implemented. The outer loop would [illegible] through the input flow. The inner loop would then perform the accumulation for that sub-index value [illegible]

[illegible] that the input be sorted appropriately (possibly by introducing a sorting step).

II.2.4 Aggregate Computations

In certain cases, it may be advantageous [illegible]

[illegible] in the same loop [illegible]

[illegible]

In particular, if two or more computations [illegible]

[illegible]

If the output flow of one [illegible]

implementing them in the same loop in such a way that as the records of the flow are produced by the first computation they are [illegible]

[illegible] described; that is, no new loop structures are introduced.

II.3 General Loop Structure and Description

Now that we have informally discussed the kinds of data driven loops with which we shall be dealing, we will formalize the structure of such a loop [illegible] nested loop bodies.

II.3.1 Formal Representation of Nested Loop Structures

We have seen that the basic control structure used in implementing a computation is the totally nested loop. Associated with each loop in the nesting is a set of keys that it will fix and which will remain constant in the loops it contains. It is easy to see that this constraint means that the set of keys fixed within any loop is necessarily a (proper) superset of the set of keys fixed within any of its enclosing loops. Thus, the set of keys fixed within a loop is sufficient to determine its level in the nesting.

Now notice that the body of every loop (except the innermost one) contains exactly one top-level loop; thus, the body is naturally divided into three parts:

the prolog--those actions performed before the enclosed loop
the enclosed loop
the epilog--those actions performed after the enclosed loop.

Conceptually, then, a totally nested loop can be represented as a list of loop descriptions, one for each of the component loops. Each such description would consist of a level identifier (indicating at which level of nesting it occurs) and the prolog and the epilog. However, during the design stage, while implementations are being developed and, in particular, when computation aggregations are being considered, it is useful to distinguish 3 classes of actions within the body of a loop:

Prolog--those actions that *must* be performed before the enclosed loop
Epilog--those actions that *must* be performed after the enclosed loop
General--those actions that could end up in either the prolog or the epilog

It is also useful to separate I/O actions from the other actions. Thus, we represent each loop in the nesting as a structure of the following form:[15]

[15] This representation, and the theory of computation aggregation associated with it are due largely to the work of R. C. Fleischer [2], who improved on the earlier work of R. V. Baron.

```
(Level,
        (InputsP, Prolog,  OutputsP)
        (InputsG, General, OutputsG)
        (InputsE, Epilog,  OutputsE))
```

where

- Level indicates the depth of the loop in the nesting

$Inputs_P$ are the files (necessarily) read in the Prolog section.

$Inputs_G$ are the files (necessarily) read in the General section.

$Inputs_E$ are the files (necessarily) read in the Epilog section.

$Outputs_P$ are the outputs generated in the Prolog section (possibly used in the enclosed loop or in the Epilog section)

$Outputs_G$ are the outputs generated in the General section.

$Outputs_E$ are the outputs generated in the Epilog section.

II.3.2 Computation Implementation

The implementation of a computation as a nested loop structure reduces to the problem of determining how many and which levels are to be in the totally nested loop and where the I/O and computations go. The answers to these questions are constrained by the forces of necessity and efficiency.

II.3.2.1 Level Position of I/O and Calculations

The levels at which each input should be read, each output should be written and each calculation should be performed are determined by the following guidelines:

Inputs: Each input flow of a computation should be read at a loop level whose associated key-tuple is identical to that of the flow's index (and on this account the totally nested loop for a computation must contain a loop corresponding to the index of each input flow). It cannot be read a higher level because at such a level the key information is incomplete. To read it at a lower level would be inefficient, because it would cause unnecessary re-reads of the flow's records.

<u>Outputs</u>: Similarly, each output flow of a computation must be written at a loop level whose associated key-tuple is identical to that of the flow's index. It cannot be written at a higher level because of insufficient key information, and to output it at a lower level would cause multiple writes of the records.

<u>Calculations</u>: A flow expression should also be calculated at a loop level whose associated key-tuple is identical to that of the flow expression's index. Again, the key information at a higher level would be insufficient to calculate the expression, and to perform it at a lower level would be redundant. Further economy can be realized, however, in a mixed-index flow expression if it contains a sub-expression whose associated index is a sub-index of the flow expression as a whole; such a sub-expression should be split off and calculated at its appropriate (higher) level.

II.3.2.2 Position of I/O and Calculations Within Their Assigned Levels

The placement of a read, write or calculation within a given loop level (i.e. in either the Prolog, Epilog or General section) should be done with a view toward imposing the minimum constraint on implementation. If done in this manner placement preserves the maximal flexibility in subsequent aggregation. For instance, if a calculation could go into either the Prolog or the Epilog it should be placed in the General section. If instead it were arbitrarily placed in the Epilog this unnecessary constraint would preclude subsequent aggregations that would require it to be in the Prolog (loop merging in computation aggregation is discussed below).

From this consideration and the definitions of the Prolog, Epilog and General sections, the following placement guidelines emerge:

__Inputs__: An input should be included in [illegible] if and only if it is used in some enclosed loop; otherwise it should be placed in [illegible] [illegible] an input will never end up in [illegible] the [illegible] have been determined) it may be found [illegible] [illegible] to release its input buffer. However, before [illegible] placing an input in [illegible] would be an unnecessary constraint.

__Outputs__: An output should be included in [illegible] if and only if its calculation depends on something done in some enclosed loop; it should be included in [illegible] if and only if it is used in some enclosed loop; otherwise it should be placed in [illegible].

__Calculations__: A calculation should be included in the Prolog section if and only if it must be performed before the enclosed loop; it should be placed in the Epilog section if and only if it depends on something [illegible] in the enclosed loop; otherwise it should be placed in the General section.

As an obvious consequence of these guidelines it can be seen that in the case of any single-level loop or innermost loop all inputs, outputs and calculations will go into [illegible], [illegible] and the General section, respectively; [illegible], [illegible], [illegible], the Prolog section and the Epilog section will all be empty.

II.9.3 Examples

Let us illustrate the use of the loop structure representation by examining a few examples. First, consider the PAY [illegible] discussed above.

```
PAY IS RATE * HOURS IF RATE PRESENT AND HOURS PRESENT
```

Here, both inputs have the same index (employee-id) so there is only one loop:

```
Level: (employee-id)
          Inputs_P: empty
          Prolog:      empty
          Outputs_P empty

          Inputs_G: {HOURS, RATE}
          General:calculate PAY
          Outputs_G:{PAY}

          Inputs_E: empty
          Epilog:      empty
          Outputs_E empty
```

As explained above, everything is placed in the general sections.

Now consider a simple reduction flow equation:

```
ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID
```

We have seen that the implementation of such a flow equation will always have two loop levels:

<u>Loop 1 (outer loop)</u>

```
Level: (item-id)
          Inputs_P: empty
          Prolog:      initialize sum
          Outputs_P empty

          Inputs_G: empty
          General:empty
          Outputs_G empty

          Inputs_E: empty
          Epilog:      empty
          Outputs_E:{ITEMDEMAND}
```

Loop 2 (inner loop)

```
Level: (item-id, store-id)
        Inputs_A: empty
        Prolog:        empty
        Outputs_A: empty

        Inputs_G: DEMAND
        General: calculate sum
        Outputs_G: empty

        Inputs_E: empty
        Epilog:        empty
        Outputs_E: empty
```

The input DEMAND has the keys item-id and store-id in its index and so must be read in the (item-id, store-id) level loop (the inner one). Likewise, the summing calculation must be performed at that level. Since this is the innermost level, everything ends up in the general section.

On the other hand the output ITEMDEMAND has the single key item-id in its index and so must be written from the outer (item-id) level. Furthermore, because its value depends on the calculation performed in the inner loop, it must be written from the epilog of the outer level loop.

A mixed-index matching computation like:

```
EXTENDEDPRICE IS CURRENTORDER * PRICE IF CURRENTORDER PRESENT
                                      AND PRICE       PRESENT
```

must have two loop levels when implemented, one for each distinct index of its inputs. Its representation looks like:

<u>Loop 1 (outer loop)</u>

```
        Level: (item-id)
                Inputs_P: {PRICE}
                Prolog:         empty
                Outputs_P empty

                Inputs_G: empty
                General: empty
                Outputs_G empty

                Inputs_E: empty
                Epilog:         empty
                Outputs_E empty
```

<u>Loop 2 (inner loop)</u>

```
        Level: (item-id, store-id)
                Inputs_P: empty
                Prolog:         empty
                Outputs_P empty

                Inputs_G: {CURRENTORDER}
                General: calculate EXTENDEDPRICE
                Outputs_G: {EXTENDEDPRICE}

                Inputs_E: empty
                Epilog:         empty
                Outputs_E empty
```

Part III: *Computation Aggregation and Loop Merging*

As explained above the aggregation of two or more computations must result in a single totally nested loop structure. The process of computation aggregation can be performed most simply on two loops at a time (thus if it is desired to aggregate three computations, the first two are aggregated and then the result is aggregated with the third). Without loss of generality, then, we will confine the treatment that follows to pair-wise aggregation.

When two computations are found to be candidates for aggregation, first their suitability for aggregation must be tested, and then, if they are aggregatable, their respective totally nested loops must be merged to form a single totally nested loop that [illegible] both. These two problems are the subjects of the next sections.

III.1 Loop Aggregatability

A little thought will show that when two nested loops are aggregated each action (read, write or calculation) in the aggregate must be performed at the same loop level as before aggregation; there is no possibility of moving an action to a different level of nesting than where it originally appeared. Thus, for two loops to be aggregatable it must be possible to construct a totally nested loop structure that contains all of the levels necessary to both. Two loops for which this is possible are said to be *level compatible* with each other.

Furthermore, there are certain ordering constraints that the actions of the individual loops satisfy and which must be satisfied by the aggregate loop: a files must be produced before it can be used; a Prolog action must occur before its associated inner loop; and an Epilog action must occur after its inner loop.

If two computations have level compatible loops and if the ordering constraints of the two loops can be mutually satisfied in a single totally nested loop, aggregation is possible.

III.1.1 Level Compatibility Between Loops

It is easy to show that two loops are level compatible if and only if their level structures are identical or empty levels (levels at which no actions are performed) can be inserted to make their level structures identical. Some examples of level compatible totally nested loops (TNL's) and the level structures of their aggregated results are:[16]

loop	levels	levels in aggregate
TNL_1	(K), (K,L)	(K), (K,L)
TNL_2	(K,L)	
TNL_1	(K,L)	(K,L), (K,L,M)
TNL_2	(K,L,M)	
TNL_1	(K), (K,L)	(K), (K,L), (K,L,M)
TNL_2	(K,L), (K,L,M)	

It is interesting to note that when aggregation occurs loop levels are neither added nor deleted; that is, the set of loop levels in the aggregate is simply the union of the sets of loop levels in the component computations.

Some examples of loops whose level structures are incompatible are:

loop	levels
TNL_1	(K)
TNL_2	(L)

[16] In this section the symbols K, L and M denote different keys.

```
TNL₁    (K),  (K,L)
TNL₂    (L),  (K,L)

TNL₁    (K),  (K,L),  (K,L,M)
TNL₂    (K),  (K,M),  (K,L,M)
```

III.1.2 Order Constraint Compatibility Between Loops

Consider the computations for the following two flow equations:

ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID

FRACTION IS DEMAND/ITEMDEMAND IF DEMAND PRESENT

It would seem immanently reasonable to aggregate these two computations since they have a common input (DEMAND) and the output of the first is an input to the second. Yet they cannot be aggregated into a totally nested loop! Their implementation descriptions reveal why. Recall that the description of the first is:

Loop I (outer loop)

```
Level: (item-id)
        Inputs_P: empty
        Prolog:       initialize sum
        Outputs_P: empty

        Inputs_G: empty
        General: empty
        Outputs_G: empty

        Inputs_E: empty
        Epilog:       empty
        Outputs_E: (ITEMDEMAND)
```

Loop 2 (inner loop)

```
        Level: (item-id, store-id)
                Inputs_P: empty
                Prolog:       empty
                Outputs_P empty

                Inputs_G: {DEMAND}
                General:calculate sum
                Outputs_G empty

                Inputs_E: empty
                Epilog:       empty
                Outputs_E empty
```

The FRACTION computation also has two nested loops:

Loop 1 (outer loop)

```
        Level: (item-id)
                Inputs_P: {DEMAND}
                Prolog:       empty
                Outputs_P empty

                Inputs_G: empty
                General:empty
                Outputs_G empty

                Inputs_E: empty
                Epilog:       empty
                Outputs_E empty
```

Loop 2 (inner loop)

```
        Level: (item-id, store-id)
                Inputs_P: empty
                Prolog:       empty
                Outputs_P empty

                Inputs_G: {DEMAND}
                General:do division
                Outputs_G {FRACTION}

                Inputs_E: empty
                Epilog:       empty
                Outputs_E empty
```

Clearly these computations are level compatible since they have identical level structures. But the (item-id) level loop of the first requires that ITEMDEMAND be an output of the epilog, whereas the (item-id) level loop of the second requires that it be an input in the prolog. Their aggregate would thus require records of ITEMDEMAND before they are computed, an obviously impossible condition.

The basis for all ordering constraints is the simple dictum that *data must be produced or read before it is used*. Totally nested loop implementations are structured in such a way that this rule is observed exactly. That is, things are in a loop's Prolog if and only if they must be done before the enclosed loop(s); things are in a Epilog if and only if they must be performed after the enclosed loop. Listed explicitly, the constraints that must hold for every loop level when merging two totally nested loops are:

- an output of an Epilog cannot be an input to a Prolog
- every Prolog action must remain in the Prolog
- every Epilog action must remain in the Epilog
- Prolog I/O must remain in the Prolog
- Epilog I/O must remain in the Epilog

Implicit are the constraints that

- an action cannot be moved from its original level to another in the aggregate
- I/O cannot be moved from its original level to another in the aggregate

The only thing that can change is that actions and their corresponding I/O can be moved from the General action to either the Prolog or Epilog of the same loop level. Such a move merely reflects the addition of a constraint that does not alter the validity of the implementation. Thus, such a move may be made when necessary to mutually satisfy the constraints of the two loops to be merged, but should never be done arbitrarily, so as to preserve maximum freedom in subsequent mergers.

Computations whose totally nested loops are level compatible and satisfy the above order constraints are aggregatable.

III.2 Merging Loops

Because each action and all I/O must be performed at the same level in the aggregate as it was before aggregation, the loop structure of the aggregation of two computations can be obtained through a level-by-level merge of the loop levels of the two computations to be aggregated.

The algorithm for merging two totally nested loops is:

For each loop in one:

> If the other has no loop at the the same level, just add the representation of that level to the description of the aggregate.
>
> If there is a corresponding loop, the two loops must be merged into one for the aggregate.

The full details of merging loops are complicated, but a rough sketch follows. Let the corresponding loops be L_1 and L_2, where no output of L_2 is an input to L_1.[17] There are three cases:

I. Some output F of the Epilog of L_1 is an input to L_2.

> a. F is an input to L_2's Prolog section: aggregation impossible.
>
> b. F is used by an action in L_2's General section: move that action to the Epilog of the the corresponding level in the aggregate, along with any actions in L_2's General section which use, as input, some output produced by the action; all other actions remain in the same sections in the aggregate as they were in L_1 and L_2.
>
> c. All other cases: all other actions remain in the same sections in the aggregate as they were in L_1 and L_2.

---

[17] Obviously, the case where no output of L_1 is an input to l_2 will be handled exactly the same, *mutatis mutandis*. The remain case, where each has some output that is an input to the other, is impossible.

2. Some output F, generated by some action A in the General section of L_1, is an input to L_2.

a. F is an input to L_2's Prolog section: move A from the General section to the Prolog section of the aggregate, along with any actions in the General section which have, as output, something used as input to that computation; all other actions remain in the same sections in the aggregate as they were in L_1 and L_2.

b. All other cases: all actions remain in the same sections in the aggregate as they were in L_1 and L_2.

3. Neither 1 nor 2: all actions remain in the same sections in the aggregate as they were in L_1 and L_2.

Basically, what this means is that a General action must move to the Prolog of the aggregate if it must come before some action in that Prolog or if it must come before another General action which must be moved to the Prolog; a General action must move to the Epilog if it must come after some action in the Epilog or if it must come after another General action which must be moved to the Epilog.

III.3 Non-Totally-Nested Loops

In this report the treatment of data driven loop implementations is restricted to loop structures that are totally nested. Totally nested implementations are not only broadly applicable, but generally simple and efficient as well. In fact they often provide the *most* efficient and expeditious implementations, especially when sequentially organized files, sorted by key values, are used. For the sake of completeness, though, something should be said here about non-totally-nested loops. Indeed, a great deal could be said about such implementations—enough, certainly, to make one or more separate reports. Because of this the discussion here is necessarily brief and incomplete.

Most importantly, it should be said that non-totally-nested loop structures are by no means inefficient or uninteresting. They are used all the time and for good, solid reasons. Their use is perhaps most interesting when two or more computations cannot be performed entirely concurrently (i.e. in the same loop), but they *can* be performed with partial concurrency. The following two examples illustrate.

III.3.1 Example I: Aggregating Computations with Incompatible Order Constraints

Recall the flow equations:

```
ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID

FRACTION IS DEMAND/ITEMDEMAND IF     DEMAND      PRESENT
                                 AND ITEMDEMAND PRESENT
```

and their implementing computations. We saw in Section III.1.2 that the implementing computations for these flow equations could not be merged into a totally nested loop structure because the inner loop for the first had to be completed before the inner loop of the second could be performed. They can, however, be aggregated into a single loop with a structure like:

```
for each (item-id) from DEMAND

    sum = undefined

    for each (store-id) from DEMAND(item-id)
        <calculate sum>
    end

    if defined[sum] then ITEMDEMAND(item-id) = sum

    for each (store-id) from DEMAND(item-id)
        <calculate FRACTION>
    end

end
```

This is a non-totally nested loop structure, since two loops (the inner ones) appear at the same level.

It is interesting to compare this aggregate implementation with the unaggregated implementation of the two computations involved (as separate loops in separate job steps). On the one hand, in either implementation every record of the DEMAND flow must be accessed twice, so no accesses are eliminated by aggregation. On the other hand, accesses of the records of the ITEMDEMAND flow are eliminated by aggregation. If the computations are implemented separately, every record of ITEMDEMAND must be written into a file by the first computation and then read back by the second; whereas in the aggregate implementation the records are used as they are generated, so no re-reading is necessary.[18]

In general we have seen that when two implementations are level-compatible, the only case in which their aggregate cannot be implemented as a totally nested loop is where, for some loop level, the output of the Epilog section of one is an input to the Prolog section of the other (as is the case with ITEMDEMAND above). In such a case the corresponding loop level of the aggregate can be implemented (as above) as two loops of the same level performed in sequence, and re-reads of the flow in question will be saved.

III.3.2 Example 2: Aggregating Computations That Are Not Level-Compatible

In Section III.1.1 we saw that computations with the following level structures were not level compatible with one another:

TNL_1 (K), (K,L), (K,L,M)
TNL_2 (K), (K,M), (K,L,M)

The fact that they are not level-compatible means that it is impossible to devise a total

[18] In fact, if these records are not used by any other computation in the data processing system, it is not necessary to write them out into a file either.

nesting of loops that will implement their aggregate. They might, however, be said to be *partially* level-compatible, since the outermost levels have identical keys. If a common driver set can be found for that level, they might be implemented as a non-totally-nested loop structure. The following is a possible implementation skeleton:

```
for each (K) from D0

    for each (L) from D1
        for each (M) from D2
        .
        .
        end
    end

    for each (M) from D3
        for each (L) from D4
        .
        .
        end
    end

end
```

where the D_i are distinct drivers.

This is another commonly found construct in file data processing. It is the case where, for a common set of values for the sub-index (K), two or more independent computations are to be performed. As in the previous example, there is some I/O saving (over separate implementations of the computations involved) because each record of D_0 has to be read only once.

Part III Driving Flow Sets

We have seen that, by definition, a data driven loop must have a set of flows (its driving flow set) to drive it. These flows are used to effectively enumerate a set of values for the loop index. The body of the loop is performed once for each [illegible] set.

We have also seen that, in general, computations and their aggregates are implemented by nested loop structures. That is, an implementation involves one or more nested loops, *each of which must have a driving flow set.*

In Part I we saw that for a computation as a whole correct implementation requires the effective enumeration of the critical index sets of each of its inputs. This constraint obviously extends to the individual loop levels. Additionally, the index set of every output of a level must be effectively enumerated so that all records will be written. Combining these constraints in terms of drivers we have

The Fundamental Data Driven Loop Driving Constraint

In the nested loop structure implementing a data driven computation, the drivers for each level, i, must enumerate an index set B_i such that

1. for every input flow F_I at level i, involved in calculating an output flow F_O,

$$CIS_{F_O}(F_I) \subseteq B_i$$

2. for every output F_O at level i,

$$IS(F_O) \subseteq B_i$$

In order to discuss the determination of loop level drivers we must first develop a precise theory of index sets and critical index sets.

IV.1 A Theory of Index Sets and Critical Index Sets for Data Driven Loops

Let us begin with some definitions and useful consequences of these definitions.

IV.1.1 Definitions and Useful Lemmas

We redefine the notions of a flow's index set and critical index set formally and introduce the operators Proj, Inj and Restr:

Definition: The *index set* of a flow F with index I is defined as

$$IS(F) = \{I \mid \text{there is a record in F for I}\}$$

Definition: The *critical index set* of a flow F (with index I) with respect to a flow X is defined as

$$CIS_X(F) = \{I \mid \text{there is a record in F for I that is necessary to generate some record in X}\}$$

Definition: The *projection* of an index set S with index $(k_1, \ldots, k_m, k_{m+1}, \ldots, k_n)$ onto the sub-index $(k_1, \ldots, k_m)$ is defined as

$$Proj(S, (k_1, \ldots, k_m)) = \{(k_1, \ldots, k_m) \mid \exists (k_{m+1}, \ldots, k_n) \text{ such that } (k_1, \ldots, k_m, k_{m+1}, \ldots, k_n) \in S\}$$

Definition: The *injection* of an index set S with index $(k_1, \ldots, k_m)$ by the index set T with super-index $(k_1, \ldots, k_m, k_{m+1}, \ldots, k_n)$ is defined as

$$Inj(S, T) = \{(k_1, \ldots, k_m, k_{m+1}, \ldots, k_n) \mid (k_1, \ldots, k_m) \in S \wedge (k_1, \ldots, k_m, k_{m+1}, \ldots, k_n) \in T\}$$

Definition: The *restriction* of an index set S with index $(k_1, \ldots, k_n)$ by the condition C (whose truth depends on the values of the keys $k_1, \ldots, k_n$) is defined as

$$Restr(S, C) = \{(k_1, \ldots, k_n) \in S \mid C \text{ is true}\}$$

From the last three definitions the following simple but useful results (stated without proof) can be obtained:

Lemma 1: If A is an index set with index I, then

Proj(A,I) = [illegible]

Lemma 2: [illegible], then

Inj(A,B) = B ∩ Inj(A,B)

In particular, if A and B are index sets with the same index, then

Inj(A,B) = B ∩ A

Lemma 3: If S and T are index sets where the index of T is a super-index of that of S, then

Inj(S,T) ⊆ T

Lemma 4: If T is an index set with index I_T and S is an index set with index I_S a sub-index of I_T, then

Proj(Inj(S,T), [illegible]

Lemma 5: If S is an index set with index I_S [illegible] a sub-index of I_S, then

Restr(S, F PRESENT) = Inj(IS(F),S)

IV.1.2 Critical Index Set Theorems for Computations

We begin with two theorems concerning the critical index sets of flows involved in computations. The results are expressed in terms of the index sets of the inputs and outputs.

Theorem 1: If F is a flow defined in terms of the flows $F_1, \ldots, F_n$ by a non-reduction flow equation, where each flow F_i has index I_i, then

$CIS_F(F_i)$ = Proj(IS(F), I_i)

That is, an input record is needed in the calculation of a flow F if and only if it corresponds (by index or sub-index value) to some record of F. [illegible] By Lemma 1 we have that

Corollary 1: Let F be defined as in Theorem 1. Then for any flow F_j with index identical to that of F

$$CIS_F(F_j) = IS(F)$$

Theorem 2: If R is a flow (with index I_R) described by the application of a *reduction* operator to a flow expression expr in terms of the flows $F_1, \ldots, F_n$, where each flow F_i has index I_i, (e.g. the flow equation for R is: R IS SUM OF expr FOR EACH <I_R>), then

$$CIS_R(F_i) = Proj(IS(expr), I_i)$$

(Note that the index of expr must be a super-index of I_R.)

This theorem simply says that when a flow (as that described by expr) is reduced every record of that flow is used in calculating the result. From Theorem 1 we have in turn that the critical index set of each F_i with respect to the flow to be reduced is given by the expression on the right-hand side of the above equation.

Corollary 2: If R is a flow (with index I_R) described by the application of a reduction operator to a flow F (e.g. R IS SUM OF F FOR EACH <I_R>), then

$$CIS_R(F) = IS(F)$$

The following theorems concern the nature of the index sets of flow expressions. First, a simple result about flows described by reduction:

Theorem 3: If R is a flow (with index I_R) described by the application of a *reduction* operator to a flow expression expr (e.g. the flow equation for R is: R IS SUM OF expr FOR EACH <I_R>), then

$$IS(R) = Proj(IS(expr), I_R)$$

This theorem says that there will be a record in the result if and only if the flow to be reduced has at least one corresponding record.

For flows described by non-reduction flow expressions a more extensive treatment is necessary. We begin with simple arithmetic expressions and go on to consider various conditional expressions.

In FE-HIBOL every arithmetic flow expression must be qualified by the presence of all of the flows involved in it (it can, of course, be qualified further). We call such an expression a *simple arithmetic flow expression (safe)* and denote it thus:

$safe(F_1, \ldots, F_n)$ = arith-expr IF F_1 AND F_2 [illegible] PRESENT

where arith-expr is an arithmetic flow expression containing exactly the flows $F_1, \ldots, F_n$.

Theorem 4: The index set of a simple arithmetic flow expression $safe(F_1, \ldots, F_n)$, where F_1 is a flow having the same index as the overall flow expression (in legal FE-HIBOL there must be at least one such flow) is

$$IS(safe(F_1, \ldots, F_n)) = \begin{cases} IS(F_1) & \text{if } n = 1 \\ IS(F_1) \cap IS(F_2) & \text{if } n = 2 \\ \bigcap_i Inj(IS(F_i), IS(F_1)) & \text{if } n > 2 \end{cases}$$

If all of the F_i have the same index as the overall flow expression, this result simplifies further (by use of Lemma 2):

Corollary 3: Let $safe(F_1, \ldots, F_n)$ be a flow expression defined as in Theorem 4 with the additional constraint that the F_i are of uniform index. Then

$$IS(safe[F_1, \ldots, F_n]) = \begin{cases} IS(F_1) & \text{if } n = 1 \\ \bigcap_i IS(F_i) & \text{if } n > 1 \end{cases}$$

As mentioned above the only legal arithmetic flow expression in FE-HIBOL is a safe or a safe further qualified by some condition. This further qualification must take the form of a logical expression ANDed with the safe. Thus, to complete our treatment of arithmetic flow expression we only need the following simple theorem:

Theorem 5: The index set of a simple arithmetic flow expression safe qualified by the condition C is given by

IS(safe AND C) = Restr(IS(safe),C)

Consideration of special cases leads to three simple corollaries:

Corollary 4: By Lemmas 2 and 5

IS(safe AND G PRESENT) = Inj(G,IS(safe))
= IS(safe) $\cap$ Inj(G,IS(safe))

Corollary 5:

IS(safe AND (C_1 AND C_2)) = Restr(IS(safe),C_1) $\cap$ Restr(IS(safe),C_2)

Corollary 6:

IS(safe AND (C_1 OR C_2)) = Restr(IS(safe),C_1) $\cup$ Restr(IS(safe),C_2)

For conditional expressions with two cases[19] we have the following result:

Theorem 6: Let E be a conditional flow expression of two terms:

```
E  =        expr₁  IF C₁
      ELSE  expr₂  IF C₂
```

[19] The extension of this theorem to more than two cases is trivial.

where $expr_1$ and $expr_2$ are legal FE-HIBOL flow expressions and C_1 and C_2 are logical expressions. Define the flow expressions E_1 and E_2 (using the same flow and logical expressions):

$$E_1 = expr_1 \text{ IF } C_1$$

and

$$E_2 = expr_2 \text{ IF NOT } C_1 \text{ AND } C_2$$

Then

$$IS(E) = IS(E_1) \cup IS(E_2)$$

IV.13 Examples

To illustrate the above theorems we give a few examples of loop implementations and verify that the loop level drivers satisfy the fundamental driving constraints.

Example I: R IS THE SUM OF F FOR EACH k_1

where R has index (k_1) and F has index (k_1, k_2). As we have seen above the typical implementation is:

```
for each (k1) from F
    sum = undefined
    for each (k2) from F(k1)
        get F(k1, k2)
        sum =+...
    end
    if defined(sum) then begin
                         R(k1) = sum
                         write R(k1)
                         end
end
```

In level 1 we have the output R and the driver F. The index set D_1 enumerated by this driver at this level is[20]

D_1 = Proj(IS(F),(k_1)) = IS(R)   (by Theorem 3)

thus satisfying the driving constraint for the input R.

In level 2 we have the input F and the driver F. The index set D_2 enumerated by this driver at this level is

D_2 = IS(F) = CIS_R(F)   (by Corollary 2)

thus satisfying the driving constraint for the output F.

Example 2:

```
PAY IS      HOURS * 3.00                IF HOURS PRESENT AND
                                           NOT HOURS > 40

         ELSE 120 + (HOURS - 40) * 4.5  IF HOURS PRESENT
```

We shall use this example to illustrate Theorem 6. Define E_1 and E_2 by

```
E1  =  HOURS * 3.00  IF HOURS PRESENT AND  NOT HOURS > 40
```

and

```
E2  =  120 + (HOURS - 40) * 4.5  IF HOURS PRESENT AND
                                    NOT (HOURS PRESENT
                                       AND NOT HOURS > 40)
```

By pure logical simplification the last equation can be rewritten:

```
E2  =  120 + (HOURS - 40) * 4.5  IF HOURS PRESENT AND
                                    HOURS > 40
```

From Theorem 6 we have that

[20] Theorem 8 of the next section provides a formal treatment of enumerated index sets.

IS(PAY) = IS(E₁) ∪ IS(E₂)

= Restr(IS(HOURS), NOT HOURS > 40) (by Theorem 5)
  ∪ Restr(IS(HOURS), HOURS > 40) (by Theorem 5)

= Restr(IS(HOURS), NOT HOURS > 40 OR HOURS > 40)

= Restr(IS(HOURS), T)

= IS(HOURS)

and by Corollary 1

CIS_{PAY}(HOURS) = IS(PAY) = IS(HOURS)

<u>Example 3:</u>

EP IS P * C IF P PRESENT AND C PRESENT

where EP and D have the index [illegible]-id, store-id) and P has the index (item-id). (This is our familiar EXTENDEDORDER [illegible] with [illegible], PRICE, and CURRENTORDER abbreviated by EP, P and C, [illegible]

We have that

CIS_{EP}(C) = IS(EP) (by Corollary 1)
  = [illegible] (by Theorem 4)

CIS_{EP}(P) = Proj(IS(EP), (item-id)) (by Theorem 8)
  = Proj([illegible]) (by [illegible])
  = IS(P) ∩ Proj(IS(C), (item-id)) (by Lemma 4)

As we have seen above, [illegible] implemented using a two-level loop, with both loop levels driven by C:

```
for each (item-id) from C

   get P(item-id)

   for each (store-id) from C(item-id)

      get C(item-id, store-id)

      EP(item-id, store-id) = ...

      if defined(EP(item-id, store-id))
         then write EP(item-id, store-id)

   end

end
```

In level 1 the input is P and the driver is C. The index set D_1 enumerated by this driver at this level is

$$D_1 = \mathrm{Proj}(IS(C), (item\text{-}id)) \supseteq IS(P) \cap \mathrm{Proj}(IS(C), (item\text{-}id)) = CIS_{EP}(P)$$

In level 2 the input is C, the output is EP and the driver is C. The index set D_2 enumerated by this driver at this level is

$$D_2 = IS(C) \supseteq \mathrm{Inj}(IS(P), IS(C)) \quad \text{(by Lemma 3)} = CIS_{EP}(C) = IS(EP)$$

Thus we see that the flow C is (at least) adequate to drive both levels.

IV.1.4 Driving Flow Set Sufficiency

We wish to be able to determine whether a set of input flows is sufficient to drive a computation loop level. Let us begin by defining the notion of the necessary index set for a computation level:

Definition: The *necessary index set* at level i for a computation C (denoted $NIS_i(C)$) is defined as the set of index values necessary to drive level i of the totally nested loop implementing C.

By the fundamental driving constraint we have

Theorem 7: The necessary index set for level i of a computation C is[21]

$$NIS_i(C) = \left(\bigcup_{\substack{F \in \mathbb{O}(C),\\ I \in \mathbb{I}_i(C)}} CIS_F(I) \right) \cup \left(\bigcup_{G \in \mathbb{O}_i(C)} IS(G) \right)$$

where $\mathbb{O}(C)$ = outputs of computation C
$\mathbb{O}_i(C)$ = outputs of level i in C
$\mathbb{I}_i(C)$ = inputs of level i in C

Now a loop level can be driven by inputs only at the same or lower levels (those at higher levels do not have enough keys in their indices). Obviously the index set enumerated by a driving input at the same level is its index set. The index set enumerated at a level by a driving input at a lower level is given by the following theorem:

Theorem 8: The index set $S_F(I)$ enumerated in a loop level with index I by an input F read at a lower level is

$$S_F(I) = Proj(IS(F), I)$$

Using the terminology just introduced we have

Theorem 9: A set $\mathbb{D}$ of flows is sufficient to drive level i (with index I) if and only if

$$NIS_i(C) \subset \bigcup_{F_j \in \mathbb{D}} S_{F_j}(I)$$

that is, if and only if the index set enumerated by $\mathbb{D}$ at level i includes the necessary index set for that level.

[21] There is some redundancy in this expression. The critical index set of any level i input with respect to a level i output is identical to the index set of that output (by Corollary 1 and the fact that the input and output must have identical indices). That is, for any $F' \in \mathbb{O}_i(C)$, $CIS_{F'}(I) = IS(F')$. Thus $IS(F')$ is already taken into account in the union over $IS(G)$. It would suffice to perform the first union over just those $F \in (\mathbb{O}(C) - \mathbb{O}_i(C))$.

IV.1.5 Minimal Driving Flow Sets

The set of all inputs of a computation is sufficient to drive that computation. We are interested in finding the smallest subsets of this set that will provide sufficient drivers for each level. This interest stems from our implementation constraint that all drivers must be read sequentially and must have compatible sort orders. If all contained inputs were used to drive each level of a computation loop, all inputs to that computation would have to have compatible sort orders and all would have to be read sequentially, a constraint that is often unnecessarily severe.

Moreover, from an efficiency point of view, we generally want the set of indices enumerated by the drivers at any level to be as small as possible (while satisfying the fundamental driving constraints) so as to minimize the number of iterations. For example, if we are trying to minimize I/O accesses and we have a loop that reads some (non-driving) flow by random access, the fewer iterations there are the fewer attempts there will be to access records from that flow.

Consider, for example, the EP computation (Example 3 above). The inputs contained in the outer loop are P and C. Both together could have been used as a driving flow set for that level. We were able to show, however, that C alone was sufficient to drive the outer loop. Thus, we came up with an implementation in which only the flow C had to be sorted and read sequentially. Additionally, in this implementation only those records of P that can actually be used are fetched.

It is important to note that the using some smallest driving flow set for each level does not always improve efficiency. In the computation above it can be shown that P alone is sufficient to drive the outer loop. However, such an implementation would be no better than one in which the outer loop is driven by both inputs. Since the inner loop must be driven by C in any case, we would still end up using both inputs as drivers; both would have to be sorted compatibly and read sequentially; and more records of P would be read than would actually be used.

We want to minimize the number of drivers over the computation as a whole. Thus, instead of independently determining a minimal driving flow set for each level, we look for a smallest flow set that will suffice to drive every loop level. We call a subset of a computation's inputs minimal if it is sufficient to drive every level in the computation but it contains no proper subsets that are sufficient to drive every level. It is these minimal driving flow sets that we wish to find.[22]

To find the minimal driving flow sets we need only systematically enumerate the power set of the inputs of a computation and determine (by application of Theorem 1) which of these subsets are legal driving flow sets not properly containing any other legal driving flow set.

IV.2 Determination of Index Set Inclusion

To show that a flow set is a legal driving flow set we must prove the set inclusion relationship vis a vis $NIS_i(C)$ given above. In an ad hoc way, we have in fact already shown some such inclusions for some of the examples given above. There is a simple, formal method of proving set inclusion[23] based on the use of characteristic functions.

The characteristic function of an index set is a logical expression (or predicate) whose variables are the keys of the index of that index set. For a given index value it evaluates to T (true) if and only if the index set contains that value; otherwise it evaluates to F (false). Our proof technique rests on the equivalence of set inclusion with an implication (⇒) relationship between the characteristic functions of the two sets involved. In particular, for sets A and B with characteristic functions A_{char} and B_{char}, respectively,

---

[22] Not every minimal driving flow set may be usable. In Prototype I, the implementation policy requires that [illegible] be sorted in an order consistent with the nesting order of the loop levels. See Chapter 3 for more details.

[23] Insofar as set inclusion is provable. It can be shown that the general problem of proving set inclusion is not solvable.

$A \supset B \leftrightarrow B_{char} \rightarrow A_{char}$

The expression on the right of the equivalence symbol ($\leftrightarrow$) is a formula in the first order predicate calculus. If this formula can be shown to be a tautology the corresponding set inclusion is proved. Showing that a formula is a tautology is equivalent to showing that it simplifies to T. Since powerful first order predicate calculus simplifiers exist, the task of proving set inclusion can be solved by recasting the hypothesis as a predicate calculus formula and trying to simplify it. If it can be simplified to T inclusion is proved; if it simplifies to F inclusion is disproved.

When the formula cannot be simplified to either T or F, the meaning of the result is not clear. Either the simplification is correct (in which case the formula is not a tautology, and thus set inclusion does not hold) or the simplifier has run up against a fundamental limitation[24] and has failed to simplify the formula completely. In the latter case the formula may in fact be equivalent to T (implying set inclusion), but the simplifier is unable to determine it. Because of this ambiguity, the wisest assumption is the conservative one: whenever simplification to T does not occur, set inclusion does not hold.

IV.2.1 Characteristic Functions for Index Sets

In this section the particulars of the syntax[25] and semantics of characteristic functions for index sets are presented.

The *characteristic function for an index set* is a logical expression (predicate) in terms of its the keys of its index that is true for an assignment of values to those keys in exactly those cases in

[24] It is a well-known fact that it is impossible to devise a procedure that will correctly simplify every formula in the first order predicate calculus.

[25] Because our work is implemented in the LISP programming language the notation is unabashedly LISPish.

which the index set contains a corresponding index value. That is, if $S_{char}(k_1, \ldots, k_n)$ denotes the characteristic function for the index set S then

$$S_{char}(\ell_1, \ldots, \ell_n) = T \quad \text{iff S contains an index value with } k_1 = \ell_1, \ldots, k_n = \ell_n$$

The logical operators from which characteristic functions are formed are:

1. Standard logical operators[26]

   a. AND (AND $p_1, \ldots, p_n$) = T for a particular key-tuple instance iff all of the p_i are true for that instance

   b. OR (OR $p_1, \ldots, p_n$) = T for a particular key-tuple instance iff any of the p_i are true for that instance

   c. NOT (NOT p) = T for a particular key-tuple instance iff p is false for that instance

   d. FOR-SOME (FOR-SOME ($k_1, \ldots, k_m$) p($k_1, \ldots, k_m, k_{m+1}, \ldots, k_n$)) = T for a particular key-tuple instance ($k_{m+1}, \ldots, k_n$) iff there exist values for the keys $k_1, \ldots, k_m$ such that the predicate p($k_1, \ldots, k_n$) is true; this is existential quantification.

2. Standard arithmetic comparison operators (their arguments must be arithmetic expressions in terms of variables (see below) and constants formed using the arithmetic operators +, −, * and /)

   a. EQUAL (EQUAL $expr_1$ $expr_2$) = T iff $expr_1$ and $expr_2$ have the same numerical value

   b. GREATERP (GREATERP $expr_1$ $expr_2$) = T iff the numerical value of $expr_1$ is greater than that of $expr_2$

3. The special operator DEFINED; (DEFINED (V per $k_1, \ldots, k_n$)) = T iff there is a record in the variable V in period per for the key-tuple instance ($k_1, \ldots, k_n$). The argument to a DEFINED operator must be a variable.

The terms introduced here are explained in greater detail in the following sections.

[26] The symbols p and p_i denote predicates.

IV.2.1.1 Variables

A *variable* is a representation of a HIBOL flow with key and period information attached. The period uniquely identifies the variable in time (i.e. it specifies a particular "incarnation" of the flow). An assignment of values to a variable's index and its period specifies an *instance* of that variable and this instance is said to be *defined* if there is a datum (and thus record) corresponding to the key and period values named in the assignment.

The general form for a variable is

```
(flow-name  period  key_1 ... key_n)
```

where flow-name is the name of the associated flow[27], the slot period contains the name of the period in which the variable is generated or input, and the slots key_i contain the names of the keys of the variable. An example of a variable specification is

```
(ENROLLED term student subject-number)
```

where

ENROLLED is the name of the variable
term is the name of a period
student and subject-number are the names of the variable's keys

An occurrence of a variable in a predicate is called a *variable reference*. In a variable reference the form in the period slot identifies a particular incarnation of the variable (e.g. if the period slot contains TERM that means that this term's incarnation of the variable is being referred to; if it contains (PLUS TERM -1.), last term's incarnation is referred to).

[27] The variable and the flow have the same name.

IV.2.1.2 (DEFINED variable-reference)

This expression is true if and only if variable-reference is defined. In particular an expression like

```
(DEFINED (ENROLLED term student subject-number))
```

is true for an assignment of constant values to each of its keys and its period if and only if the variable ENROLLED in the specified period contains a record corresponding to the specified index value; otherwise it is false. Thus, for example, the predicate above is true for subject-number = 33 and term = TERM if and only if in this term's incarnation of ENROLLED there is a record for the index value (JOE 33) (i.e. if and only if Joe is enrolled in subject = 33 during the current term).

IV.2.1.3 Correspondence Between Logical and Set Theoretic Notations

In our characteristic function/index set duality the general correspondence between logical and set operators is given by:

logical operator		set operator
AND	↔	$\cup$
OR	↔	$\cap$
(FOR-SOME $(k_{m+1},\ldots,k_n)$ S_{char})	↔	Proj(S, $(k_1,\ldots,k_m)$)
(AND S_{char} C)	↔	Restr(S,C)
(AND S_{char} T_{char})	↔	Inj(S,T)
(DEFINED (V ...))	↔	IS(V)

That is:

the characteristic function of the intersection of two sets is the logical AND of their characteristic functions;

the characteristic function of the union of two sets is the logical OR of their characteristic functions;

the characteristic function of the projection Proj(S,I') of an index set S onto the sub-index I' is the FOR-SOME operator applied to the characteristic function of S and the remaining keys;

the characteristic function of the restriction Restr(S,C) of an index set S by the condition C is the logical AND of the characteristic function of S and the condition C;

the characteristic function of the injection Inj(S,T) of an index set S by the index set T is the logical AND of their characteristic functions;

the characteristic function of the index set IS(V) of a variable V is the DEFINED operator applied to that variable.

This mapping can be used to determine the characteristic function of any set expression encountered above.

Examples:

The index set

```
IS(P)
```

has the characteristic function

```
(DEFINED (P DAY item-id))
```

The index set

```
IS(P) ∩ Proj(IS(C),(item-id))
```

has the characteristic function

```
(AND (DEFINED (P DAY item-id))
     (FOR-SOME (store-id) (DEFINED (C DAY item-id store-id))))
```

The index set

```
Restr(IS(HOURS), NOT HOURS > 40)
```

has the characteristic function

```
(AND (DEFINED (HOURS WEEK employee-id))
     (NOT (GREATERP (HOURS WEEK employee-id) 40)))
```

IV.2.2 Back-Substitution of Characteristic Functions

We would like our characteristic functions to contain as much information as possible so as to be able to determine as much as possible about the inclusion properties of index sets.

The only possible characteristic function for a variable (V per $k_1, \ldots, k_n$) that is a system input (i.e. a variable whose flow is *not computed* by the system; for example a supplier list) is the *trivial* one (DEFINED (V per $k_1, \ldots, k_n$)), because all that can be said is that it contains a record iff it contains a record.

In some cases an input variable may have the special property that it will always contain a record for every allowable index value. (Knowledge of such a property cannot be deduced from the HIBOL specification of a data processing system; it must be supplied separately.) Such a variable is termed *dense* or *full*. An example might be the PRICE variable, which in every incarnation should have a record for every possible value of the index (item-id). In such a case the characteristic function of such a variable is simply T.

We could use the trivial characteristic function for a computed variable as well, but more (useful) information can be obtained through the application of Theorems 3-6 to the defining HIBOL flow equation. Likewise, we can use Theorems 1 and 2 to obtain useful characteristic functions for critical index sets. Characteristic functions thus obtained are called *one-step characteristic functions*.

It should be easy to see that for any characteristic function if an occurrence of (DEFINED variable) is replaced by the characteristic function for variable, the result will be a logically equivalent characteristic function. This is termed *back-substitution* of characteristic functions. If back-substitution is applied recursively, the result will be a characteristic function containing only

DEFINED's whose arguments are non-computed variables. This is called *total* back-substitution. Total back-substitution of all characteristic functions has the advantage of making them all into a uniform form, thus facilitating comparison and logical manipulation.

IV.2.3 Example

Consider the flow equations:

```
S  IS  H * R   IF H PRESENT AND R PRESENT

X  IS  (H - 40) * R / 2  IF H PRESENT AND R PRESENT AND H > 40

P  IS       S + X  IF S PRESENT AND X PRESENT

       ELSE S      IF S PRESENT

       ELSE X      IF X PRESENT
```

where the flows H and R are system inputs, all flow have the index (key) and all computations are performed daily. The one-step characteristic functions of the necessary input sets are:[28]

$NIS(S)_{char}$ = (AND (DEFINED (H DAY key))
(DEFINED (R DAY key)))

$NIS(X)_{char}$ = (AND (DEFINED (H DAY key))
(DEFINED (R DAY key))
(GREATERP (H DAY key) 40))

$NIS(P)_{char}$ = (OR(DEFINED (S DAY key))
(DEFINED (X DAY key)))

From these we deduce (by Theorem 9) the following results

1. Computation S can be driven by either H or R, since both

[28] We use the outputs as the computation names and drop the level subscript since there is only one level.

```
NIS(S)char  →  (DEFINED (H DAY key))                (1.a)
```

and

```
NIS(S)char  →  (DEFINED (R DAY key))                (1.b)
```

are true

2. Computation X can be driven by either H or R, since both

```
NIS(X)char  →  (DEFINED (H DAY key))                (2.a)
```

and

```
NIS(X)char  →  (DEFINED (R DAY key))                (2.b)
```

are true

3. Computation P must be driven by both S and X, since neither

```
NIS(P)char  →  (DEFINED (S DAY key))                (3.a)
```

nor

```
NIS(P)char  →  (DEFINED (X DAY key))                (3.b)
```

are true, but

```
NIS(P)char  →  (OR  (DEFINED (S DAY key))           (3.c)
                    (DEFINED (X DAY key)))
```

is true

However, we know that

```
IS(S)char = (AND(DEFINED (H DAY key))
                (DEFINED (R DAY key)))

IS(X)char = (AND(DEFINED (H DAY key))
                (DEFINED (R DAY key))
                (GREATERP (H DAY key) 40))
```

so back-substitution of characteristic functions yields

```
NIS(P)char = (OR (DEFINED (S DAY key))
                 (DEFINED (X DAY key)))

           = (OR (AND  (DEFINED (H DAY key))
                       (DEFINED (R DAY key)))
                 (AND  (DEFINED (H DAY key))
                       (DEFINED (R DAY key))
                       (GREATERP (H DAY key) 40)))

           = (AND  (DEFINED (H DAY key))
                   (DEFINED (R DAY key)))
```

Thus, formula (3.a)

```
NIS(P)char → (DEFINED (S  DAY key))
```

becomes

```
(AND (DEFINED (H DAY key)) (DEFINED (R DAY key)))
→
(AND (DEFINED (H DAY key)) (DEFINED (R DAY key)))
```

which is obviously true. Thus, back-substitution has revealed that computation P can be driven by S alone.

Part V: *Loop Implementation* [illegible] *(PLII)*

Each (aggregate) computation (job step, program) in the design produced by ProtoSystem I's Optimizing Designer is merely a loop [illegible] of its input files to generate the records of its output file(s). Implementation is [illegible] the appropriate specific control and data structures necessary for this loop and the I/O involved.[29] The main complications that arise in this process stem from the [illegible] of the loops to be implemented and the hybrid nature of files and loops resulting from aggregation. It is easiest to explain [illegible] through a series of examples, beginning with [illegible] in the [illegible] general case. In this way the full complexity [illegible] easily digestible parts corresponding to the layers of [illegible].

V.1 Single-Level Loops

In order to show the barest necessities of computation implementation, let us begin by looking at the implementation of the most basic single level loop: the simple computation.

V.1.1 Simple Computations

Consider the HIBOL flow equation:

```
PAY IS HOURS * 3.00   IF HOURS PRESENT
```

(recall that PAY and HOURS are files keyed on employee-id). Suppose that the design specifies that both files are to be stored on disk in sequential format.

The basic implementation of this computation is a PL/I DO WHILE loop, whose body will:

read a record of the HOURS file

[29] We make a distinction between implementation and *code generation*, which is the problem of writing the actual code. Although we will show a good deal of detail, we will not go into a detailed discussion of code generation here.

extract the data item (the number of hours worked)

multiply it by 3.00,

assemble the corresponding record of PAY

whose employee-id key is the same as the record read

whose data item's value is the result of multiplying the value of the data item of the record read by 3.00

write the newly created record to the file PAY

To support this iteration, there must be
declarations of the data objects to be used

loop initialization

EOF (end-of-file) checking (to terminate the loop)

V.1.1.1 Necessary Data Objects and Their Declaration

First there must be declarations for all input and output files. Assume that the files PAY and HOURS are known by these names to the PL/I environment (JCL code can be generated to make this happen). Then the following declarations must appear in the PL/I code:

```
DECLARE  HOURS INPUT FILE SEQUENTIAL RECORD,
         PAY OUTPUT FILE SEQUENTIAL RECORD;
```

There must also be declarations for data structures ancillary to the I/O and control to be performed. In particular, for every input file there must be a record image data structure into which a record of that input can be read. Likewise, for every output file there must be a record image data structure into which a record of that output can be built so that it can be written out. In our simple example, the HOURS and PAY files must have such associated data objects. The PL/I structure can be used for this purpose:

```
DECLARE  1 PAY_RECORD,
             2 EMPLOYEE FIXED DECIMAL (4),
             2 PAY FIXED DECIMAL (4),
         1 HOURS_RECORD,
             2 EMPLOYEE FIXED DECIMAL (4),
             2 HOURS FIXED DECIMAL (3);
```

Finally, for each input a flag is needed to indicate the EOF condition for that input. Thus, for the HOURS file we would have the declaration:

```
DECLARE  1 EOF ALIGNED,
             2 HOURS BIT (1) UNALIGNED INITIAL ('0'B);
```

When EOF occurs on the associated file this flag is set to '1'B

V.1.2 Loop Initialization

Before iteration all flags must be initialized. This can be done by the use of the INITIAL statement in the declaration (as above for EOF.HOURS). Also all drivers must be read to establish initial values for their indices. In our example, the initialization section would consist of merely:

```
READ FILE (HOURS) INTO (HOURS_RECORD);
```

V.1.3 EOF Checking and Loop Termination

To detect an EOF condition on a file and set its corresponding flag the PL/I ON construct can be used. For the HOURS file the appropriate code would be:

```
ON ENDFILE (HOURS) EOF.HOURS = '1'B;
```

To enforce iteration termination upon EOF of the driver, the loop is constructed using the form DO WHILE (¬ EOF.driver).

V.1.1.4 The Loop Itself

Given this supporting structure, the rest of the implementation is easy. The loop itself can be written simply as:

```
DO  WHILE (¬ EOF.HOURS);
    PAY_RECORD.PAY = HOURS_RECORD.HOURS * 3.0;

    PAY_RECORD.EMPLOYEE =  HOURS_RECORD.EMPLOYEE;

    WRITE FILE (PAY) FROM (PAY_RECORD);

    READ FILE (HOURS) INTO (HOURS_RECORD);
END ;
```

When the loop terminates, the job step is ended and the input and output files are automatically closed. The complete PL/I program for the pay calculation computation is given in Fig. 1.

V.1.2 Uniform-Index Matching Computations *

Let us extend our treatment of single-level loop implementations to those with more than one input. We use as our vehicle the variation of the pay calculation that includes a rate file (indexed by employee-id):

```
PAY  IS  RATE * HOURS IF RATE PRESENT AND HOURS PRESENT
```

Suppose that the input files RATE and HOURS are to be read sequentially, that their records are sorted by employee-id and that HOURS is used as the loop driver.

Again because the loop is driven by a single input file, it is implemented using the form DO WHILE (¬ EOF.driver). However, the computation description dictates that a record of the output file PAY for a given value of the key employee-id is to be produced if and only if there is a record for that employee in HOURS and there is a *corresponding* record in the RATE file. Therefore, *in the body of the loop, before the output record can be calculated*, the record (if any) of the non-driving input that matches the current value of the driver's index must be found.

V.11.1 The Loop Body

Given this supporting structure, the [illegible] written simply as:

```
DO WHILE (~ EOF (HOURS));
   PAY_RECORD.PAY = HOURS_RECORD.HOURS [illegible]
   PAY_RECORD.EMPLOYEE = HOURS_RECORD.[illegible]
   WRITE FILE (PAY) FROM (PAY_RECORD);
   READ FILE (HOURS) [illegible]
END ;
```

When the loop terminates, the job step is ended and the input and output files are automatically closed. The complete PL/I program for the pay calculation computation is given in Fig. 1.

V.12 Uniform-Index Matching [illegible]

Let us extend our example [illegible] input. We use as our example the [illegible] by employee-id):

PAY IS RATE * HOURS IF RATE PRESENT AND HOURS PRESENT

Suppose that the input files RATE and HOURS are to be read sequentially, that their records are sorted by employee-id [illegible]

Again because the loop is driven by a single input file, it is implemented using the form DO WHILE (~ EOF-of-hours). However, the computation description dictates that a record of the output file PAY for a given value of the key employee-id is to be produced if and only if there is a record for that employee in HOURS and there is a corresponding record in the RATE file. Therefore, in the body of the loop, before the output record can be calculated, the record (if any) of the non-driving input that matches the current value of the driver's index must be found.

To find the matching record of the non-driving input we read successive records from its file comparing the index value of each record with the current loop index. The general matching algorithm consists of the following loop:

For each non-driving input:

1. If FOUND.input is true (indicating that the record currently held in the input's image structure has been used) read the next record of the input.

2. If an EOF condition has occurred on the input, set FOUND.input to false (0) and exit the loop.

3. Otherwise, check the index of the current input record against the index of the current driver record:

   If =, set FOUND.input to true and exit.

   If <, read the next record of the input and go to step 2.

   If >, there is no corresponding record in the input. Set FOUND.input to false (in case the index of the record just read may match that of some subsequent driver record) and exit

To support this algorithm a flag FOUND.input must be declared for each non-driving input and initialized to true (1) before the main loop.

The implementation of the rest of the main loop's body (following the matching code) consists of code that attempts to compute the output record *using only those non-driving inputs whose* FOUND *flags are true.* Basically, in this code, the PRESENT checks of the HIBOL description become checks on the corresponding FOUND flags.

This matching process must be implemented for every non-driving input in a data driven

```
PAY_COMP: PROCEDURE;

(declarations)

ON ENDFILE (RATE) EOF.RATE = '1'B;
ON ENDFILE (HOURS) EOF.HOURS = '1'B;

READ FILE (RATE) INTO (RATE_RECORD);
LEVEL_1_MINIMUM.EMPLOYEE = RATE_RECORD.EMPLOYEE;

DO WHILE ( EOF.RATE);
   IF  EOF.HOURS
      THEN DO;/* THIS READS ITEMS, SEQUENTIALLY, FROM A FILE UNTIL THE REQUESTED
                 RECORD IS FOUND (SET FLAGS TO TRUE) OR PASSED (SET FLAGS TO FALSE). */
              IF FOUND.HOURS_RECORD
                 THEN READ FILE (HOURS) INTO (HOURS_RECORD);

          HOURS_RECORD_COMPARE:
              IF EOF.HOURS
                 THEN FOUND.HOURS_RECORD = '0'B;
                 ELSE IF HOURS_RECORD.EMPLOYEE = LEVEL_1_MINIMUM.EMPLOYEE

                         THEN FOUND.HOURS_RECORD = '1'B;
                         ELSE IF HOURS_RECORD.EMPLOYEE > LEVEL_1_MINIMUM.EMPLOYEE

                                 THEN FOUND.HOURS_RECORD = '0'B;
                                 ELSE DO; READ FILE (HOURS) INTO (HOURS_RECORD);
                                            GO TO HOURS_RECORD_COMPARE;
                                      END;
          END;

   IF FOUND.HOURS THEN DO; PAY_RECORD.PAY = RATE_RECORD.RATE * HOURS_RECORD.HOURS;
                           PAY_RECORD.EMPLOYEE = LEVEL_1_MINIMUM.EMPLOYEE;
                           WRITE FILE (PAY) FROM (PAY_RECORD);
                       END;

   READ FILE (RATE) INTO (RATE_RECORD);
   LEVEL_1_MINIMUM.EMPLOYEE = RATE_RECORD.EMPLOYEE;
END;
END PAY_COMP;
```

Figure 2: PL/I code for PAY IS RATE * HOURS

First, notice that the iteration structure is fundamentally different from that for a single driver loop. The index value determination and EOF checking is now performed at the beginning of the loop body.[31] As always, the iteration is terminated when all drivers are exhausted (when the flag EOF_SO_FAR ends up true after all drivers have been read). Thus the loop exit must appear before the output calculations and the form DO WHILE ('1'B) is used instead of DO WHILE (¬ EOF.driver) (as in the single driver case). This is just a minor variation on the basic scheme.

What is interesting in the implementation of Fig. 3 is the use of the PL/I ACTIVE structure and the ACTIVE_DRIVER_COUNT variable in determining the proper next index value. The idea is to look through the drivers in succession. The first is used to establish a tentative index value for the current iteration. The first driver is also given a number that marks it active (for the time being). If the next driver has the same index value it is given the same number, indicating that it will be active when the first is; if it has a lower index value the loop index is reset and the second driver is assigned a higher number, meaning that it is tentatively active (and, effectively, that the first is inactive). When all drivers have been examined, those sharing the highest ACTIVE number (held in ACTIVE_DRIVER_COUNT) are marked defined, and the rest are marked not defined.

V.2 *Multiple-Level Loops*

Multiple-level loops introduce the need for maintenance of current index values for each distinct loop level and for control structures to implement loop driving from loops at lower levels. Multiple-level loops arise from two basic sources: reduction computations and mixed-index matching computations. Let us examine the implementation of each in turn.

[31] It could be done at the end of the body if the same code were duplicated as an initialization before the loop were entered. We have refrained from doing this to minimize code.

First, notice that the iteration structure is fundamentally dif[illegible] driver loop. The index value [illegible] of the loop body[31] As [illegible] the [illegible]

What is interesting in the implementation of Fig. 5 is the use of the PL/I ACTIVE structure and the ACTIVE_DRIVER_COUNT variable in determining the proper next index value. The idea is to look through the drivers in succession. The first is used to establish a tentative index value for the current iteration. The first driver is also given a counter that makes it active for the time being. [illegible]

[illegible]

[illegible] shown in Fig. 5.

[31] It could be done at the end of the body if the same code were duplicated as an initialization before the loop were entered. We have refrained from doing this to minimize code.

```
ITEMDEMAND_COMP: PROCEDURE;

(declarations)

ON ENDFILE (DEMAND) EOF.DEMAND = '1'B;
READ FILE (DEMAND) INTO (DEMAND_RECORD);
IF  EOF.DEMAND
   THEN DO; LEVEL_2_MINIMUM.ITEM = DEMAND_RECORD.ITEM;
            LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_2_MINIMUM.ITEM;
        END;
   ELSE LEVEL_1 = '0'B;

DO WHILE (LEVEL_1);
   DEFINED.ITEMDEMAND = '0'B;

   DO WHILE (LEVEL_2);
      IF DEFINED.ITEMDEMAND
         THEN ITEMDEMAND_RECORD.ITEMDEMAND = ITEMDEMAND_RECORD.ITEMDEMAND + DEMAND_RECORD.DEMAND;
         ELSE DO; ITEMDEMAND_RECORD.ITEMDEMAND = DEMAND_RECORD.DEMAND;
                  DEFINED.ITEMDEMAND = '1'B;
              END;

      READ FILE (DEMAND) INTO (DEMAND_RECORD);
      IF  EOF.DEMAND
         THEN DO; LEVEL_2_MINIMUM.ITEM = DEMAND_RECORD.ITEM;
                  IF LEVEL_2_MINIMUM.ITEM > LEVELS_1_THRU_2_MINIMUM.ITEM
                     THEN LEVEL_2 = '0'B;
              END;
         ELSE DO; LEVEL_2 = '0'B;
                  LEVEL_1 = '0'B;
              END;
   END;

   ITEMDEMAND_RECORD.ITEM = LEVEL_1_MINIMUM.ITEM;
   WRITE FILE (ITEMDEMAND) FROM (ITEMDEMAND_RECORD);

   IF  EOF.DEMAND
      THEN  LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_2_MINIMUM.ITEM;

END;
END ITEMDEMAND_COMP;
```

Figure 4: PL/I code for ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID

The reader should have little difficulty in understanding this code. Note that the variables LEVEL_1 and LEVEL_2 are used as flags to control the iteration of the outer and inner loops, respectively. LEVEL_1 is essentially equivalent to EOF, [illegible] the outer loop terminates when the file to be reduced is exhausted. LEVEL_2 becomes true when the [illegible] being accumulated is exhausted (i.e. [illegible] EACH clause changes value.

The variables LEVEL_2_MINIMUM.ITEM and LEVELS_1_THRU_2_[illegible] are used to keep track of the current input record's item-id value and the current accumulator's item-id value, respectively. A ">" comparison between these two is sufficient to [illegible] item-id because the input's records are sorted by ascending [illegible].

V.2.2 Mixed-Index Matching Computations

Consider the mixed index computation

EXTENDEDPRICE IS [illegible] PRESENT [illegible] PRESENT

where EXTENDEDPRICE and CURRENTORDER have the index (item-id, [illegible]) and PRICE has the index (item-id). Suppose that, as above, the Optimizing Designer has specified that the records of CURRENTORDER are sorted by the key item-id. As we have shown above, CURRENTORDER can be used to drive the computation.

Because CURRENTORDER is sorted by item-id first, the [illegible] computation [illegible] can proceed as follows:

0. (Initialize) Read a record of the CURRENTORDER file.

1. Read records from the PRICE file until either:

a. one is found that has an item-id value matching the driver's item-id value, in which case all EXTENDEDPRICE records for that value can be generated; or

b. one is found that has an item-id value greater than the driver's, or the PRICE file is exhausted, in which case there is no matching value and the inner loop can be skipped.

2. (Inner loop) Generate all output records for the given item-id value, reading records from the driver as you go. When a driver record is read that has an item-id value greater than that of the current PRICE record, or the driving file is exhausted, exit.

3. If neither input file is exhausted go to step 1 and repeat; otherwise exit.

In this way each record of the PRICE file is read only once[32]

A PL/I implementation of this algorithm is shown in Fig. 5. The reader will notice that this implementation is unnecessarily inefficient because when a matching PRICE record is not found the inner loop is executed anyway. This is done to illustrate what happens in the general case where there may be calculations in the inner loop that can still be performed without the use of a missing input.

V.3 Aggregated Computations

The aggregation of two or more computations into one nested loop introduces a consideration not seen before: the synchronization of computations at different loop levels. Consider the two HIBOL computations:

```
EXTENDEDPRICE IS PRICE * CURRENTORDER IF     PRICE PRESENT
                                         AND CURRENTORDER PRESENT

VALUESHIPPED  IS PRICE * ITEMDEMAND   IF     PRICE PRESENT
                                         AND ITEMDEMAND PRESENT
```

[32] If CURRENTORDER had been unsorted or sorted differently, records from PRICE would generally be read more than once.

a. one is found that has an item-id value matching the driver's item-id [illegible] case all EXTENDEDPRICE records for this item are to be [illegible]

b. one is found that has an item-id [illegible] exhausted, in which case [illegible] loop can be stopped.

2. Once [illegible] value reading records from the driver [illegible] greater than that of the current PRICE [illegible]

3. If neither input file is exhausted go to step 1 and repeat otherwise exit.

In this way each [illegible]

A PLA [illegible] will notice that this implementation is [illegible] found is not found the inner loop [illegible] in the general case where there may be [illegible] without the use of a missing input.

8.3 Associated [illegible]

The aggregation [illegible] and each follows the [illegible] the one GENSL computation

EXTENDEDPRICE IS [illegible] PRICE [illegible] AND CURRENT [illegible]

VALUE [illegible] IS [illegible] PRICE [illegible]

[illegible] If CURRENTPRICE had been sorted [illegible] PRICE would generally be [illegible] where CURRENTORDER is the same as above (with index (item-id, store-id)) and ITEMDEMAND is a file with index (item-id). As we have seen above, the first computation can be implemented as a two-level nested loop. The second computation iterates over the single key item-id and so has only one level.

When aggregated the result is a two level loop:[33]

<u>Loop 1 (outer loop)</u>

```
    Level: (item-id)
           InputsP: (PRICE, ITEMDEMAND)
           Prolog:     calculate value-shipped
           OutputsP: empty

           InputsE: empty
           Epilog:     empty
           OutputsE: (VALUESHIPPED)
```

<u>Loop 2 (inner loop)</u>

```
    Level: (item-id, store-id)
           InputsP: (CURRENTORDER)
           Prolog:     calculate extended-price
           OutputsP: (EXTENDEDPRICE)

           InputsE: empty
           Epilog:     empty
           OutputsE: empty
```

What is significant here is that the computations in the aggregate occur in different levels. Suppose that the PRICE file is guaranteed to have a record for every item-id. Then ITEMDEMAND is the natural choice for a driver for the value-shipped computation because a record of the output will be generated if and only if there is a record in ITEMDEMAND for the same key. As for the extended-price computation, CURRENTORDER is the only possible choice for the driver.

Now the outer loop iterates over item-id values determined by both drivers. Suppose the first record of each driver is read. There are three cases, distinguished by the relative values of the item-id keys in these records:

---

[33] Notice that in finalized loop description there is no General section.

```
(declarations)
(ON conditions)
(read CURRENTORDER and initialize  LEVEL_2_MINIMUM.ITEM = CURRENTORDER_RECORD.ITEM;)
(read ITEMDEMAND and initialize    LEVEL_1_MINIMUM.ITEM = ITEMDEMAND_RECORD.ITEM;)

(code to set the synchronization flag for each level to false if its driver had no records)
(comparison of ITEM values to set synchronization flags:
     IF LEVEL_2_MINIMUM.ITEM > LEVEL_1_MINIMUM.ITEM
        THEN DO; DO_LEVEL_1 = '1'B;
                 LEVEL_2 = '0'B;
                 LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_1_MINIMUM.ITEM;
             END;
        ELSE IF LEVEL_2_MINIMUM.ITEM < LEVEL_1_MINIMUM.ITEM
                THEN DO; DO_LEVEL_1 = '0'B;
                         LEVEL_2 = '1'B;
                         LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_2_MINIMUM.ITEM;
                     END;
                ELSE DO; DO_LEVEL_1 = '1'B;
                         LEVEL_2 = '1'B;
                         LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_1_MINIMUM.ITEM;
                     END; )

 DO WHILE (LEVEL_1);
   (read PRICE record)
   IF DO_LEVEL_1 THEN (calculate value-shipped)  /* Prolog LEVEL_1 */;

   DO WHILE (LEVEL_2);
      IF FOUND.PRICE_RECORD THEN (calculate and write extended-price)
      (read CURRENTORDER and reset  LEVEL_2_MINIMUM.ITEM = CURRENTORDER_RECORD.ITEM;)
      (check for eof)
      IF LEVEL_2_MINIMUM.ITEM > LEVELS_1_THRU_2_MINIMUM.ITEM THEN LEVEL_2 = '0'B;
                                                          ELSE LEVEL_2 = '1'B;
   END /* LEVEL_2 */;

   IF DO_LEVEL_1 THEN DO /* Epilog LEVEL_1 */;
                         IF DEFINED.VALUESHIPPED THEN (write value-shipped record)
                         (read ITEMDEMAND and reset
                               LEVEL_1_MINIMUM.ITEM = ITEMDEMAND_RECORD.ITEM;)
                      END /* Epilog LEVEL_1 */;

(synchronization code exactly as above)

END /* LEVEL_1 */;
```

Figure 6: Illustration of synchronization code for aggregated computations

```
PAY_COMP: PROCEDURE;

DECLARE  DSAG1 INPUT FILE SEQUENTIAL RECORD,
         PAY OUTPUT FILE SEQUENTIAL RECORD;
DECLARE  1 PAY_RECORD,
             2 EMPLOYEE FIXED DECIMAL (4),
             2 PAY FIXED DECIMAL (4),
         1 DSAG1_RECORD,
             2 EMPLOYEE FIXED DECIMAL (4),
             2 DEFINED ALIGNED,
                  3 HOURS BIT (1),
                  3 OVERTIME BIT (1),
             2 HOURS FIXED DECIMAL (3);
             2 OVERTIME FIXED DECIMAL (3);
             2 EMPLOYEE FIXED DECIMAL (4),
             2 HOURS FIXED DECIMAL (3);
DECLARE  1 EOF ALIGNED,
             2 DSAG1 BIT (1) UNALIGNED INITIAL ('0'B);

ON ENDFILE (DSAG1) EOF.DSAG1 = '1'B;

READ FILE (DSAG1) INTO (DSAG1_RECORD);

DO  WHILE (¬ EOF.DSAG1);

    IF DSAG1.DEFINED.HOURS
       THEN DO;

            PAY_RECORD.PAY = DSAG1_RECORD.HOURS * 3.0;

            PAY_RECORD.EMPLOYEE =  DSAG1_RECORD.EMPLOYEE;

            WRITE FILE (PAY) FROM (PAY_RECORD);

            READ FILE (DSAG1) INTO (DSAG1_RECORD);

            END;
       ELSE;

            READ FILE (DSAG1) INTO (DSAG1_RECORD);

END ;
END PAY_COMP;
```

Figure 7: PL/I code for PAY IS HOURS * 3.00 with Aggregated Flow

V.5.1 Sequential Access

Sequential access of sequentially organized files has already been illustrated and explained. Sequential access of indexed sequentially organized files is identical[26]. Sequential access of files with regional (2) organization is not permitted.

V.5.2 Core Table Access

When the records of an input file are to be accessed by the core table method, code is generated to read them all into core before the [illegible]. This requires a PL/I structure that holds not just a single record, but one large enough to hold the maximum number of records in the file. If, for example, the PRICE file in the [illegible] computation above were organized sequentially and were to be accessed by core table the following declaration would be generated:

```
1 PRICE_RECORD (4000),
     2 ITEM FIXED DECIMAL (4),
     2 PRICE FIXED DECIMAL (6),
```

and the code to fill up this table would be

```
DO PRICE_RECORD_INDEX = 1 TO 4000;
   READ FILE (PRICE) INTO (PRICE_RECORD (PRICE_RECORD_INDEX));
   IF EOF.PRICE THEN DO; PRICE_RECORD_SIZE = PRICE_RECORD_INDEX -1;
                  GO TO ENDFILE_PRICE;
              END;
END;
ENDFILE_PRICE: PRICE_RECORD_SIZE = PRICE_RECORD_INDEX - 1;
```

If the input file is sequentially or indexed sequentially organized, the entries in this table are sorted in same order by the record keys. The [illegible] of this table depends on whether the sort order of the input is compatible with the sort order associated with the data of the computation in which it is

[26] The only difference is in the JCL declaration of the file.

used. If the sort orders are compatible the method of access is completely analogous to sequential access except that "records" are "read" from the table instead of secondary storage (see Fig. 8).

If the input file is "randomly" organized (regional (2)) the access code generates a hash index and then mimics the PL/I access procedure: compare the key values of the indicated table entry with the desired ones; if identical stop; otherwise examine successive entries in wrap-around fashion until an empty slot is found (end of the bucket) or a complete cycle has been made. If the sort orders are not compatible a more complicated binary search is implemented.

V.5.3 Random Access

When the records of an input are directly (regional (2)) organized the file is randomly accessed. Instead of using a loop, as with sequential access, a single read, using a calculated key is executed. For example, if the PRICE file in the EXTENDEDPRICE computation (above) were randomly accessed, the accessing part of the code would be:

```
PRICE_RECORD_HASH_VALUE = MOD (5 * (MOD (LEVEL_2_MINIMUM.ITEM,)),);
PRICE_RECORD_HASH_VALUE_STRING = PRICE_RECORD_HASH_VALUE;
PRICE_RECORD_HASH_KEY =
        LEVEL_2_MINIMUM.ITEM || PRICE_RECORD_HASH_VALUE_STRING;
FOUND.PRICE_RECORD = '1'B;
READ FILE (PRICE) INTO (PRICE_RECORD) KEY (PRICE_RECORD_HASH_KEY);
```

The first three statements calculate the source key string which has two parts: the region number (rightmost 8 characters) and the comparison key (the remaining characters). The case where the record is not present is handled by the statement:

```
ON KEY (PRICE) IF ONCODE = 51 THEN FOUND.PRICE_RECORD = '0'B;
```

which resets the FOUND flag if a "keyed record not found" error occurs.

```
IF  EOF.PRICE
  THEN DO; IF FOUND.PRICE_RECORD
             THEN IF PRICE_RECORD_INDEX < = PRICE_RECORD_SIZE
                    THEN PRICE_RECORD_INDEX = PRICE_RECORD_INDEX + 1;
                    ELSE EOF.PRICE = '1'B;

       PRICE_RECORD_COMPARE:
           IF EOF.PRICE
             THEN FOUND.PRICE_RECORD = '0'B;
             ELSE IF PRICE_RECORD.ITEM = LEVELS_1_THRU_2_MINIMUM.ITEM

                    THEN FOUND.PRICE_RECORD = '1';
                    ELSE IF PRICE_RECORD.ITEM > LEVELS_1_THRU_2_MINIMUM.ITEM

                           THEN FOUND.PRICE_RECORD = '0'B;
                           ELSE DO; IF FOUND.PRICE_RECORD
                                      THEN IF PRICE_RECORD_INDEX < = PRICE_RECORD_SIZE
                                             THEN PRICE_RECORD_INDEX =
                                                  PRICE_RECORD_INDEX + 1;
                                             ELSE EOF.PRICE = '1'B;

                                    GO TO PRICE_RECORD_COMPARE;
                                END;
       END;
```

Figure 8: PL/I Code for Reading PRICE by Core Table in the Extended Price Computation

V.6 The General Case--A Summary

We have seen that the basic code structure for a computation consists of the following four parts:[35]

declarations

on-conditions

loop initialization

the nested loop[36]

The basic structure of the body of each loop in the nested loop is as follows:

read & match non-driving inputs

Prolog calculations

inner loop (if any)

Epilog calculations

write outputs

read active drivers

determine new active drivers
and index values for the next iteration

loop synchronization code

exit on EOF or (for inner loop) sub-index change

[35] It may be interesting to note that ProtoSystem I's code generator generates these sections simultaneously as four separate output streams (rather than sequentially) that are catenated together when they are all finished.

[36] There is no clean-up code following the loop because the end of the job step which is the computation does everything necessary, including the closing of files.

Appendix I: The Simple Expositional Artificial Language (SEAL)

As an aid to discussing loops we invent an artificial language similar in form to traditional high-level languages such as ALGOL, PL/I and FORTRAN. The basic constructs of this language are:

<u>Iteration</u>: expressed by the construct:

```
for each <loop-index> from <driving-flow-set>
    <body>
end
```

which has the meaning: perform the actions contained in the <body> for each value of the <loop-index> obtained from the flows in the <driving-flow-set>. <loop-index> is the either the name of the index associated with the flows in the <driving-flow-set> or (for reasons that become evident in this paper) a sub-index of corresponding sub-flows. The set of values that the <loop-index> takes on is the union of the index sets of the drivers. This set is enumerated at execution time by reading successive records of the drivers.

<u>I/O and defined:</u> input (record fetching) is expressed by the get operator, thus:

```
get <variable-instance>
```

where <variable-instance> specifies a flow and a particular value for its index, represented as a variable (see below). A statement like this means: fetch the indicated record if it exists.

Output is expressed by the write operator, similarly:

```
write <variable-instance>
```

The defined operator is a logical operator for use in conditional expressions. It is applicable only to flow variable instances. The form

```
defined[<variable-instance>]
```

evaluates to "true" if the specified record or the indicated flow exists. In particular, if the record is an input (obtained through a get) it is "defined" if and only if the get succeeded; if the record is an output it is "defined" if and only if the generating code produced a datum for the record.

Conditional Execution: expressed by the familiar if-then-else construct:

```
if <condition> then <statement-list>1
               else <statement-list>2
```

which means that if the logical expression <condition> evaluates to "true" perform the statements in <statement-list>$_1$; otherwise, perform the statements in <statement-list>$_2$.

Logical expressions can be formed using the arithmetic comparison operators, the defined operator, and the logical connectives and, or and not.

Conditional Expressions: expressed by the construct:

```
if <condition> then <expression>1
               else <expression>2
```

which evaluates to the value of <expression>$_1$ if the logical expression <condition> evaluates to "true" and to the value of <expression>$_2$ otherwise.

Variables and Assignment: expressed by the construct:

```
<variable> = <expression>
```

where = is the assignment operator.

A variable can be either a scalar or an indexed variable. Flows are represented as indexed variables with an index identical to the flow's index. Thus, DEMAND(item-id, store-id) is the variable corresponding to the DEMAND flow and an instance of its index selects the datum of the corresponding flow record. That is, for example, the statement

```
DEMAND(1234, 5678) = CURRENTORDER(1234, 5678) +
                     BACKORDER(1234, 5678)
```

means that the datum of the record of DEMAND for item #1234 ordered by store #5678 is to get the value obtained by adding the data of the corresponding records from CURRENTORDER and BACKORDER.

Typically, the record-by-record computation implied by a HIBOL flow equation would look like that equation translated into our artificial language (with a generalized index), such as

```
DEMAND(item-id, store-id) =

        if defined[CURRENTORDER(item-id, store-id)]
           and defined[BACKORDER(item-id, store-id)]

           then CURRENTORDER(item-id, store-id) +
                BACKORDER(item-id, store-id)

           else if defined[CURRENTORDER(item-id, store-id)]

                then CURRENTORDER(item-id, store-id)

                else if defined[BACKORDER(item-id, store-id)]

                       then BACKORDER(item-id, store-id)

                       else undefined
```

and would appear somewhere in the body of loop.

<u>Sub-flows:</u> A sub-flow (for use in the for each construct) is expressed by:

```
<flow-variable>(<sub-index>)
```

For example,

```
CURRENTORDER(item-id)
```

denotes the sub-flow of CURRENTORDER consisting of just those records whose indices correspond to the value of the sub-index (item-id). Generally, the value of the indicated sub-index is fixed by an enclosing loop.

References

1. Baron, Robert V., "Structural Analysis in a Very High Level Language", Master's thesis, MIT, 1977.

2. Fleischer, Richard C., "Loop Merger in ProtoSystem I", Bachelor's thesis, MIT, 1978.

3. Ruth, Gregory. R., "ProtoSystem I--An Automatic Programming System Prototype", Proceedings of the National Computer Conference, 1978.

Index