MIT/LCS/TM-70

AUTOMATIC DESIGN OF DATA PROCESSING SYSTEMS

GREGORY R. RUTH

FEBRUARY 1976

MIT/LCS/TM-70

Automatic Design of Data Processing Systems

February 1976

Gregory R. Ruth

Laboratory for Computer Science
(formerly Project MAC)
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

This work was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract #N00014-75-C-0661.

ABSTRACT

The design of data organization and data accessing procedures for data processing systems operating on large keyed files of data is a common and recurrent activity in modern data processing applications. A considerable amount of understanding and expertise in this area has been developed and it is time to begin codifying and automating this process. It should be possible to develop a system where the user has merely to specify the characteristics of his data objects and their interrelations and the system will automatically determine the data organizations and accessing procedures that are optimal for his application. The optimizer for Protosystem I (an automatic programming system prototype at MIT) provides an example of how such automation can be accomplished.

Automatic Design of Data Processing Systems

The design of data organization and data accessing procedures for data processing systems operating on large keyed files of data is a common and recurrent activity in modern data processing applications. A considerable amount of understanding and expertise in the this area has been developed and it is time to begin codifying and automating this process. It should be possible to develop a system where the user has merely to specify the characteristics of his (input, output, and intermediate) data objects and their interrelations[2] and the system will automatically determine the data organizations and accessing procedures that are optimal for his application. The optimizer for Protosystem I (an automatic programming system prototype at MIT's Project MAC) provides an example of how such automation can be accomplished. This paper describes the theory and algorithms behind the optimizer that is currently operational at Project MAC.

Previous Work

The first reported efforts in the area of formalizing data processing system design are those of Langefors[5,6]. He conceives a data processing system as a collection of processes or computations that operate on data files. In this model a file is a set of records, each with a single key and one or more data items. Each computation makes passes over all of the records of its input files to produce records of its output files. A transport volume is associated with each file-computation pair (where the computation reads or writes that file). This is defined to be the volume of the file (measured, say, in bytes) multiplied by the number of passes made by the computation over that file. The design objective is the minimization of total system transport volume. Transport volume reduction is accomplished by merging computations (so that a file can be read once for all, rather than separately for each) and by merging files (thus eliminating key redundancy and making the resultant file physically smaller than the sum of the sizes of its component files). Nunamaker et al[7] have analyzed this formulation and developed an algorithm for system optimization that generates all feasible mergings (by implicit enumeration) and corresponding system configurations to find the one that minimizes transport volume.

Because the computations generally involve very simple calculations, the bulk of the processing cost for such systems is usually due to I/O charges. Thus, an optimization strategy that focuses mainly on I/O reduction is appropriate. But implicit in the Langefors model are two simplifying assumptions that seriously weaken it as a design optimization tool for practical systems: (1) that I/O costs depend directly on total transport volume as defined above, and (2) that all files have a single key. (Also ignored is the cost due to core residency charges, but in systems dominated by secondary-primary storage I/O activity, this is a second order effect).

In most operating systems, the user is charged primarily by the I/O event (the transfer of a single second storage block to or from primary storage), rather than by total volume. Furthermore, the number of I/O events necessary when a computation processes a file depends on not just the total volume of the file, but also (1) the number of records in a block, (2) the number of records used by the computation, and (3) the access method (e.g. sequential, random).

The limitation of one key per record is completely artificial and at variance with practical reality. Files frequently have more than one key in their records (cf. Codd[1]). This means that they can be sorted in more than one way. By the same token, a computation that processes such a file can process its records in more than one order by their keys. This affords the possibility of conflict either between two computations that process the records of a file in two different key orders, or between two files with different sort orderings that are processed by the same computation. The manner in which such conflicts are resolved (e.g. re-sorting a file, using random instead of sequential access) will affect the total I/O cost of the system. The possibility of sort order conflicts also complicates the evaluation of I/O savings due to file and computation merging.

The Protosystem I optimizer takes all of these design considerations into account.

The Protosystem I MIS Model

Protosystem I is an automatic programming system for generating batch oriented MIS's. Such systems involve a sequence of runs or job steps that are to be performed at specified times. They are assumed to involve significant I/O activity due to repetitive processing of keyed records from large files of data. Systems such as inventory control, payroll, and employee or student record keeping systems are of this type.

Central to the design of such systems is the notions of the *data set*. A data set is a collection of similar data that are to be processed in a similar way. An example is the set of all inventory levels in a warehousing MIS. In the domain of Protosystem I a data set is assumed to consist of fixed format *records* (e.g. one for the level of each inventory item). Associated with each record are *data items* and *keys*. The key values of a record uniquely distinguish it (e.g. the inventory data set can be keyed by item since there is only one level [record] per item) and so can be used to select it. Thus, a data set is essentially the same as a Codd relation[1] and its keys are what Codd calls primary keys.

The repetitive application of an operation to the members of a data set or sets is termed a *computation.* The order of application of the operation by a computation is assumed to be unimportant to the user; in fact, he may think of them as being performed in parallel. However, every computation does, in fact, process its inputs serially, according to a particular ordering (chosen by Protosystem I) on their keys. Computations typically *match* data items from different data sets by their keys and operate on the matching items to produce a corresponding output data item. For example, an order filling computation matches the total orders for a given item against the inventory level of that item and determines the amount of that item to be shipped. Computations may also *group* the members of a data set by common keys and operate on each group to produce a single corresponding output. Following our example, suppose that item orders can come from several sources, so that both the item and the source of an order are needed (as keys) to distinguish it. To form the total of all orders for each item, a system must group the orders by item and sum over the order amounts in each group.

In this context, then, data processing system design is the process of constructing computations and data sets and determining the best access methods and organizations for them. In the domain of Protosystem I only three types of file organizations and three types of accessing methods are considered. The organizations are:

Consecutive — Records are organized solely on the basis of their successive physical location in the file. For Protosystem I records in a sequentially organized data set are required to be completely ordered.

Index Sequential Records are organized in such a way that by reference to indices associated with the data set it is possible to quickly locate any record and read or update them. Additions or deletions do not require a rewrite of the file.

Each index-sequential data set has a particular sort ordering (determined by the optimizer) associated with it and usually the records are *nearly* ordered in this way physically.

Regional (2) Records are stored in a "random" manner but there is a mapping function from the keys of a record to its physical address.

The accessing methods are:

Sequential For regional (2) and consecutive data sets, records can be read and processed in the physical order in which they appear in the data file. For an index sequential data set the operating system's sequential access software delivers records in the sort order associated with the data set.

Random Records are referenced (read, updated, added, or deleted) through the values of keys supplied by the computation. This method is applicable only for accessing regional (2) data sets or for reading index sequential datasets.

Core Table An input file is initially read into core in its entirety; an output file, after being assembled in core, is sequentially copied onto the device where it will reside when finished with. Records are accessed in core sequentially if their sort order agrees with the processing order of the computation; otherwise, by binary search. Directly organized data sets are accessed by hashing.

The Optimization Criterion

Optimization, as viewed by this part of Protosystem I is simply cost minimization. Further, because the MIS's are assumed to be I/O intensive, this is equated with access minimization. An *access* is defined as the reading or writing of a single secondary storage block, which corresponds to a single operating system I/O event. In Protosystem I, for a particular data set a *block* consists of a fixed number of records.

Example

A simple example of an MIS in the domain of Protosystem I is the store chain inventory and warehousing system shown in Fig. 1 (boxes indicate computations and arrows represent data flows). This system contains computations that update the inventory levels file to reflect shipments received from suppliers, find the total amount of each item ordered by all stores, fill the orders, determine the total reductions in inventory, adjust the levels file accordingly, check for the necessity of restocking orders, and make orders to the proper suppliers.

This example illustrates some of the important factors in data processing design. Consider the FILL ORDERS computation. For each item that is ordered by some store a shipment quantity is determined. It could be implemented by considering each possible combination of values for the keys STORE and ITEM, determining whether the conditions are suitable for producing a shipment record (viz. that the item in question has been ordered by the store under consideration), and if so applying the associated operation to generate such a record. This requires a test for every possible combination and each test involves a probe to one or more of the input data sets. Since on a given day not all of the stores will order and most that do will order only a fraction of the possible items, such an implementation would involve an unnecessarily large number of useless accesses. In contrast, since it can be determined (by analyzing the FILL ORDERS computation) that the operation will be applicable only when there is a record in QUANTITIES ORDERED, the computation could be designed to consider only those store and item combinations for which there are records in QUANTITIES ORDERED. In this case the computation is said to

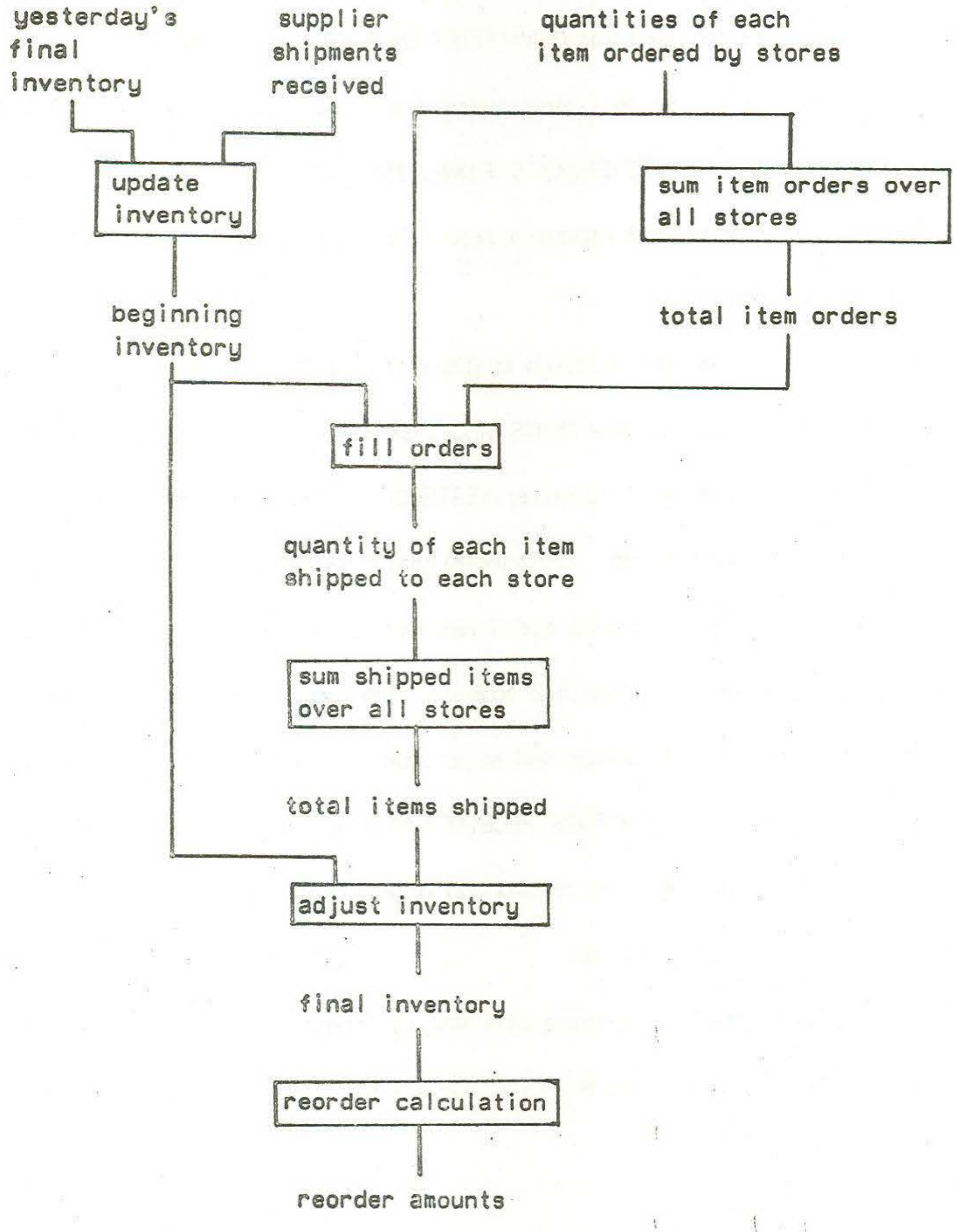
yesterday's final inventory
supplier shipments received
quantities of each item ordered by stores
update inventory
sum item orders over all stores
beginning inventory
total item orders
fill orders
quantity of each item shipped to each store
sum shipped items over all stores
total items shipped
adjust inventory
final inventory
reorder calculation
reorder amounts

Fig. 1
Simple Store Chain Inventory and Warehousing System

be *driven* by QUANTITIES ORDERED, and QUANTITIES ORDERED is said to be its *driving data set*. Note that not every input to a computation is a suitable driver; for the UPDATE INVENTORY computation only YESTERDAY'S FINAL INVENTORY will do--if SHIPMENTS RECEIVED were used it might not contain a record for each item, causing BEGINNING INVENTORY to be incomplete.

Another factor in optimization is illustrated by the UPDATE INVENTORY computation. If the records of SHIPMENTS RECEIVED are consecutively organized and sorted in the same order as those of the driver (YESTERDAY'S FINAL INVENTORY) it can be used in the order in which it is stored. This means that blocking can be used to minimize accesses; if there are B records per block and N records in shipments received, the number of accesses necessary will be no greater than $N/B + 1$. However, if the two inputs are not sorted in the same order, the computation will have to search (usually requiring more than one access) for each record of SHIPMENTS RECEIVED when it is needed. Alternatively, SHIPMENTS RECEIVED might be given regional (2) organization and accessed randomly; but the blocking advantage would be lost.

These considerations of driving data sets, consecutive data set organization, compatible sort orderings and blocking will figure importantly in the design process.

The Optimizer

In Protosystem I the optimizing designer of data set organization and data processing (called the *optimizer*) is given a relational description of the basic data aggregates and the relations among them. (Fig. 2 is an example of such a description for the MIS of Fig. 1.) Its job is to

CALCULATIONS EVERY DAY

```
BEGINNING-INVENTORY IS FINAL-INVENTORY(1 DAY AGO) + SHIPMENTS-RECEIVED


TOTAL-ITEM-ORDERS IS SUMS OF GROUPS QUANTITY-ORDERED-BY-STORE BY ITEM


QUANTITY-SHIPPED-TO-STORE IS

      QUANTITY-ORDERED-BY-STORE       IF BEGINNING INVENTORY IS GREATER
                                              THAN TOTAL-ITEM-ORDERS

      QUANTITY-ORDERED-BY-STORE
      * (BEGINNING-INVENTORY / TOTAL-ITEM-ORDERS)
                                              IF BEGINNING INVENTORY IS NOT
                                              GREATER THAN TOTAL-ITEM-ORDERS


TOTAL-SHIPPED IS SUMS OF GROUPS OF QUANTITY-SHIPPED-TO-STORE BY ITEM


FINAL-INVENTORY IS BEGINNING-INVENTORY - TOTAL-SHIPPED

REORDER-AMOUNTS IS 1000 IF FINAL-INVENTORY IS LESS THAN 100
```

DATA SET TABLE

```
(BEGINNING-INVENTORY          DAY  ITEM)
(FINAL-INVENTORY              DAY  ITEM)
(ORDERS-TO-SUPPLIERS          DAY  ITEM)
(QUANTITY ORDERED-BY-STORE    DAY  ITEM)

(QUANTITY-SHIPPED-TO-STORE    DAY  ITEM STORE)
(TOTAL-ITEM-ORDERS            DAY  ITEM)
(TOTAL-SHIPPED                DAY  ITEM)
```

Figure 2
Relational Description for the MIS of Figure 1

(1) design the keyed files--in particular their

  (a) contents (information contained)

  (b) organization (direct, consecutive, or index sequential)

  (c) storage device

  (d) associated sort orderings (by key values)

(2) design each job step of the MIS--namely

  (a) which computations it includes

  (b) its accessing methods (sequential, random, core table)

  (c) its driving data set(s)

  (d) the order (by key values) in which it processes the records of its input data sets

(3) determine whether sorts are necessary and where they should be performed

(4) determine the sequence of the job steps

All design decisions are made in an effort to minimize the total number of accesses that must be performed in the execution of the MIS. In preparation for this design process an analysis of the MIS description is performed in order to determine properties relevant to making good decisions. Among these properties are the candidates for driving data sets for each computation and the average and maximum sizes of each data set (the latter may require some interrogation of the user). During the design process the user may be asked to give further information[9] that can be used to determine the sizes of newly designed data sets.

Access Minimization

There are three major techniques that the Protosystem I uses in designing MIS's so as to minimize accessing: (1) making use of data set blocking, (2) aggregating data sets, and (3) aggregating computations.

Blocking

Since an access is defined as a reading or writing of a block of records, accesses can be reduced if blocking factors greater than one are used and if processing and data set organizations are arranged in such a way that the records of a block can be used at the same time. Where core table access is not possible (i.e. when the whole file will not fit in available core) this means that a data set must be accessed sequentially (and therefore have consecutive or index sequential organization) and it must be sorted in an order that is the same as processing order(s) of the computation(s) accessing it. When core table access is possible for a data set the number of accesses is just the number of blocks in the data set; regardless of whether the data set's sort order matches the processing order of accessing computations, each block need be accessed just once to get it into or out of core.

Because blocking is the most effective way of reducing accesses, the optimizer tries to make it possible for the sort orders of data sets that are too big for core table access to match the processing orders of their accessing computations in as many cases as possible. However, unresolvable conflicts among sort order preferences can and will arise. Sometimes it is advantageous to introduce resorting computations so that a data set can have two or more differently sorted versions.

Aggregating Data Sets and Computations

The relational description of the MIS identifies the *basic* data entities and operations. The straightforward implementation of such a description is to make one data set for each data entity and one computation for each operation. This can be considered as the *initial configuration* to be manipulated by the designer. The designer combines or *aggregate* data sets and computations in order to reduce accesses.

The aggregation of data sets produces a data set in which there is one record for each set of records in the original data sets that match (by keys). For example, the aggregate of the data sets in Figs. 3.a and 3.b is shown in Fig. 3.c

$(N_{11}, 3)$	$(N_{21}, 2)$	$(N_{11}, ---, 3)$
$(N_{12}, 2)$	$(N_{22}, 4)$	$(N_{12}, N_{21}, 2)$
		$(---, N_{22}, 4)$
male-employees(dept)	female-employees(dept)	male-&-female-employees(dept)
a.	b.	c.

Figure 3: Data Set Aggregation

where a record is represented by a tuple whose last element is the key value and whose first elements are the data items.

Data set aggregation is advantageous when two or more data sets are read or written by the same computation. Accesses can be saved if the shared data sets are aggregated *and* processing is arranged so that a single record of the aggregate can be accessed where more than one record from each of the unaggregated data sets would have to be accessed without aggregation.

Two other ways of reducing the number of accesses are illustrated in Figs. 4.a and 4.b (circles represent data sets, boxes represent computations). It may be advantageous to combine computations that read the same data set so that they can all access a record at the same time; if their processing orders for the shared data set agree, each record to be accessed can be read once for all, rather than once for each computation. This is called *horizontal aggregation* (Fig. 4.a).

Vertical aggregation refers to combining two computations when the output of one is used as the input to the other and their processing orders agree (Fig. 4.b); in that case, the second computation does not have to read the output records of the first--they can simply be passed from the one computation to the other as needed.

The aggregation of computations is more than a simple merging. Looping redundancy is eliminated so that the composite result is more efficient than its components separately.

General Design of the Optimization

The access minimizations techniques given above require that the key order of processing agree in a special way with the organization of the data being processed. This is where the fundamental difficulty in optimization lies. A data set's organization and the accessing method of a computation using it cannot be determined independently of each other or of other data set organizations and computation accessing methods. The organization of a data set limits the ways in which it can be practically accessed by a computation, and, conversely, the accessing method of a computation restricts the practicable

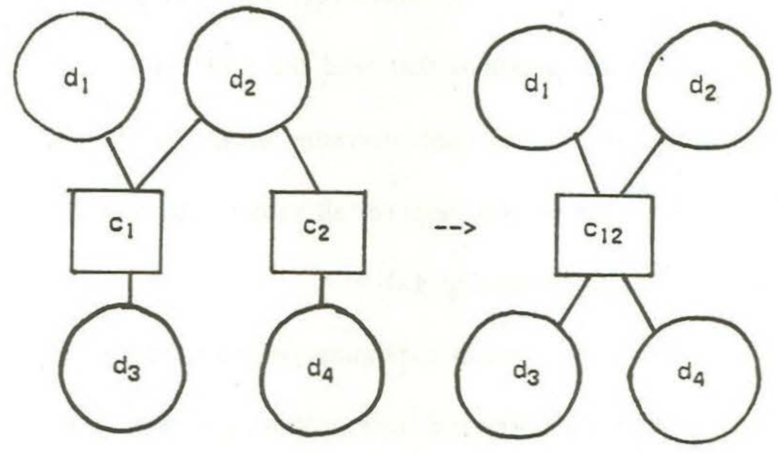

Figure 4.a: Horizontal Aggregation of Computations

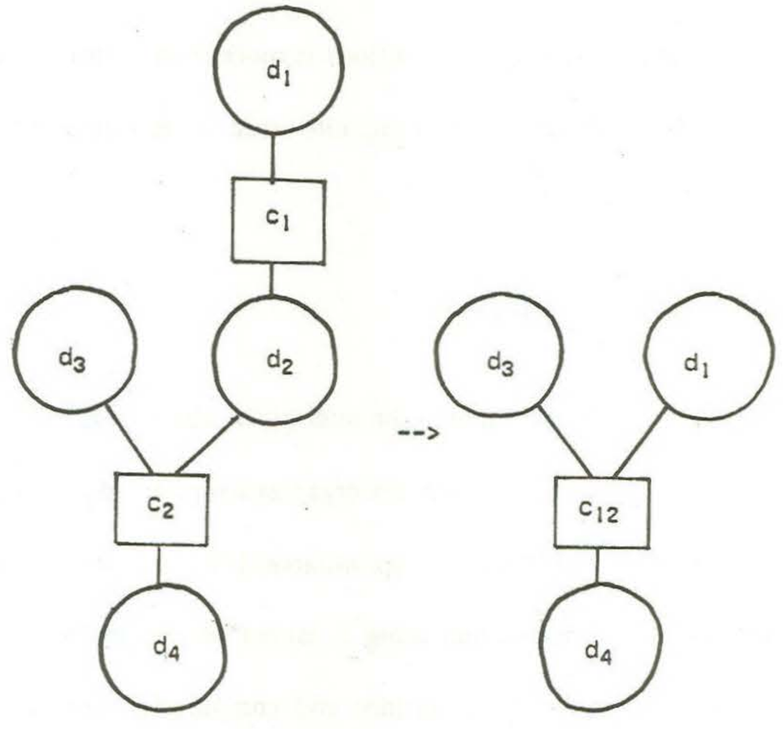

Figure 4.b: Vertical Aggregation of Computations

organizations of a data set that it accesses. Furthermore, a data set is typically accessed by more than one computation with possibly conflicting preferences for its organization; and a computation accesses more than one data set with conflicting preferences for accessing methods. Finally, data set organization constraints tend to propagate *through* computations, because it is most efficient for a computation to write its outputs in the same key order in which it reads its inputs, since that is the order in which the output records will be generated. So, optimization of the type we are considering is necessarily be a problem in global compromise.

The straightforward solution of evaluating the cost of every possible combination of assignments of sort order, device, organization, and access method for data sets and computations in every possible aggregation configuration to determine the least expensive is ruled out by the sheer combinatorics involved. Even with mathematical and special purpose tricks it would be impossibly slow.

To make optimization tractable a heuristic approach must be taken. First different kinds of decisions (e.g. choice of driving data sets, which objects to aggregate) must be decoupled wherever possible. Further decoupling must be judiciously introduced where it is not strictly possible, for the sake of additional simplicity. Such forced decoupling does not mean, though, that decisions that are in fact coupled are treated as if they were independent. The decoupled decisions are still made with a certain awareness of their effects on other decisions. Finally, as a first order approximation, the optimizer does what is reasonable locally, and then adjusts somewhat for global realities.

The Optimization Algorithm

Our optimization algorithm consists of the following 7 steps:

1. Development of maximal potential for reducing accesses through blocking for the initial configuration.

2. Aggregating computations where advantageous in the current context.

3. Aggregating data sets where advantageous in the current context.

4. Iteration over steps 2 and 3 until no further aggregation is suggested.

5. Determination of driving data sets.

6. Determination of device and organization for each data set and of access method for each computation.

7. Determination of optimal blocking factors.

Step 1: Setting Up the Initial Configuration to Take Advantage of Blocking

The determination of mutually agreeable sort orders for computations and data sets that will allow maximal advantage to be taken of blocking (which requires matching sort orders and sequential accessing where core table access is not possible) must be considered first in the optimization process. Aggregation and other optimization techniques are of little value if they force sorting or other methods of accessing (that can require orders of magnitude more I/O events than sequential accessing).

As explained above it is necessary for the sort order of a data set that is not core table accessible to be made the same as the processing order of a computation that accesses it in order to reduce accesses by that computation by blocking. But there are user constraints on data set sort orders and inherent preferences of computations for constraints on their sort orders that must be taken into account, too. As a result of these each computation and data set will have a sort order constraint (SOC) restricting the possible sort orders it may be given.

For example, a data set D(key_1, key_2, key_3) may have the SOC that its records be sorted on key_2 first. Then the optimizer could decide to sort them first by key_1 and then by key_3 under key_2, or vice versa.

The factors that affect SOC's are:

1. User constraints The user may specify that inputs or outputs have a particular associated SOC.

2. Uniqueness A particular computation presents the same SOC to all of the data sets that it reads or writes; and a particular data set presents the same SOC to all of the computations that read or write it

3. Parallelism A computation outputs records in the same key order in which its inputs are read.

4. Computation Preferences: The optimizer prefers that the input data set to a grouping computation be sorted first by the keys that distinguish the groups it operates on. That way, it can be designed to process one group at a time. Otherwise groups must be processed in parallel, requiring costly accesses if the core available for buffering is insufficient.

5. Sequential Access Preference (to take advantage of blocking)

Two SOC's for which there is a common possible sort order are called *consistent*. If a data set and a computation accessing it have consistent SOC's both can be assigned the same SOC (one that conforms to each of their original SOC's), thus insuring potential for sequential access and blocking.

In general it will not be possible to satisfy all preferences and constraints simultaneously. Conflicts arise preventing the sequential accessing of every data set. At best, therefore, the optimizer can only try to find the cheapest compromise. What it aims for is the maximization of the total volume of data accessed sequentially. Even here it must to make concessions to practicality. That is, it tries to come as close as possible to this maximum without expending an unreasonable amount of effort.

Its method is to follow the implications of the initial constraints and preferences throughout the network of computations until conflicts arise, and then try to resolve those conflicts as advantageously as possible. It tries to associate with each computation and data set a minimimally restrictive SOC that is consistent with its own preference (if any) *and* with all of the constraints on and preferences for the SOC's of all of the immediately adjacent objects in the net (for data sets this is the set of all accessors, and for computations this is the set of all inputs and outputs). Insofar as this is possible it has what it wants; otherwise it has discovered a conflict of interest, which it attempts to resolve in such a way that the greatest volume possible of records involved can be accessed sequentially.

We have found that this simple SOC assignment algorithm produces good, if not optimal, results in the systems we have tested. This occurs because typically (1) there are very few conflicts and (2) those that do occur are generally simple and local phenomena.

When an assignment of minimally restrictive, "consistent-as-possible" SOC's has been determined for all of the objects in the system heuristic aggregation of computations and data sets can be performed. It is not advantageous to aggregate computations and data sets if this will increase the I/O cost by precluding sequential accessing that may have been possible before aggregation. So, the aggregation of two data sets or computations will be permitted only if their SOC's are consistent. On the other hand, unnecessarily restrictive

SOC constraints would prevent aggregations that would otherwise be possible. Thus the emphasis on finding *mtntmally* restrictive SOC's.

Step 2: Computation Aggregation

Computations are considered for aggregation if they are have consistent SOC's and access a common data set. This data set may be either a common input (horizontal aggregation) or a common intermediate data set--that is, the output of one and and input to the others (vertical aggregation). In order to preserve existing sequential accessing potentials, the SOC of the aggregate must be consistent with each of the SOC's of the aggregated computations. Further, to avoid unnecessarily precluding otherwise possible aggregations, the SOC assigned to the aggregate is the minimally restrictive mutual restriction of the SOC's of the components. Nevertheless, it will not be uncommon for the SOC of the aggregate to be more constrained than the SOC's of the computations that went into it. This means that the decision to make a particular aggregation may prevent other aggregations (by virtue of SOC inconsistency) that were formerly possible. For example, computations A, B, and C may be pairwise aggregatable but the aggregation of all three may be impossible; that is, if, say, A and B are aggregated, the SOC of the result may be inconsistent with that of C, even though the SOC of A is consistent with that of C, and the SOC of B is consistent with that of C.

Finding an exact optimum by considering all possible combinations of all possible aggregations is again precluded by sheer combinatorics. The heuristic approximation is made that what is optimal locally is good for the system, and again we have observed (and conjecture) that the simplicity of typical systems is such that this is a viable approach.

The optimizer considers aggregation candidates two at a time. As the order of treatment is significant, the policy is (locally) to consider aggregations in order of the number of I/O events they are expected to save. Furthermore, classes of aggregations are performed in the following order:

1. vertical aggregations
2. horizontal aggregation of computations reading common system inputs
3. other horizontal computation aggregations

Vertical aggregations are considered to have a higher priority than the horizontal variety because the former may result in the entire elimination of data sets; that is, the data sets involved will neither have to be read nor written. Horizontal aggregations that eliminate reads of system inputs are preferred over other horizontal aggregations because such inputs often reside on such storage media as magnetic tape and cards, which are relatively inconvenient and costly to access repeatedly.

Step 3: Data Set Aggregations

Data sets are considered for aggregation if they have consistent SOC's, are not system inputs or outputs, and are both outputs of a common computation and inputs to another common computation. As with computations, care must be taken so that the SOC of the aggregate is consistent with each of the SOC's of the aggregated data sets, in order to preserve whatever sequential accessing potentials may exist. As the introduction of arbitrary constraints can prevent otherwise possible aggregations here too, the SOC assigned to the aggregate is the minimally restrictive mutual restriction of the SOC's of the components. But again, the SOC of the aggregate may be more constrained than the SOC's of the data sets that went into it, so a particular aggregation may prevent others, and the order in which they are performed is important.

The optimizer considers those computations with more than one output in order of the total volume in records that they process. For each computation its largest output data set is considered for aggregation with as many of the others (in order of their size) as possible, the criteria for aggregation being that SOC's be consistent and that the total cost of reading the aggregate (by all computations for which it is an input) is lower than the total cost without aggregation.

Step 4: Aggregation Iteration

Computation and data set aggregation are repeated until no further aggregation is possible. This iteration is necessary because the aggregation of data sets may make further pairwise computation aggregations feasible, and similarly, the aggregation of computations may make additional pairwise data set aggregations feasible.

Step 5: Driving Data Set Determination

Choosing driving data sets for each computation is straightforward. Empirically, about 85% of the computations in the initial (before aggregation) system configuration are found to have a only a single candidate. Aggregation further decreases the number of cases where there are multiple candidates. For each of the (few) remaining computations for which a candidate must be chosen, determinations of the total average accesses necessary for each are made (possibly requiring the user to supply information about data set sizes) and the accessing minimizing candidate is chosen.

Step 6: Device, Organization, and Access Method Determination

Some assignments of device and organization for data sets will already have been made by the user. Typically he has specified the devices and organizations for the system inputs and outputs. Reports generally have device PRINTER and so must be consecutively organized. Additionally, driving data sets are constrained to be accessed sequentially by the computations they drive; and so their organizations are either consecutive or index sequential.

Within this context the remaining assignments are made by cases. The optimizer considers one data set at a time and binds its device and organization and the accessing methods of each accessing computation before considering the next. If a data set has no SOC conflicts with its accessors it is given consecutive organization, sequential access, and (unless otherwise specified by the user) device DISK. If it is a system input that is only partially sorted core table access is used if its size permits; otherwise a sorting computation is inserted.

If a data set has SOC conflicts with its accessors and it is core table size, all accessors access it by core table.

If a data set has a SOC conflict and is too big for core table access there are three alternatives:

1. give it consecutive organization, have the compatible computations access it sequentially, and insert sorting computations to produce versions that can be accessed by the rest of the computations.

2. give it index sequential organization, have the compatible computations access it sequentially and the others randomly

3. give it direct organization and have all accessors access it randomly

Each alternative assignment is sent to the job cost estimator and the one that minimizes cost is chosen.

Step 7: Determination of Blocking Factors

Choice of blocking factors can be postponed to the very last step in the optimization process. The cost of accessing as a function of the block size drops initially as block size increases; but then it hits a minimum and begins to rise as core residency charges become significant. However, for the costing scheme we have studied the minimum usually occurs either beyond the operating system's block size limit or so close to it that the core residency does not play a significant role in cost.

So the optimal assignment of blocking factors by minimizing total system cost (as a function of the blocking factor) is determined subject to the constraints that the blocking factor be less than the operating system prescribed upper limit (e.g. no block can be larger than a single disk track) and that the total buffer space at any time cannot exceed the available core.

Experience

While this process will not find the true optimum, it produces a good and usually near-optimal solution for real and honest problems. The reason for this is that, from our experience so far, in typical systems global effects are weak, Step 1 (SOC determination) does a good job of bringing the consequences of these global effects down to a local level, and after that what is "reasonable" to do in a fairly local context is almost always good (if not best) for the overall system design.

The approach described in this paper is wholly within the spirit of state of the art MIS design. The goal is to get a system that works and works well. Designers do not expend inordinate amounts of time and energy to get as close to an optimum as possible unless timing and/or cost constraints are absolutely critical. Although our optimizer lacks the cleverness and special case knowledge of a competent system designer we believe that the indefatigable, meticulous application of sound (though not perfect) optimization rules will more than compensate for this. Even with the few, relatively simple, systems that have been worked on so far, the optimizer has proposed designs which were not at all obvious, but which, under careful scrutiny, proved to be quite good.

References

1. Codd,E.F. A Relational Model of Data for Large Shared Data Banks, *Comm. ACM* 13, 6 (June 1970), 377-387.

2. Early, J. Relational Level Data Structures For Programming Languages, Comp. Sci. Dept., U. of California, Berkeley, 1973.

3. Hammer, M., Howe, W. and Wladawsky, I. An Overview of a Business Definition System, *ACM SIGPLAN Notices*, 9, 4, April 1974.

4. Kornfeld, W. Methodology for Optimization in Automatic Programming Systems, unpublished B.S. thesis, Project MAC, MIT, 1975.

5. Langefors, Borge Somne approaches to the Theory of Information Systems, *BIT*, 3 (1963), pp. 229-254,

6. Langefors, Borge Information System Design Computations Using Generalized Matrix Algebra, *BIT*, 5 (1965), pp. 96-121.

7. Nunamaker, J. F. Jr., Nylin, W. C. Jr., and Konsynski, B. Jr. Processing Systems Optimization tahrough Automatic Design and Reorganization of Program Modules, *Informations Systems* (tou ed.), pp. 311-336, Plenum, 1974.

8. Ruth, G. Internal Memo 16: Status of Protosystem I, Project MAC, MIT, 1975.

9. Ruth, G. Internal Memo 21: The New Question Answerer, Project MAC, 1975.