Copyright © 1982 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 9, November 1982
Printed in U.S.A.

Database Systems:

Issues in the Design of a Distributed Record Management System

By J. P. LINDERMAN

(Manuscript received September 15, 1981)

Inexpensive processors will lead to powerful new tools for constructing software systems. However, the introduction of intelligent hardware may be limited by software architecture. If intelligence has been stripped away in the process of top-down refinement, new devices will be constrained to behave like their unintelligent predecessors. Software design will have to be flexible in distributing intelligence to accommodate processor-based tools gracefully. I discuss a conventional design for a record-retrieval facility. Some features that are acceptable and even attractive in a single-processor environment are shown to limit use of multiple processors and smart peripherals. I propose a less conventional design that can exploit special-purpose hardware and provide a smooth growth path from single-processor systems.

I. INTRODUCTION

The ideas in this paper are a result of my participation in the design of an experimental data-management system. The goal of the design is less to produce a database manager than to produce a system of components from which a database manager, or many database managers, can be constructed. Major components will have several interchangeable implementations exhibiting a range of price and performance. The design team hopes to assemble individual database managers by matching the various components to the requirements of the database system. The intent is to provide variety, not only through a spectrum of algorithms, but also by allowing the components to be distributed over general-purpose or special-purpose hardware.

Section II defines several of the record management functions I wish to provide and outlines a conventional implementation. In Section III, I consider the incorporation of additional processors into the design described in Section II. Section IV offers an alternative design in response to the problems raised in Section III. The advantages of the second design are presented in Section V. A summary in Section VI concludes the paper.

II. A CONVENTIONAL DESIGN FOR A RECORD MANAGER

2.1 Record management functions

What we call a *record manager* is not a database manager. The record manager is meant to provide functions that are indispensable for data management. These functions include

(*i*) Assignment of data to and retrieval of data from secondary storage.

(*ii*) Protection of data from a class of device and software failures.

(*iii*) Support of the notion of transactions with access control to prevent interference between concurrent transactions. (See Ref. 1 for a summary of the important properties of transaction-oriented systems.)

Database designers differ about the merits of relational and hierarchical views and the means for defining and preserving the relationships between a database and the enterprise it models. Few, however, would dispute the importance of surviving a system crash with the database intact. Our record manager will provide the functions most would agree are necessary, thereby allowing the database designer to concentrate on other problems.

2.2 Retrieval

For the purposes of this paper, it is sufficient to consider the design of the data retrieval functions of the record manager. For additional details on our approach to concurrency control and crash recovery, see Ref. 2.

The interface presented by the record manager is intended to

(*i*) make users independent of the physical storage of data, and

(*ii*) make it possible to retrieve many records with a single request.

Data independence makes software more maintainable. If programs do not make implicit assumptions about the physical storage of data, a database administrator is free to reorganize data to optimize global performance. Dealing with records in the aggregate rather than one at a time is often more natural, contributes to data independence, and reduces communication overhead between the user and the (potentially remote) record manager.

2.3 Retrieval requests

A retrieval request contains several components.

(*i*) A *selection expression* that identifies, by content, which records are to be retrieved.

(*ii*) A *sort specification* that determines the order in which records are to be retrieved.

(*iii*) A *projection specification* that identifies those attributes to be returned from the records that are selected.

For example,

```
SELECT NAME=L* & RATING=INCOMP & SALARY>30000
SORT SALARY{r}, NAME
PROJECT NAME, SALARY, DEPARTMENT
```

might be a request to retrieve the values of the NAME, SALARY, and DEPARTMENT fields for all incompetent employees whose NAME begins with letter L and whose SALARY is greater than $30,000. The records retrieved are to be sorted in reverse order of SALARY, and on NAME for groups of records with the same SALARY.

2.4 Designing the retrieval function

I can now outline the design of the retrieval function and discuss the major components. Figure 1 shows a possible design. I will refer to this as the conventional design. In this design, a retrieval-manager module orchestrates the retrieval process. High-level operations are performed by the record manager, using three submodules to carry out basic operations. The next three subsections describe the interaction between the retrieval manager and the submodules in the course of satisfying a retrieval request.

2.4.1 Page storage operations

Assume that secondary memory is divided into pages of a fixed size. The record manager gains access to secondary memory using a few simple page-storage operations.

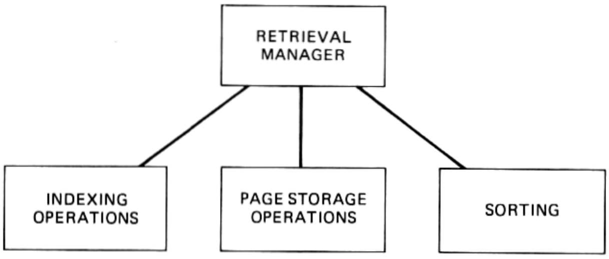

Fig. 1—Conventional design.

1. Begin a transaction. To control concurrent access, all references to stored records must be associated with a transaction.

2. Commit or abort a transaction.

3. Allocate a page of secondary memory. I will assume that each page is assigned a simple, numerical *page id* when it is allocated. This id will be used to reference the page in read and write operations.

4. Write a page of secondary memory, given its page id and contents.

5. Read a page from secondary memory, given its page id.

Since the retrieval manager does not modify records, it only requires the transaction start, read, and commit operations. In our system, concurrent access is controlled by the page storage manager.

The retrieval manager has the responsibility for understanding how records are packaged into pages and for determining the whereabouts of individual records. The page-storage operations deal with pages as featureless chunks of data.

2.4.2 Indexing operations

To speed retrieval, the record-management system will map the values of certain *indexed* fields onto the records containing those values. For example, if NAME is indexed, for each different value of the NAME field, the record manager will keep track of the record or list of records containing the value. A convenient way to identify a record is to combine the page id of the page in which the record is stored with the offset of the record in the page. This compact, numerical quantity can be used by the retrieval manager to fetch the appropriate page (using the page-storage operations) and then to locate the record in it.

A collection of records can be identified by a list of these *record id*s. If the lists are kept in numerical order by record id, the indexing operations become very simple.

1. Open the list associated with the given value of a given field for reading or writing.

2. Get the next record id from a list opened for input.

3. Put the next record id into a list opened for output.

4. Close an opened list.

Assuming the NAME and RATING fields are indexed, the retrieval manager could find the records for incompetents whose names begin with L in the following manner.

1. Merge all the lists for NAMEs beginning with L.

2. Intersect this master list with the list having incompetent RATINGs.

The resulting list of records could be retrieved and further selection, say on the basis of SALARY, could be performed by the retrieval manager.

It should be noted that the indexing operations described in this example, although correct, may be ill-advised. The increasing order of record ids within lists makes it possible to merge or intersect two lists with only a single pass through each list. Still, pairwise merging of a large number of NAME lists might require much more effort than simply retrieving all records for incompetents and doing further selection for SALARY and NAME on the records themselves.

2.4.3 Projection and sorting

The previous sections have suggested how the retrieval manager can use the indexing operations and page-storage operations to retrieve the records that satisfy a selection expression. With the records in hand, it is a simple matter to build a sort key that will produce the specified sort sequence, to strip out unrequested fields, and to hand the records and sort keys to a general-purpose sort utility. The retrieval manager can return the sorted records to satisfy the original retrieval request.

2.5 Summary

Although much of the rest of the paper will be devoted to discussing the flaws in the design that has just been presented, the design has several attractive features that deserve attention.

(*i*) The design is tool oriented. For example, the sort routine employed by the retrieval manager is likely to exist on many systems, eliminating the need to write and maintain a major component of the design. It is less likely that a transaction-oriented page-storage manager can be picked up off the shelf, but once it is written, it can be used in other applications, such as a transaction-oriented editor. The indexing operations will use the page-storage module for reliable storage of the index data. The design therefore takes advantage of existing tools and provides useful new tools for other applications.

(*ii*) The simple interfaces make it easy to replace components. Because the indexing operations are so primitive, it is possible to support several different index implementations. For example, one simple implementation stores the record ids as an ordered array. However, it is often possible to save space by storing the *difference* between record ids instead of the ids themselves, since the ids might occupy 32 bits or more, while the difference between two ids might fit in 16 bits or less. If record ids in the lists are numerically close, it may save even more space to represent the lists as bit strings in which bit n is on if and only if the record with id n is in the list. The most economical representation depends on properties of the lists, but all three representations can coexist because it is easy to support the open/close/get/put operations that the retrieval manager requires.

(*iii*) Information is localized. For example, the syntax and semantics of the selection expression are known only to the retrieval manager, so changes would have no effect on the other modules.

These are all important features for flexible, maintainable systems. We will want to monitor the effects of proposed design changes to make sure the features are not lost without adequate compensation.

III. THE NEXT STEP IN SOFTWARE DESIGN EVOLUTION

3.1 Cheap processors

Two principles have been in effect while software design methods have evolved.

(*i*) Processors were expensive.

(*ii*) Peripheral devices had limited capabilities.

A single processor was at the bottom of most software design, so there was little pressure for high-level languages and operating systems to address issues of multiprocessing. Since peripheral devices were scarce, shared resources with limited capabilities, isolating them from designers by an operating system, a high-level language, and several levels of utility libraries seemed like a service rather than a flaw.

The price of processors has had twenty years to shape the tools through which we use those processors. In a few years, the price of processors will be negligible compared with the expense of designing the software to use them. We cannot expect the design methods based on the old processor economics to apply, without change, to the exploitation of cheap processors.

3.2 Content addressable storage devices

As an example of the problems of integrating the new economics with the old design methods, consider secondary storage devices. If the price of a disk storage unit dwarfs the price of a processor, one can produce a "smart disk" for the same price as a conventional one. Such a device could

(*i*) Package variable length records into fixed length pages.

(*ii*) Look through the records in a page and retrieve only those records containing specified values of certain fields.

(*iii*) Reformat records to eliminate unwanted fields.

These are operations our record manager must perform, so it would appear that the device should fit in nicely with our retrieval system. Unfortunately, with the design that has been presented, this is not the case. The problem is that the device replaces a conventional disk, and would therefore be under control of the page-storage module. However, the responsibility (and the information necessary) for carrying out record packaging, selection, and projection resides in the retrieval-manager module. To take advantage of the smart disk, the design must change.

3.3 Another look at the problem

The weakness in the original design is not limited to accommodating smart disks. A similar problem exists in taking advantage of hardware support for indexing operations. The hardware should reside next to the indexes, but the operations are performed by the retrieval manager.

The availability of special-purpose processors presents a real problem to a software designer. Anything I do in a combination of software and hardware can, at least in principle, be done in hardware alone. If I implement a function in software, I lose the advantage of having it carried out by special-purpose hardware. On the other hand, if I insist that a function be performed by hardware, I tie my design to the existence of that hardware. What I really want is a design that can take advantage of hardware capabilities if they are available and attractive, but that does not depend on the presence of those capabilities.

The problem with the retrieval-system design presented earlier is that it committed the solution to software by withholding important information at the level of the retrieval manager. By building the retrieval manager on top of simpler abstractions that did not *need* to know what operations were to be performed on the indexes and records, I ruled out a solution in which the operations *could* be done in hardware.

This is not to say that problems cannot be decomposed into smaller problems. It appears that this is the *only* approach that will keep software intellectually manageable. However, designers must break the habit of assuming that all modules draw on the same processor and are thereby limited in what they can accomplish. In the next section, I will redesign the retrieval function to make it possible to take advantage of special-purpose hardware without requiring that it be present.

IV. A RETRIEVAL DESIGN THAT ANTICIPATES INEXPENSIVE PROCESSORS

4.1 Ask much—demand little—withhold nothing

Let us redesign the retrieval function introduced in Section II but avoid the assumption that a single processor must support all the modules in our design. We can assume that each module has at least a share of a conventional processor underlying it, but a module may also have a dedicated, special-purpose processor supporting it. In this design environment, the capabilities of a module are bounded from below by current software design techniques but limited from above only by one's willingness and ability to employ hardware. Stated differently, it is much safer to define what such a module can do (anything that a share of a conventional processor can now do) than to define what it cannot do.

A good rule for design with these modules is

(*i*) To ask much (since the module can do no more than we ask),

(*ii*) To demand little (since we may have to rely on conventional implementation techniques), and

(*iii*) To withhold nothing (so the module has the wherewithal to do what we ask).

The little that I demand is what I call the *primary function* of the module. By asking a module to do everything that remains to be done, and giving it access to everything that has already been done, I can be sure that I have not artificially limited its ability to contribute to the solution.

4.2 A revised retrieval system design

The retrieval function of the record manager (see Fig. 2) can be redesigned in accordance with the new principles as follows.

I will refer to this as the *distributed* design to distinguish it from the conventional design already presented and to emphasize that the modules in the design may be distributed over more than one processor.

Initially, of course, the retrieval request is everything that remains to be done. By turning the entire request over to the indexing module, I can be sure I have not withheld any critical information. In general, the request together with the results from the previous module will be made available to the next module as a statement of what has already been done and what remains to be done. A brief description of the primary function of each of the modules should help clarify the design.

4.2.1 Index module

The primary function of the index module is to identify a superset of the records that satisfy the selection expression in the retrieval request.

It would be convenient if the index module identified precisely those records that satisfied the expression, but this would be demanding too much. For one thing, an expression may involve fields that are not indexed. Returning to the example of

```
SELECT NAME=L* & RATING=INCOMP & SALARY>30000
```

assume NAME and RATING are indexed but SALARY is not. The indexed fields can be used to identify all records that might satisfy the expression, but the question of SALARY cannot be decided until the

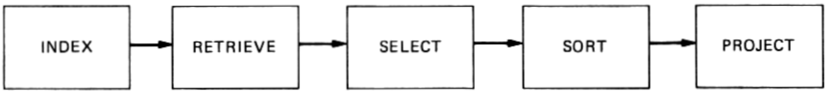

Fig. 2—Distributed design.

records are accessed. As was pointed out earlier, it might be better to ignore the NAME information, which will require considerable effort to extract, and simply use the RATING field to identify the superset. Demanding too much precision from the indexing module would deny us these important optimizations.

The index module can perform its primary function by ignoring the request and identifying all records. This observation is not entirely facetious. Such a module is easy to implement and provides a functionally correct module for testing. For small collections of records, this simple approach may be all that is necessary to provide adequate performance.

4.2.2 Retrieval module

The primary function of the retrieval module is to retrieve a superset of the records that satisfy the selection expression.

The retrieval module will have access to the complete retrieval request and to the results of the index module. If the retrieval module does no more than retrieve all the records identified by the index module, it will perform its primary function. However, it is also free to eliminate any records the index module identified that fail to satisfy the selection expression.

4.2.3 Selection module

The primary function of the selection module is to evaluate the selection expression on each record passed to it, and to pass along only those records satisfying the expression.

If the index, retrieval, and selection modules correctly perform their primary functions, the output of the selection module will always be the records that satisfy the selection expression.

4.2.4 Sort module

The primary function of the sort module is to put the records passed to it into an order compatible with the sort specification of the retrieval request.

4.2.5 Projection module

The primary function of the projection module is to eliminate unrequested fields from the records passed to it.

V. A COMPARISON OF THE TWO DESIGNS

The two retrieval system designs presented in this paper perform the same functions, but in distinctly different ways. Both designs permit us to hide *how* functions are performed, but in the conventional design, we know precisely *where* each function is performed, while the distributed design hides this information as well. The importance of hiding this information is best seen by considering how changes can be introduced into the designs.

5.1 Getting a conventional start

Since conventional, single-processor systems are likely to be with us for some years, we begin with the question of implementing the alternative designs on a single processor.

At first glance, the conventional design seems to have the advantage, since it is constructed from simple components that may already exist or that contribute to the collection of software tools. The indexing and retrieval modules in the distributed design are not likely to exist and not likely to contribute to unrelated applications.

This criticism is only skin deep. The distributed design can be realized by repackaging the same general-purpose modules that go into a conventional design. For example, the indexing module in the distributed design can use the same index operations found in the conventional design. The major difference is that the selection expression must now be processed in two places, once in the index module and again in the selection module. This doesn't mean additional software—one has the subroutines to process the expressions in both designs—it means the routines are no longer localized as nicely. In exchange for this loss, one gains the genuinely useful selection module that does not appear in the conventional design.

In a single-processor implementation of the distributed design, only the primary function of each module would be implemented. This affords the same simplification of the software as does the conventional design. An important distinction is that the distributed modules are made simpler by consciously choosing to ignore information available to them, while the conventional modules are constrained to be simple because they lack the information to be profound.

5.2 Making conventional improvements

To improve the conventional design, there are two primary approaches—build on a more powerful processor or improve the algorithms. These approaches are equally applicable to a single-processor implementation of the distributed design.

There are some algorithmic improvements that are easier to make in the distributed design. For example, two indexes represented as bit strings can be intersected or merged efficiently using common logical operations. In the distributed design, the representation and operations are contained in a single module, so the change could be introduced without affecting other modules. In the conventional design, the representation is hidden in one module and the operations performed by another. Both would have to be modified to support the proposed change.

5.3 *Adding conventional processors*

When a conventional design uses the biggest processor, the best algorithms, and the fastest secondary storage devices, there is nowhere else to go. Sorting and projecting might be offloaded to a separate processor, but the processing bottleneck in the conventional design is likely to be the retrieval manager module. It is responsible for much of the work in processing a request, so even if all the other modules can be removed to other processors without penalty, the retrieval manager will continue to limit performance. Given the level of interaction between the retrieval manager and the indexing and page-storage modules, it is unreasonable to expect that an inexpensive separation could be made.

The distributed design fares much better. Since modules interact much less than in the conventional design, transporting one or more modules to separate processors need not introduce excessive interprocessor overhead. It therefore becomes a realistic possibility to move the busiest module onto a processor of its own.

Further tuning is made possible by the open-ended assignment of functions to modules. If retrieval is the bottleneck, more thorough indexing can be done to reduce the number of records that must be retrieved. If selection is the bottleneck, some of the selection can be carried out by the indexing and retrieval modules.

5.4 *Special-purpose hardware*

The processors involved in the distributed design need not be conventional. Special-purpose processors such as content addressable storage devices fit nicely into the design. Since it has the selection and projection expressions as well as the identifiers provided by the indexing module, the retrieval module can lend a hand with other functions as it retrieves records. *Any* extra work the selection module can perform is just that much less that needs doing later. If a smart disk can do the entire job of selection, the retrieval request can be modified to indicate that no further selection is necessary, and, with suitable interprocess primitives, the selection module can drop completely out of the path.

For a software designer, this is a delightful state of affairs. The distributed design does not call for any special capabilities from peripheral devices, but the chances are good that it can exploit whatever useful features are available.

VI. SUMMARY AND CONCLUSIONS

This paper suggests that if inexpensive processors and smart peripherals are the wave of the future, software design must change in two related areas.

Software design has often proceeded by stripping away "intelligence" at each level of abstraction. In the case at hand, a logical record manager that knows about the contents of records employs a physical record manager that knows how to pack objects of variable length into pages of fixed length. The physical record manager uses standard library routines to move the pages between primary and secondary memory. At each refinement, there is less information about what is being manipulated. This is fine until we attempt to introduce "intelligent" devices at the bottom of the design hierarchy, and discover that they are several levels removed from the information they require. If intelligence is to be allowed in from below, more intelligence must propagate down from above.

The distinction between the function of a module and the primary function of a module is a multiprocessor phenomenon. On a single processor, the idea that the same function might be carried out in several different modules would be viewed as fuzzy thinking, at best, and a profligate use of instruction space. In a multiprocessor implementation, it is a reasonable hedge against the uncertainty of where additional processor support may appear and the capabilities of that additional support.

The principles that led to the distributed design are not limited to database applications, but they are not completely general. The retrieval request is a concise summary of everything that must be accomplished. The effort of passing the request from module to module and of locating information of interest within the request is negligible in comparison to the work carried out by each module. If an application lacked such a convenient way of communicating the big picture, the distributed approach would be less appealing.

VII. ACKNOWLEDGMENTS

Bill Roome, Dan Fishman and Rudd Canaday have been reliable sources of support and insight. Thanks also to my colleagues and the referees whose suggestions substantially improved the form and content of this paper.

REFERENCES

1. M. J. Rochkind, "Structure of a *UNIX* Database File System," B.S.T.J., this issue.
2. W. D. Roome, "A Content-Addressable Intelligent Store," B.S.T.J., this issue.