EDITING REQUIREMENTS FOR DATA BASE APPLICATIONS AND THEIR IMPLEMENTATION ON THE INDY BACKEND KERNEL

Allen J. Otis
George P. Copeland
Tektronix, Inc.
Beaverton, Oregon 97077

ABSTRACT

An extrapolation of user and hardware cost trends indicate that future systems should provide more complete functionality, simplicity of use, and reliability by increasing the amount of hardware present in the system. For data base systems, this can be done by providing very high level data languages and by implementing them directly in hardware. These goals are realized with a simple hardware arrangement called INDY, which uses inexpensive memory technologies (such as charge coupled devices, magnetic bubbles, or discs). This paper first discusses the data definition and editing requirements of data languages. Then the implementation of these requirements on the INDY backend kernel is described. Finally, a comparison is made of the various memory technologies for suitability to INDY.

1 Introduction

The rate at which hardware function costs are decreasing is drastic. No other industry has experienced such a sustained and incredible reduction in cost per function. However, total system cost involves many other costs: the amortized manufacturer's hardware and software development cost; user software development to provide functionality that the manufacturer did not provide; tuning the system to the working environment; daily operating costs, such as personnel training and salaries; software and hardware maintenance costs; and many opportunity losses resulting from the time delays caused by each of the above. The rapid decrease in hardware function cost has significantly contributed to a reduction in total system cost. However, this alone has a diminishing effect since many other costs are involved. Periodically, a rebalancing of these costs is required to effectively reduce total system cost.

This rebalancing can be done by making several improvements in future systems. System functionality and user simplicity must be simultaneously increased. This will allow more applications to be reached, because the costs and delays involved in putting systems to practical use are reduced. Also, system reliability must be increased. This will allow more complex systems to be applied to more critical functions and will reduce system maintenance costs. Reliability is an increasingly critical economic and social issue, since we are expecting the system to do more for us than ever before and, at the same time, we must depend on it more than ever before.

Unfortunately, providing more complete and easy-to-use functionality with a strictly software approach results in increased system complexity, which increases the system cost and development time and decreases reliability (Brooks 1975, many others too numerous to list). But implementing existing software techniques directly in hardware does not address the basic problem of total system complexity. Hardware has similar complexity limitations. The consideration of hardware flexibility in addition to software provides much more freedom to simplify the total system. If the basic hardware functions are closely matched to the basic operations of the user language, then software development is greatly simplified. Furthermore, if simple hardware arrangements can be found that directly implement these language operations, the overall system is simplified.

More complete and easy-to-use functionality in data base systems can be achieved by providing very high level data languages. Section 2 discusses the functionality that is strived for by all such languages, laying the foundation for the functionality required of a data base kernel.

System simplicity can be achieved by a direct implementation of the full functionality required by these languages. This approach makes more feasible the task of developing a compiler or an interpreter. Section 3 briefly describes the INDY backend kernel (Copeland 1978a), which provides an implementation of this language functionality directly in hardware with simple and generalized algorithms. A storage structure is described using inexpensive segmented serial memories, such as charge coupled devices, magnetic bubbles, and discs. The storage structure directly implements the data definition requirements discussed in Section 2. Section 4 describes a direct implementation of the editing requirements discussed in Section 2, using the storage structure described in Section 3. Section 5 compares tape, disc, magnetic bubbles, and charge coupled devices for their suitability to INDY.

2 Language requirements for a data base kernel

The language facility of a data base kernel should be generalized enough to support any user data model. The following two goals are common among all data models. First, a capability to define the semantic integrity constraints of the data should be provided that is generalized enough to allow adequate modeling (abstraction and normalization). The semantic integrity constraints (or semantic model) give meaning to the data. Any data that violates the semantic integrity constraints is therefore excluded from the data base, since it has no meaning. Secondly, a query language should be provided that is flexible enough to allow any question to be asked (data independence). These issues are briefly discussed in this section. A more detailed discussion of data base kernel language requirements can be found in Copeland (1978b, 1978c).

2.1 Semantic modeling

The data base of an enterprise (for example, a business unit, a government agency, or other organization) is a model of the information required to operate that enterprise. As in any modeling process, the enterprise must be understood well enough to include only those characteristics that describe the most significant aspects of the enterprise, excluding those characteristics that are least important. Accountants must design the information system to include whatever information is necessary to report the status and progress of the enterprise for decision makers and for government control. Accountants must also incorporate a system of internal control into the information system to insure that clerical errors can be found and that cheating can be prevented through a simple auditing procedure. Systems analysts must design the information system so that data entry and data processing systems can be both cost-effective and functionally adequate. The resulting model has well defined semantic integrity constraints, including natural and well defined units of information for each basic editing transaction.

The semantic model should simplify the task of maintaining the semantic integrity of the data base. This simplicity is best achieved if the information is partitioned into units that directly match the units of editing. This model, which includes both abstraction and normalization, must insure that the editing of one portion of the data does not force the editing of other portions of the data when this is not desired. Also, the model must insure that editing of one portion of the data does force the editing of other portions of the data when this is desired. Thus, the semantic modeling facilities of a data base language should be powerful enough to allow adequate modeling, and the data storage facilities of a data base system should directly support these powerful semantic models.

2.2 Data independence

Data independence requires that data representation and storage place no restrictions on (be independent of) what can be done with the data. The semantic model should not be based on any particular question. Instead, it should directly facilitate maintenance of the semantic integrity constraints of the data through editing. The extreme danger of destroying the integrity of the data by insertion and deletion dictates that editing should be given priority over querying when modeling the data. To insure data independence, a query language must be provided that allows any question to be asked of this model, even though the model was not tailored for any specific question.

2.3 Proposed data languages

Codd's (1970, 1971a) Third Normal Form relational model was a significant step toward an adequate modeling tool. Later, the Boyce-Codd Normal Form (Codd 1974) simplified the understanding of normalization. Fagin (1977) introduced a Fourth Normal Form based on multivalued dependencies, which further improved the understanding of the normalization process. Then Makinouchi (1977) described a Normal Form based on a modified definition of Fagin's multivalued dependency, which can be achieved if embedded relations are used. Embedded relations allow a one-to-many dependency to be directly represented. Multivalued dependencies can be removed from a relation by creating an embedded relation containing the repeating group. This has three advantages. Redundancies are not required to represent a one-to-many dependency. Secondly, several one-to-many dependencies based on the same key can be represented in the same relation without causing insertion and deletion anomalies that can destroy the semantic integrity of the data base. Thirdly, embedded relations improve the implementation efficiency (Copeland 1978c). Reducing unnecessary redundancies has the obvious advantage of increasing storage utilization. Also query implementation speed is greatly decreased when information is unnecessarily broken apart into separate relations.

Smith and Smith (1977a-b) and others argued that normalization alone is not adequate for modeling the semantics of data. Examples of properly normalized models were cited which had no real-world-meaning. Hierarchical classification was suggested as a first step in the modeling process to allow the model to correspond more closely to the real world. Then normalization was suggested as a second step to remove insertion and deletion anomalies. Both aggregation and generalization abstractions are used together to define the semantic model of the data.

Various user-oriented query languages have been proposed for these information models: for example, the predicate calculus based ALPHA (Codd 1971b), SQUARE (Boyce et al. 1973), SLICK (Copeland and Su 1974), SEQUEL (Chamberlin and Boyce 1974), and Query-By-Example (Zloof 1975,

1976a, 1976b). Of these, only SLICK and Query-By-Example can query information defined by embedded relations, as well as relations without embedding. In addition, Query-By-Example offers a more user-oriented human interface (Thomas and Gould 1975) and more complete querying capability (Zloof 1976b).

2.4 The OBJECT language as a kernel

The importance of character strings for defining abstract data types for data base applications is described by Copeland (1978). A language called STRING was introduced which provides semantic data modeling at the character string level and allows specification of patterns within the strings for querying. STRING is intended as a powerful extention to the data languages described above, which define the semantic relationships between strings, and whose querying facilities can only conveniently treat strings as whole units. STRING definition allows items to be represented as variable-length character strings.

The OBJECT language described by Copeland (1978c) uses strings and classical sets together to define the semantic integrity constraints, as well as any question asked of the data. OBJECT uses STRING to define the semantic integrity constraints of strings and to specify string pattern membership for querying and editing. OBJECT uses classical set membership conditions to define the semantic integrity constraints that relate strings to one another and to define queries and editing transactions. The methods used in Query-By-Example for expressing set membership are used by OBJECT to enhance ease of use. OBJECT makes no arbitrary distinction among strings used as names of relations, names of attributes, names of values, names of objects, or names of classes. Each string can potentially be used to describe the name of a set of sets of strings, allowing a generic of aggregates as suggested by Smith and Smith. Thus, no arbitrary distinction need be made between generalizations and aggregations, since both are provided for each string. This approach allows users to dynamically make these distinctions, so that various user views can easily be superimposed upon the more objective and generalized OBJECT view. Because of its generality, OBJECT can easily describe the view imposed by the various user models, including the embedded relational model, described previously. Thus, OBJECT is an excellent candidate for a kernel language.

The simplified employee personnel file of Figure 1 is used to illustrate modeling and editing using OBJECT. The semantic integrity constraints can easily be expressed as follows:

```
employee {name {$ $},
    salary-history {date{$/$/$},
                    salary {$k}},
    children {name{$},
              birthdate {$/$/$}}}.
```

Each employee can be described by three strings: name, salary history, and children. Each salary history can be described by date

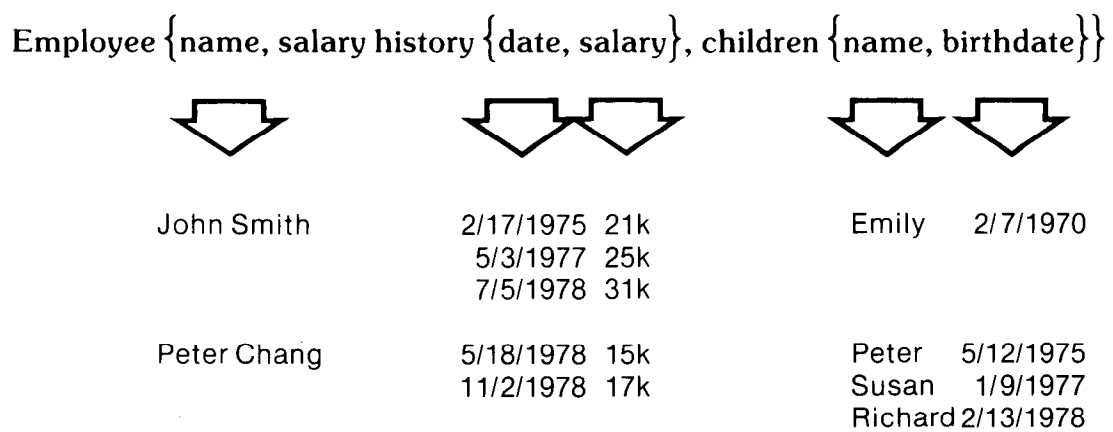


Figure 1. Example data base.

and salary. Each child can be described by name and birthdate. These descriptions involve an exhaustive listing of strings in this semantic model. The remaining descriptions involve string abstractions. For example, each employee name is described by two substrings of arbitrary length (indicated by $) and separated by a blank. Each date has three substrings separated by slashes. Any entry into this data base is constrained to follow these patterns. Otherwise, the semantics of the entry cannot be understood.

Various insertions and deletions can be made independently of one another, except when constrained by the semantic model. This integrity is insured because the semantic model is used as a template for specifying editing. For example, if the employee, John Smith, is given a raise on 3/18/79 to $33K, the required insertion can be specified by

```
employee {name{John Smith},
    salary-history i.{date{3/18/1979},
                      salary {33K}}}.
```

The i. indicates that the date and salary are inserted into the embedded salary history description for the employee named John Smith. This insertion is made independently of the description of the employee's children.

If John Smith leaves the company, all information concerning him can be deleted from the personnel information by specifying

```
employee d.{name{John Smith}}.
```

The d. indicates that the entire employee entry for John Smith is deleted.

3 INDY Backend storage system

The INDY backend storage system provides direct hardware implementation of the data base kernel requirements. INDY is based on an associative memory cell developed as a data base version of the CASSM cell (Su et al. 1973, Copeland et al. 1973, Copeland 1974). Use of this array of cells allows a direct logical to physical mapping of the OBJECT model and a direct implementation of searching and editing operations.

This section briefly describes the system architecture which has evolved from earlier work on associative searching and shows how objects (strings and sets) can be mapped onto the physical storage. Finally, editing requirements are used to further specify the memory architecture within the associative cell.

3.1 Storage system architecture

3.1.1 Requirements

The architecture of INDY is designed to meet three primary requirements of a data base system. First, a structure for describing, storing, and editing information is provided which is generalized enough to allow adequate modeling. This makes more feasible the task of preserving the semantic integrity of the data base while editing. Secondly, a set of searching operations is provided which directly supports the OBJECT query language capabilities This feature allows data independence to be maximized. Thirdly, economical storage media are exploited, and the architecture allows simple control structure so that hardware implementation and language interface are made simple and reliable.

A simple implementation of maximum data independence requires an exhaustive search capability as discussed in Copeland (1978a). The exhaustive search must operate efficiently on a serial memory, since economical storage requires use of memory technologies which are serially organized within the basic memory element. Examples of this class of memories include magnetic and optical tapes and discs, charge coupled devices and magnetic bubbles.

These technologies do not allow efficient random accessing at the bit level since the physical storage elements must be circulated past an access point to read or write information. Because of this limitation, they are generally classified as slow access. However, this is only true when such technologies are used to emulate a random access memory. When used to implement an exhaustive search, they are more efficient than a truly random access memory. Access speed is determined only by their shift rate and the number of points of simultaneous access. Furthermore, Section 4 shows how the dynamic nature of these segmented circulating devices allows a much more efficient implementation of data base editing requirements (without the use of pointers) than is possible on randomly addressed devices. Thus, these memory technologies are fully exploited when used to implement an exhaustive search for improved data independence.

Efficient hardware implementation and simplification of control structures are best achieved using a one-dimensional array of hardware storage elements, as opposed to two-dimensional or higher order arrays (Copeland 1974, 1978a).

3.1.2 Implementation

INDY is configured as a linear array of storage cells connected to a controller, as shown in Figure 2. Each cell contains a pipelined associative search processor and one or more segments of serial memory. Both the common bus and the intercell bus are serial busses. The common bus is used to transfer data between the controller and the cells, and instructions from the controller to the cell processors. The intercell bus is used for communications between cells while searching items or structures that overlap across cell boundaries, and for arbitration

of access of the cells to the common bus. Within each cell, the processor and memory are connected as shown in Figure 3.

This hardware architecture is described further in Copeland (1974 and 1978a). The cell interconnection of Figure 2 provides a one-dimensional hardware structure. Exhaustive search on a serial memory is provided by the associative processor in each cell. This search capability includes the ability to handle variable-length objects. The serial memory may be implemented using a variety of technologies, provided separate read and write ports are available on each segment as shown in Figure 3.

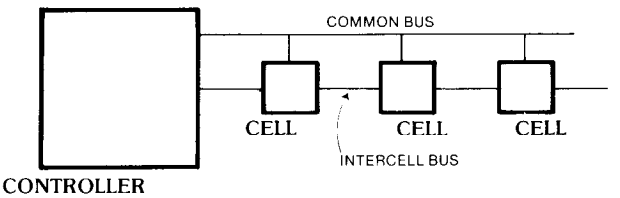


Figure 2. INDY storage array.

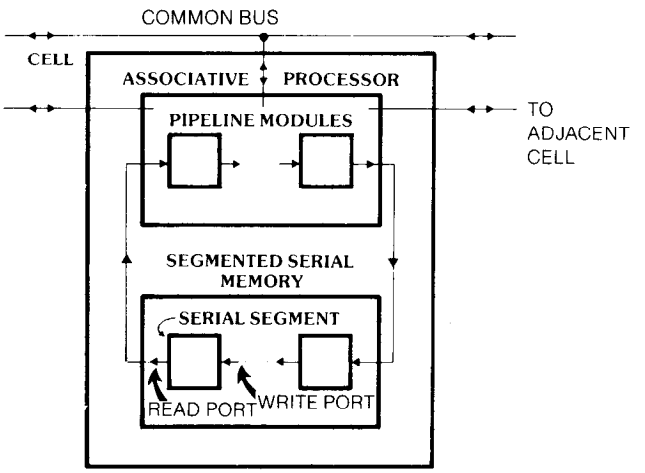


Figure 3. A cell.

3.2 Data storage model

Mapping the embedded objects (strings and sets) onto the INDY hardware makes use of a linearized encoding scheme (Copeland et al. 1973, Copeland 1974) which directly stores the embedded objects in the serial memory. This transformation is compatible with the INDY hardware and provides a conformable mapping as defined by Date and Hopewell (1971).

In transforming the logical data base of Figure 1 to a linearized variable-length form, both the embedding of sets and the string boundaries must be preserved. Sets of sets can be preserved by assigning level numbers to the objects as shown in Figure 4. To store this encoded variable-length data on a serial

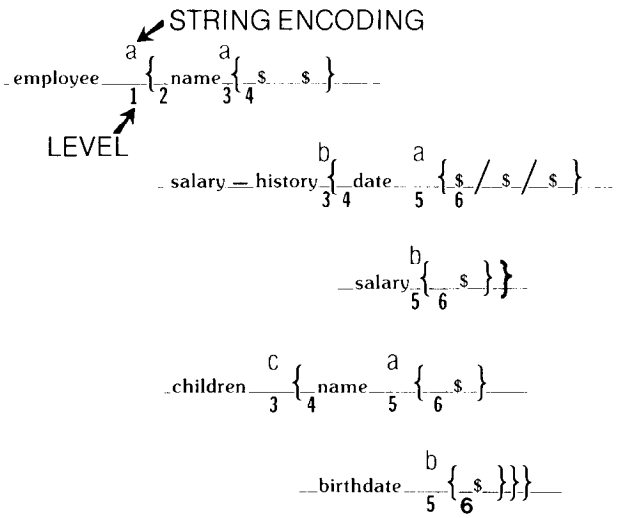


Figure 4. Level assignment and string encoding.

memory, a unique delimiter character Ø is added to identify the beginning of each object. The contents of the example data base can be represented in storage in the same form as Figure 1, but with a two character (delimiter and level) prefix on each object. This prefix is the structural encoding. Furthermore, whenever an exhaustive listing of strings occurs in the semantic model, the strings can be encoded into smaller strings as shown in Figure 4. Figure 5 shows how the example data base is stored on the INDY hardware using the (delimiter, level, and string) encoding. This mapping from the logical data base to the linearized storage model of Figure 5 preserves all associations between data items, and does not introduce any other associations. Therefore, it is a conformable mapping (Date and Hopewell 1971) which preserves the data independence and generalized querying properties of the OBJECT model.

The absence of pointers or indexes in this data storage model are both a result of and a requirement for the generality of the associative search technique used by the cell processor. If the structural information (delimiter, level, and string encoding) is considered a part of the information to be stored, then this storage technique reduces overhead (in the form of duplicate data and pointers), while providing much more generalized accessing capabilities than those feasible with indexes and inverted file structures.

4 Editing on the INDY storage system

In this section, the requirements for editing the linearized storage model are presented and a memory architecture is developed to satisfy them. Examples of insertion and deletion are presented using the data base of Figure 1, the encoding of Figure 4, and the storage structure of Figure 5. Expressions for time and storage utilization of the editing operations are developed.

Cell	Contents			
1	Ð 1 a Ð 2 Ð 3 a	Ð 4 J o h n S	m i t h Ð 3 b Ð	4 Ð 5 a Ð 6 2 /
2	1 7 / 1 9 7 5 Ð	5 b Ð 6 2 1 K Ð	4 Ð 5 a Ð 6 5 /	3 / 1 9 7 7 Ð 5
3	b Ð 6 2 5 K Ð 4	Ð 5 a Ð 6 7 / 5	/ 1 9 7 8 Ð 5 b	Ð 6 3 1 K Ð 3 c
4	Ð 4 Ð 5 a Ð 6 E	m i l y Ð 5 b Ð	6 2 / 7 / 1 9 7	0 Ð 2 Ð 3 a Ð 4
5	P e t e r C h	a n g Ð 3 b Ð 4	Ð 5 a Ð 6 5 / 1	8 / 1 9 7 8 Ð 5
3	b Ð 6 1 5 K Ð 4	Ð 5 a Ð 6 1 1 /	2 / 1 9 7 8 Ð 5	b Ð 6 1 7 K Ð 3
7	c Ð 4 Ð 5 a Ð 6	P e t e r Ð 5 b	Ð 6 5 / 1 2 / 1	9 7 5 Ð 4 Ð 5 a
8	Ð 6 S u s a n Ð	5 b Ð 6 1 / 9 /	1 9 7 7 Ð 4 Ð 5	a Ð 6 R i c h a
9	r d Ð 5 b Ð 6 2	/ 1 3 / 1 9 7 8	Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ	Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ
10	Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ	Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ	Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ	Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ
⋮				

Ð = DELIMITER

Ⱥ = AVAILABLE SPACE

Figure 5. Linearized storage of example data base, using 8 characters per segment and 4 segments per cell.

4.1 Additional hardware for editing

4.1.1 Requirements

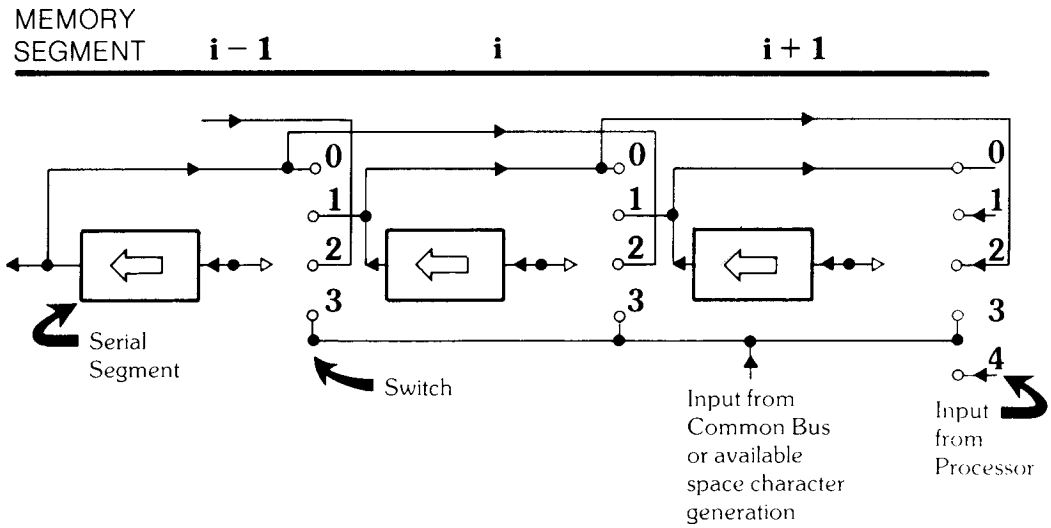
The two basic operations required for editing are the insertion of information at a specific point in the embedded structure and deletion of a specific structure in the existing data base. The linearized (and pointer free) storage structure of Figure 5 requires that insertion and deletion be directly implemented in-place on the memory. This means free space must be physically created at the point of insertion, after this point has been marked by an associative search, and deleted data must be physically erased from the linear storage.

Since the INDY hardware uses a segmented serial memory, insertion can be accomplished by shifting the memory before and after the insertion point by different amounts to create free space at the desired point. Deletion can be accomplished by the inverse of this process.

4.1.2 Implementation

Adding insertion and deletion capability to the serial memory is accomplished by partitioning the memory into segments and interconnecting the segments with four-position switches as shown in Figure 6. The length of the segments and number of segments per cell are implementation parameters of the INDY system. In general, at least one segment is required per cell, and the segments in adjoining cells are chained together using the intercell bus. Figure 7 shows the four modes of operation of the segmented memory.

The segmented memory provides insertion and deletion at segment boundaries. Control of the insertion by the processor in the cell containing the insertion point allows insertion to begin between segment boundaries. Using the segment switches, storage collection is possible in units of one segment. Free space increments of less than one segment can be recovered by the controller at data base back-up or during paging when the INDY system is used as a staging device for very large storage devices.



SWITCH POSITION	SEGMENT **i** INPUT	SEGMENT MODE
0	**i output**	**segment recirculate**
1	**i + 1 output**	**search circulation; collection**
2	**i − 1 output**	**shift down for insertion**
3	**external input**	**insertion into segment i**
4	**processor output**	**search circulation, at cell boundary**

Note: Data storage within a segment is left to right.

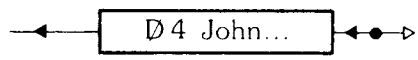
example: 

Figure 6. Memory segment interconnection.

4.2 Editing examples

Implementation of the editing examples of section 2.4 is shown here using the segmented memory of Figure 6 and the linearized storage model of Figure 5. For illustration, the cells of Figure 5 are each divided into four segments of eight characters, and the segments are consecutively numbered starting with 1 at the beginning of the data base. The segments are not addressable by the controller. Instead, segments are identified by associative searching of the data, and each search processor can control the switching of segments within its cell.

The insertion example of Figure 8 shows how an insertion is begun between segment boundaries, by inserting partial segments of data and available space characters. This allows the remainder of the insertion to be performed at segment boundaries, by partitioning the segments into two groups, recirculating in the upper group and shifting down one segment in the lower group, with new data entering at the partition point. This method requires that enough available space exist at the end of the data base so that valid data is not shifted off the end of the array.

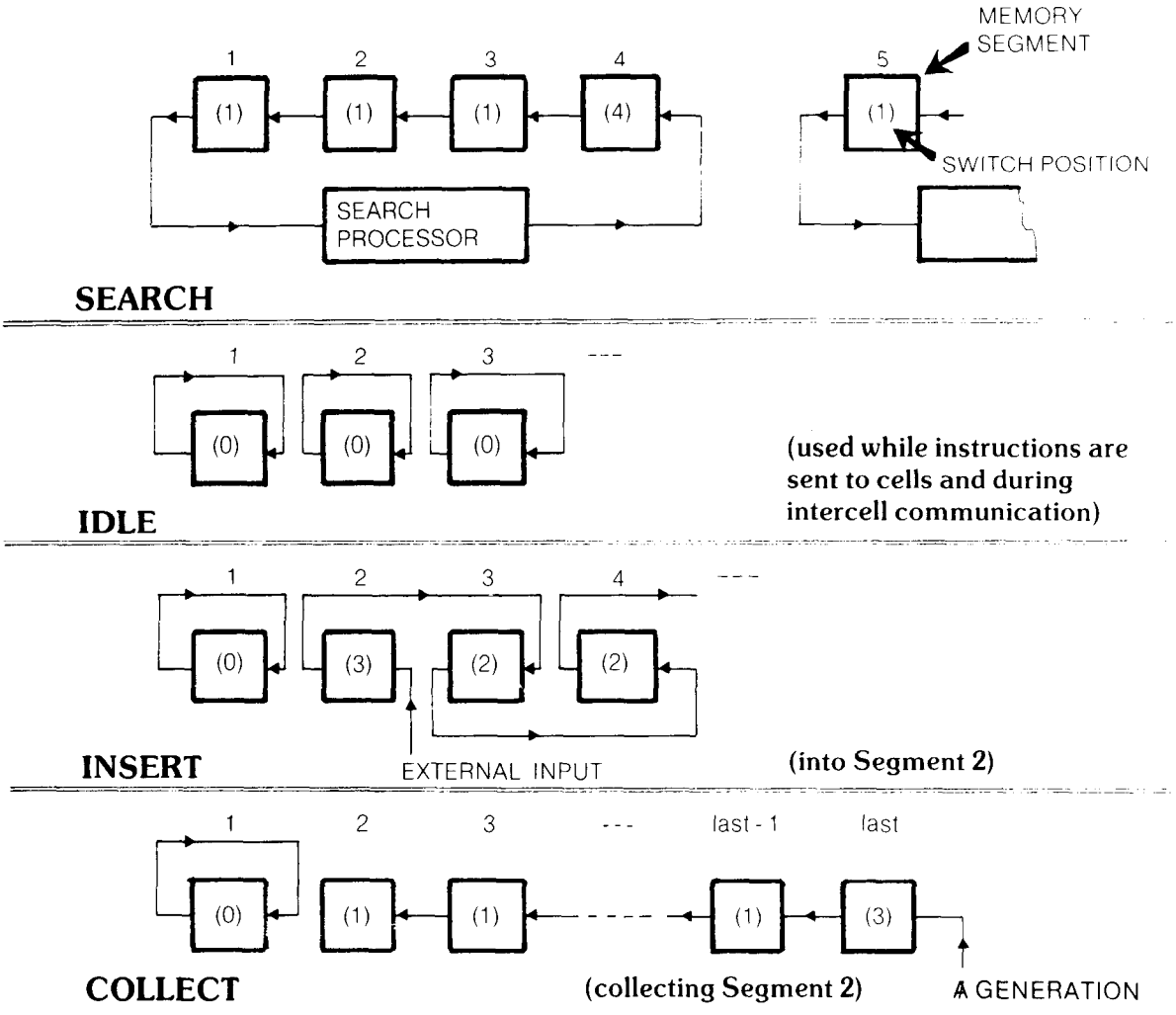
LOGICAL OPERATION **employee { name { John Smith }, salary — history i. { date { 3/18/1979 }, salary { 33K } } }.**

ENCODED INSERTION **Ð4Ð5aÐ63/18/1979Ð5bÐ633K**

MEMORY OPERATION	SEGMENT CONTENTS AT END OF OPERATION			
	2	3	4	5
Search & Mark	Ð 4 J o h n S	m i t h Ð 3 b (Ð)	4 Ð 5 a Ð 6 2 /	1 7 - -
		marked for insertion before		
	switch positions used			
Insert Cycle 1	2 (Ø)	3 (Ø,3)	4 (3,2)	5 (2)
	Ð 4 J o h n S	m i t h Ð 3 b Ð	A A A A A A A (Ð)	4 Ð 5 a
Insert Cycle 2	3 (Ø)	4 (3)	5 (2)	6 (2)
	m i t h Ð 3 b Ð	4 Ð 5 a Ð 6 3 /	A A A A A A A (Ð)	4 Ð 5 a

Repeat Cycle 2 until insertion is complete. The insertion point moves down one segment per cycle. A characters are shifted off end of last segment.

Figure 3. Insertion example.



Note: Switches are omitted for clarity. Only the active connections between segments are shown.

Figure 7. Memory modes.

The deletion example of Figure 9 shows the storage collection operation after the search processors have overwritten the structure to be deleted with available space characters. Storage collection is the inverse of the insertion process, with all segments below the collection point being shifted up one segment while available space characters are generated at the end of the storage array. Search operations can be performed with available space characters present in the memory. Thus the collection operation need not occur immediately after deletion, but can be scheduled as a background operation to avoid delaying the execution of searches. Collection is required only to satisfy the insertion requirement of enough available space at the end of the memory.

4.3 Editing performance

The performance of the editing operations is measured by execution time and memory utilization. The following parameters describe execution time:

$$\text{Segment period} = \frac{\text{Bits per segment}}{\text{Memory shift rate}}$$

The minimum continuous shift of the memory is one segment, thus the execution times for the editing and search operations are expressed as multiples of the segment period.

$$\text{Insertion time} = \left\lceil \frac{\text{Bits inserted}}{\text{Bits per segment}} \right\rceil \times (\text{Segment period}),$$

where $\lceil X \rceil$ :: = the smallest integer $\geq X$.

$$\text{Collection time} = \text{No. segments filled with } \text{Ⱥ} \times (\text{Segment period})$$

$$\text{Search period} = (\text{Segments per cell}) \times (\text{Segment period})$$

$$\text{Deletion time} = \text{No. cells spanned by set} \times (\text{Search period}).$$

The insertion process generates a string of available space (Ⱥ) characters with each insertion as overhead for alignment. The string has:

Minimum length: 0

Maximum length: 2 segments - 2 characters

Average length: 1 segment - 1 character

Since this string crosses segment boundaries, it must be handled as two strings of less than one segment during collection, as discussed in section 4.1.2.

5 Memory technology comparison

Memory technologies which have a segmented serial organization have been cost effective when used to emulate a random access memory in spite of their slow access times (Pohm 1975). When properly organized for exhaustive searching, however, their segmented serial organization is more fully exploited. Both cost per bit and access times are minimized, while providing generalized algorithms for improved data independence and ease of language implementation. A comparison of these technologies for suitability to INDY is complex. However, the major issues become clearer if the comparison is based on three parameters: cost per bit,

LOGICAL OPERATION **employee d. { name { John Smith } }.**

MEMORY OPERATION	SEGMENT CONTENTS AT END OF OPERATION			
	1	2 — 15	16	17
Delete Data	Ð 1 a Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ	Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ	Ⱥ Ð 2 Ð 3 a Ð 4	P e t e r
	← entire { 2 } set overwritten by search processors			
	1 (Ø)	2≤i≤[last − 1] (1)	last (3)	
Collection Cycle 1	Ð 1 a Ⱥ Ⱥ Ⱥ Ⱥ Ⱥ	contents of segment i + 1		

Repeat collection cycle until original segment 16 has been moved to Segment 2.

Figure 9. Collection Example

cost per concurrent access point, and shift rate. Cost of storage is determined by cost per bit, while cost of access is determined by both cost per access point and shift rate.

Tape technologies have the lowest cost per bit, but very high cost per access point. This high cost per access point causes an INDY system based on tapes to be very expensive if moderate response times are desired.

Disc technologies generally have a higher cost per bit, but a lower cost per concurrent access point. Thus, discs are more suitable for moderate to fast response times than tapes. Moving head discs offer low cost per bit at the expense of slow response, while head per track discs (with more access points) offer faster response at higher cost.

Magnetic bubble technologies currently have higher cost per bit than moving head discs, but are rapidly improving in comparison (Arnold 1978). Practical high density devices are organized as multiple segments of serial memory using the major-minor-loop arrangement (Texas Instruments 1977b). These small segments, typically a few hundred bits, offer a more efficient unit for insertion than a disc track. Bubbles suffer from low shift rate (a factor of 10 to 100 slower than discs or CCD's), so that more points of concurrent access are needed for a given response time. The cost per access is high if an incompatible logic technology (for example, NMOS or TTL) is used, but potentially very low if bubble logic (Lee and Chang, 1974) can be used effectively. Bubble logic would greatly reduce the need for magnetic to electric transducers between memory and logic, which require large chip area. Thus, the suitability of bubbles to the INDY architecture is dependent upon either a significant increase in shift rate or the future success of bubble logic.

Charge coupled devices currently are higher in cost per bit than moving head discs, but are improving in comparison (Armstrong 1977). Representative high density CCD's (Rosenbaum 1976, Fairchild 1977, Texas Instruments 1977a) are organized as multiple segments of serially accessed memory, each segment consisting of one series-parallel-series (SPS) array. The SPS arrays, with typically 4k bits each, offer a more efficient unit for insertion than a disc track. CCD's have a shift rate that is comparable to discs. Also, since CCD's store information in the form of electrical charge, expensive magnetic to electric transducers are not required to interface to a variety of logic technologies. Based on projected VLSI densities (Altman 1977), a single chip INDY cell with 128k bits of CCD memory and an NMOS associative processor should be technically feasible at the same time that a 256k bit CCD memory or a 64k bit RAM is feasible. Furthermore, complex logic functions have successfully been implemented directly in CCD technology (Zimmerman 1977), offering significant improvements in chip area and power dissipation.

6 Summary and conclusion

The cost and time involved in designing and using future computer systems should be reduced by exploiting the drastically decreasing cost of hardware technology. For data base systems, this can be done by providing more complete and easy-to-use data language functionality which is implemented by considering the flexibility of hardware as well as software.

The INDY kernel is intended to allow a simple implementation of any of the various data languages using a single storage representation. The INDY associative storage array is intended to provide those capabilities that are difficult to implement on conventional architectures, by implementing the language capabilities much more directly in hardware. The simplicity and generality of the exhaustive search technique of INDY allows compilers or interpreters for very high level data languages to be written with ease and reliability. Parallelism and technological efficiency are used to provide adequate response time.

Implementation of storage and editing requirements was described, using inexpensive segmented serial memory technologies. A comparison was made of these technologies for suitability to the INDY architecture. The combinations of logic and memory technologies that seem most promising for the near future are CCD memory with either NMOS logic or CCD logic, and magnetic bubble memory with magnetic bubble logic.

REFERENCES

L. Altman and C. L. Cohen, "The Gathering Wave of Japanese Technology," Electronics (June 9, 1977).

L. Altman, "Five Technologies for Squeezing More Performance From LSI Chips." Electronics (August 18, 1977).

L. Armstrong, "The CCD's Future Takes On a Bright Hue." Electronics (November 10, 1977)

W. F. Arnold, "Memory Makers Brace For Bubble Battle." Electronics (February 16, 1978)

R. F. Boyce, D. D. Chamberlin, W. F. King III, and M. M. Hammer, "Specifying Queries as Relational Expressions." Proceedings of ACM SIGPLAN/SIGIR Interface Meeting on Programming Languages and Information Retrieval, Gaithersburg, Maryland (November 1973).

F. P. Brooks, The Mythical Man-Month, Addison-Wesley, Reading, Mass. (1975).

D. D. Chamberlin and R. F. Boyce, "SEQUEL: A Structured English Query Language." Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control (May 1974).

E. F. Codd, "A Relational Model of Data for Large Shared Data Banks." Communications of ACM, Vol. 13, No. 6 (June 1970).

E. F. Codd, "Further Normalization of the Data Base Relational Model." Courant Computer Science Symposia, Vol. 6, Data Base Systems, New York; Prentice-Hall (May 1971a).

E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus." Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access and Control San Diego (November 1971b).

E. F. Codd, "Recent Investigations in Relational Data Base Systems." Information Processing 74, North-Holland Publishing Co., Amsterdam (1974).

G. P. Copeland, G. J. Lipovski and S.Y.W. Su, "The Architecture of CASSM: A Cellular System for Non-Numeric Processing." Proceedings of the First Annual Symposium on Computer Architecture (December 1973).

G. P. Copeland and S.Y.W. Su, "A High Level Data Sublanguage for a Context-addressed Segment-sequential Memory." Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control (May 1974).

G. P. Copeland, "A Cellular System for Non-Numeric Processing." Ph.D. Dissertation, University of Florida (December 1974).

G. P. Copeland, "String Storage and Searching for Data Base Applications: Implementation on the INDY Backend Kernel." Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing (August 1978a).

G. P. Copeland, "The Importance of Strings for Data Base Abstractions." in preparation (1978b).

G. P. Copeland, "Language Requirements for a Data Base Kernel." in preparation (1978c).

C. J. Date and P. Hopewell, "Storage Structure and Physical Data Independence." Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control (November 1971).

R. Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases." ACM Transactions on Data Base Systems, Vol. 2, No. 3 (September 1977).

Fairchild Semiconductor, "F464 65536x1 Dynamic Serial Memory." Data Sheet (April 1977).

S. Y. Lee and H. Chang, "An All-Bubble Text-Editing System." IEEE Transactions on Magnetics (September 1974).

A. Makinouchi, "A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Model." Proceedings of the Third International Conference on Very Large Data Bases (October 1977).

A. V. Pohm, "Cost/Performance Perspectives of Paging with Electronic & Electromechanical Backing Stores." Proceedings of the IEEE (August 1975).

S. D. Rosenbaum, C. H. Chan, J. T. Caves, S. C. Poon, and R. W. Wallace, "A 16384 Bit High Density CCD Memory." IEEE Transactions on Electron Devices (February 1976).

J. M. Smith and D. C. P. Smith, "Database Abstractions: Aggregation." Communications of the ACM 20,6 (June 1977a).

J. M. Smith and D. C. P. Smith, "Database Abstractions: Aggregation and Generalization" ACM Transactions on Database Systems 2,2 (June 1977b).

Texas Instruments, "TMS3064 65536 Bit CCD Memory." Data Sheet (November 1977a).

Texas Instruments, T1B0101 Data Sheet (1977b).

J. C. Thomas and J. D. Gould, "A Psychological Study of Query-By-Example." Proceedings of the National Computer Conference (1975).

T. A. Zimmerman, R. A. Allen and R. W. Jacobs, "Digital Charge Coupled Logic." IEEE Journal of Solid State Circuits (October 1977).

M. M. Zloof, "Query-By-Example." Proceedings of the National Computer Conference, AFIPS Press (1975).

M. M. Zloof, "Query-By-Example--Operations on Hierarchical Data Bases." Proceedings of the National Computer Conference, AFIPS Press (1976a).

M. M. Zloof, "Query-By-Example: Operations on the Transitive Closure." IBM Research Report RC5526 (October 1976b).