A TEMPORAL AND SPATIAL LOCALITY THEORY FOR CHARACTERIZING VERY LARGE DATA BASES

Stuart E. Madnick

Allen Moulton

February 1987 #WP 1864-87

A Temporal and Spatial Locality Theory for Characterizing Very Large Data Bases

Introduction

This paper summarizes some recent theoretical results in formally defining and measuring database locality and demonstrates the application of the techniques in a case study of a large commercial database. Program locality has a long history of theoretical investigation and application in virtual memory systems and processor caches. Database locality has also been investigated but has proven less tractable and reported research results are mixed with respect to the strength of database locality. Nevertheless, engineering pragmatism has incorporated mechanisms analogous to virtual memory into disk caches and database buffer management software on machines as small as the personal computer and as large as the largest mainframe. The research presented here is aimed at formally defining and better understanding database locality.

This paper starts with a brief overview of the locality concept and program locality models, identifies the different stages at which database locality might be observed, highlights the differences between database and program locality, and summarizes prior database locality research results. Next a technique is presented for adapting the program locality model to both temporal and spatial dimensions at all stages of database processing. Finally, a case study of a large commercial database provides the context for elaborating on the application of the database locality model and the interpretation of the results.

Program Locality

Program locality refers to the tendency of programs to cluster references to memory, rather than dispersing references uniformly over the entire memory range. Denning and Schwartz [1972, p. 192] define the "principle of locality" as follows:

> (1) during any interval of time, a program distributes its references non-uniformly over its pages; (2) taken as a function of time the frequency with which a given page is referenced tends to change slowly, i.e. it is quasi-stationary; and (3) correlation between immediate past and immediate future reference patterns tends to be high, whereas correlation between disjoint reference patterns tends to zero as the distance between them tends to infinity.

Denning[1980] estimated that over two hundred researchers had investigated locality since 1965. Empirical work has shown that programs pass through a sequence of phases, during which memory references tend to cluster in a relatively small set of pages -- the working set. Transitions between phases can bring a rapid shuffling of the pages in the working set before the program stabilizes again. Reasons advanced for the existence of program locality include the sequential nature of instructions, looping, subroutine and block structure, and the use of a stack or static area for local data.

The seminal work on program locality is the working set model developed by Denning[1968] and presented at the original Gatlinburg Symposium on Operating Systems Principles. The elegant simplicity of the model permits the extensive exploration of the load that programs place upon a virtual memory system and the means for balancing that load in a multi-programming situation. The working set model postulates a reference string, *r(t)*, generated by a process (the execution of a program). Each element in the reference string corresponds to the virtual address of a word fetch or store to the memory system. The virtual memory is usually considered to be composed of equal sized pages, which may be stored either in primary or secondary memory as determined by the memory management system.

The working set model uses two basic measures of load placed on the memory by a program -- the working set size (the quantity of memory demanded), and the missing page rate (the frequency and volume of data that must be retrieved from secondary storage). The sequence of overlapping, equal-sized intervals of width θ along the reference string is examined. The working set at each point t along $r(t)$ consists of the pages referenced in the interval $r(t-\theta+1) \ldots r(t)$. The average working set size, $s(\theta)$, is the average of the number of pages in the working set at each point along the reference string. The missing page rate, $m(\theta)$, is the frequency with which $r(t+1)$ is missing from the working set at t. Generally, increasing the interval width increases the working set size but decreases the missing page rate, resulting in a tradeoff between the two resources.

Most work on program locality assumes a fixed page size. Madnick[1973] addresses the issue presented by counter-intuitive results when page size was experimentally varied. The <u>page size anomaly</u> occurs when reducing the page size more than proportionally increases the missing page rate with the working set size held constant. Madnick suggests that the locality phenomenon should be viewed in two dimensions -- temporal and spatial. The temporal dimension measures the degree to which references to the same segments of memory cluster along the reference string. The spatial dimension measures clustering along the address space of the program. In program locality, where processor architecture requires a fixed page size, the spatial dimension is often ignored. The application of locality theory to database systems, however, requires attention to the spatial dimension.

The study of program locality benefits from the stylized interface between the processor and memory. The processor requests a fetch or store passing a virtual address to the memory; the memory fetches or stores the word at that address and returns the result to the processor. If an instruction references more than one word, the processor generates a sequence of requests for memory access. The reference string corresponds to the sequence of addresses that appear on the processor-memory bus. The address space is also generally linear and anonymous. The same virtual address may be reused for different semantic purposes at different times without informing the memory due to programs being overlaid or new records read into buffers. Logically different programs and data appear the same to the memory. The simplicities of the processor-memory interface are not naturally present in interfaces to database systems.

Database Locality

Database references begin with external demand for access to data. This may take the form of ad hoc queries, execution of transaction programs, or batch processes that prepare reports or update files. External demand is mediated by an application program that generates a series of data manipulation language (DML) requests in the course of its execution. The simplest form of application program, such as IBM's Query Management Facility (QMF), takes a request presented in the DML (SQL in this case) and passes it to the DBMS, presenting the response to the user in readable format. When the DML request is received by the DBMS, it is analyzed and transformed by an interpretation, or binding, stage into a sequence of internal functional steps referencing internally defined data objects, such as searching an index or retrieving a row from a table. The transformation from DML to internal functions may be compiled, as in the bind operation of IBM's DB2, or performed each time the request is executed. The execution of internal functions results in accesses to data in buffers controlled by a <u>storage</u> manager, which, in turn, must read or write blocks of data in secondary storage. This sequence of processing stages presents five different points at which database load may be measured: (1) external, (2) application, (3) interpretation, (4) functional, and (5) storage. Each stage has different distinctive characteristics and is normally described in different terms. The methodology presented here permits the load at each stage to be described in commensurable units.

The storage stage of database processing is analogous to program memory paging, and most of the published work on database locality has concentrated there. Easton [1975, 1978] found support for the existence of locality in the database of IBM's Advanced Administrative System (AAS). Rodriguez-Rosel [1976] found sequentiality, but not locality by his definition, in disk block accesses generated by an IMS application. Loomis and Effelsberg [1981], Effelsberg [1982] and, Effelsberg and Haerder [1984] find mixed evidence of locality in buffer references by several applications using small CODASYL databases. Concentrating on the storage stage means that the locality observed will be that of the external demand filtered through the database system's internal logic. The sequentiality observed by Rodriguez-Rosel, for example, may be more the characteristic of IMS's search strategy than the application's use of data. Observing locality in external demand requires a different approach. By examining accesses to entities drawn from a single entity set, McCabe [1978] and Robidoux [1979] found evidence of locality.

Circumstantial evidence also suggests that the disk accesses observed by Easton were one-to-one mappings of entities in AAS.

The reference string in program locality consists of a sequence of word references. Each element corresponds to an equal amount of useful work. The requests generated by the first three stages of database processing are, however, anything but uniform in size. Table 1 summarizes database stages, types of requests generated, and data objects operated upon.

	Stage	Type of Request	Data Operated Upon
1	external	application task	implicit in task definition
2	application	DML statement	instantiation of schema
3	interpretation	internal function	internal objects
4	functional	buffer reference	contents of buffer
5	storage	disk access	disk blocks

Table 1

To achieve a common unit of measure, permitting comparison of locality measures across stages of processing, each type of request must be transformed into a sequence of elemental data references. Since the byte is usually the basic unit of data storage, byte references will be used here. Clearly, an actual reference to a single byte alone will be rare. A reference to a data field consisting of a number of bytes can be transformed into a sequence of byte references. The order of bytes within a field is immaterial as long as a consistent practice is maintained. The same principle can be applied to convert a request for access to multiple fields within one or more records (attributes within a row) into a sequence of field references and thence into a byte reference string. The order of fields should match the order specified in the request.

An Illustrative Example

To clarify the process of transforming a request string into a uniform reference string, let us consider an simple example with two tables: one to represent stocks and another to represents holdings by an account in a stock. In SQL these tables might be defined as follows:

```
CREATE TABLE STOCKS
    (SYMBOL          CHAR(8) NOT NULL,          -- Key
     NAME            CHAR(40))

CREATE TABLE HOLDINGS
    (ACCOUNT         CHAR(8) NOT NULL,          -- Key
     STKSYM          CHAR(8) NOT NULL,          --     combined
     SHARES          INTEGER)
```

Assume that account "82-40925" holds stock in General Motors and Delta Air Lines. Processing the external request "Show holdings for account 82-40925" an application program might generate the following query:

```
SELECT  SHARES, STOCKS.NAME
  FROM  STOCKS, HOLDINGS
  WHERE ACCOUNT = '82-40925'
    AND STOCKS.SYMBOL = HOLDINGS.STKSYM
  ORDER BY STOCKS.NAME
```

The field references seen at the application level by this query would be:

```
HOLDINGS.SHARES[ACCOUNT='82-40925',STKSYM='DAL']         4 bytes
STOCKS.NAME[SYMBOL='DAL']                               40 bytes
HOLDINGS.SHARES[ACCOUNT='82-40925',STKSYM='GM']          4 bytes
STOCKS.NAME[SYMBOL='GM']                                40 bytes
```

Using the data types from the table definitions, this query can be transformed into a sequence of 88 byte references . Each byte reference can be identified by a database address consisting of:

- file or table identifier,
- field or attribute identifier,
- record or row identifier (primary key above), and
- byte number within field or attribute.

The method of transformation and the form of the database address will differ from stage to stage and from one type of database system to another. Although byte reference strings are the basis of the database locality models presented here, it is not necessary to collect tapes full of individual byte reference traces for analysis. The point is that all requests can be transformed conceptually into a uniform string of byte references for analysis.

Database temporal locality measures

Given a uniform reference string, a working set model can be developed for database locality. Each element, $r(t)$, of the reference string is the database address of the byte referenced at point t. The parameter θ determines the width of the observation interval along the reference string. At each point $t \geq \theta$, the working set $WS(t,\theta)$ consists of the union of the database addresses in the string $r(t-\theta+1) \ldots r(t)$. For $0 \geq t > \theta$ the interval is considered to begin at $r(1)$. For $t<0$ the working set is defined to be empty. The working set size, $w(t,\theta)$, is the total number of distinct database addresses in $WS(t,\theta)$. The <u>average</u> <u>working</u> <u>set</u> <u>size</u> is defined as

$$s(\theta) = (1/T) \cdot \sum_{t=1}^{T} w(t,\theta)$$

where T is the length of the reference string. We now define the binary variable

$$\Delta(t,\theta) = \begin{cases} 1 & \text{if } r(t+1) \text{ is not in } WS(t,\theta) \\ 0 & \text{otherwise.} \end{cases}$$

The missing ratio is then

$$m(\theta) = (1/T) \cdot \sum_{t=1}^{T} \Delta(t,\theta)$$

The volume ratio, $v(\theta)$, is defined as the average number of bytes that must be moved into the working set for each byte referenced. For purely temporal locality $v(\theta) = m(\theta)$.

The definitions of $s(\theta)$ and $m(\theta)$ follow Denning and Schwartz [1972, pp. 192-194]. The properties of the working set model elaborated in that paper also can be shown to hold for the database byte reference string. Some of these properties are:

(P1) $1 = s(1) \leq s(\theta-1) \leq s(\theta) \leq \min(\theta, \text{size of database})$

(P2) $s(\theta) = s(\theta-1) + m(\theta-1)$

(P3) $0 \leq m(\theta+1) \leq m(\theta) \leq m(0) = 1$

The interval width, θ, is measured in bytes referenced. The working set as defined here is composed of individual bytes. Since the page size effect has been completely eliminated, these measures reflect purely temporal locality. The average working set size, $s(\theta)$, is measured in bytes and can be shown to be concave down and increasing in θ. The missing ratio, $m(\theta)$, on the other hand, decreases with θ. The tradeoff curve of $s(\theta)$ against $m(\theta)$ can be determined parametrically.

Spatial locality measures

To introduce the spatial dimension, the reference string must be transformed. Let σ be the spatial dimension parameter and $P_\sigma[r]$ be a function which transforms any database byte address into a segment number. Then the spatial reference string of order σ is defined by:

$$r_\sigma(t) = P_\sigma[r(t)].$$

The limiting case of $\sigma=0$ is defined to be purely temporal locality:

$$r_0(t) = r(t).$$

Let $Size[r_\sigma]$ denote the size of the segment containing a byte reference. Segments may be of constant size, in which case q_σ may be used for any segment size. Alternatively, segments may be defined of varying sizes. If comparisons are to be made across the spatial dimension, we require

$$Size[r_{\sigma+1}(t)] \geq Size[r_\sigma(t)] \quad \text{for all } r.$$

Although the reference string now consists of segment numbers each reference is still to a single byte. The temporal parameter θ is measured along the reference string in byte references. The working set of order σ at t, $WS_\sigma(t,\theta)$, is defined as the union of the segments included in the reference substring $r_\sigma(t-\theta+1) \ldots r_\sigma(t)$. The working set size, $w_\sigma(t,\theta)$, is the total of the sizes of the segments in $WS_\sigma(t,\theta)$. The average working set size, $s_\sigma(\theta)$, and the missing ratio, $m_\sigma(\theta)$, are defined as in purely temporal locality. The volume ratio is defined to be:

$$v_\sigma(\theta) = (1/T) \cdot \sum_{t=1}^{T} Size[r_\sigma(t)]\cdot\Delta_\sigma(t,\theta)$$

For constant segment size we have

$$v_\sigma(\theta) = q_\sigma \cdot m_\sigma(\theta).$$

The missing ratio measures the rate at which segments enter the working set; the volume ratio measures the average quantity of data moving into the working set for each byte reference.

The properties of the working set model mentioned above, when adapted to constant segment size spatial locality, are:

(P1′) $q_\sigma = s_\sigma(1) \leq s_\sigma(\theta-1) \leq s_\sigma(\theta) \leq \min(q_\sigma \cdot \theta, \text{ size of database})$

(P2′) $s_\sigma(\theta) = s_\sigma(\theta-1) + v_\sigma(\theta-1)$

(P3′) $0 \leq m_\sigma(\theta+1) \leq m_\sigma(\theta) \leq m_\sigma(0) = 1$

(P4) $0 \leq v_\sigma(\theta+1) \leq v_\sigma(\theta) \leq v_\sigma(0) = q_\sigma$

From these properties we can derive an alternative method for calculating the average working set size from the volume ratio:

$$s_\sigma(\theta) = \begin{cases} q_\sigma & \text{for } \theta = 1 \\ q_\sigma + \sum_{i=1}^{\theta-1} v_\sigma(i) & \text{for } \theta > 1 \end{cases}$$

The constant segment size spatial locality measures are the same as the traditional Denning working set measures with a change in unit of measure. The adoption of a common unit of measure and the addition of the volume ratio allow direct comparisons along the spatial dimension.

Interpreting the measures

Having defined the locality measures formally, let us consider what they represent. The reference string corresponds to the sequence of bytes that must be accessed to service a series of requests. The length of the reference string generated by a process, T, measures the total useful work performed by the database system for that process. The temporal parameter θ determines the measurement interval in units of byte references. It serves as an intervening parameter to determine the tradeoffs among the measures. The spatial parameter σ can be used to test the effect of ordering and grouping data in different ways. In its simplest form, σ can be thought of as a blocking factor.

The average working set size, $s_\sigma(\theta)$, estimates the quantity of buffer memory required to achieve a given degree of fast access to data without resort to secondary storage. If several processes are sharing a database system, as is the usual case, the sum of working set sizes can be used to determine total memory demand and to assist the database system in load leveling to prevent thrashing.

The missing ratio, $m_\sigma(\theta)$, measures the average rate at which segments must be retrieved from outside the working set. Generally, $m_\sigma(\theta)$ decreases with θ, while $s_\sigma(\theta)$ increases, requiring a tradeoff of retrieval rate with buffer memory size. The product $T \cdot m_\sigma(\theta)$ measures the expected number of accesses to secondary memory required by a process reference string. On many large computer operating systems, such as IBM's MVS, there is a large cost for each disk read and write initiated -- both in access time on the disk and channel, and in CPU time (perhaps 10,000 instructions per I/O).

The volume ratio, $v_\sigma(\theta)$, measures the average number of bytes that must be transferred into the working set for each byte of data referenced. If there is a bandwidth constraint on the channel between secondary storage and buffer memory, $T \cdot v_\sigma(\theta)$ can be used to estimate the load placed on that channel.

In summary, the three locality measures describe different kinds of load implicit in the demand characterized by a reference string. In the following sections an actual application will be analyzed to demonstrate how these measures are determined and used to characterize a database.

A case study

SEI Corporation is the market leader in financial services to bank trust departments and other institutional investors in the United States and Canada. The company operates a service bureau for three hundred institutional clients of all sizes, managing a total of over 300,000 trust accounts. The processing workload completely consumes two large IBM mainframes -- a 3084 and a 3090-400, using the ADABAS database management system. The applications software has developed over a period of fifteen years, originally on PR1ME computers. Five years ago the company used six dozen PR1ME 750's and 850's to process a smaller workload. At the time of this study approximately two thirds of the accounts had been migrated to the IBM mainframes. Extrapolated to the complete workload, the database requires approximately 230 IBM 3380 actuators (140 gigabytes). Of this total, 40% is occupied by the "live" current month database, 20% by an end of month partial copy retained for at least half of the month, and 40% by a full end of year copy needed for three months after the close of the year. Each of these databases is actually divided into ten to fifteen sub-databases because of limitations imposed by ADABAS. The locality model and measures described in this paper are being used to gain additional understanding of the nature of the database and its usage in order to find opportunities for improving performance and reducing unit cost of the applications.

Approximately 60% of the live and year-end databases are occupied by the "transactions" file, containing records of events affecting an account. On average there are approximately ten transactions per account per month, although the number can range from none into the thousands for a particular account. At the end of each month, except for year end, all the files in the live database, except transactions, are copied into the month end database. A limited volume of retroactive corrections are made to the copy. Transactions are read from the live database. End of month reports are prepared for the client. In addition, statements are prepared for the beneficiaries, investment advisors, and managers of each account. Statements may cover a period of from one to eighteen months. A statement schedule file determines the frequency and coverage of the statements prepared. The statement preparation application program prepares custom format statements according to specifications contained in client-defined "generator" files. The end of year procedure is similar to end of month, except that the entire database is copied, transactions are read from the copy, and the workload is much larger. Full year account statements frequently are prepared at the end of the calendar year, along with a large volume of work required by government regulations.

Access to the transactions file for statements and similar administrative and operational reports uses a substantial amount of system resources. Statement preparation alone consumes 20% of total processor time during a typical month. Preparation of a statement requires a sequential scan of the transactions entered for an account within a range of dates. An SQL cursor to perform this scan might be defined as follows:

```
DEFINE CURSOR trans
     FOR SELECT A1, ..., An
     FROM TRANSACTIONS
     WHERE ACCOUNT = :acct
       AND EDATE BETWEEN :begdate AND :enddate
     ORDER BY EDATE
```

Each transaction would then be read by:

```
FETCH CURSOR trans INTO :v1, ..., :vn
```

The group of fields selected requires approximately 300 bytes -- substantially all of the stored transaction record. These records are blocked by ADABAS into 3000 byte blocks.

The PR1ME version of this application stored transactions in individual indexed files by client "processing ID" and month of entry. A processing ID was a group of several hundred accounts. At the end of each month, before producing statements, the current transactions file for each processing ID was sorted by account and entry date, and a new file begun for the next month. Managing the large number of small files was perceived to be a problem. During the IBM conversion, a database design exercise was performed, resulting in consolidation of the monthly and ID-based transactions files into a single file in ADABAS. Multiple clients were also combined together into ADABAS databases covering up to 30,000 accounts. The transactions file in each of these databases contains five to ten million records. The dump to tape, sort, and reload process to reorganize a single transactions file takes over twenty hours of elapsed time, with the database unavailable for other work. Reorganization is attempted only once or twice a year, and never during end of month or end of year processing. Thus, the physical order of the file is largely driven by entry sequence. As a result, instances of more than one transaction for an account in a block are rare.

As the migration of clients from the PR1ME system to the IBM version has progressed over the past three years, the estimate of machine resources to process the full workload has increased five-fold. A surprising observation is that the limiting factor is CPU capacity, although the application itself does relatively little calculation and a lot of database access. The resolution to this riddle seems to lie in the large amount of CPU time required to execute each disk I/O. The transaction file has been the center of considerable controversy. One school of thought holds that some form of buffering of data would eliminate much of the traffic to the disk. The buffering might be done by the application program, or the interface programs that call the DBMS, or it might be done by adjusting the buffer capacity in the DBMS itself, or by adding disk cache to the hardware. The analysis to follow will show how the database locality model can be used to characterize this database application and gain insight into the impact of various buffering strategies.

Predictive estimation of locality measures

There are two ways to apply the locality model to a case: empirical measurement and predictive estimation. The empirical approach uses a request trace from a workload sample to calculate the curves for $s(\theta)$, $m(\theta)$, and $v(\theta)$. The database locality model can be applied at each stage of database processing. At the application level a trace of calls to the DBMS, such as the ADABAS command log, can be used. Similarly, disk block I/O traces can be used at the storage level. Request traces from inside the DBMS -- at the interpretation, internal, and buffer stages -- are substantially more difficult to obtain unless the database system has been fully instrumented. The empirical approach can also only be used with observed workload samples. Application of the model to planning and design requires techniques for predictive estimation of the measure curves from specifications and predicted workload. In this section we will show how to develop predictive estimators of the locality measures from patterns of data access.

The sequential scan is a fundamental component of any data access. Each FETCH CURSOR statement, or its counterpart in another DML, retrieves a sequence of fields from a logical record (a row in the table resulting from the SELECT statement the cursor definition). A series of FETCH CURSOR statements occurring between the OPEN CURSOR and CLOSE CURSOR statements constitutes another level of sequential scan (all transactions for a statement in the case study). Although sequentiality might seem to be the antithesis of locality, since each item of data in a scan may be different from all others, we shall see that exploitation of the spatial dimension can result in considerable locality even for the sequential case.

To derive formulas for the locality measures of a sequential scan, we first recast the problem in formal terms. Let r be the number of bytes of data accessed with each FETCH. We can reasonably assume that there are no repeated references to the same data within a single FETCH. Assume that all the data referenced in a fetch comes from single record of length ℓ. Each FETCH is then a sequential scan of r bytes in any order from a record of length $\ell \geq r$. Let us consider the class of segment mapping functions that will place whole records into segments of constant size $b \geq \ell$. Each mapping function can be defined as an ordering of records by some fields in each, followed by the division of the ordered records into equal-sized segments of b bytes. For example, we might segment transaction records, ordered by account and entry date, into blocks of ten records, or 3000 bytes each. It is important to remember that in this type of analysis the ordering is only conceptual and need not correspond to the physical order or blocking of the file.

Given the definitions of r, ℓ, and b above, we can proceed to derive formulas for the locality measures. First note that each of the r bytes in a FETCH is from the same record, and by extension, from the same segment. Therefore, only the first byte reference of each string of r can be missing from the prior working set even if $\theta=1$. Now let $f(\theta)$ denote the likelihood that a record is missing from the working set when its first byte is referenced. We then have

$$q_\sigma = b$$
$$m_\sigma(\theta) = (1/r) \cdot f(\theta)$$
$$v_\sigma(\theta) = (b/r) \cdot f(\theta)$$

and from the alternative method for calculating average working set size

$$s_\sigma(\theta) = b + (b/r) \cdot F(\theta - 1)$$

where

$$F(\theta) = \sum_{i=1}^{\theta} f(i)$$

Note that $F(\theta)=\theta$ when $f(\theta)=1$. For the limiting case of purely temporal locality for a sequential scan we can apply these formulas by using $b=r=1$ and $f(\theta)=1$. Since no byte of data is ever referenced twice, each byte in the working set must be unique. The working set size will always be equal to the interval width θ, and the missing ratio 100%, independent of θ. The volume ratio will be unity.

Application of locality measures to the case study

Table 2 shows the results for several examples of sequential access in the context of the case study. Case 1 is the purely temporal locality base case. Case 2 shows access to whole records or rows without blocking. Each segment is defined to be a single transaction record of 300 bytes. No assumptions are made about the ordering of records, but we assume that they are never referenced twice: $f(\theta)=1$. In case 3 we examine the effect of blocking where only one logical record of 300 bytes is referenced out of each block of 3000 bytes. This case might occur where the segment mapping function placed only one transaction record in each segment (the remainder of the block might be filled with other data). Alternatively, we might use this case to model the situation where records are so widely dispersed that there is essentially zero likelihood of repeatedly referencing a block. Again $f(\theta)=1$, since the block is never reused. Case 4 shows the effect of accessing all the data in a 3000 byte block, by using a segment mapping function that orders records in processing sequence -- by account and entry date here. Since all records in a block are accessed together, only the first record in a block will be missing and $f(\theta)=(b/r)^{-1}$. Note that this ordering is performed by the mapping function. We do not assume any physical ordering. In fact, the results depend only on the clustering of data into segments, not on the ordering of segments or of data within segments.

	access case	$f(\theta)$	$m(\theta)$	$v(\theta)$	$s(\theta)$
1	purely temporal ($b=1$)	1	100%	1	θ
2	whole records ($b=300$)	1	.33%	1	$300 + (\theta - 1)$
3	one record per block ($b=3000$, $r=300$)	1	.33%	10	$3000 + 10(\theta - 1)$
4	whole blocks ($b=3000$, $r=300$)	r/b	.03%	1	$3000 + (\theta - 1)$

Sequential access to transaction records
Table 2

Cases 1, 2, and 4 show the effect of increasing block size when all data in each block is accessed. The volume ratio is one, since only data needed is accessed. The missing ratio drops inversely with block size, independent of interval width θ. The minimum, and optimal, buffer size is one block. As we would expect, there is no benefit to obtained from the larger buffer sizes which result from increasing θ. Cases 3 and 4 show the opposite extremes for blocked records. In Case 3 only one record out of each block is used; in Case 4 all the records are used. The missing ratio, the volume ratio, and the working set size growth factor all differ by a factor of b/r.

Although Case 3 above is a simple approximation of the effect of a mapping function that orders transactions in order of entry, the model can be further refined. Since a transaction is entered for an account approximately every other day, we can expect a uniform distribution of account transaction records across the file. Let a be the number of accounts in the file. Then

$$p = (b/\ell)\cdot(1/a)$$

is the likelihood that a given block will contain a record for a particular account. We then have

$$f(\theta) = \left[1 - p\right]^k \qquad \text{where } k = floor(\theta/r)$$

The expression for k represents the number of records fully contained in the interval width θ. Here $f(\theta)$ is the probability, at the point when a record is first referenced, that it is contained in none of the blocks containing any of the k most recently referenced records. Vander Zanden, et al. [1986] discuss alternative formulations of this expression when a uniform distribution does not apply.

Table 3 shows representative values for access to a single transactions file with 30,000 accounts, typical of the ADABAS databases in the case study. Interval widths, θ/r, are shown in multiples of the logical record size, r.

θ/r	$m(\theta)$	$v(\theta)$	$s(\theta)$ (Kbytes)
1	.33%	10	3
...			
100	.32%	9.7	300
200	.31%	9.4	600
300	.30%	9.0	800
400	.29%	8.8	1,100
500	.28%	8.5	1,300
...			
1000	.24%	7.2	2,400
2000	.17%	5.1	4,200
$b=3000$ $r=\ell=300$ $a=30000$			

Transaction record access with uniform distribution.
Table 3

Since ADABAS uses an LRU block replacement algorithm, the average working set size, $s(\theta)$, can be used to estimate the buffer memory required to achieve a reduction in I/O load. From Table 3 it is evident that increasing the buffer memory sufficiently can reduce I/O load resulting from the dispersed file by approximately 30% (for 2.4 megabytes) or 50% (for 4.2 megabytes). Such a large buffer size is not unreasonable on current generation mainframes with over 100 megabytes of memory. The observed improvement from use of large buffer memory allocations in the case study situation was comparable to the predictions.

Conclusion

A very large database, such as described in the above application, imposes an inertia on the organization. The cost of change is large, and the wrong choice can make matters substantially worse. Thus, gaining a good understanding of the application is important. The predictive estimation techniques demonstrated here permit the analysis of load at the logical level from specifications of the type of access in each application and predicted application demand. At the same time these locality measures allow consideration of various tradeoffs, such as the impact buffer size on I/O load. Comparison of predicted load to observed load generated at the application level can be used to test the predictive model and to highlight inefficiency in the application software. Comparison to observed buffer size and retrieval rate at the buffer stage can also be used for model validation and for performance evaluation of the database system. Most importantly, the locality measures presented provide a consistent way to characterize and understand a complex database environment.

REFERENCES

Denning, P.J. (1968)
The Working Set Model for Program Behavior, *Comm. ACM 11*, 5 (May 1968), 323-333.

Denning, P.J. (1980)
Working Sets Past and Present, *IEEE Trans. Softw. Eng., SE-6*, 1 (January 1980), 64-84.

Denning, P.J. and Schwartz, S.C. (1972)
Properties of the Working Set Model, *Comm. ACM 15, 3* (March 1972), 191-198.

Easton, M.C. (1975)
Model for Interactive Database Reference String, *IBM J. Res. Dev. 19*,6 (November, 1975), 550-556.

Easton, M.C. (1978)
Model for Database Reference Strings Based on Behavior of Reference Clusters, *IBM J. Res. Dev. 22*, 2 (March 1978), 197-202.

Effelsberg, W. (1982)
Buffer Management for CODASYL Database Management Systems, MIS Technical Report, Univ. of Ariz. (1982).

Effelsberg, W. and Haerder, T. (1984)
Principles of Database Buffer Management, *ACM Trans. Database Syst. 9*, 4 (December 1984), 596-615.

Loomis, M.E.S. and Effelsberg, W. (1981)
Logical, Internal, and Physical Reference Behavior in CODASYL Database Systems, MIS Technical Report, Univ. of Ariz. (1981).

Madnick, S.E. (1973)
Storage Hierarchy Systems, MIT Project MAC Report MAC-TR-107, MIT, Cambridge, Mass. (April 1973).

McCabe, E.J. (1978)
Locality in Logical Database Systems: A Framework for Analysis, unpublished S.M. Thesis, MIT Sloan School, Cambridge, Mass. (July, 1978)

Robidoux, S.L. (1979)
A Closer Look at Database Access Patterns, unpublished S.M. Thesis, MIT Sloan School, Cambridge, Mass. (June, 1979).

Rodriguez-Rosel, J. (1976)
Empirical Data Reference Behavior in Database Systems, *Computer 9*, 11 (November, 1976), 9-13.

Vander Zanden, B.T., Taylor, H.M., and Bitton, D. (1986)
Estimating Block Access when Attributes are Correlated, *Proceedings of the 12th International Conference on Very Large Data Bases*, 119-127.