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

Allen Moulton and Stuart E. Madnick

Sloan School of Management
Massachusetts Institute of Technology
30 Wadsworth Street, Cambridge, Massachusetts 02139

ABSTRACT

Very large databases represent an important asset for most organizations, but efficient operation poses a major challenge requiring new theory and techniques. This paper presents theoretical results in formally defining and measuring database locality. Database locality has proven less tractable than program locality, which has a long history of theoretical investigation and application in virtual memory systems and processor caches. The stages at which database locality may be observed and the differences between database and program locality are identified. A technique for adapting the program locality model to both temporal and spatial dimensions at all stages of database processing is developed. Finally, a case study of a large commercial database provides the context for illustrating the application of the database locality model and the interpretation of results.

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

0073-1129/89/0000/0612$01.00 © 1989 IEEE

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-θ+1) ... r(t). The average working set size, s(θ), is the average of the number of pages in the working set at each point along the reference string. The missing page rate, m(θ), 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 *page size anomaly* 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.

Denning and Slutz [1978] treat the application of the working set model to generalized segment references strings, where fixed size pages are replaced by irregular size segments. Each reference is implicitly, however, still to a single memory word and consideration is not given to potential repackaging of words into different size segments along the spatial dimension.

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 *storage* 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. Effelsberg [1982], Effelsberg and Loomis [1984], and Effelsberg and Haerder [1984] found 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. Peinl, Reuter, and Sammer [1988] report a high degree of locality in a stock trading system.

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.

Table 1 -- Stages of data access

	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

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,   -- Key
   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 are shown in Table 2 below.

Table 2 -- References by example query

Size	Table/Field	Primary Keys
4	HOLDINGS .SHARES	ACCOUNT='82-40925' STKSYM='DAL'
40	STOCKS .NAME	SYMBOL='DAL'
4	HOLDINGS .SHARES	ACCOUNT='82-40925' STKSYM='GM'
40	STOCKS .NAME	SYMBOL='GM'

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 (e.g., primary key), 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, $\mathbf{r(t)}$, 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 $\mathbf{WS(t,\theta)}$ consists of the union of the database addresses in the string $\mathbf{r(t-\theta+1)}$ ... $\mathbf{r(t)}$. For $0>t>\theta$ the interval is considered to begin at $\mathbf{r(1)}$. For $t \leq 0$ the working set is defined to be empty. The working set size, $\mathbf{w(t,\theta)}$, is the total number of distinct

database addresses in $WS(t,\theta)$. The *average working set size* 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) $1 = s(1) \le s(\theta-1) \le s(\theta) \le \min(\theta,$ size of database$)$
(P2) $s(\theta) = s(\theta-1) + m(\theta-1)$
(P3) $0 \le m(\theta+1) \le m(\theta) \le m(0) = 1$

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 $Seg_\sigma[r]$ be a function which transforms any database byte address into a segment number. Then the spatial reference string of order σ is defined by:

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

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

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

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') $q_\sigma = s_\sigma(1) \le s_\sigma(\theta-1) \le s_\sigma(\theta)$
$\le \min(q_\sigma \cdot \theta,$ size of database$)$
(P2') $s_\sigma(\theta) = s_\sigma(\theta-1) + v_\sigma(\theta-1)$
(P3') $0 \le m_\sigma(\theta+1) \le m_\sigma(\theta) \le m_\sigma(0) = 1$
(P4) $0 \le v_\sigma(\theta+1) \le v_\sigma(\theta) \le v_\sigma(0) = q_\sigma$

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 operating 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 in 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

As mentioned above, there had been considerable controversy at the case study site over whether the transactions file should be partitioned; if partitioned, whether by bank or by month; and whether the whole file or some partitions should be reorganized before major processing runs. Measurements showed that the transactions file accounted for 15% of the database load. Any reduction in the cost of accessing this file would have substantial impact on the operating cost of the business.

To resolve the issue, the company set up an experiment using two samples of actual customer workload. The Nightwork sample consisted of the entire end of quarter work stream of approximately 400 jobs (4.8 million database calls) for an ADABAS database covering thirteen banks with about 25,000 accounts. The Statements sample consisted of the major statement jobs (monthly, quarterly, annual) from the three largest clients in the same database (2.3 million database calls). Both samples were run against a copy of the end of quarter database three times: unchanged, partitioned by bank, and partitioned by month.

The database contained 6.2 million transaction records. Because of uneven bank sizes, the bank partitioning resulted in about two million transactions per partition. The month partitioning resulted in 350,000 to 450,000 transactions for each of the sixteen months. For the partitioned cases the database was reorganized and ordered by bank, account, and date of entry within each partition.

Table 3 -- Missing record rate for test samples

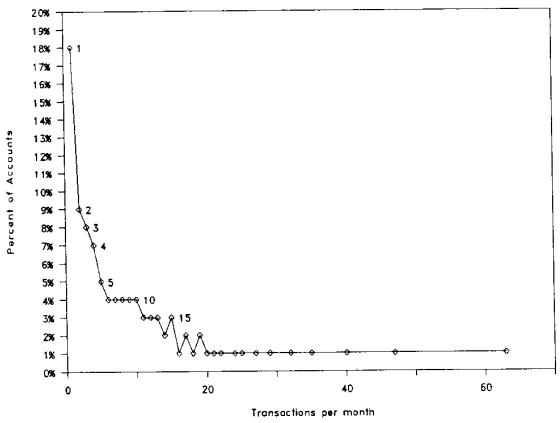
Workload Sample	Run 1 Unchanged	Run 2 By Bank	Run 3 By Month
Nightwork	87.6%	87.3%	86.5%
Statements	88.1%	77.6%	75.8%

The partitioning and reorganization required 56 hours of elapsed time in each case. Each test run of the two workload samples took up most of a weekend. All test runs were made with the maximum buffer size of three megabytes. The results are shown in Table 3.

The missing record rate measures the percent of database fetch requests that required a block to be read from disk. The results shown above are an average rate across all files. Measured CPU seconds were also reduced proportionately. Based on the results of this experiment, the company decided to partition the transactions file by month and regularly reorganize the latest month. The planning, preparation, execution, and analysis of this experiment took about six months, including modification of the database interface programs that accessed the transactions file.

We can use the spatial locality model to better understand the results of the experiment and, if fact, to show how the results could have been predicted. For Run 1 the database was not reorganized: transaction records occur in entry sequence. Figure 1 shows the distribution of number of transactions per month for customer accounts, which constitute 60% of the file. The remaining 40% of the records referred to internal bank suspense accounts used to match accrual to cash accounting when needed. Suspense account transactions do not appear on statements. Since suspense account transactions are primarily generated in batch runs, they cluster separately from customer account transactions. On average there were 9.45 customer account transactions per month in the sample observed, or about one transaction every other day.

Figure 1 -- Customer Account Transactions



Given the nature of the processes that produce transactions, we can reasonably assume that any customer account transaction is equally likely to fall in any relative position among the customer account transactions for a month. From inspection of the application code we know that

every transaction file access is to the entire 300-byte record ($r = \ell = 300$). These records are stored in blocks of ten ($b = 3000$). The statement application uses a combination of transaction data and other data that remained unchanged in the experiment, with transactions accounting for about 15% of the total load. Since the pattern of access is consistent and repetitive (transactions are read for every statement in the midst of other data needed for the statement), and the buffer replacement discipline is known to be LRU (Least Recently Used), we can estimate that 15% of the buffer (450,000 bytes or 150 blocks) will contain transactions at any point in time, and use this amount as the effective transaction buffer size. From the LRU assumption we can use the effective buffer size as an estimate of working set size: $s(\theta) = 450000$ bytes.

The likelihood that a record will be found in a given block in the working set can be approximated as:

$$p = (b/\ell - 1)\ (1/n)\ (b/\ell)$$

where n is the number of records spanned by the run.

Each block in the working set contains b/ℓ records (3000/300 = 10 for this case). Since the block is in the working set we know that one record was used for a prior reference to this or another account. Given the statement application's access pattern we also know that a previously accessed record cannot be the next record needed. Thus $(b/\ell - 1)$ represents the number of records in each block that might contain the next one needed. The divisor, n, is the number of records in the file that might contain the desired record. The factor b/ℓ converts number of records to number of blocks. For statements, only customer account transaction records are included. Since records are naturally ordered by date, only the records in the date range of the statements being prepared are included. For Run 1 we can use $n = 250{,}000$ for a monthly statement run (60% of the average number of transactions per month), 750,000 for a quarterly run, and 3,000,000 for an annual run. We can then estimate

$$f(\theta) \approx \left[1 - p\right]^k \quad \text{where } k = \text{int}\left[\frac{s(\theta)/r}{b/\ell}\right]$$

which gives the rate at which a record is missing from the buffer when it is first referenced. For Run 1 this works out to 94.7% for a monthly statement run, 99.8% for a quarterly run, and in excess of 99.9% for an annual run. From this analysis it is clear that the observed missing record rate of 88.1% is almost completely due to locality in accessing data other than transactions. Since the 88.1% rate is a weighted average of 15% transactions and 85% other data, the missing record rate for the other data must be approximately 86%.

Run 2 used a transaction file partitioned into groups of banks. For Run 3 the file was partitioned into single months. In both cases the transaction file was reordered by bank, account, and entry date within each partition. For brevity, we shall consider only Run 3 since results of both were similar. Let t be the number of transactions for an account for a month. The first of these transactions could fall in any relative position in a block. The remaining $t-1$ transactions follow in sequence. The expected number of blocks spanned by the transactions for an account in a month is, therefore:

$$1 + (t-1)/(b/\ell)$$

Using the frequency distribution shown in Figure 1, the average number of blocks referenced for an account for a month can be estimated at 1.84, to access an average of 9.45 transactions. The average missing record rate is then about 19.4%. Using a weighted average of the estimated missing record rate for transactions on the partitioned and ordered file and the estimated rate for other files determined above, we can estimate the overall missing rate at (.15)19.4% + (.85)86% = 76%, which closely matches the observed 75.8% from Run 3.

The results of estimating the spatial locality measures for the statements sample are shown in Table 4 below. From the locality analysis it is clear from the change in the missing ratio and the volume ratio that reorganizing the transactions file produced an 80% reduction in the transaction file I/O load for statement processing.

Table 4 -- Transactions File Estimated Locality Measures

Parameter / Measure	Run 1 Unpartitioned Entry order	Run 3 Partitioned by month Account/Date Order
Interval width θ	450,000	450,000
Working set size $s(\theta)$	450,000	87,000
Missing record rate $r \cdot m(\theta)$	99.9%	19.4%
Missing ratio $m(\theta)$	.33%	.06%
Volume ratio $v(\theta)$	10	1.9

The improvement found for the nightwork sample was marginal. The estimation of the locality measures for statements was based on the characteristics of the application and the significant role of transactions. The nightwork sample consists of a number of applications, most of which do not access transactions in account groups. Transactions are created in pending trade settlement, income collection, and cash movement applications. These transactions will

not be reordered when read in later processing. The only close analog to statements are transaction journals prepared at the end of the sequence. From the general characteristics of the nightwork sample, we would conclude that there would be little improvement obtained from reorganizing the transactions file.

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. In this paper we have demonstrated how locality measures can be estimated using knowledge about the application and the database. We have also illustrated how these locality measures can be used to predict performance.

This approach has valuable, practical significance, since the locality measures can be obtained directly from a trace of a workload execution. In this case, the trace could have been taken from the runs against the unchanged database (using the command log mechanism provided in ADABAS). The locality measures then would be computed from the single trace for each of the alternative database reorganizations. It would not have been necessary to have expended the enormous cost in people and computer time to actually reorganize the database, reprogram the database access software, or perform test runs 2 and 3. We plan to carry out further empirical experiments of this type on samples from large databases and will report on the results in a future paper.

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.

Denning, P.J. and Slutz, D.R. (1978)
Generalized Working Sets for Segment Reference Strings, *Comm. ACM 21*, 9 (September 1978), 750-759.

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.

Effelsberg, W. and Loomis, M.E.S. (1984)
Logical, Internal, and Physical Reference Behavior in CODASYL Database Systems, *ACM Trans. Database Syst. 9*, 2 (June 1984), 187-213.

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)

Peinl, P., Reuter, A., and Sammer, H. (1988)
High Contention in a Stock Trading Database: A Case Study, *Proc. ACM SIGMOD Conf.*(Chicago, IL June 1-3, 1988) ACM, New York, 1988, pp 260-268.

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.