Sequentiality and Prefetching in Database Systems

ALAN JAY SMITH

University of California-Berkeley

Sequentiality of access is an inherent characteristic of many database systems. We use this observation to develop an algorithm which selectively prefetches data blocks ahead of the point of reference. The number of blocks prefetched is chosen by using the empirical run length distribution and conditioning on the observed number of sequential block references immediately preceding reference to the current block. The optimal number of blocks to prefetch is estimated as a function of a number of "costs," including the cost of accessing a block not resident in the buffer (a miss), the cost of fetching additional data blocks at fault times, and the cost of fetching blocks that are never referenced. We estimate this latter cost, described as memory pollution, in two ways. We consider the treatment (in the replacement algorithm) of prefetched blocks, whether they are treated as referenced or not, and find that it makes very little difference. Trace data taken from an operational IMS database system is analyzed and the results are presented. We show how to determine optimal block sizes. We find that anticipatory fetching of data can lead to significant improvements in system operation.

Key Words and Phrases: prefetching, database systems, paging, buffer management, sequentiality, dynamic programming, IMS
CR Categories: 3.73, 3.74, 4.33, 4.34

1. INTRODUCTION

Database systems are designed to insert, delete, retrieve, update, and analyze information stored in the database. Since every query or modification to the database will access at least one target datum, and since the analysis associated with a query is usually trivial, the efficiency of most database systems is heavily dependent on the number of I/O operations actually required to query or modify and the overhead associated with each operation.

One method used in some systems to reduce the frequency of I/O operations is to maintain in a main memory buffer pool a number of the blocks of the database. Data accesses satisfied by blocks found in this buffer will take place

much more quickly and usually with much less computational overhead. IMS (Information Management System/360) [10–12] uses this strategy, as we discuss later. The INGRES database system [9] makes use of the I/O buffer pool maintained by UNIX [19] for this same purpose. There is considerable scope for optimizing the operation of the buffer pool, principally by controlling the selection of those blocks to enter into the buffer pool and those to be removed from the buffer pool. In this paper we will primarily be concerned with sequential prefetching as an algorithm for the selection of blocks for the buffer; we shall also look briefly at other aspects of the selection and replacement problems.

A. Sequentiality

A characteristic of the database system that we study in this paper, and we believe a characteristic of many other systems, is sequentiality of access. Many queries require scans of an entire database in order to compute an aggregate. An example might be: "Find the average salary of all residents of New York City," which would probably be calculated by scanning sequentially over the appropriate section of the database. In some systems a simple lookup evokes a sequential scan of the entire database, if the database is not indexed on the key, or a partial scan, if the index is not precise. For example, a query about "John Jones" might be answered by using an index to find the beginning of the records for names starting with "J," and then searching forward. "Range" queries, in which the database is searched for records with keys in a range, will also result in sequential accesses, either to the records themselves or at least to an index if the index exists for that field of the record. Implicit in this discussion of sequentiality is the assumption that the logical sequentiality of access is reflected in physically sequential accesses to the stored data. If the physical storage organization of the data differs significantly from the logical one, then little sequentiality can be expected.

A consistently sequential pattern of access will allow us to easily anticipate which data blocks, segments, or tuples are likely to be accessed next and to fetch them before their use is required or requested. In systems in which anticipatory data fetching is less costly than demand fetching, we can expect a decrease in the cost of I/O activity. It is obvious, and under certain conditions has been shown formally [1], that when multiple block or anticipatory fetching is no "cheaper" per block than demand fetching, demand fetching is an optimal policy. We contend, and we shall discuss this in greater detail in a later section of this paper, that fetching segments or blocks in advance of their use, and in particular fetching several segments or blocks at once, is significantly less costly per block fetched (with our cost functions) than individual demand fetching.

B. Previous Research

The prefetching of data blocks into a database system main memory buffer is very similar to the prefetching of program address space (instructions/data) in a virtual memory system. Simple sequential prefetching of pages has generally been found to be ineffective [2, 13, 24], but more sophisticated methods which either analyze the program in advance, accept user advice, or maintain relevant statistics during program execution can significantly improve system operation

[2, 8, 25–28]. Sequential prefetching of lines for cache memory has been shown to work very well [24] because the extent of sequentiality is large compared to the cache page (line) size; for most programs the sequentiality is not large compared to the main memory page size. Prefetching of I/O data streams has also worked well [22, 23] because of the frequent use of purely sequential files.

C. Information Management System/360

The previous research regarding prefetching which is discussed above has all been oriented toward the analysis and use of data describing real system operation. We shall do likewise, and in later sections of this paper we present data taken from an operational and long running Information Management System/360 (IMS), a data management system marketed by the IBM Corporation. This particular system has been in operation for a number of years in an industrial environment. The results observed are believed to be illustrative of an actual application, but they cannot be taken as typical or representative for any other IMS system. Despite this disclaimer, and the fact that our research is based on data taken from a specific installation running a specific database system, we believe that sequentiality is a characteristic common to many database systems, and that therefore the methodology developed in this paper for sequential prefetching will also be applicable to other IMS installations and to other database systems.

In order to provide some background for the analysis of our data, we describe below some of the significant features of IMS. A readable and more complete discussion can be found in a book by Date [5], which also discusses other database systems and organizations. The IMS manuals [10–12] provide much more thorough coverage.

IMS is a hierarchical database system; that is, segments (tuples) in a database are arranged logically in a tree structure such as that indicated in Figure 1. An IMS implementation may consist of several databases, each of which consists of
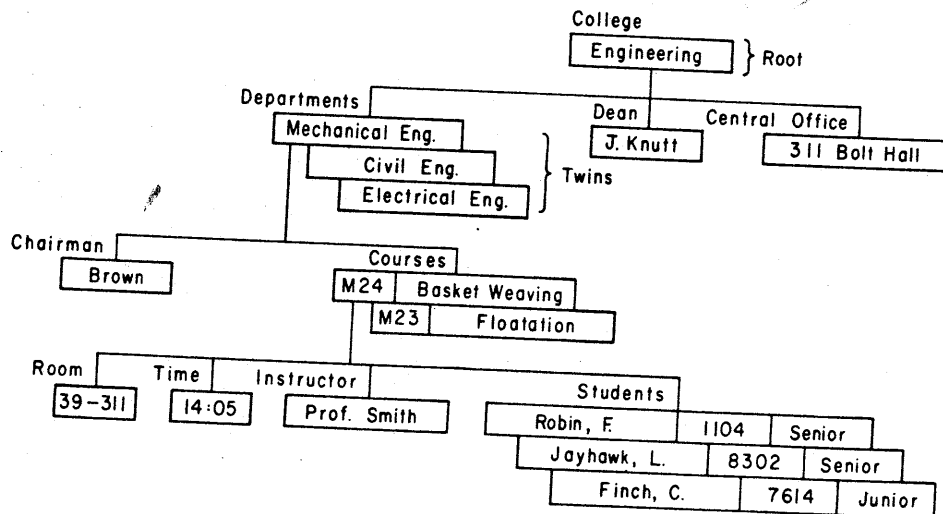

Fig. 1

a large number of such "trees." The roots may or may not be sorted and may or may not be indexed. In the system measured there were several databases; each had the roots indexed. Access to units within a tree is through the root. Within the tree, data is stored in a top-down, left-to-right order. That is, the following items from Figure 1 would be stored in this order: Engineering, Mechanical Engineering, Brown, M24 Basket Weaving, 39-311, 14:05, Prof. Smith, Finch C 7614 Junior, Jayhawk L 8302 Senior, Robin F 1104 Senior, M23 Floatation, ..., and so on.

Two physical storage organizations are possible, hierarchical sequential and hierarchical direct. In the former case segments are examined consecutively in the order in which they are stored in order to find some position in the tree. In the latter case, which describes the databases in this system, certain segment instances are related by two-way pointers and these pointers can be used to skip over intervening data. In either case the data is accessed using DL/1 (Data Language/1), which implements nine operations. These are: GET UNIQUE (find a specific item), GET NEXT (get the next item), GET NEXT WITHIN PARENT (get next item under the same root), GET HOLD UNIQUE, GET HOLD NEXT, GET HOLD WITHIN PARENT, INSERT, REPLACE, and DELETE. In each of the first six cases, the command can or must be qualified to partially or completely identify the tuple in question. GET HOLD (items 4, 5, and 6) is usually user prior to INSERT, DELETE, and REPLACE. Eighty-one percent of all accesses measured (see below) proved to be for lookup only (GET) rather than modification (INSERT, REPLACE, DELETE). Average frequency for INSERT, REPLACE, and DELETE was 11 percent, 0.7 percent, and 8 percent, respectively, with substantial variation between databases.

A search for a uniquely qualified item, such as GET UNIQUE (College(Engineering).Department(Mechanical).Chairman) will involve a search starting at the appropriate root and either scanning in sequential storage order if the storage organization is sequential, or following pointers if the organization is direct. Each of the segments referenced in the course of finding the target segment is called a *path segment*. A "GET NEXT" will also involve either sequential search or search following pointers. Unless the pointers lead immediately to the desired item, a significant amount of sequentiality will be evident in both cases; in following a path to the target the direction of search is always forward in the database.

A segment can be referenced only if it is in memory. For several efficiency reasons, IMS groups segments into fixed size storage units called *physical blocks* (henceforth called *blocks*). In the particular implementation in question the block size was 1690 bytes for data blocks and 3465 bytes for blocks used to hold index entries for the root nodes. The blocks are the unit of physical storage and data transmission; a request for a segment will result in the block containing the segment being transmitted, if necessary, from secondary storage to a *buffer pool* of blocks kept in main memory.

The search for a target segment proceeds as follows:
 i. Determine the first path segment to be examined.
 ii. Search the buffer pool for the block containing the segment. If the block is missing, then fetch the block (and remove a block if necessary to make room).

iii. Find the segment within the block. If this is the target segment, then perform the desired operation. Otherwise, determine the next path segment and continue at step 2.

We note that segments are commonly much smaller than blocks; for our data the average segment size was 80 bytes. A large block size will result in many segments being fetched at once; because of the sequential nature of segment access, this often means that one block fetch will satisfy many segment requests.

D. The Data

The experimental data discussed in this paper was obtained as the result of a two-step process. From the source IMS installation a trace of each DL/1 call issued over a period of a week was made. The key portion of the database was unloaded to tape, and the entire database system was reloaded at the IBM San Jose Research Laboratory. The DL/1 calls were then run against the copy of the database and a record of the target and path segments and block references was made. No effort was made to tune the copied system; the original database design was used without change, except that the reloading ensured that the logical and physical database organizations coincided.

The database system used was IMS/360 version 2.4, running under OS/VS2, release 1.6. The total size of the entire database was about 200 megabytes.

Our data analysis, in later sections, uses three sections of the entire seven-day block reference trace. The part labeled "full tape" is the trace for the first day. This first-day tape actually was generated in seven sections of approximately equal size; the parts of the trace referred to as "part 1" and "part 2" are just the first two segments of this day-long trace.

This data was gathered by researchers at the IBM San Jose Research Laboratory, and further discussions of IMS, the data gathering methodology, and the data analysis may be found in a number of papers. The reader is referred to papers by Tuel and Rodriguez-Rosell [30], Rodriguez-Rosell and Hildebrand [21], Ragaz and Rodriguez-Rosell [18], Rodriguez-Rosell [20], Gaver, Lavenberg, and Price [7], Lavenberg and Shedler [14], Lewis and Shedler [16], Lavenberg and Shedler [15], and Tuel [29] for further information.

2. OPTIMIZATION OF PREFETCHING

A. The Model

As was discussed in the beginning of this paper, the efficiency of operation of a database system can be increased by increasing the fraction of data references captured by the main memory buffer. The only general method known by the author to be effective in increasing the buffer hit ratio (for a fixed size buffer) is to take advantage of sequentiality in the series of accesses—either by prefetching or by increasing the block size. This issue is discussed further in the data analysis section of this paper; for the formulation of our optimization results we rely on a *model* as follows:

i. The physical layout of the database is the linear projection (as defined earlier) of the logical structure of the database. Segments in the database will be assumed to be numbered consecutively from 1 to S. Blocks in the database

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

fastcase
Smarter legal research.