Predictive Caching

Mark L. Palmer
Stanley B. Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-90-29
November 1990

Predictive Caching

Mark L. Palmer
Stanley B. Zdonik

Technical Report CS-90-29
November 1990

Abstract

Prefetching data objects or program pages in advance of their use is a powerful means of improving performance which has proved difficult to realize. Current OODB systems that maintain object caches use policies for fetching and replacing referenced objects approximating those used for virtual memory demand paging. These policies usually assume no knowledge of the future. Object cache managers prefetch objects by using explicit representations of data structure, or employ demand fetching combined with data clustering to effect prefetching. The latter two methods can be awkward to implement and ineffective in systems where encapsulation is present, or where the usage patterns serviced are incompatible.

Recent work at Brown [MP89] has introduced the notion of a black box, the *Estimating Prophet*, that assimilates access patterns of individual users over time and predicts access for each user. The Prophet samples an access sequence on-line and generates predictions which are used to prefetch data. This approach can be supplemental, and is in many instances preferable to prefetching via data clustering, prefetching based on static representations of structure, and demand paging. We present a cache management model for use with prefetching and the design of a Prophet which learns to predict simple access sequences.

1 Introduction

1.1 Problems with Prefetching

One of the worst performance problems in current OODB systems is the time required to fetch objects from secondary storage into a local object cache as they are demanded. In networked designs such as Observer [FEZ89] and ORION-1SX [G+88] the severity of this problem is compounded, as data crosses several I/O boundaries on its path from the server's secondary storage to an application's local memory. Given the high relative cost of performing I/O, the ability to anticipate movement of data across communication links in order to prefetch objects presents a potent means of improving performance. In database systems, however, several difficulties have prevented elegant designs for prefetching.

One traditional method ([MS77],[C+82],[JS84],[CK89],[HK90]) of speeding I/O is to cluster data that tends to be accessed together into units of contiguous storage, artificially creating locality of reference. This also allows a server to prefetch entire clusters for a client in response to a request for any member of the cluster, as in Observer [HZ87]. However, a major purpose of databases is to allow data sharing. The reason that prefetching clusters can be ineffective is that different users often follow incompatible patterns of access to the same set of data. Devising a partitioning of data which is both equitable and efficient then becomes problematic, and finding an optimal partitioning can be [BN78] computationally infeasible.

Another approach to prefetching is to construct explicit representations of static structure in the data to determine possible future accesses during execution. The DBMS maintains usage statistics with the structure to use in deciding which objects to prefetch. However, access patterns are often directed by data values, the organization of which may not be explicitly described by a structure in advance. A further difficulty arises when strict encapsulation is desired. OODB systems maintain encapsulation by disallowing direct access to the internal states of objects, enforcing information hiding. Allowing structures to be defined over arbitrary types and making knowledge of these structures available to other parts of the system can violate encapsulation.

A final difficulty with determining good clustering or secondary access structures arises from the lack of generally useful access monitoring and analysis tools, and often of human resources to tune performance for specific users. Consequently, decisions to cluster or provide secondary access structures usually rely on the database designer or administrator's intuitions about access patterns. The database administrator decides when and how to periodically re-configure the database as changes in access patterns and data make the old physical structures inefficient. This tuning process is very much labor and expertise intensive.

1.2 Solution: Adaptive, Predictive Prefetching

The Estimating Prophet is an experimental prototype of a "black box" component which learns the access patterns of each user over time well enough to predict object access for individual users. During a database session, the Prophet monitors client/server communication and generates access predictions which are then used by the object cache manager to prefetch data - which may or may not be clustered. This solution addresses the difficulties described above and has several desirable properties. The Prophet gains experience with each access pattern individually, and tailors its predictions for each user according to that user's access history without considering the usage patterns of others. Also, the Prophet monitors access sequences in a non-invasive way, requiring no knowledge of the data model or schema. Its operation is automatic and invisible to the database administrator, requiring little in the way of time, intuition, or expertise to operate. Through continued monitoring of access patterns, the Prophet adjusts its predictions to reflect changes in usage quickly, incrementally, and in a uniform way. Existing OODB systems which already fetch demanded objects into a client cache may be able to add predictive prefetching without major architectural changes, since the concomitant cache management model is a simple refinement of common demand fetching strategies. A major concern about augmenting an existing system in this way is whether the performance could be ultimately degraded instead of improved. The cache replacement policy, MLP, is designed to ensure that the faulting performance of a system which employs predictive prefetching will not be worse than if it employed the common cache management strategy of demand-fetching and replacing the least recently used objects.

1.3 Paper Structure

This paper summarizes current research, which consists of two primary subjects for exploration. The first subject concerns how to best incorporate predictive prefetch activity into cache management, given the costs of prediction and prefetching. The second topic involves studying various means of implementing the prediction component. Accordingly, this paper presents a model for predictive cache management and outlines a design for a Prophet that recognizes and predicts access patterns on-line.

Section 2 describes the way the Prophet fits into a typical OODB design. In section 3, a cache model for managing prefetches is presented, and its faulting behavior is discussed. An algorithm which learns to predict simple access patterns is presented in section 4, along with a broad description of the resource costs of the design. Metrics for measuring prediction performance are discussed in section 5. Section 6 characterizes the kinds of applications which will benefit most from predictive caching, discusses other applications and related work, and offers some concluding observations.

2 System Component Overview

The purpose of this section is to summarize only the aspects of the encompassing database architecture which are essential to illustrate the operation of the Prophet. This description covers a subset of functions provided by the system, which entails some simplifying assumptions. For example, the problems of variable object size, cache validation and distributed concurrency control are assumed to have feasible solutions. The problem of managing a cache of variable-sized objects is orthogonal to the issues of accuracy and costs of prefetching examined in this paper. The distributed system architect must consider methods of validation and synchronization of cached objects against other caches and secondary storage. However, these issues have been addressed by others (e.g. [G+88]), and are not covered here. Prefetched objects are assumed to be locked and validated as if they were demanded. Low contention for write locks is assumed, not unrealistically since design applications often have high read/write ratios ([RKC87], [CK89]). The operation of predictive caching per se does not rely on these simplifications.

2.1 Database Server

The Prophet operates within an OODB system where applications retrieve objects from secondary storage and cache them in local memory. This arrangement is known as a client/server, or interpreter/storage manager, architecture. It consists of a central server machine responding to requests for data shared between applications running independently on workstations. The target system is Observer [FEZ89], a database server that runs on a central machine and acts as a typeless back end to applications, managing all access to the secondary storage of the database. Strong identity [KC86] is maintained by the system, which is important because it simplifies recognition of regularly reoccuring parts of an access pattern. Identity is provided for each object via an external unique identifier (UID) which acts as an immutable handle for the object and is not recycled. Some OODBs assign meanings to individual bits of a UID, however these are not of concern to the Prophet, which interprets all UIDs as symbols. In Observer, objects can be clustered into segments and can migrate and be replicated between segments. Whole segment clusters can be prefetched in response to a read, but this feature is not used when predictive prefetching is in effect.

Observer provides operations for object read, creation, deletion, and update which are activated by messages from the application. To facilitate prefetching, the basic read function is augmented in several ways. A read message supplies a list of UIDs to the server, which gets the identified objects from disk or a server-managed buffer and returns them to the requestor. The requestor marks UIDs for either *demand* or *prefetch* processing. In order to ensure that outstanding prefetch requests do not delay completion of demand requests, the server implements a *pre-emptive* read operation which the client uses for all demand requests. The server maintains separate queues for pre-emptive reads, and services all pre-emptive reads before any other read. The network expedites the sending of demand (i.e. pre-emptive) read and completion messages over prefetch messages, and pre-emptive reads complete in-order.

2.2 Client

The workstation application needs a way of interfacing easily to the server. The ENCORE client component provides this by acting as an interpreter, mapping the data model used by the application onto operations understandable by Observer. ENCORE implements object type semantics - executing methods and enforcing encapsulation, and is typically bound into the application's image. The client also handles validation of the state of local object copies with respect to what is stored in the server, and supports other database functions related to persistence and distribution of objects, but the operation of these is unrelated to the prefetch mechanism.

For the purposes of this paper, the salient responsibility of the client is that it allows the application to reference objects by UID, without knowledge of how or where they are stored. The client ensures that objects referenced by an application are ferried between the server and the applications's local memory transparently. It is the performance of this responsibility that is greatly improved by prefetching. ENCORE maintains a cache of objects which are being currently used by the application, sending demand requests to the server when the application references an object not in cache and "flushing" modified objects back to the server's secondary storage as needed. Several current systems (ORION [G+88], Mneme [M89], O2 [B+86], and LOOM [KK83]) have similar architectures. They take various approaches to translating object references to memory addresses, but their common feature is that the application references each object via its UID. The client then either hashes or indexes a Resident Object Table (ROT) to obtain a pointer to the object's location in memory, and may augment the UID with the pointer. In an environment where the client is on a workstation, object caching is desirable because it is effective in reducing network traffic [RKC87]. However, the relative cost of each fault is still quite high. The disparity between secondary storage and main memory access times is growing as compute technology matures [G87], and is magnified by "staging" objects at multiple I/O boundaries as it travels from disk to application. Thus, the ability to use compute time to predict access and prefetch objects will continue to present a desirable means of improving performance.

2.3 Prophet

The job of the Prophet component is to predict the next references to be made during a user session. The Prophet can be configured as a separate process or embedded in the client image. It has two primary modes of operation: *training* and *prediction*. Given a small sample of the current access sequence, the Prophet predicts which UIDs will be requested next. The Prophet "learns" access patterns over time via training, and becomes increasingly better at predicting accesses, until it reaches a stable state where learning ceases. This state may be reached because the Prophet is predicting well and not encountering any changes in access pattern, or because it has exceeded user-specified resource parameters. The information the Prophet stores about an access pattern known as *pattern memory*.

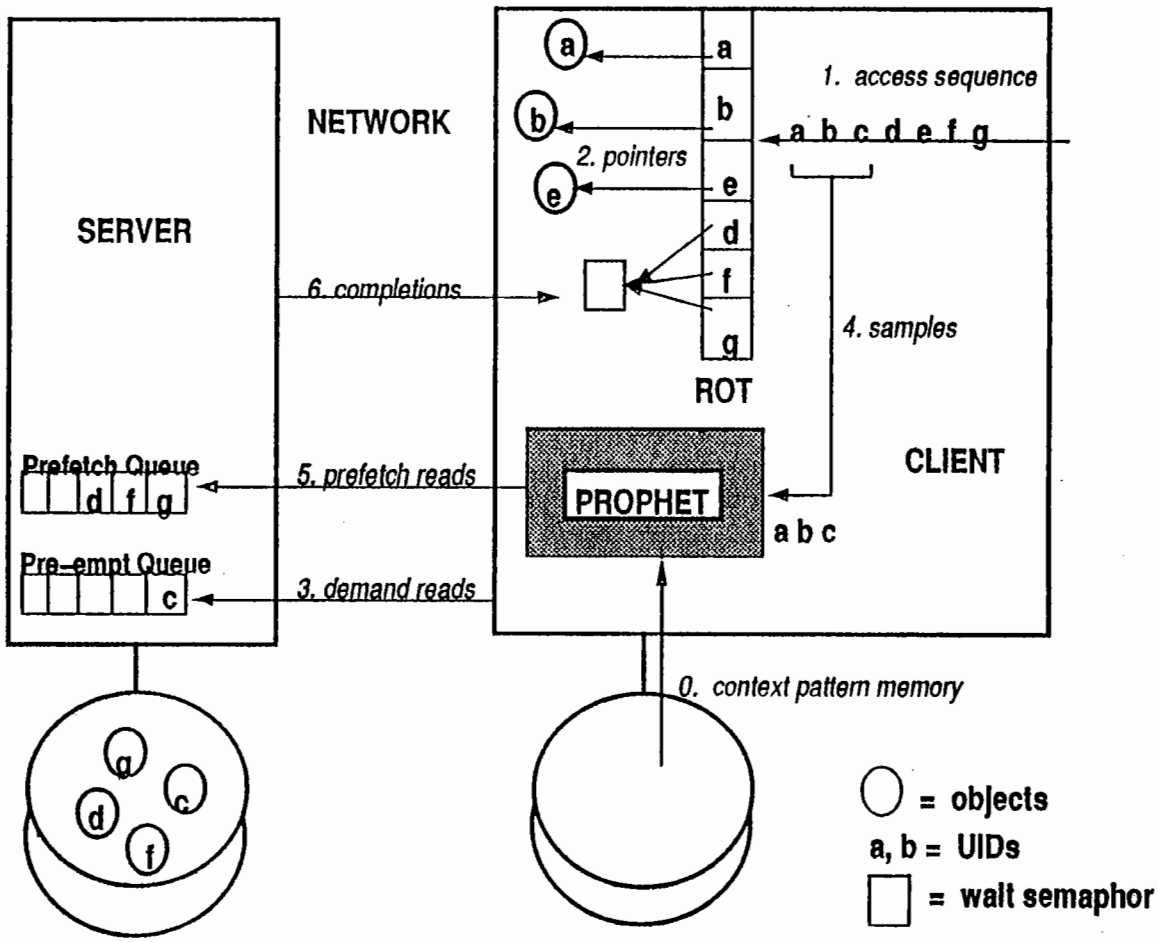


Figure 1: Prophet, Client, and Server

Figure 1 shows how the Prophet interacts with the client and server. UIDs are represented as lowercase letters and objects as circles. As the application begins a session, the Prophet loads in (0) the pattern memory for a context. The application makes a sequence of references (1) to UIDs "a b c d e f g...", to be handled by the client object cache manager. The client converts these UID references to memory addresses of objects using a Resident Object Table (ROT). References to UIDs a or b return pointers (2) to those objects in the local cache. If an object (e.g. c) is referenced but not found in cache, the client issues a pre-emptive demand read (3) to the server and blocks. During object access, the Prophet samples the current reference sequence (4), recognizes the beginning of a pattern, and completes it - predicting "d e f g". The client sends a prefetch request (5) for objects d, f, and g - whose UIDs were not found in the ROT - to the server prefetch queue, and points the ROT entries for objects being prefetched to the flag which will be set by the prefetch read completion (6).

The Prophet learns and recognizes access patterns specific to individual *contexts*. It uses a context identifier (CID) to associate access traces generated by the same source. The client assigns a CID to indicate the unique combination of user, application, and sometimes function which generated a particular access sequence. For example, a CAD application might provide one function that displays a diagram of a circuit board design, and another function to allows ad hoc queries. The access pattern of the display function might be very predictable, allowing the Prophet to learn it efficiently, while sequences generated by ad hoc queries might be arbitrary and difficult to learn. The client would establish different contexts corresponding to these two functions, even when the functions are invoked by the same user. By using contexts, a designer can exploit knowledge of an application to "focus attention" of the Prophet, preventing conflicting access patterns from interacting in pattern memory. This reduces pattern memory size and speeds prediction within a context. The client stores a pattern memory for each unique CID separately between sessions and reloads it at the start of any session which invokes that context.

2.3.1 Training and Prediction

The Prophet learns an access pattern through *training*, the first of its two modes of operation. The client records traces of object references generated during a session within each particular context. The training algorithm then processes each reference trace - normally (but not necessarily) off-line, between sessions, and incrementally improves its pattern memory for that context. Training reinforces sequences which are seen frequently from session to session in pattern memory, while sequences which appear sporadically or which consist of obsolete information are repressed. This causes pattern memory to focus on regularly reoccuring phenomena and develop an internal prototype, or generalization, of each access pattern.

The second mode, *prediction*, operates during an ENCORE/Observer session to determine which objects to prefetch. During initialization, the pattern memory for a specific context is loaded into the Prophet. During the session, current UID accesses move through a sampling window. The client sends the sample periodically to the Prophet's prediction algorithm. The Prophet returns a prediction and provides some information regarding its confidence in the accuracy of the prediction, which the client uses to prefetch data.

A set of parameters (see section 4.6) dynamically throttles the performance of both training and prediction modes. For example, the maximum amount of memory and time used to generate a prediction can be limited. The minimum confidence level required before a prediction will be prefetched can also be set. A system designer can use these and other parameters to exploit knowledge of an application's semantics and access patterns, or leave the parameters to "self-adjust" over time.

3 Predictive Cache Management Model

As mentioned above, the client ferries objects referenced by the application between the server and the applications's local memory transparently. Assuming there is some way of making predictions, the client must have a way of managing prefetched objects. This section outlines a cache management model for use with predictive prefetching that extends ENCORE's current cache management algorithm. ENCORE keeps a set of objects in a local object cache, and maintains a table of resident objects (ROT) to resolve application UID references. Each ROT entry includes a main memory address and timestamp for the object in cache. When an application references a UID not found in the ROT, the client issues a demand read to the server and blocks. When the read completes, the demanded object replaces the object with the lowest timestamp value, i.e. ENCORE's current cache manager implements a Least-Recently-Used (LRU) [HS90],[ST85] replacement policy.

When adding prefetching to a client that uses demand fetching and LRU replacement, an important question is whether enough prefetch errors could actually degrade performance. A predictive cache manager should guard against the effects of poor predictions. There are three ways that prefetching could degrade performance - 1) by causing additional object faults, 2) by congesting the communications subsystem, and 3) by adding prediction overhead to the cost of accessing objects in cache. To avoid incurring these costs, the cache manager must attend to three things. First, it must ensure that the faulting behavior does not degrade, i.e. - that bringing objects into cache in anticipation of their use does not cause faulting of objects actually used, since prediction errors will occur. Second, the cache manager must avoid loading the communication bandwidth with prefetches to the extent that it delays demand requests. Third, the cost of computing predictions during object access must be kept as small as possible. The following subsections address these concerns.

3.1 "Most Latter Predictions" Replacement Policy

The predictive cache manager should ensure that its handling of prefetched objects does not cause more faults than would be the case were traditional cache management used for the same application. The set of decisions a cache manager makes about which objects to replace with new objects is known as a *replacement policy*. The strategy the Prophet's replacement policy follows is to limit the space occupied by unreferenced prefetches in cache and assign timestamps such that unreferenced prefetches have lower timestamp values than referenced objects. Normal LRU replacement is then allowed to operate, causing all prefetches to be evicted before any referenced object, and ensuring that the remaining cache space (containing referenced objects) behaves as if demand LRU were in operation if no correct prefetches are made. The replacement policy also governs how prefetched objects interact when arriving prefetches replace existing prefetches, evicting those prefetched objects which are the Most Latter Predictions (MLP) within each estimate of the future.

3.1.1 Operation

A predictive cache $C = \{R \cup P\}$ mixes two disjoint sets of objects in memory - R, the set of most recently used objects, and P, a set of prefetched objects which are unreferenced. Two virtual time counters are used - T_P , which is the largest prefetch timestamp, and T_R , the timestamp of the object last referenced. Both virtual timers are advanced in a positive direction in such a way as to guarantee that T_P never "catches up" with T_R . Each time a non-resident object is referenced it is faulted in, and T_R is incremented by p_{max} to timestamp the fetched object. Prefetched objects, however, are timestamped by starting with T_P and decrementing in a *negative* direction, *in order of expected reference.* T_P is only incremented when objects are evicted from R, and the new value of T_P cannot exceed the value of the evicted object's timestamp (see figure 2). Because LRU always evicts the lowest timestamp first, T_P will always be less than any timestamp in R. Objects are moved from P to R by stamping them with the value of T_R , but may not migrate from R to P.

The cache management algorithm handles one of two situations on a reference to an object o with timestamp o.time at cache address o.ref. An object which is being prefetched from the server is marked by the set flag o.promise, in which case o.ref points to the I/O completion semaphor that gets set when the object arrives. The following algorithm illustrates how T_R is advanced and when prefetching is begun:

1) If ($o \in C$)
BEGIN
PREFETCH(o)
If (o.promise) WAIT(o.ref)
RETURN(o.ref)
END

2) If ($o \notin C$)
BEGIN
QUEUE-DEMAND-READ(o,Faultflag)
PREFETCH(o)
WAIT(Faultflag)
$o1 \leftarrow$ LRU(C)
If ($o1 \in R$) $T_P \leftarrow T_P + p_{max}$
REPLACE($o1$, o)
END

$o.time \leftarrow T_R$
$T_R \leftarrow T_R + p_{max}$

PREFETCH is called on each reference but when a fault occurs, PREFETCH executes during the demand read. PREFETCH calls PREDICT, which may generate a prediction[1], ω , consisting of $o_1 \ldots o_{|\omega|}$ UIDs, arranged in order of expected access. When a prediction is generated, the cache manager begins a prefetch read for predicted but non-resident objects. Note what happens to objects $o \in C$ which are predicted. If an unused prefetch $o \in P$ is re-predicted, its timestamp is left untouched to avoid "demoting" it and making it a more likely candidate for eviction. If $o \in R$ is predicted, it is timestamped with T_R to "promote" its status in cache.

PREFETCH(o)
If (PREDICT(o,ω) > Threshold)
BEGIN
request $\leftarrow \omega - \{C \cap \omega\}$
QUEUE-PREFETCH-READ(request,Waitflag,COMPLETION)
$\forall o \in$ request: o.promise $\leftarrow$ T; o.ref $\leftarrow$ Waitflag
$\forall o \in \{R \cap \omega\}$: o.time $\leftarrow T_R$; $T_R \leftarrow T_R + p_{max}$
END

When prefetched objects arrive, the COMPLETION routine executes asynchronously, and decides which objects to evict when placing prefetches in cache. Because $T_P < T_R$, unused prefetches $o \in P$ are the first to be evicted by incoming prefetches. The promises entered in the ROT are "pinned" for the duration of the prefetch read, and at completion the promises are updated with cache addresses and timestamped:

COMPLETION(ω)
for $i = 1, |\omega|$
BEGIN
$o1 \leftarrow$ LRU(C)
If ($o1 \in R$) $T_P \leftarrow T_P + p_{max}$
EVICT(o1,C)
END

for $o_{i=1 \ldots |\omega|} \in \omega$
BEGIN
o_i.time $\leftarrow T_P - i$
o_i.ref $\leftarrow$ addr(o_i)
o_i.promise $\leftarrow$ FALSE
INSERT(o_i, C)
END

[1] actually, the prediction may consist of several alternate estimates of the future, interleaved - see 4.4

The rule MLP follows in deciding which unused prefetches to evict is adapted from the proven optimal replacement algorithm for demand paging, OPT [HS90]. OPT always replaces that item in cache which will be referenced furthest in the future, but necessarily operates off-line. MLP replaces those objects which are *expected* to be referenced furthest in the *forseen* future. This rule is implemented by the timestamping mechanism described above. The Prophet generates a prediction, ω , consisting of $o_1 \ldots o_{|\omega|}$ UIDs, in estimated order of access. The cache manager timestamps the objects in a negative direction starting with T_P , so that the last UID, $o_{|\omega|}$ gets the lowest timestamp value of ω , and is later evicted first by MLP. Successive predictions may add prefetches with duplicate timestamps, but as T_P is advanced, the more recent prefetches receive higher timestamps, causing prefetches from older predictions to be evicted first. Among unused prefetches resulting from a single prediction ω , the lowest timestamps are always on those objects whose UIDs appear in the latter part of ω , since these objects are anticipated to be accessed furthest in the future. For $|\omega| < p_{max}$ arriving prefetches, MLP replaces $|\omega|$ objects having the lowest timestamp values with arriving prefetches. The difference between MLP and OPT is that MLP uses estimated and partial knowledge of the future, instead of the perfect and complete prescience assumed by OPT.

3.1.2 Faulting Behavior

It is possible to ensure that the faulting behavior of a predictive cache will not be worse than that of a demand LRU cache, for the same access sequence, by adding extra space to the predictive cache. The worst case is where some maximum number of predictions are generated on every reference, and all predictions are wrong. However unlikely this worst case scenario is, it is instructive to consider because it illustrates how prediction accuracy affects fault rate, and suggests that extra cache space is not required for an expected-case scenario. The cache manager maintains several invariants:

Invariant 1 *The timestamps of all $o \in P$ are always less than timestamps of any $o \in R$.*

Invariant 2 *The objects $o \in C$ replaced by x arriving objects (demanded or prefetched) are always those with the x lowest timestamps.*

Invariant 3 *P may not contain more than p_{max} objects at any one time.*

Claim 1 *Invariants 1,2, and 3 ensure that the worst case faulting behavior of predictive cache $C = R \cup P$, of size $r_{max} + p_{max}$ equals that of a demand LRU cache R of size r_{max}, processing the same sequence.*

To compare the behavior of a demand LRU cache with that of a predictive cache for a given sequence, observe what happens in each cache as the number of referenced objects grows and then stabilizes. Let $p = |P|$, the number of prefetched objects under MLP. Let m be the number of referenced objects under MLP, l the number of referenced objects under LRU, and c be the number of correct prefetches.

Case 1: $l, m < r_{max}$

Initially, $m = l = 0$. With each new reference, m and l increase as object fault into R.

LRU: produces exactly r_{max} faults in filling cache, i.e. before $l = r_{max}$.

MLP: produces $r_{max} - c$ faults before $m = r_{max}$. As many as p_{max} prefetches may be present in cache on each reference, but by Invariant 3, new cache locations are used until $m + p = r_{max} + p_{max}$ and the cache is full. By Invariants 1 and 2, no $o \in R$ is evicted before the cache fills. In the worst case, $c = 0$ - no are prefetches correct, and $m = r_{max} = l$ faults are produced.

Case 2: $l, m \geq r_{max}$

LRU: Once $l = r_{max}$, l does not decrease, since objects are only replaced. The LRU cache (by definition) consists of the r_{max} most recently used objects.

MLP: After $m = r_{max}$ m can increase, but in the worst case, each time a demanded object sets $m = r_{max} + 1$, p_{max} erronious prefetches then arrive to replace P, and one demanded object (by Invariants 1 and 2), before the next reference. But by Invariant 3, m will not decrease below r_{max}, so R consists of the r_{max} most recently used objects.

Prediction accuracy governs the fault rate, which in turn determines cache memory utilization. If $c > 0$ and $p < p_{max}$, objects are moved from P to R. R then consists of the $m > l$ most recently used objects, a superset of the l most recently used objects maintained by LRU. For each prediction error, one prefetch location and one demand location are required to ensure that MLP will not produce more faults than demand LRU would for the same sequence. As prediction accuracy increases, the space which must be reserved to store demand faults, m, decreases - as does the space p required for unused predictions. An interesting case occurs when $m = p$ and the prediction accuracy is fifty percent. In this case, only $m/2 + p/2 = r$ locations are required for MLP to approximate the fault rate produced on the same sequence by demand LRU. This suggests that in practice, if a minimum prediction accuracy is expected (see MinAccuracy parameter below) or can be guaranteed, p_{max} can be set as a function of prediction accuracy.

3.1.3 Example

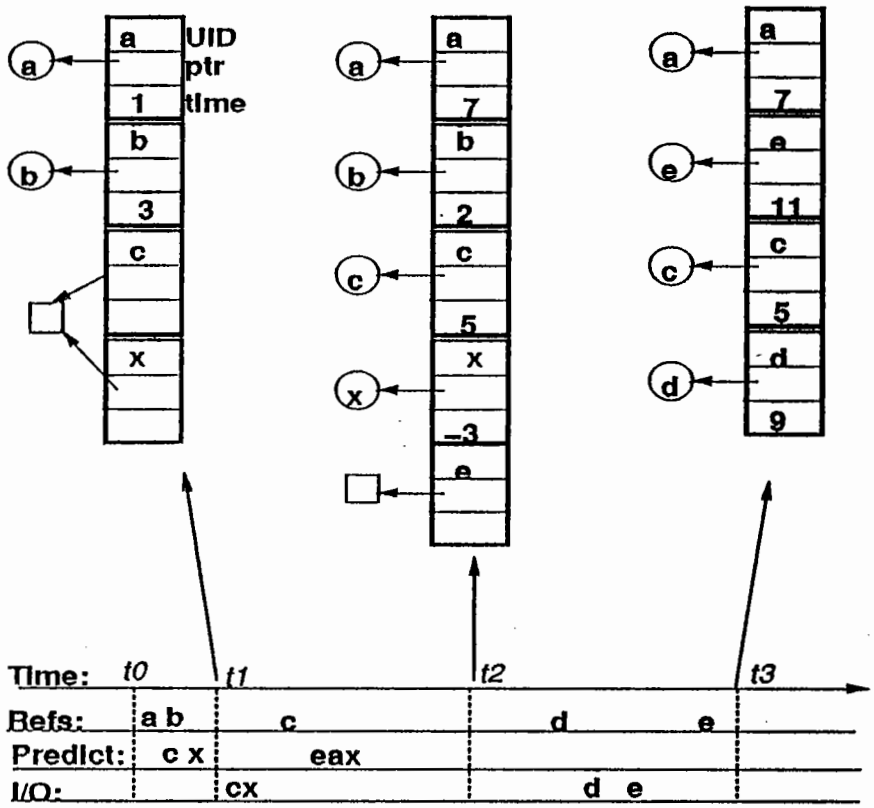


Figure 2: Example of MLP Operation

Figure 2 illustrates some of the situations handled by MLP by showing "snapshots" of the ROT at different times during processing of an access sequence. The cache dedicates space for two prefetch and two demand objects. Three events are depicted along the timeline at the bottom: references, predictions, and I/O completions. Each ROT entry includes UID, pointer, and timestamp fields. Between *t0* and *t1*, the application references UIDs a and b - objects a and b are already in cache and are stamped $T_R = 1$ and 3. On the reference to b, UIDs c and x are predicted and a prefetch read for c and x is sent. "Promises" for c an x are entered into the ROT, pointing to the read completion semaphor. Between *t1* and *t2*, the prefetch read completes and updates ROT entries for c and x, giving them real pointers. Timestamps T_P-2 and -3 are initially used for c and x. On the next reference, c's ROT entry gets T_R =5. The Prophet predicts "e a x", but only issues a prefetch for e, since a and x are already in cache, adding a ROT promise for e. Note that now a has timestamp T_R=7, as if it had been referenced again instead of predicted, but x's timestamp is unchanged. After *t2*, d is referenced but is not in cache - so the client issues a blocking pre-emptive read for d, which completes before the prefetch of e. Since the lowest timestamp in C is on x, the cache manager evicts x and replaces it with d, using T_R=9. Next, a reference to e finds the dummy ROT entry and blocks (not issuing a read) until e's prefetch completes. At read completion, e gets timestamp T_P =-2 but the reference then stamps it using T_R =11.

3.2 Softening Prefetch Collisions

The second design goal mentioned above is that prefetch requests should not impede the processing of demand requests. Several extensions to the client and server have been defined to prevent prefetching activity from interfering with normal demand communication. The server distinguishes and favors demand reads over prefetch reads. The client issues prefetch reads to the server without blocking, and handles prefetch read completions asynchronously. On each cache miss, the client issues one pre-emptive demand read and blocks until it completes. When a prefetch read is issued, the UIDs requested are entered in the cache residency table as "promises". If an application references the UID of an object which is promised, the client blocks until the prefetch read completion routine marks the entry as resident, without issuing any demand read. This ensures that at most one pre-emptive read per session is outstanding at a time, and prevents an application from waiting for any object which it does not need.

Another factor affecting the worst-case cost of predictive caching is the communications cost of prefetching bad predictions. The worst situation imaginable is where some prefetch read(s) are always queued to the server when each object fault occurs. This is called a *prefetch collision*. Prefetch collisions add at most a small expected-time constant (the cost of suspending or waiting for one prefetch read to complete, by our assumptions about the server) to the cost of each object fault. The end result of handling many prefetch collisions is similar to the effect of slightly increasing disk latency for demand reads. However, increased disk latency can in turn be compensated by using larger disk buffers [HS90]. In addition, if many prefetches are made, anticipating far into the future, the possibility of new prefetch requests being delayed by "old" prefetch requests arises. This situation can be avoided by keeping the number of outstanding prefetch reads small and providing a mechanism to cancel old prefetch requests. Given current communications technology, prefetch collisions should be rare in practice.

3.3 Minimizing Prediction Overhead in Object Access

The third way that predictive prefetching could potentially slow the system is by incurring the overhead of calculating predictions in the cost of accessing objects, but this overhead is easily paid for, in an amortized sense, by the benefits of prefetching.

The PREDICT algorithm executes on every few object reference(s), adding to the cost of accessing objects which are in cache. The client must check whether each UID predicted is already in cache before beginning to prefetch it from the server. This adds the cost of several more ROT searches to the cost of cache access. Because of the high relative cost of I/O, one correct prefetch will "pay" for many prediction computations. Additionally, as prediction accuracy increases, cache space requirements decrease. This means that a smaller object table is needed, and that each UID lookup becomes faster. Finally, the client can use context and parameters (see MemMax, below) to limit the size of pattern memory and the worst-case time for generating predictions.

4 Estimating Prophet

Previous sections of this paper have outlined a model for managing a cache with prefetched objects but have assumed an ability to predict references. This section presents a simple design intended to illustrate how a Prophet can learn to predict. To understand why this design is presented, an historical aside is in order. The first implementation of the Prophet [MP90] was constructed using a neural network similar to that described in [KA84]. Initial experiments with that implementation indicated that learning and prediction of access patterns were indeed feasible. However, a model of how such predictive capability could actually be incorporated into a system was needed, as was some analysis of the possible performance costs. Observations of how the neural net worked suggested an implementation using traditional technology which would imitate some functioning of the neural net but have better performance. Thus it is important to note that this design is presented in the context of explaining the Prophet concept and examining database issues, but that it performs limited kinds of learning useful for relatively simple access patterns. Other learning algorithms are currently being explored for use in pattern memory.

4.1 Operation of Pattern Memory

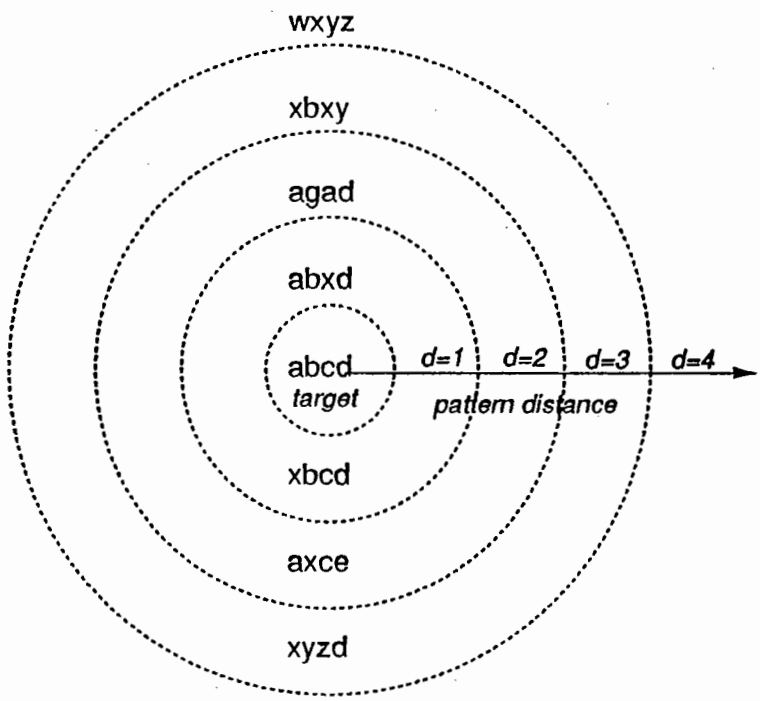


Figure 3: Distance Between Unit Patterns of Size $|\sigma| = 4$

The principle behind the prototype pattern memory is simple and centers around a *nearest-neighbor* pattern recognizer. A *unit pattern* is a row of UIDs, interpreted as symbols with a partial ordering. Figure 3 shows the distance (i.e.number of column mismatches) between one target pattern and a set of unit patterns. Similar patterns are "near" each other.

A pattern memory consists of $j = 1 \ldots n$ unit patterns, denoted as Sigmas (σ_j) in equations. Unit patterns are actually exemplary samples of the reference traces saved by the client. Any given unit pattern represents all observed access sequences within a predetermined similarity distance, or radius, from the unit pattern in pattern space.

Figure 4 shows how a pattern memory is structured as a table of n rows and $|\sigma|$ columns. Each row stores a unit pattern. Each unit pattern is logically divided into a *prefix* and *suffix* (denoted as alpha (α) and omega (ω) in equations). The prefix functions as a trigger for the suffix of each unit pattern during prediction - an input prefix is matched against all prefixes, incrementing a count of matches between the input and each prefix. During prediction, the Prophet only uses enough indices to match inputs against the $|\alpha|$ prefix columns, but during training it constructs indices for all $|\sigma|$ columns. The indices speed the nearest-neighbor function NN, a pattern recognition algorithm central to both training and prediction. In this illustration, the input prefix "d a g" has matched the last two unit pattern prefixes equally well. Since the prediction buffer is small, only the suffix from the unit pattern with the better rating is output.

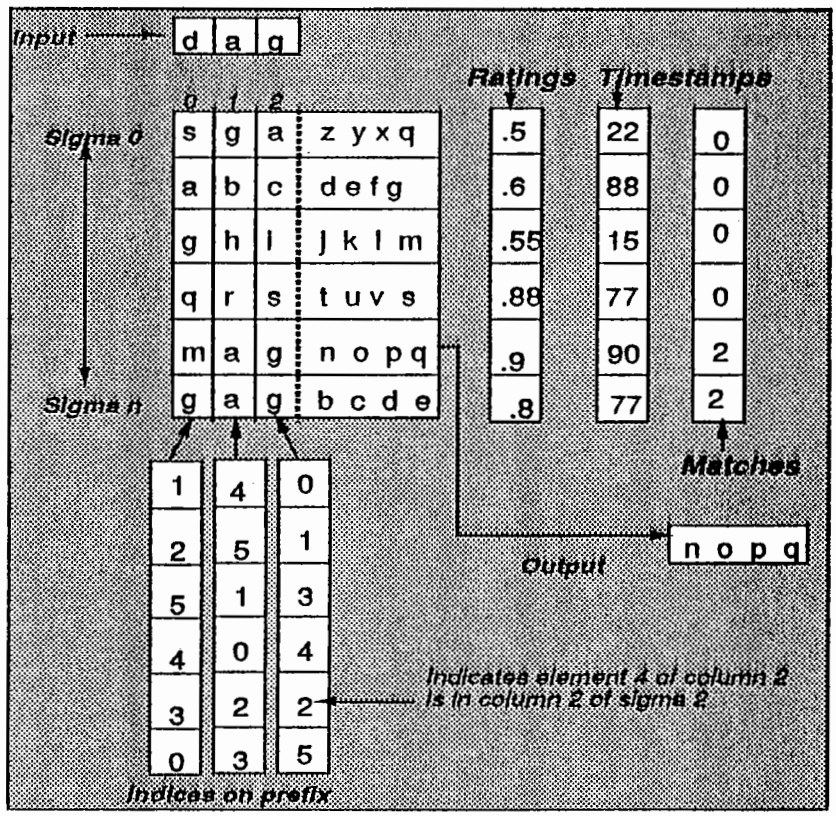


Figure 4: Structure of Pattern Memory

4.2 Costs of Nearest-neighbor Finding Using NN

The nearest-neighbor function NN counts the columns with matching values between a target and all unit patterns. NN finds all neighbors of a target pattern by performing binary search for each UID symbol in column k of the target in column k of the table. If a match is found at row j, the match tally for σ_j is incremented. The index represents the ordering of each column, so all further matches in column k are found by "seeking" up and down the index (starting at row j) until a mismatch occurs in each direction, and incrementing the match counter for each unit pattern in between. This process is repeated for all target columns, leaving the match tally for σ_j equal to its "closeness' to the target.

Both training and prediction utilize the NN function heavily - the difference is that training uses NN to match across all columns of the unit patterns, while prediction only matches the few columns of the prefix. NN operates in log time of the size of pattern memory. Matching a target pattern of k columns requires k binary searches, one for each of k columns of the pattern memory, plus some number of linear "seek" comparisons to find other near neighbors in each column. The number of seek comparisons is bounded by fk where f is the number of neighbors actually found. If the pattern distance between any two σ_j in the system is small, many near neighbors might be found for a given pattern - the worst case would be to find that all n patterns match the target somehow. However, if the minimum distance between patterns is kept above a user-specified threshold (see discussion of Resolution parameter below), the expected value for f is reduced to a small number. Thus the worst case cost of finding $f = n$ neighbors among n patterns for a target with k columns is bounded by:

$$O(k\log(n) + kn) = O(k\log(n) + n),$$

and the expected cost, where f is the expected number of neighbors, is:

$$O(k\log(n) + f)$$

Over many training runs, the set of unit patterns which comprise the Prophet's pattern memory is forced to evolve and adapt by performing credit assignment to individual unit patterns. Only those unit patterns which produce good predictions are allowed to survive over time. Implementing this "survival of the fittest" patterns requires maintaining a couple of columns of values for rating each unit pattern according to its age and historical accuracy (see below), in addition to the column used to tally matches. Thus the storage required to store n patterns of length $|\sigma|$ for matching against a sample of size k is about $O(n(\ |\sigma| + k + 3))$. This design of pattern memory is not storage optimal and could be improved.

4.3 Training Mode

The goal of training mode is to force pattern memory to adapt over time so it consists of unit patterns which are commonly encountered and are exemplary of the way the application accesses data. After each user session, the client saves a trace of all references generated within each context - prefaced with the appropriate CID, then invokes the Prophet's training algorithm to process the trace. There are two phases to the training algorithm: recognizing new mutations in an access pattern, and "natural selection", or determining which unit pattern samples will survive.

Each time a user invokes an application, he or she may try new functions or access data in different ways, revealing more about the total access pattern. To recognize new mutations in an access pattern which appear from one session to the next, the Prophet scans an input access sequence of UIDs by shifting it through a window of unit pattern length ($|\sigma|$). Each time a shift is done, the nearest neighbors of the window contents are found as described above. Each access sequence is thus matched against the pattern memory, and portions of the access sequence which resemble some unit pattern are marked. When parts of a sequence appear which are different from anything seen before (appearing as "gaps" large enough to constitute unit patterns), the training algorithm adds them to the pattern memory and timestamps them. On the initial training run, pattern memory is empty. This causes all unique parts of the first reference trace to be chopped into overlapping unit patterns and stored in pattern memory. If successive access sequences do not change much over time the pattern memory reaches a stable state where no further unit patterns are added because they have all been "seen" before.

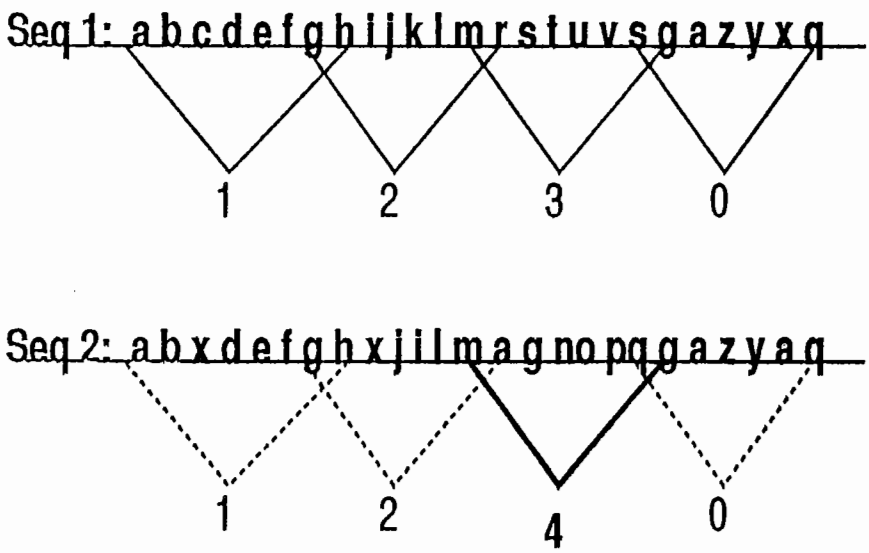


Figure 5: Detecting Mutations Between Sequences

Figure 5 illustrates how mutations are identified. The Prophet, during training, breaks Sequence 1 into four unit patterns. Training on Sequence 2, it recognizes minor variations (which are within a specified pattern radius) of three unit patterns (1,2, and 0) it has seen before, and identifies a new unit pattern (4) to add to pattern memory.

Some access sequences may appear a few times but never again, or may occur with low frequency from session to session. Also, the window boundaries of unit patterns are initially arbitrary. If sample boundaries fall in the wrong places, the resulting unit patterns will not function well as predictors. Over time, the pattern memory must adapt its set of unit patterns to better reflect structure in the access pattern and updates to the data. An "evolutionary" strategy enables this. As described, the first phase identifies new mutations in the reference trace. The second phase of training performs credit assignment to unit patterns involved in predicting the input sequence. A "prediction" run is simulated on the input sequence using the pattern memory just updated in phase 1. Each time the prefix of a unit pattern "fires" and contributes its suffix to a prediction, two things happen. First, the unit pattern's timestamp is reset. Second, an historical accuracy average is updated by counting the number of UIDs in the unit pattern suffix which appear within a fixed distance in the "future" of the reference trace. This distance is typically $p + r$, the size of the object cache. If the rating of an individual unit pattern falls below a threshold (see MinAccuracy parameter), it is deleted.

Old unit patterns which first functioned well as predictors might stay around unused, since their accuracy ratings would be set high. Timestamps prevent pattern memory from becoming full of unused unit patterns. Supplied coefficients (see Age, Accuracy parameters below) assign relative importance to the age of unit patterns versus accuracy ratings. A ranking function then orders the unit patterns according to age since last use and accuracy. If the number of unit patterns exceeds the specified maximum, n, the unit patterns with the lowest rankings are removed until the pattern memory fits within its designated space. The end effect of training is that unit patterns which function well as predictors are assigned higher ratings, and unit patterns which are not good predictors or which are obsolete are supressed by the ranking scheme and fade away. Over time, the pattern memory converges to a set of unit patterns which, when used for prediction, achieve a greater prediction accuracy than a specified threshold (see MinAccuracy parameter).

4.3.1 Training Cost:

It is likely that most applications will choose to perform training between sessions, providing ample time. Training itself is efficient enough that it could be conducted incrementally during the session, however. This would be useful if the time between references is typically long, as with Hypermedia. The mutation phase performs one NN call for each of l UIDs in the training sequence in $O(l(\log(n) + |\sigma|\ f))$ time. The selection phase is a matter of performing at most l predictions at cost $O(\ |\alpha|\ \log(n) + f)$ each. At each prediction a credit assignment calculation is done at cost $fp|\omega|$. Sorting the pattern memory according to ranking takes $O(n \log(n))$ time, and finally, reconstructing the indices for prediction takes $O(n \log(n))$ time.

4.4 Prediction

The training algorithm updates pattern memory for a context after each session, and stores the results, ready for the next user session which invokes that context. The pattern memory consists of unit patterns which appear regularly, with limited variation, from session to session. Thus the task of the prediction mode algorithm is to quickly recognize prefixes of these unit patterns, as similar sequences appear during a session, and produce the associated suffixes. As a user session initializes, it presents its CID to the Prophet, which loads the pattern memory for the particular context. Also allocated are one column to to tally matches, $|\alpha|$ indexes to allow matching over the prefix columns, and the column of historical accuracy ratings. The training algorithm prefaces each reference trace with its CID. Therefore, the client feeds the CID to the Prophet PREDICT routine to produce an initial prediction, corrsponding to those UIDs usually referenced first. During the session, the current access sequence is shifted through a sampling window of variable size. The sample is sent to the Prophet's prediction algorithm on each cache reference. If the Prophet is not operating as a separate process, this adds the cost of one prediction to each cache reference.

The PREDICT routine finds the nearest-neighbor unit patterns whose prefixes partially or totally match the input sample. The t best matches are tracked during matching. These are then ranked according to both the historical accuracy and degree of match with the input sample. As with training, supplied coefficients (see Similarity, Accuracy) determine the relative importance of these factors in composing the prediction. The suffix of each unit pattern is then copied to the prediction output buffer, with elements from the "best" unit pattern appearing in the "front" of the buffer. As elements are added to the prediction output buffer, duplicates are detected and avoided. Thus UIDs of multiple unit pattern suffixes are interleaved in the prediction output buffer with UIDs from the best matches and historically more effective unit patterns appearing first.

4.5 Prediction Costs

Prediction relies on the NN routine, the performance of which was described above. NN generates the nearest neighbor matches for a sample of size $|\alpha|$ in time $O(\ |\alpha|\ \log(n) + f)$, where f is the expected number of neighbors found. A constant number, t, of best match pointers is updated during NN. These t top choices are then interleaved into a prediction output buffer of size p. It requires at most $pt\ |\omega|$ comparisons to eliminate duplicates using linear search of the output buffer for each of the t predictions of size $|\omega|$. Thus the expected time for a prediction is $O(\ |\alpha|\ \log(n) + pt\ |\omega|\ + f)$, where f, p, t, and $|\omega|$ are small numbers. To ensure that predictions will complete in a given maximum time (e.g. for real-time appliations) it may be useful to calculate the expected or worst-case prediction time - where $f = n$, and then derive the memory size limit n for use during training. The amount of physical memory required by this (suboptimal) implementation for a pattern memory storing n subpatterns is $O(n(\ |\sigma|\ +\ |\alpha|\ +2))$

4.6 Parameters

The following parameters are used to balance learning rate against the amount of resources used by the Prophet, and to control various aspects of prediction. Most are specific to the nearest-neighbor implementation. In effect, these parameters are used to customize training and prediction behavior of the Prophet to individual applications. Some of the parameters can be made to self-adjust based on performance statistics. For example, if many nearest neighbors are found during the average prediction, the Resolution value can automatically be decreased in an attempt to increase the average orthogonality in the pattern memory.

MemMax: Controls the amount of memory allocated to store pattern memory, which in turn limits the worst-case and expected time needed to generate predictions. The pattern memory for a context should be locked in the process working set during predictions so as not to incur extra I/O from paging virtual memory during prediction. More physical memory is required to learn access patterns when Resolution is increased or when the access pattern is complicated.

MinAccuracy: Specifies the minimum Accuracy needed for unit patterns to survive Training. Unit patterns whose average Accuracy rating falls below this threshold are deleted.

Resolution: Specifies the maximum distance, permitted between unit patterns before they are considered distinct. High Resolution values cause the Prophet to notice small changes in patterns and require the Prophet to take more time and space when predicting.

Overlap: This parameter governs how the reference trace is sampled during training. High values cause unit patterns to overlap greatly and require more physical memory for pattern memory, however allow the Prophet to recognize the pattern starting at more points. Typically, overlap between unit patterns should be greater than the size of the prefix sample used, so that a new unit pattern is matched as the application accesses UIDs in the tail end of a unit pattern whose UIDs were just prefetched into cache.

Age, Accuracy, Similarity: The first two coefficients govern how unit patterns are rated for survival during training. They specify the relative importance of two factors respectively - age and historical accuracy, in rating unit patterns. The Accuracy coefficient is used during both training and prediction. The Similarity coefficient is used during prediction to assign a relative importance to the degree of match as opposed to the historical accuracy.

5 Assessing Prediction Accuracy

The previous parts of this paper have illustrated how a prototype of the Prophet works and fits into an OODB system, discussed the system resources required by the prototype, and reasoned that the worst case costs of using predictive prefetching instead of demand paging are small. The remaining questions concern the best case - how accurately can a Prophet predict? This section outlines approaches to studying a Prophet's prediction accuracy under various conditions.

Methods for studying the learning and prediction performance of different Prophet designs originate from both theoretical and empirical points of view. Formal methods exist to express learning performance as a function of how complicated the pattern is to learn [V84], as well as for assessing paging performance [ST85]. These theoretical results are paired with experimental results obtained by building Prophet prototypes and testing them using simulated access traces as well as real traces obtained from applications. The final phase of research is of course to add predictive prefetching to a system and measure performance.

Learning and prediction performance vary according to how "predictable" an access pattern is. Access pattern predictability is a function of both indeterminacy in the application which generates the pattern, and of the noise present during training and prediction. The following sections outline means of characterizing the indeterminacy inherent in an application and discuss some observed effects of noise on the performance of the prototype.

5.1 Access Patterns

In order to function efficiently, the Prophet must abstract its observations of specific access traces into generalizations about the access pattern. Some access patterns are less predictible than others. Access sequences which change very little from one session to the next are easy to learn and predict. If few similarities exist between sessions, the pattern may be more unpredictable and will require more resources to learn. Analyzing the predictive performance of the Prophet on access patterns which vary in degree of predictability requires some means of characterizing access patterns both qualitatively and quantitatively. One such approach is to imagine that every program traverses a directed *generative graph* which determines possible and probable access sequences. Each vertex of the generative graph is labeled with a UID and each edge has an associated traversal probability. Each application session produces an access sequence by traversing the generative graph, using the out-edges of a vertex, according to their assigned probabilities, to choose the next vertex. This produces an access sequence consisting of the UIDs of vertices visited.

Applications which traversing small, simple generative graphs produce few possible access sequences, which makes learning and prediction easy. Conversely, a large graph with high branching factor allows many possible traversals. A generative graph with branching factor b has maximum indeterminacy, or entropy, if the probability of traversal at each node is independently $1/b$. For example, a node with four outgoing edges, each with independent traversal probabilty of 0.25, would have maximum entropy - it is equally likely that any of the available paths through the graph would be taken each time. Intuitively, an access pattern has zero indeterminacy if it is generated from an (acyclic) graph with branching factor of one, i.e. - a list. A pattern generated from such a graph which has not been subject

to noise can be perfectly predicted (trivially) by a Prophet which stores the sequence, and when given any element present in the sequence, simply returns the rest of the sequence.

Real applications tend not to make such random decisions about access sequences. Rather, they generate short "dependent" sequences which result from bursts of deterministic action. Factors besides the data structure alone operate in applications to constrain access sequences. For example, an OODB may expand composite objects using breadth-first search, starting with a top level UID, and following pointers in a fixed order at each level of the hierarchy to retrieve an entire composite object. Thus the same short sequence is generated to expand the composite object each time, even though the composite object has a tree structure which admits many different traversals. Another example is where an application initializes by reading a set of objects in the same order each time, and maintains a "display list" for graphics operations. The display list may determine most of the access sequences for an interactive user who causes a CAD design to be redrawn with each operation. As the number and length of these dependent sequences in the access pattern increases, the recent access history becomes a more reliable means of generating predictions.

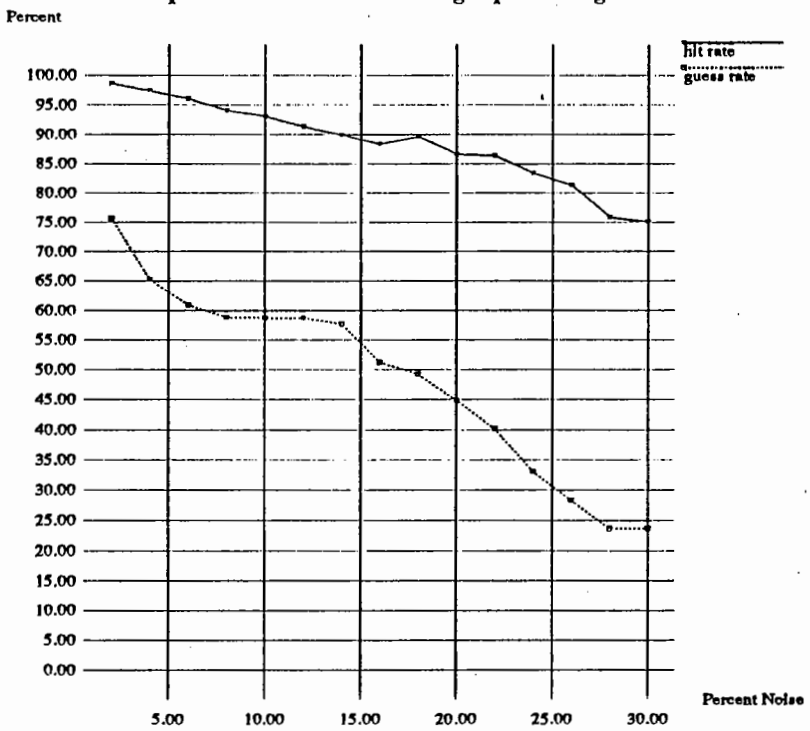
Current experiments (results forthcoming), construct generative graphs with specific amounts of indeterminacy and known probable paths, and produce simulated access sequences by traversing them. These are then fed to the Prophet for training and prediction, and prediction accuracy and resource usage are measured.

5.2 Response to Noise

A second critical aspect of prediction performance is how prediction accuracy degrades as increasing amounts of noise are present in the access pattern. There are two data operations affecting learning and recognition of access patterns - object creation and deletion. These can affect even a read-only application, since other users update the database in an unpredictable way. The update rate typical of OODB applications (e.g. CAD tools) is slower than that of transaction processing applications. However, it is reasonable to expect that 10 or 20 percent of the UID symbols may change or disappear between sessions - after training and before prediction. New UIDs appearing in the resulting access sequences are drawn arbitrarily from a large alphabet of possible UID symbols.

Experiments with an initial neural net implementation of the Prophet [MP90] revealed an interesting property of the prototype's handling of create/delete noise. In one experiment, an access sequence simulator produced a single sequence of UIDs, which was then used to train the Prophet once. UID inserts and deletes were made to the original sequence at random points in equal numbers, maintaining its length. Then prediction was run on the mutilated sequence, and prediction accuracy and volume (i.e. number of predictions) were measured. This process (adding additional insert/delete noise to the sequence, then predicting without re-training) was repeated until the original sequence contained thirty percent noise. We observed that both the prediction accuracy and prediction volume degraded linearly and gracefully as increasing amounts of noise were added to the learned sequence before prediction. Also, the prediction volume degraded more quickly than prediction accuracy.

Graph 1b: Effect of Inverting alpha/omega Ratio



This encouraging result was easy to explain. If create and delete operations occur in equal numbers and over a uniform random distribution within an access sequence, the sequence length stays the same. Then the mean distance between "breaks" (i.e. inserted or deleted UIDs) decreases linearly with increasing noise. This has two effects: a) correspondingly fewer predictions are made as the average size of subsequence declines to be less than the size of the unit pattern prefix, and b) fewer of the UIDs in a prediction are correct. Also, if the prediction sample size ($|\alpha|$) is increased, fewer predictions will be produced, since fewer subsequences of larger size will be recognized. However, the predictions which are produced will be more accurate. This effect was confirmed in a second experment which repeated the previous one, except that the ratio of prefix size to suffix size was decreased, producing a larger number of predictions but lower accuracy at each noise level. This capability of dynamically varying the sample size to can be used to combat the effects of noise.

6 Conclusions

We presented a cache management model for prefetching which limits the performance degradation that can be introduced by prefetching. It requires extra cache space to store prefetch errors, but less space for demanded objects. If prediction accuracy is better than fifty percent, no extra cache space is required to ensure emulation of LRU on the same access sequence. The implication is that with an ability to make *some* correct guesses about the immediate future, MLP prefetching will produce a better hit rate than LRU, provided there is enough time in between (or during) I/O operations to generate predictions. We then described the design of a simple pattern memory for generating such predictions which had well-defined resource costs and scaling properties, and argued that the pattern memory would adapt to changes in access pattern and data over time, and improve in prediction accuracy. We expect that the performance of this design could be improved by employing more sophisticated techniques e.g. [ZL77], [MJ86], [RM86]. The importance of the current design is that it suggests that prediction can be performed quickly enough for use in real systems, and used for virtual memory paging.

6.1 Applications

Predictive caching seems to be a useful technique. A picture of the kinds of applications which will find it useful is emerging. These applications tend to have some or all of the following characteristics:

1) It is not convenient to hand-develop access structures because data or access patterns change too quickly, for lack of manpower, or because of encapsulation.

2) The rate of change to data is slow enough that it can be tracked by a Prophet via training.

3) The environment permits computation during I/O and use of a cache.

4) There is relatively low contention for object locks.

5) The application is data intensive, and faces I/O bottlenecks or poor memory utilization.

6) The applications's access to data is structurally constrained or otherwise has regularity.

7) The data objects accessed are of a medium granularity. At too fine a granularity, prediction begins to demand resources. If too coarse, mistakes are too costly.

6.2 Related Work in Adaptive Databases and Neural Nets

Previous work in adaptive databases ([BN78], [C76], [HC76]) in the mid 1970s explored means of automatically configuring the physical and secondary access structures of databases according to usage. These solutions involved running statistical analysis procedures periodically and generating schema reconfigurations or indices to improve performance. The database did not adapt to changes in usage patterns between reconfigurations, and was unavailable during reconfiguration. Such statistical procedures required customization to access each database schema, resulting in brittleness and lack of generality. The adaptive mechanisms were not transparent to the database administrator and attempted to optimize access over all users rather than for each user.

Several advances in technology have renewed possibilities for adaptive database research. First, OODBs have provided explicit maintenance of immutable object identity. This makes learning usage patterns for a particular set of data objects easier than constructing indirect means to identify data over time. Also, recent work in neural networks and cognitive science has produced models for machine learning and self-organizing systems which are less ad hoc, more robust, and better understood than previously. These models of human cognition (e.g. [KA84], [PK88]) have provided a rich set of tools which can "make sense" out of patterns, sometimes by generalizing and forming internal representations [RM86] to achieve learning. In addition, neural models are generally quite robust, adapting and functioning well in the presence of noise. In recent years the technology has matured to a point where special hardware implementations of the models enable use of neural net techniques for practical applications. Also, much more compute power is now available to perform the sorts of learning and statistical computation often required by adaptive systems. More elegantly, such models operate as black boxes, and so are inherently non-invasive in observing system interactions.

A unifying goal of our work is to find ways to mate appropriate cognitive and machine learning models with various system components to exploit regularities in system operations. In this endeavor, approximate solutions such as those developed by neural models can have high payback, especially where I/O bottlenecks and memory utilization are problems. The function of neural and cognitive models in the early cycles of this research is as tools to sharpen statistical intuition and allow study characteristics of various patterns. The inspiration for the NN design presented in this paper came from an experimental attempt to perform pattern learning and prediction using the "Brain State in a Box" model developed by Jim Anderson at Brown [KA84]. Modelling problems initially at a cognitive level allows finding an appropriate representation for inputs and outputs of the black box. Once an appropriate cognitive/neural model and problem representation are identified, the resulting function can be "imitated" by a design that uses faster but less flexible technology.

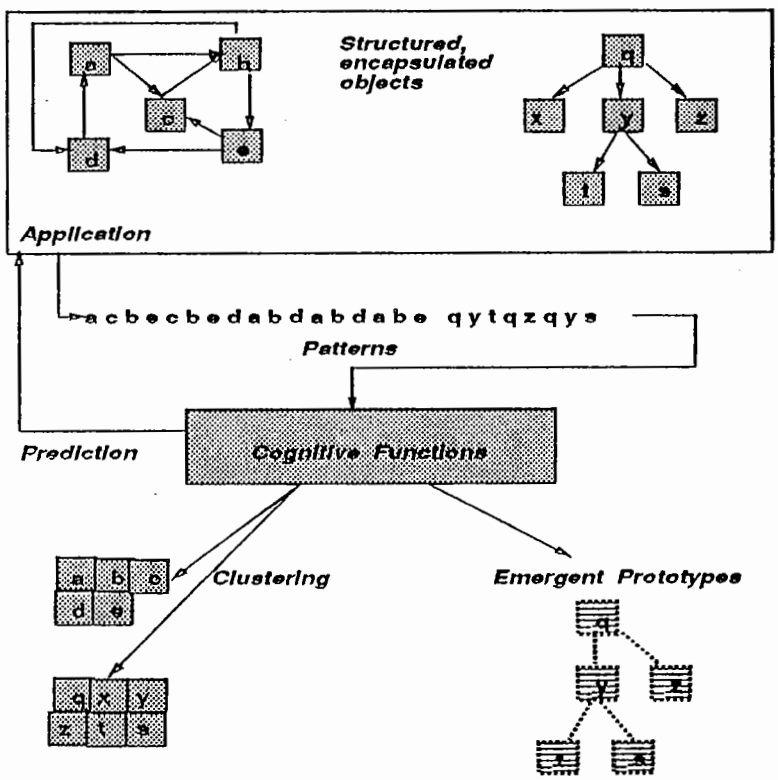
6.3 Ongoing Work

A small prototype along the lines described herein has been constructed and is being tested on simulated access sequences. We are beginning to test it on "real" access sequences and incorporating it into ENCORE. We would like to pursue more ways of using adaptive technology in database and operating systems, as well as to further experiment with using predictive caching in real systems. Prophet implementations using true distributed/neural

representations such as backpropagation [RM86] and others which represent probability explicitly are being considered for the prediction function. Special-purpose parallel associative memory hardware presents the possibility of vastly increasing the size of a pattern memory which can be processed on-line. Pattern memory requirements may be further decreased by sampling only the access sequences before actual faults. We would like to augment the training algorithm to make use of available database schem ainformation. This would probably involve using the schema to intialize pattern memory, and as a "sanity check" when training.

Another area we are beginning to explore is clustering. A Prophet which has learned an access pattern can then generate a clustering for the data a clustering ability is implicit in the ability to predict. Actually, each "unit pattern" in the current prototype represents a kind of small ordered cluster, except the clustering is specific to one usage context. The way to use Prophet as a heuristic for clustering is to train one pattern memory over all contexts.

The backdrop against which we have discussed the Prophet is that of OODBs. However, the concept is quite general-purpose and could be used in other databases as well. Within a database, prediction could be used in more components than just the client - perhaps at each I/O boundary, prefetching pages of disk storage into the server buffer as well as from server to client. Obviously, an actual server and client model is not needed to make predictive prefetching useful - it may be useful anywhere there is an I/O bottleneck. The approach may be adapted to other kinds of systems and to other dimensions of system operation. In particular, the use of predictive prefetching for virtual memory paging is an attractive possibility. Paging may actually be a simpler problem, since variable object sizes and cache validation aren't issues. In summary, we feel that there are several interesting avenues for continuing this research and hope to report further results soon.



7 References

The following list of references is divided into two parts. The first part lists papers on various OODB systems and issues related to storage management, while the second provides sources for various neural net and learning paradigms which are relevant to learning access patterns.

7.1 Database and Systems References

A good discussion of storage management tradeoffs can be found in [ZM90], in the introduction to the chapter on Implementation. Papers on the Observer storage manager [HZ87], object identity [KC86], and physical storage management in Adaplex [C+82] are also found in the [ZM90] collection. The ORION paper [G+88] provides a description of client cache management. The Niamir [BN78], Chan [C76], Hammer [HC76], and Schkolnick [MS77] papers are included as examples of adaptive database research of the mid 1970s.

[BN78] B. Niamir, "Attribute Partitioning in a Self-Adaptive Relational Database System". MS thesis, MIT/LCS/TR-192, January 1978.

[B+86] F. Bancilhon, et al. "The Design and Implementation of O2, an Object-Oriented Database System," Proceedings of the 2nd International Workshop on Object-Oriented Databases, Springer-Verlag Lecture Notes in Computer Science 334, September 1988.

[C76] A. Y. Chan, "Index Selection in a Self-Adaptive Relational Data Base Management System," SM Dissertation, MIT, September 1976.

[CK89] E. E. Chang, R. H. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS," Proceedings ACM-SIGMOD International Conference on Management of Data, Portland, OR, June 1989.

[C+82] A. Chan, A. Danberg, S. Fox, W.T. Lin, A. Nori, and D. Ries, "Storage and Access Structures to Support a Semantic Data Model," Proceedings of the VIII International Conference on VLDB, Morgan Kaufmann Publishers, September 1982.

[FEZ89] M. F. Fernandez, A.N. Ewald, S. B. Zdonik, "ObServer: A Storage System for Object-Oriented Applications," Tech Report CS-90-27, Brown University, November 1990.

[F+88] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, N. Young, "Competitive Paging Algorithms," CMU-CS-88-196, Carnegie-Mellon University, November 1988.

[G87] J. Gray, "The 5 minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time," Proceedings ACM-SIGMOD International Conference on Management of Data, San Francisco CA, May 1987.

[G+88] J.F. Garza, H. Chou, W. Kim, D. Woelk, "ORION Object Server - Architecture and Experiences," MCC TR ACA-ST-423-88, 1988.

[HC76] M.M. Hammer, A.Y. Chan, "Acquisition and Utilization of Access Patterns in Relational Data Base Implementation," Pattern Recognition and Artificial Intelligence, Academic Press, 1976.

[HK90] S. Hudson, R. King, "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System," ACM Transactions on Database Systems, 1990.

[HS90] H. S. Stone, High-Performance Computer Architecture, Addison-Wesley Publishing Company 1990.

[HZ87] M.F. Hornick, S.B. Zdonik, "A Shared, Segmented, Memory System for an Object-Oriented Database". ACM Transactions on Office Information Systems, 5:1, 1987.

[JS84] J.W. Stamos, "Grouping Objects to Enhance Performance of a Paged Virtual Memory". ACM Transactions on Computer Systems, 1984.

[KC86] S. N. Khoshafian, G.P. Copeland, "Object Identity". ACM Proceedings on the Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, OR, 1986.

[KK83] T. Kaehler, G. Krasner, "LOOM: Large Object-Oriented Memory for Smalltalk-80 Systems," in Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, 1983.

[MP90] M. L. Palmer, "(the) Estimating Prophet." term paper, Brown University, May 1990.

[MP89] M. L. Palmer, Thesis Proposal. (memo) Brown University, December 1989.

[MS77] M. Schkolnick, "A Clustering Algorithm for Hierarchical Structures". ACM Transactions on Database Systems, Vol. 2 No. 1, March 1977.

[M89] J.E.B. Moss, "Addressing Large Distributed Collections of Persistent Objects: The Mneme project's approach", In Second International Workshop on Database Programming Languages, Gleneden Beach, OR, June 1989.

[RKC87] W.B. Rubenstien, M.S. Kubicar, R.G.G. Cattell, "Benchmarking Simple Database Operations", Proceedings ACM-SIGMOD International Conference on Management of Data, San Francisco CA, May 1987.

[ST85] D. Sleator, R. Tarjan, "Amortized Efficiency of List Update and Paging Rules", Communications of the ACM 28, February 1985.

[ZM90] S.B. Zdonik, D. Maier (eds), Readings in Object-Oriented Database Systems. Morgan Kaufmann Publishers, 1990.

7.2 Neural Net and Learning References

A collection of papers which provides a good overview of the neural network field is found in Neurocomputing [AR90]. Papers on associative memory, probability learning, and the Brain-State-in-a-Box model [ASRJ77], [KA84] as well as backpropagation learning [RM86] may also be found in the Neurocomputing collection. The Kanerva book [PK88] provides interesting ideas regarding nearest-neighbor memories used for predicting sequences (in chapter 8) and implementation.

[AHS85] D. H. Ackley, G. E. Hinton, T. J. Sejinowski, "A Learning Algorithm for Boltzmann Machines," Cognitive Science 9, 1985.

[ASRJ77] J.A. Anderson, J. W. Silverstein, S.A. Ritz, R.S. Jones, Distinctive Features, Categorical Perception, and Probability Learning: Some Applications of a Neural Model. Psychological Review 84:413-451

[AR90] J.A. Anderson, E. Rosenfeld, Neurocomputing. MIT Press, 1988.

[KA84] A.G. Knapp, J.A. Anderson, "Theory of Categorization Based on Distributed Memory Storage," Journal of Experimental Psychology: Learning, Memory, and Cognition 10:616-637.

[LF87] A. Lapedes, R. Farber, "Nonlinear Signal Processing Using Neural Networks: Prediction and System Modelling," Tech Report, Los Alamos National Lab Theoretical Division, July 1987.

[MJ86] M. Jordan, "Serial Order: A Parallel Distributed Processing Approach," UCSD Institute for Cognitive Science, Report 8604, May 1986.

[PK88] P. Kanerva, Sparse Distributed Memory. MIT Press, 1988

[RM86] D.E. Rumelhart, G.E. Hinton, R.J. Williams, "Learning Internal Representations by Error Propagation," Parallel Distributed Processing Volume 1, MIT Press 1986

[V84] L.G. Valiant, "A Theory of the Learnable," Artificial Intelligence and Language Processing 27 N0. 11, November 1984.