Correlating Database I/O Accesses at the Storage Server

by

Madalin Mihailescu

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Your file Votre référence
ISBN: 978-0-494-27293-0
Our file Notre référence
ISBN: 978-0-494-27293-0

Abstract

Correlating Database I/O Accesses at the Storage Server

Madalin Mihailescu

Master of Science

Graduate Department of Computer Science

University of Toronto

2007

In this dissertation we investigate the usage of a known data mining technique, frequent sequence mining, for correlating database accesses in storage systems. We look at I/O prefetching as an application of disk block correlations. We design, implement and compare dynamic and static block correlation mining techniques. For the static techniques, we study schemes with and without correlation rule retraining.

In our experimental evaluation, we use the MySQL database engine and two applications: DBT-2, a TPC-C-like benchmark and the RUBiS auctions benchmark. We perform the mining at the storage level and measure the respective application hit rates in the storage cache. Our results show that, by using block correlations, we can improve the storage cache hit rate by 3-21% for DBT-2 and 5-47% for RUBiS, compared to the baseline. Furthermore, we show that dynamic mining outperforms static mining in terms of higher hit rates and more accurate block correlation rules.

Acknowledgements

First and foremost, I would like to thank my supervisor Cristiana Amza. Thank you for bearing with my unsteadiness and for your constant support and guidance during the past two years.

Second, I would like to thank Angela Demke Brown for taking the time to read my thesis and providing useful comments and suggestions.

I also want to thank my labmates, especially Gokul, for his ideas and graphics skills, and Livio, for always sharing his great knowledge in systems.

Last but not least there is Iubi, for her courage and strength and for always believing in me.

Contents

1 Introduction **1**

2 Background **6**

2.1 Block correlations 6

2.2 Frequent sequence mining 7

2.2.1 Static mining 8

2.2.2 Dynamic mining 9

3 Design **11**

3.1 Context Definition 11

3.2 Overview 12

3.3 DBMS Instrumentation 12

3.4 Rule Generation 13

3.5 Block Prefetching 14

3.6 Advantages of Contexts 14

3.7 Tradeoffs 16

3.8 Data Evolution 17

3.8.1 Static mining 17

3.8.2 Dynamic mining 18

4 Evaluation **20**

4.1 Benchmarks 20
4.1.1 DBT-2 20
4.1.2 RUBiS Auction Benchmark 21
4.2 Evaluation Methodology 21
4.2.1 Trace Collection 22
4.2.2 CacheSim 22
4.2.3 Performance Metrics 23

5 Results **25**
5.1 Overview 25
5.2 Static Mining 26
5.2.1 Hit Rate 27
5.2.2 Rule Parameters 30
5.3 Dynamic Mining 31
5.3.1 Dynamic vs. Static 31
5.3.2 Varying the Window Size 34

6 Related Work **39**
6.1 Inferring Block Correlations without Breaking the Interfaces 39
6.2 Changing the Interfaces to Bridge the Semantic Gap between Storage Clients and Servers 40
6.3 High Level Prefetching for Databases 41
6.4 Storage Cache Optimization Techniques 42
6.5 Adaptive Cache Management 43

7 Conclusions **44**

Bibliography **46**

List of Tables

5.1 Static Mining Rule Generation 29

5.2 Static Mining Rule Accuracy 30

5.3 Dynamic vs. Static Mining Rule Accuracy 36

5.4 Rule Generation when Varying the Window Size 37

5.5 Rule Accuracy when Varying the Window Size 38

List of Figures

1.1 Dynamic Content Web Servers using NAS 3

2.1 Sliding Window Example 10

3.1 I/O Access Pattern 15

3.2 Data Evolution Over Time. 17

5.1 RUBiS Static Mining Hit Rate 27

5.2 DBT-2 Static Mining Hit Rate 28

5.3 RUBiS Dynamic vs. Static Mining Hit Rate 32

5.4 DBT-2 Dynamic vs. Static Mining Hit Rate 33

5.5 RUBiS Dynamic vs. Static Mining Rule Evolution 34

5.6 DBT-2 Dynamic vs. Static Mining Rule Evolution 35

5.7 Hit Rate when Varying the Window Size 37

5.8 Rule Evolution when Varying the Window Size 38

Chapter 1

Introduction

In this dissertation, we investigate techniques for correlating disk block accesses at the storage server [18, 19, 31, 30, 29]. Disk block correlations are sequences of I/O accesses determined to occur with high probability. Disk block correlations can be very useful for a number of storage optimization policies, such as storage cache replacement, disk layout, block prediction or disk scheduling. Furthermore, predictable block correlations are inherent to access patterns of many storage clients, such as database systems or file systems. In this work we apply a known data mining technique, frequent sequence mining [32], for generating block correlations from streams of disk accesses. As an application of block correlations, we investigate block prefetching for database systems. The increasing lag between the memory access time and the disk access time motivates aggressive block prefetching, now more than ever, as a means of hiding the significant on-disk data access latency. Furthermore, our work is motivated by the recent trends towards increasingly intelligent storage systems, with large storage cache sizes and high processing capabilities. For example, current IBM Network Attached Storage (NAS) Systems can have up to 8 2.6 GHz processors and up to 64GB of memory as storage cache [1]. The processing power of the storage controller and large storage caches in such systems can be leveraged for various optimizations, including block prefetching.

For block prefetching to be effective, the storage server needs to predict future block accesses and issue prefetch requests for those blocks in advance. This is challenging because the storage server sees interleavings of page misses from application threads of several concurrent applications. For example, Network Attached Servers are nowadays commonly used as one of the many components of the complex software stack supporting several Internet services, from the traditional e-commerce such as Amazon.com, to online publishing such as Drupal content management systems [16, 15]. Figure 1.1 shows the architecture of Internet servers. Apart from the Network Attached Storage (NAS) server, which stores and provides access to data, Internet server architectures commonly consist of several other tiers : (1) the web server tier which serves the web pages to the clients, (2) the application server tier which implements the application logic, and (3) the database system tier which queries the dynamic content stored in the NAS. Such configurations are typical in large data centers where operators reduce costs by multiplexing several concurrent applications on the same server farm or even on the same physical server. Previous techniques for determining block correlations at the storage server are not applicable in these complex scenarios. "Black-box" approaches such as [19] are ineffective because of the concurrency degree inherent in data centers that causes lots of false correlations (see Section 5.2). "Gray-box" techniques require either thorough knowledge about each database server running on top of it, such as the structure of data and metadata on disk [30], or heavy changes to the storage interface [29].

The contribution of our work is an investigation of generic, DBMS-independent techniques for determining block access correlations and exploiting them for block prefetching in these environments. Our contributions are two-fold:

1. We implement and evaluate a static frequent sequence mining algorithm for block prefetching at the storage server. We augment the basic scheme to perform efficient block correlation retraining.

2. We implement and evaluate a dynamic frequent sequence mining algorithm to gen-

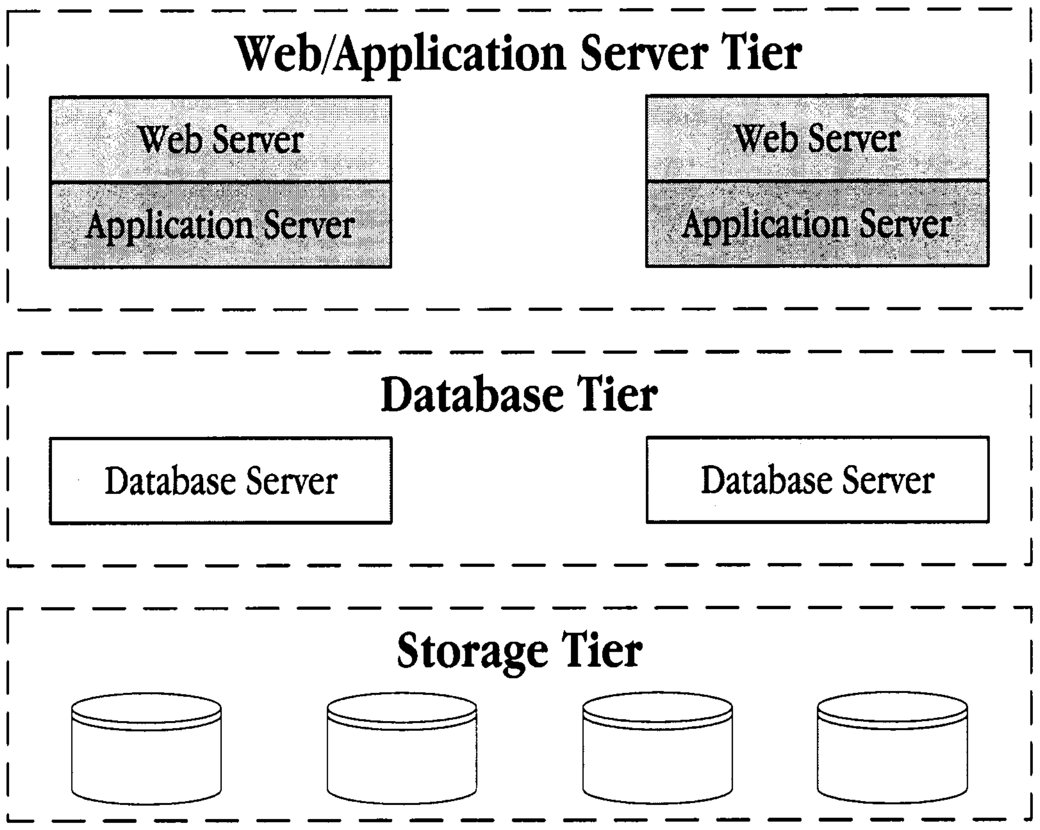

Figure 1.1: Dynamic Content Web Servers using NAS

erate block correlations on the fly.

For the static block correlation algorithm, we use as a baseline a recently proposed algorithm [19], called *C-Miner**, for frequent sequence mining on streams of disk block accesses. The authors correlate blocks fully transparently inside the storage server without any changes to the storage clients running on top. This is a "black-box" approach, making the technique very appealing since it is general and targets any storage client. In our scheme, we automatically capture and leverage application contexts for I/O block correlation at the storage server. We simply tag each database I/O block request with a context identifier corresponding to the higher level application web interaction, database transaction or database query where the I/O request to the storage manager occurred. This allows the storage server to correlate the block accesses that it sees according to their higher level context. These contexts are exposed to the storage layer by grouping based on context identifiers. The storage manager then performs data mining and on-line block predictions per-context, rather than globally. More importantly, in the *C-Miner** algorithm with or without contexts, block correlations may become inaccurate due to inserts in the database from write-intensive applications. We extend the context-guided *C-Miner** algorithm with efficient re-training decisions. In our approach, we monitor rule efficiency to recognize applications where the prediction rules have become ineffective due to an access pattern change. This helps us to perform selective rule retraining adaptively for the applications affected by the change.

Furthermore, we investigate dynamic incremental mining by using a frequent sequence mining over a sliding window technique to generate block correlations on the fly. The algorithm outputs correlations that are frequent over the most recent N context instances, where N is the window size. The benefit of this approach is that no re-training decisions have to be made since the algorithm automatically re-trains by discarding correlations that become obsolete due to data evolution and by generating new ones. Finally, we show that the dynamic approach can be effective even in the presence of memory constraints, by varying the window size and keeping a lower number of block correlations in memory.

We perform experiments with the MySQL database engine and two applications: DBT-2 (a TPC-C-like benchmark) and RUBiS, an on-line bidding benchmark modeled after eBay.com. The applications have different I/O needs. DBT-2 is I/O intensive, and hence has a very large block footprint in the storage cache, while RUBiS needs an order of magnitude less storage cache. We use a storage cache simulator to measure the block hit rate and prefetch rule accuracy. In our experiments, we vary the storage cache size.

For the static mining approach, our results show increased overall hit rates in the storage cache, up to 45% for RUBiS and 17% for DBT-2 compared to the baseline cache implementing the LRU replacement policy with no prefetching scheme. Furthermore, by using the contexts, we generate fewer and more accurate block correlation rules compared to the context-oblivious approach. Rule accuracy i.e., the ratio of useful versus useless block prefetches, is important in multi-application scenarios since useless block prefetches may interfere with the cache effectiveness of concurrent applications in addition to consuming the disk bandwidth.

When using dynamic mining we obtain up to 2% for RUBiS and 4% for DBT-2 increase in hit rate relative to the static mining scheme with re-training. Compared to the baseline the hit rate increase is up to 47% in the case of RUBiS and 21% in the case of DBT-2. Moreover, dynamic mining also improves the rule accuracy for both workloads.

The outline of the rest of this thesis document is organized as follows. Chapter 2 provides the necessary background on block correlations and the frequent sequence mining algorithms on which we build. Chapter 3 introduces the design of our prefetching solution. Chapters 4 and 5 describe our experimental platform, storage cache simulation methodology and results. Chapter 6 discusses related work and Chapter 7 concludes the thesis.

Chapter 2

Background

In this chapter we define what block correlations are, briefly introduce the frequent sequence mining algorithms that we use and give some examples.

2.1 Block correlations

Two blocks {`a`} and {`b`} are correlated if, whenever {`a`} is accessed, there is a high probability that {`b`} will be accessed as well. Correlations are useful for prefetching, disk layout or cache replacement policies. Block correlations in storage servers are a logical consequence of the structures existing in the storage clients. Database systems, for instance, access data through indexes usually represented as B^+-Trees. These structures result in a set of known access patterns. For example, point queries traverse down the B-Tree to find the data page. This access pattern typically shows a dual block correlation between a parent and child node. Similarly, scanning the leaves in a clustered index results in sequential block correlations. Three blocks a, b, c are correlated if whenever a and b are accessed close to each other, c will be accessed as well. We use the following notations to describe correlations: {`a->b`} for dual-block correlations and {`a&b->c`} for three-block correlations.

Correlations among multiple e.g., three, blocks are typically more precise than dual block correlations. We illustrate this with an example. Let us consider a merge join between two relations, R1 and R2, where the outer relation has a clustered index and the inner one has a non-clustered index on the join field. The access pattern to the blocks belonging to the two relations might have the following order: {R1:1, R2:5, R2:7, R2:9, R2:4, R1:2, R2:11, R2:8, R2:3, R2:6}, where we denote by {R1:1}, an access to block 1 belonging to relation R1. A dual block correlation in this case is {2->11} and a multiblock one is {4&2->11}. Now, let us suppose that we also have a merge join on R1 and R3 on the same join field as before. In this case the block access pattern might look like this: {R1:1, R3:12, R3:17, R3:13, R3:21, R1:2, R3:19, R3:23, R3:18, R3:14}. We obtain {2->19} and {21&2->19} as correlations. If we based our knowledge on dual block correlations only, on a subsequent encounter of block 2 in the access stream we would not know with 100% accuracy what the next block would be, since we have two conflicting dual block correlations {2->11} and {2->19}, derived for the two joins, respectively. However, if the block before 2 in the access stream is 21, we would determine that the next block is 19, according to the three block correlation {21&2->19}.

Generating block correlations from streams of I/O accesses can be translated into a frequent sequence mining problem. Frequent sequence mining algorithms aim at efficiently discovering frequent subsequences in a database of sequences [32].

2.2 Frequent sequence mining

We start this section with some preliminary concepts and then clarify the concepts through an example. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of all items. An item in this context is a general concept and in our case is translated into a block number. A sequence of items $S = \{j_1, j_2, \ldots, j_l\}$ where $j_k \in I$ for $k = 1 \ldots l$ is an ordered list of items, where the order is given by a specific criterion. For streams of block accesses the order is given by the timestamps of the disk requests. The first n-1 items in a sequence represent the *prefix* of that sequence. Similar to the item - block equivalence, a sequence is represented by a context (e.g., database transaction or database query) in our system. Let $A = \{a_1, a_2, \ldots, a_p\}$ and $B = \{b_1, b_2, \ldots, b_r\}$ be two sequences of items. We say that A is a *subsequence* of B if and only if $(\exists)$ $o_1, o_2, \ldots, o_p$, such that $1 \leq o_1 < o_2 < \ldots < o_p \leq r$ and $a_1 = b_{o_1}, a_2 = b_{o_2}, \ldots, a_p = b_{o_p}$.

A sequence database $D = \{S_1, S_2, \ldots, S_n\}$ is a set of sequences of items. The *support* of a sequence R in the database D is the number of sequences for which R is a *subsequence*. We say that a subsequence is *frequent* if it has a support greater than a predefined *min-support* threshold.

As an example, let us consider the following database of sequences: `D={ S1, S2, S3}` where `S1 = (apples, pears, cheese)`, `S2 = (milk, apples, bread, pears, cheese)`, and `S3 = (cheese, milk, butter, bread)`.

The subsequences `(apples, pears, cheese)` and `(milk, bread)` are the most frequent, each of them occurring two times. Following the above definitions, this number of occurrences is the *support* of the subsequence. Notice that the items do not have to be consecutive. If they occur within a small distance, we consider them to be part of a subsequence. Items in frequent subsequences are correlated. This small distance is called a *gap* or *lookahead distance*. The bigger the *lookahead* distance, the more aggressive the algorithm is in determining correlations.

2.2.1 Static mining

*C-Miner** [18, 19] is a recently proposed algorithm for discovering frequent subsequences from a sequence of accesses at the storage server level. Items in the previous example are now represented by disk blocks and instead of a database of sequences, mining is done on a long sequence of I/O block accesses. The basic idea behind *C-Miner** is to scan the *lookahead* distance of each prefix in the sequence. Let us assume that `{q,w,r,p,t,q,w,p,s,q,r,t,p}` is a sequence of block accesses observed at the storage server. In this example, block `{q}` appears 3 times. If the lookahead distance is 3 then we generate the following correlations for `{q}`: `{q->w}` (support 2), `{q->r}` (support 2), `{q->p}` (support 3), `{q&r->p}` (support 2) and so on. *C-Miner** falls in the category of frequent sequence mining on a static database. These algorithms work in a one-time fashion by mining the entire sequence database and generating the results. However, many applications are write intensive, thus leading to data evolution. Let us consider for example an on-line bookstore, where new books are inserted daily. As a consequence, customer preferences might get updated as well based on the content of the new books. Thus, previous correlations may now become worthless. Static algorithms are inefficient in this scenario, since they would have to mine the updated sequence database from scratch.

2.2.2 Dynamic mining

Incremental frequent sequence mining over a sliding window is a popular technique for dynamic frequent sequence mining [11, 20]. The basic idea behind this technique is to efficiently mine frequent subsequences in the most recent n sequences of items in the data stream.

The initial part of the algorithm consists of generating the frequent subsequences from the first n sequences, and putting them in a data structure, e.g a tree. Each new sequence in the data stream will replace the oldest sequence in the window. More specifically, the support of the frequent subsequences of the oldest sequence is decreased by one, while the support of the subsequences from the new sequence will be incremented by one.

Let us consider the example in Figure 2.1. For a window size of four, there are three windows during the time line of this run. Let us suppose that the *min-support* is two, meaning that we are interested only in subsequences with support higher than or equal to two. The first step of the sliding window algorithm generates the following

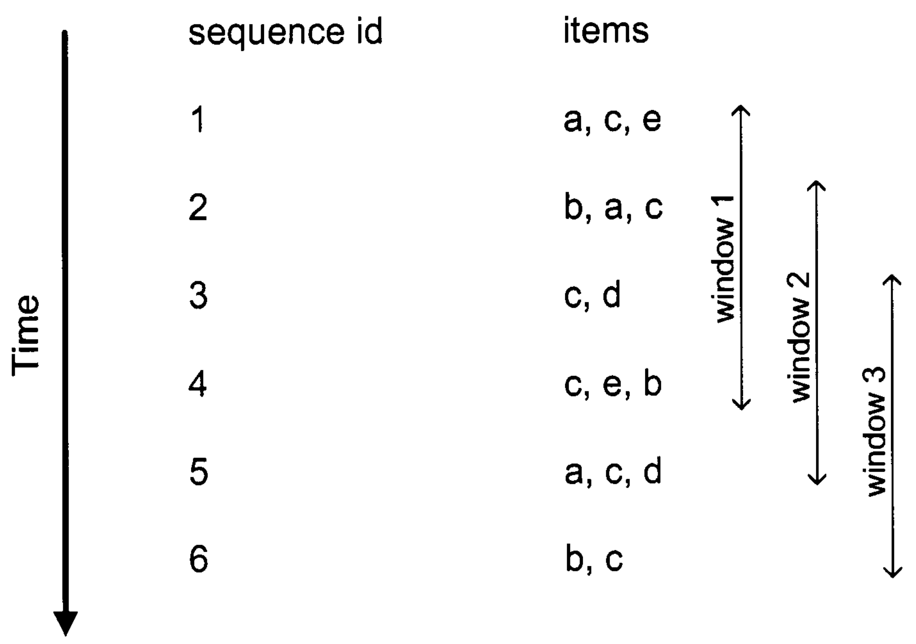

Figure 2.1: Sliding Window Example

frequent subsequences: {a->c} (support 2) and {c->e} (support 2). These correlations correspond to the first window in the data stream composed of sequences 1, 2, 3, 4. As a new sequence arrives in the data stream (sequence 5), the oldest sequence (sequence 1) is removed. Thus, the new frequent subsequences for window number two are: {a->c} (support 2) and {c->d} (support 2). The interesting observation here is that {c->e} has become infrequent (support 1), while {c->d} is now frequent. Similar for window 5, the only subsequence that is frequent is {c->d} (support 2), because, by removing sequence 2, {a->c} is not frequent in the current window.

Chapter 3

Design

In this Chapter we describe our approach for discovering block correlations and using them for prefetching. First, we define contexts and we introduce our technique for leveraging them. Then, we explain the tradeoffs associated with different context granularities. Next we discuss the mining algorithms and the rule management in the presence of data evolution.

3.1 Context Definition

In this work, we use application-level contexts in order to increase accuracy of block miss predictions and thus guide I/O block prefetching. Specifically, we group together low-level events, i.e., I/O block accesses corresponding to page misses in the buffer pool of the database system, which occur on behalf of the same higher level logical unit of work, called a *context.* A context is the logical grouping of events according to a specific level in the application's structural hierarchy i.e., the same thread, the same transaction or the same query. Contexts are delineated with begin and end delimiters and can be nested. For instance, a database query context for a particular application is nested within a database transaction, which in its turn is nested within a web interaction for that application.

3.2 Overview

Having defined contexts in the previous section, in this section, we focus on our technique for leveraging contexts for I/O block prefetching.

We use grouping by context in order to determine accurate block correlations. We instrument the DBMS to tag each I/O request with its specific context identifier. The storage server can then generate block correlation rules for each context. At the storage layer we maintain the collection of block correlations sorted in decreasing order of support numbers, since higher support means higher frequency. For each I/O block access we look at the sequence of I/O blocks that was already accessed in this context instance and the set of block correlations with the highest support. Next, we determine the set of I/O blocks that are most likely to be accessed next and we issue prefetches for these blocks. Changing the DBMS to incorporate the context into an I/O request is trivial since contexts already exist in the database system and they are easily obtained for each I/O operation. Furthermore, the contexts that we use are not DBMS-specific; they are present in any database. For the purpose of our study we use the popular MySQL database system with the InnoDB storage engine.

In the following we detail our techniques for instrumenting the DBMS, generating block correlations and issuing prefetch requests at run-time.

3.3 DBMS Instrumentation

We instrument the database system to enable tracking of contexts of various granularities, corresponding to application structure and to tag each I/O request with a context identifier.

We currently track information about three types of contexts: web interactions, database transactions and database queries. We re-use pre-existing begin and end markers, such as, connection establishment/connection tear-down with the database system from the upper tier (for the web interaction context type), begin_transaction and end_transaction queries specified by the application writer (for the database transaction context type), and command processing (for the query context type).

The context identifiers allow the storage system to differentiate block accesses occurring within each context and to group them accordingly. This allows removing false correlations due the inherent interleavings of several threads or several applications. For *web interaction* contexts, we tag block accesses with the `thread id` of the database system thread running the interaction. We differentiate *transaction* contexts by tagging all block accesses between the `BEGIN` and `COMMIT` with the transaction identifier (`tid`). A *query* context simply associates each block access with the query identifier. MySQL maintains global counters for the transaction and query identifiers. Even if this was not the case, incorporating these identifiers would be trivial in any DBMS.

3.4 Rule Generation

We train our prediction algorithms at the storage level on perceived block accesses occurring within each context instance and derive a set of block correlation rules for each context instance. Training of rules for a new application and its various contexts can be done while the system performs its regular activities, including running other applications.

Frequent sequence mining algorithms produce sequences in the form of $\{a_1a_2 \ldots a_k\}$ that correspond to block correlation rules. However, at the storage server it is easier to work with rules represented as $\{a_1 \& a_2 \texttt{->} a_3\}$. In the mining algorithms that we implemented we generate only rules in the latter form by imposing a limit on the size of the frequent sequence. For example, two frequent sequences {`acde`} (support 2) and {`bade`} (support 3) now translate into the following 3-block correlation rules: {`a&c->d`} (support 2), {`a&c->e`} (support 2), {`a&d->e`} (support 5), {`c&d->e`} (support 2), {`b&a->d`}

(support 2), {`b&a->e`} (support 2), {`b&d->e`} (support 2).

3.5 Block Prefetching

The storage server uses the block correlation rules on the fly to predict the set of blocks to be accessed next based on the most recent sequence of blocks seen, as follows. The storage manager tracks the sequence of blocks accessed by each context instance. On a cache miss at the storage cache, we determine the context instance it belongs to. Based on the sequence of I/O blocks that was already accessed and the matching set of block correlations with the highest support, we issue prefetches for the set of I/O blocks that are most likely to be accessed next. We prefetch only on storage cache misses since we do not want to interfere with the behaviour of the application. For fast access to the block correlation rules we keep them in a hashtable, with the rule prefix as key and a list of blocks to prefetch as value.

3.6 Advantages of Contexts

Compared to context-oblivious prefetching, contexts improve the efficiency of prefetch rule generation and use. As we will show in Section 4, these translate into higher cache hit rates using a lower number of rules.

The success of the context-unaware approach depends on the assumption that correlated blocks occur within a predefined distance (gap) of each other and that they are frequent enough to eliminate false correlations. However, the value of the gap is by necessity fixed during prefetch rule training. In contrast, the gap between correlated blocks can be arbitrarily large during an actual run, due to dynamic and nondeterministic application scheduling and context switches. As we will see in Section 4, the hit rate when not considering contexts can be lower than that of a simple cache with no prefetching policies implemented.

Context-guided rule training is more efficient because we can avoid generating false block correlation rules for the blocks at the context switch boundary. Thus, the rules generated during training have a high likelihood of triggering useful prefetches on-line. Furthermore, another benefit is the increased time and space efficiency, due to the lower number of rules.

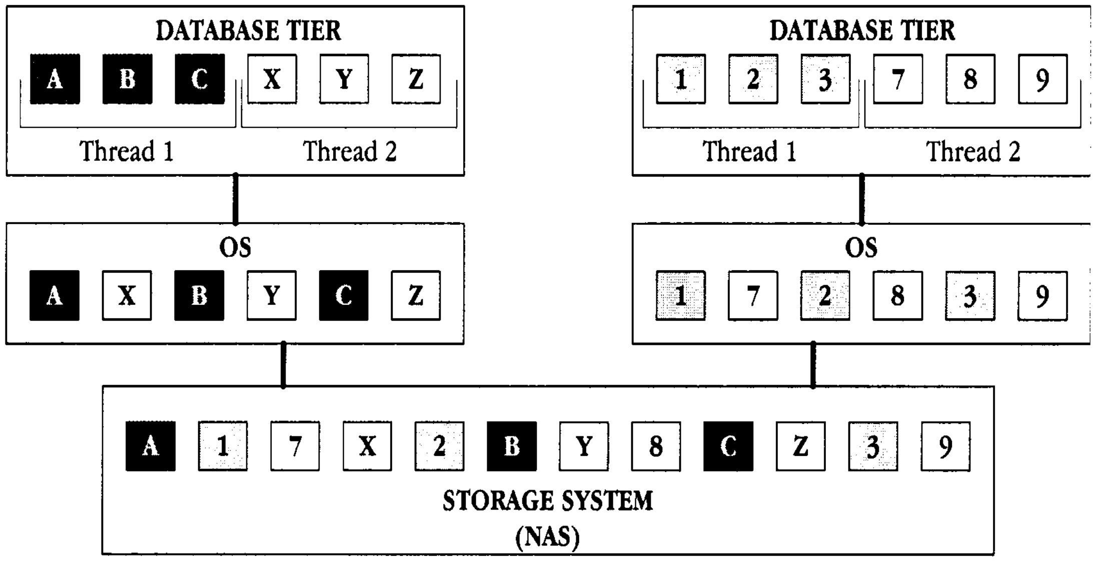

Figure 3.1: I/O Access Pattern

As an example, consider two DBMS engines running two different applications on a cluster with consolidated NAS storage server, as shown in Figure 3.1. Each DBMS is running two different queries concurrently for the same application. For example, for the DBMS on the right, one query is performing a table scan and the other an index scan through a non-clustered index. Given that the operating system may context switch among these threads, the sequence of page references that each operating system sees is intertwined between the two threads. Moreover, the sequence of block accesses at the storage server is a mixture of page misses for all 4 threads, making it difficult to decipher the original reference streams. Hence, with a sufficiently high concurrency degree, storage-level block prefetching techniques would become ineffective, unless the storage manager can track the different contexts and infer the higher-level context of each block access.

In the following we discuss the effects of using different context granularities on prefetching effectiveness.

3.7 Tradeoffs

While defining meaningful contexts is intuitive, defining the right context granularity for optimizing the prefetching algorithm is not a trivial problem. There is a tradeoff between using coarse-grained contexts and fine-grained contexts. Fine-grained contexts provide greater prediction accuracy while coarse-grained contexts provide more prefetching opportunities.

For example, consider using query templates, a fine grained context. For point queries, the access pattern is to simply traverse down the B-Tree to reach the data page. In many cases, the index pages are cached in the buffer pool and the only miss seen at the storage manager is a request to read the data block. If there is only one miss, then no prefetch rule will ever be generated or triggered for that context. This limits our prefetching aggressiveness because we may not be able to fetch several blocks at the same timc.

In contrast, coarse-grained contexts, such as, a database transaction or a web interaction are well suited for aggressive prefetching. Since web interactions for an application include more than one transaction, and each database transaction includes several read or write queries, we can derive correlations across several queries or several transactions. On the down side, having a coarser-grained context does not necessarily translate into higher prefetching accuracy if the access pattern within this context shows a lot of variability. Variability may occur due to control flow in the application code of complex database transactions or web interactions. Since a fine grain context typically has less variability, the accuracy of block correlations is correspondingly higher for fine grained versus coarse grained contexts.

3.8 Data Evolution

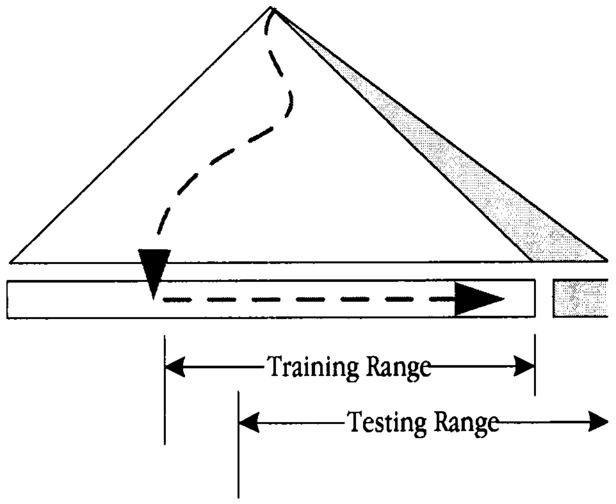

Figure 3.2: Data Evolution Over Time.

Prediction algorithms based on past access patterns are inherently limited when access pattern changes occur. Thus, in order to maintain performance, we have to adapt the rules over time. For example, consider a BestSeller query that reports the "hot" books at any given time. This query depends on the most recent orders placed at an online store to determine which books were bought recently. The access pattern of this query is to simply descend down the B-Tree and then look at the set of the most recent orders. Clearly, the access pattern of this query changes over time as more orders are placed at the online store. As more orders are inserted into the relation, the B-Tree grows towards the right (see Figure 3.2). Thus, over time, the block correlations from the left of the B-Tree become obsolete.

3.8.1 Static mining

The static mining algorithm is applied on a database of sequences and requires the entire database to be accessible at mining time. In our scheme a context instance is the equivalent of a sequence. The I/O access stream of each context instance is recorded and added to the database. When mining is to be performed (e.g., due to re-training decisions), this database sequence is used. Static mining algorithms can be improved by discarding the old context instances from the database. This approach is similar to the sliding window approach used in the dynamic mining algorithms. However, the drawback of the static algorithm comes from the fact that, unlike the dynamic one, it cannot efficiently add new correlations on the fly while the application is running. It has to be run on the entire database of context instances that we want to mine, usually because these algorithms have been designed to perform more than one pass over the sequence database.

In order to deal with data evolution, the static mining approach requires efficient re-training decisions. By monitoring rule efficiency (e.g, the increase of useful prefetches triggered) we are able to identify the moment when re-training is needed. We assume that after a mining step is performed, the rules are the most accurate. Hence, we split the running time into intervals of block accesses. For the first running interval following a mining step, we compute the rule efficiency parameter. Furthermore, we compute this parameter for the following intervals and we compare it against the first one. So long as the ratio between the two is above a predefined threshold, we consider the current rules to be effective. As soon as the ratio goes below the threshold, we decide that re-training is needed and we start the mining algorithm on the current database of context instances.

3.8.2 Dynamic mining

For the dynamic mining implementation we maintain a window of context instances and we slide it as a new context instance ends. The algorithm that we implemented is similar to the one described in [11]. The rules are updated on the fly based on the access stream in the last context instance seen. So long as the window size is below a predefined threshold, no rules are removed. When the window size becomes equal to the threshold, both the rules belonging to the oldest context instance as well as the ones belonging to the newest context instance are updated. More specifically, the support of the rules generated from the oldest context instance is decreased by one, while the support of the rules from the new context instance is incremented by one. Basically, each new context instance in the data stream replaces the oldest context instance in the window. The benefit of the dynamic algorithm compared to the static one is that it quickly adapts to any change in the I/O access pattern. No re-training decisions have to be made in this case, since the algorithm automatically re-trains by discarding correlations that become obsolete due to data evolution and by generating new ones.

Chapter 4

Evaluation

This section describes the set of benchmarks we use and our evaluation methodology, based on trace-driven simulation.

4.1 Benchmarks

We use two benchmarks to evaluate our work: the DBT-2 OLTP benchmark (a TPC-C like benchmark), and the RUBiS online bidding benchmark.

4.1.1 DBT-2

DBT-2 is an OLTP workload derived from TPC-C benchmark [24, 33]. It simulates a wholesale parts supplier that operates using a number of warehouse and sales districts. Each warehouse has 10 sales districts and each district serves 3000 customers. The workload involves transactions from a number of terminal operators centered around an order entry environment. There are 5 main transactions for: (1) entering orders, (2) delivering orders, (3) recording payments, (4) checking the status of the orders, and (5) monitoring the level of st ock at the warehouses. We scale DBT-2 by using 128 warehouses and the footprint of the database is 30GB.

4.1.2 RUBiS Auction Benchmark

We use the RUBiS Auction Benchmark to simulate a bidding workload similar to e-Bay. The benchmark implements the core functionality of an auction site: selling, browsing, and bidding. We do not implement complementary services like instant messaging, or newsgroups. We distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during the visitor sessions, during a buyer session, users can bid on items and consult a summary of their current bid, rating, and comments left by other users. We are using the default RUBiS bidding workload containing 15% writes, considered the most representative of an auction site workload according to an earlier study of e-Bay workloads [28].

4.2 Evaluation Methodology

We use trace-driven rule-training and trace-driven simulation of the block accesses in order to evaluate our context-aware prefetching algorithm. We run our web based applications on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL InnoDB (ver. 5.0.24) database storage engine. For DBT-2, we use the test harness provided by the benchmark while hosting the database on MySQL. We perform lightweight logging of page misses in the buffer pool of MySQL, which result in disk block accesses when running our benchmarks.

We collect the traces on a Dell PowerEdge with 8 Intel Xeon processors running at 2.8 GHz. The operating system on this machine is Ubuntu 6.06, Linux kernel 2.6.27-smp. Then we use the trace obtained to drive our storage cache simulator (`CacheSim`).

In the following, we describe our methodology for collecting traces and our simulation methodology for the storage cache.

4.2.1 Trace Collection

To obtain the traces of disk accesses, we instrumented the InnoDB storage engine inside MySQL. We implemented a lightweight instrumentation library that allows us to record I/O requests with negligible overhead. First, to avoid locking overhead, we create a private logging buffer per thread. In addition, we only log on buffer pool misses. Finally, we flush the logs to disk only when the buffer is full or if the thread performing the accesses is being shutdown. We log page identifiers in MySQL. The page identifiers are equivalent to disk block numbers for the purposes of our simulation since MySQL maintains database data contiguous on disk. We disabled the file system cache by opening the InnoDB data file with the `O_DIRECT` flag. Thus, the misses that we observe are not affected by any file system caching effects below the DBMS. While collecting the traces, there was no noticeable decrease in the throughput of the applications.

We generate traces by running each benchmark or set of benchmarks for 1 hour and remove a portion of the run to separate the cold storage cache effects. We split the remaining trace into 2 parts: the training trace and the testing trace. For the static mining approach, we train the block correlation rules on the training trace and drive our cache simulator with the testing trace, while for the dynamic mining one we run the whole trace but record the results only for the testing part.

4.2.2 CacheSim

We simulate the storage cache using `CacheSim`, a cache simulator we implemented in Java that allows us to evaluate different prefetching policies. `CacheSim` runs in two modes: with prefetching enabled or prefetching disabled. The size of the cache is defined by a parameter `totalCacheSize`.

When prefetching is enabled, the total cache size is divided among the *PrefetchCache* and the *MainCache*. In this design, a block may either be in the *MainCache* or the

PrefetchCache but not both. Thus, there are 3 possibilities on storage cache access: (1) the block is found in the *MainCache*, (2) the block is found in the *PrefetchCache*, (3) the block is not found (cache miss). If the block is found in the *MainCache*, a reference bit for the block is set and the block is returned to the caller. If the block access misses in the *MainCache*, then the *PrefetchCache* is checked. If the block is found in the *PrefetchCache* then, the block is promoted to the *MainCache*. On a cache miss, in addition to fetching the block, we issue prefetch requests to the underlying storage. Both the *PrefetchCache* and the *MainCache* caches implement the *Clock* replacement policy [14].

We issue prefetch requests based on the rules generated from our mining algorithm as described in Section 3. We store the rules in a hashtable with the rule prefix as the search key. For example, if the mining algorithm generates rules `[ {1&2->3,3},{1&2->100,1}, {1&2->4,2}]` then we use `1&2` as the hashtable key. We chain the collisions in a sorted linked list in decreasing order of support. Therefore, the above rules would be kept as `[{1&2->3,3}, {1&2->4,2}, {1&2->100,1} ]`. To effectively utilize the disk bandwidth, we allow the number of prefetch requests to be issued by the parameter `maxPrefetchRequests`. For example, if this parameter was set to 2, we would only issue `[{1&2->3,3}, {1&2->4,2}]`. In some cases, the block to be prefetched may already be present in the *MainCache* or the *PrefetchCache*. If the block is in the *MainCache*, we take no action. However, if the block is found in the *PrefetchCache*, we simply set the reference bit to 1. The intent is to keep the block in the cache longer since multiple rules ask for the same block.

4.2.3 Performance Metrics

During testing, we run the testing trace through `CacheSim` and measure several performance metrics including cache hit rate, rule efficiency, and rule accuracy. The hit rate is the number of blocks found in the storage cache divided by the number of accesses. Rule accuracy measures the benefit of the block correlation rules that we generate through mining. It is defined by the fraction of prefetch requests that generated hits in the prefetch cache out of the number of total prefetch requests issued. For example, if, for a specific trace, we issue 100 prefetch requests but only 50 of these are used, the rule accuracy is 50%. We define rule efficiency to be the increase in number of rules used during testing over a period of time. This parameter reflects the usefulness of the rules mined and is only valid for the static mining approach. It is used to trigger re-training when rules become obsolete, as described in Section 3.8.1. For example, suppose that during a time interval *T1* we have 100 misses in the *Main Cache.* 50 out of these 100 misses were found in the *Prefetch Cache.* The rule efficiency for interval *T1* is 50%.

Chapter 5

Results

5.1 Overview

We use trace based simulation to evaluate the benefit of our prefetching scheme. We experiment with four schemes. Our baseline approach is **No Prefetching** where prefetching is disabled. In the **No Context** scheme, we issue prefetch requests using the rules obtained from the context-oblivious mining algorithm. There is no notion of context in this case, thus, the accesses are not separated when mining is performed. For the **Web Interaction** context, we group block accesses using the connection setup and teardown delimiters. The **Transaction** context is defined by the transaction delimiters, begin and commit. Finally, for the **Query** context, we associate block accesses with the query instance. For RUBiS we noticed that transactions are usually composed of a single query, so we do not use the **Transaction** context for this benchmark. Similarly, since DBT-2 is not a web based application, we do not use the **Web Interaction** context.

For the **No Context** approach we use a *lookahead distance* of 10 when generating the block correlations with the mining algorithm. This large value is chosen for filtering out as much noise as possible, in the interest of fairness to the **No Context** scheme. For context-based mining, we use a smaller *lookahead distance* (of 5), since most of the block correlations that we generate are true correlations.

We experiment with various cache sizes from 10% to 90% of the memory footprint of the application. We measure the performance metrics with rules generated using the static mining algorithm with and without retraining, and using the dynamic mining algorithm. When using the static algorithm with retraining, we set the retraining rule efficiency threshold to 50%. This means that, when the rule efficiency for an interval decreases to less than 50% of the rule efficiency of the first interval after mining, we decide that the rules have become obsolete and retraining is performed.

In all configurations, when prefetching is enabled, we use a prefetch cache that is 6% of the total cache size and on each miss we prefetch up to 16 blocks, using the block correlation rules. We also ran experiments with various prefetch cache sizes and various limits for the maximum number of block to prefetch at once, and the results were similar.

5.2 Static Mining

In Figures 5.1 and 5.2, we show the benefit of prefetching for the two benchmarks when using the static mining approach with and without retraining. We plot the hit rate on the y-axis and the x-axis shows the different cache sizes for the methods we use in our comparison.

In Figure 5.1, we plot our experimental results for the RUBiS workload for storage caches from 16MB to 128MB, while Figure 5.2 shows the hit rates for DBT-2 for storage caches from 256MB to 4GB. The two applications have different working set sizes, thus we use different storage cache sizes in our evaluation. RUBiS, for instance, has a hit rate in the baseline case of 74% with a cache size of 128MB. On the other hand, DBT-2's baseline hit rate in the storage cache is only 14% for a cache size of 1GB.

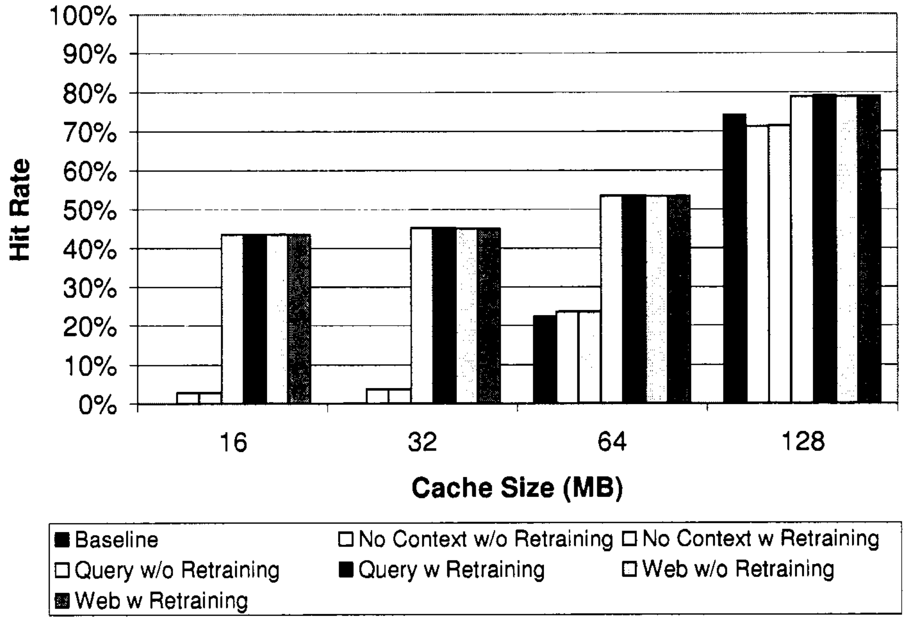

Figure 5.1: RUBiS Static Mining Hit Rate

5.2.1 Hit Rate

The results for RUBiS show that, when the cache size is small (16MB and 32MB), the baseline hit rate is less than 1%. When performing mining without the knowledge of context the hit rate is improved to about 4%. Using the query context, the hit rate improves significantly to about 45%. This improvement is achieved because the context-guided prefetching is able to accurately pinpoint the blocks that will be accessed in the near future. Using the web interaction as the context provides similar improvements as the query context. After analyzing the traces we noticed that in a web interaction composed of a number of queries, most of the time only one query has a high number of disk accesses. Thus, the rules derived through mining and the performance achieved in the two cases are similar.

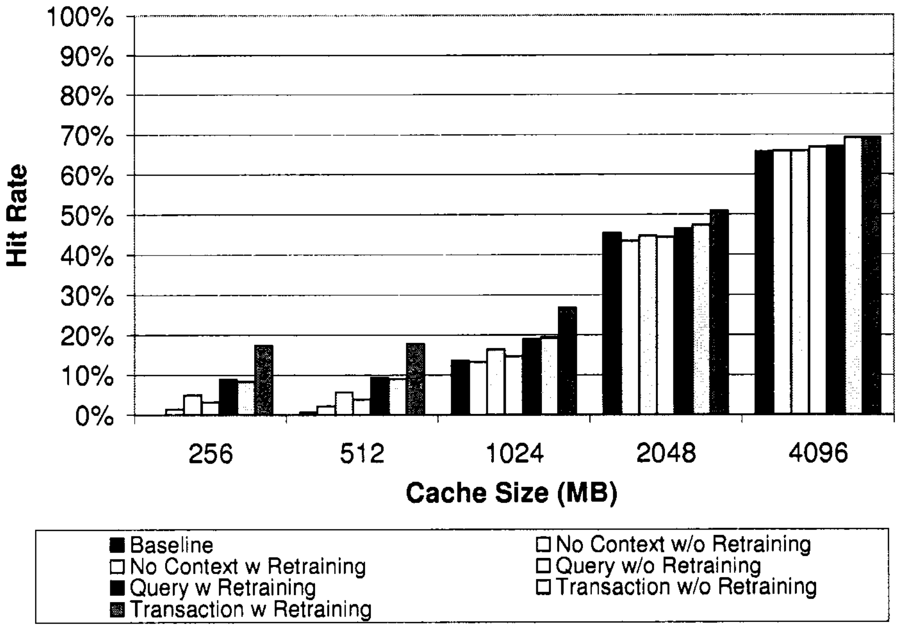

Figure 5.2: DBT-2 Static Mining Hit Rate

As the size of the storage cache increases, the baseline hit rate naturally improves and the benefit of prefetching decreases, mainly because in our scheme we prefetch only on misses. With a 64MB cache, we get a 30% increase in hit rate when using contexts. Even with a 128MB cache, context-guided prefetching provides a 5% improvement over the baseline case when prefetching is disabled. In the context-oblivious approach, the hit rate decreases and the performance is lower than the baseline. This is because the space occupied by the prefetch cache is not efficiently utilized.

The results with and without retraining are similar. As we can see in Figure 5.5 the rule efficiency for RUBiS is stable throughout the testing period and retraining is not actually triggered. This is explained by the fact that RUBiS is not a write intensive workload.

Figure 5.2 shows the results for the DBT-2 workload. Since DBT-2 has a larger I/O

RUBiS	
Scheme	**Number of Rules**
No Context (before pruning)	8,027,238
No Context (after pruning)	1,756,222
Query Context	862,287
Web Interaction Context	882,390
DBT-2	
Scheme	**Number of Rules**
No Context (before pruning)	36,415,456
No Context (after pruning)	3,019,310
Query Context	2,842,490
Transaction Context	4,600,251

Table 5.1: Static Mining Rule Generation

footprint, we use larger cache sizes for these experiments. Similar to the RUBiS results, the smaller the cache sizes the higher the prefetching benefit. When using contexts we get a much higher hit rate than when performing mining without context knowledge. This is because of more accurate predictions of future block accesses. The most interesting insight obtained from these results is that the choice of context granularity affects the prefetching benefit. As Figure 5.2 shows, prefetching using the query context performs poorly compared to the transaction context. This is due to the nature of DBT-2, where many queries are point queries which only result in 1 block access per query template. By using the transaction context, we prefetch across query boundaries and thus provide a significant benefit. Unlike RUBiS, we see that retraining improves the hit rate for DBT-2 by a lot. For a 1GB cache size for instance, using the transaction context we obtain a 5% hit rate increase without retraining and a 13% increase with retraining compared to the baseline case.

RUBiS	
Scheme	**Rule Accuracy**
No Context w/o Retraining	25%
No Context w Retraining	25%
Query Context w/o Retraining	49%
Query Context w Retraining	49%
Web Interaction Context w/o Retraining	49%
Web Interaction Context w Retraining	49%
DBT-2	
Scheme	**Rule Accuracy**
No Context w/o Retraining	60%
No Context w Retraining	67%
Query Context w/o Retraining	75%
Query Context w Retraining	79%
Transaction Context w/o Retraining	56%
Transaction Context w Retraining	62%

Table 5.2: Static Mining Rule Accuracy

5.2.2 Rule Parameters

In Table 5.1 we show the total number of rules generated with the static mining algorithm. A context-oblivious approach generates 8.03 million rules for RUBiS and 36.42 million rules for DBT-2. Since many of the rules have very low support, they are pruned by choosing a minimum support of 1. By pruning the rules with support 1, we expect to eliminate the false correlations. After pruning, the number of rules is significantly reduced to 1.76 million rules for RUBiS and 3.02 million rules for DBT-2. With context-awareness, we generate between 3 and 4 million rules for DBT-2 and 0.8 million rules for

RUBiS. The memory footprint of 1 million rules is around 16MB, by estimating that a rule can be represented by 16 bytes. Thus, keeping these rules in memory is feasible.

Table 5.2, shows the rule accuracy for both workloads. When using contexts, the accuracy is high enough for both applications. For the no context approach, the accuracy when running RUBiS is very low. This is because RUBiS has a higher context switch rate, thus more false correlations than DBT-2. For DBT-2, when performing retraining we also improve the rule accuracy compared to the run without retraining.

5.3 Dynamic Mining

In this section we present the results for the dynamic mining approach. First, we compare the dynamic mining results against the static mining ones. Next, we analyze the dynamic mining scheme with various window sizes. The dynamic mining technique that we used cannot be applied to the no context approach. That is because it requires the access stream to be split into sequences. That is the whole concept behind the sliding window technique. Thus, we only plot results for the context-based schemes. We use a window size of 10,000 for RUBiS and 80,000 for DBT-2. This numbers are around half of the total number of context instances in a trace for that benchmark.

5.3.1 Dynamic vs. Static

Hit Rate

In Figures 5.3 and 5.4, we compare the hit rates for the two benchmarks when using the dynamic mining and static mining with retraining schemes. For RUBiS, Figure 5.3, dynamic mining improves the hit rate with up to 2% when compared to the static mining with retraining. Compared to the baseline, the hit rate improvement is between 5-47%. For DBT-2, Figure 5.4, the improvement is up to 4% relative to the static mining with retraining results, the maximum benefit being obtained for the transaction context.

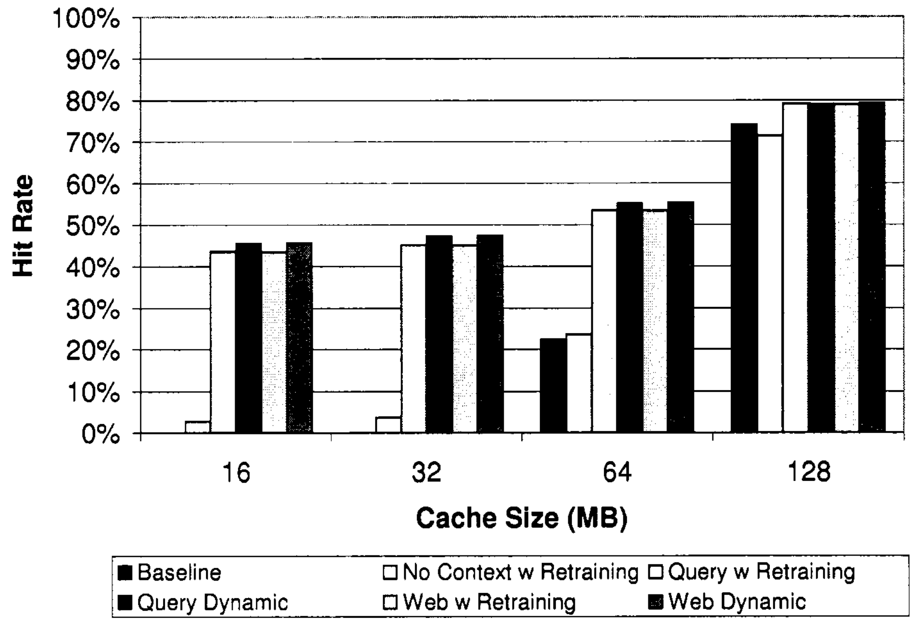

Figure 5.3: RUBiS Dynamic vs. Static Mining Hit Rate

Compared to the baseline, the increase is between 15-21% for storage cache sizes between 256MB-1GB and from 3 to 6% for cache sizes of 2-4GB. Even though retraining is very useful for static mining, when generating rules dynamically we eliminate the static threshold that triggers retraining decisions.

Rule Parameters

In Figures 5.6 and 5.5, we show the *cumulative* number of efficient rules over time for a 30 minute run. An efficient rule is a rule whose prefetched block is consumed by the database system, a rule that translates into a useful prefetch request. For RUBiS we measure with a cache size of 64MB and for DBT-2, 512MB. We use the web interaction context for RUBiS and the transaction context for DBT-2.

The two workloads show different characteristics. For RUBiS, the static mining with-

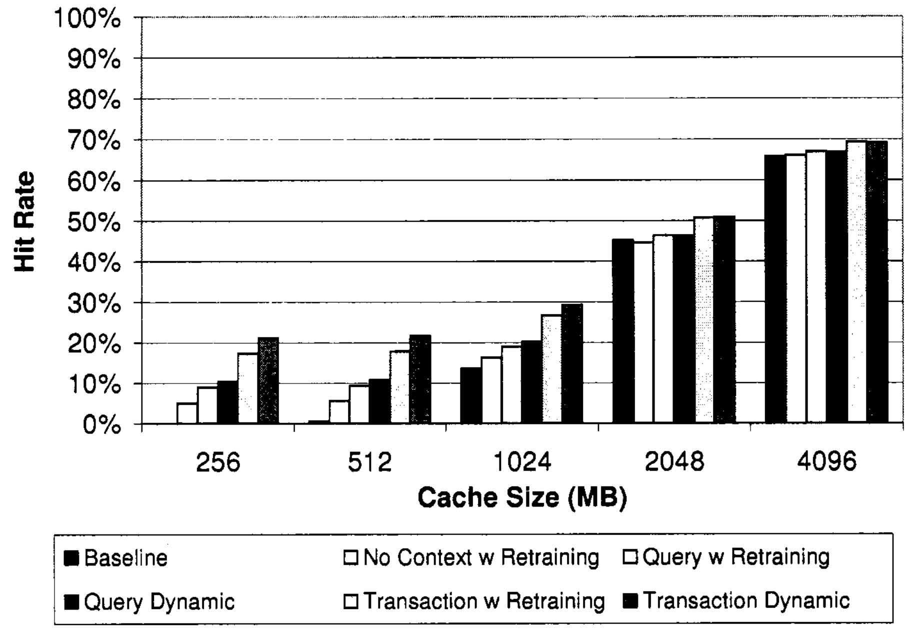

Figure 5.4: DBT-2 Dynamic vs. Static Mining Hit Rate

out retraining curve grows linearly, showing that the rules are still valid and that retraining is not needed. In fact, for this run, retraining was not triggered as we can see from the static mining with retraining curve. When using dynamic mining we obtain a slightly higher slope, as expected. RUBiS is more read intensive and rule efficiency, hence prefetching benefit for this application continues to be roughly the same during the run.

While RUBiS shows the same slope for the growth in the number of efficient rules triggered over time, the growth of DBT-2 efficient rules gradually flattens starting around the 9 minute mark, when no retraining is performed. This is due to the write intensive nature of the DBT-2 benchmark. After the first 9 minutes of the run, the significant data changes in DBT-2 make our prefetch rules for this application obsolete. As new data blocks are accessed due to INSERT queries and the block sequences change, the number of prefetch rules that are triggered for DBT-2 decreases. When performing retraining,

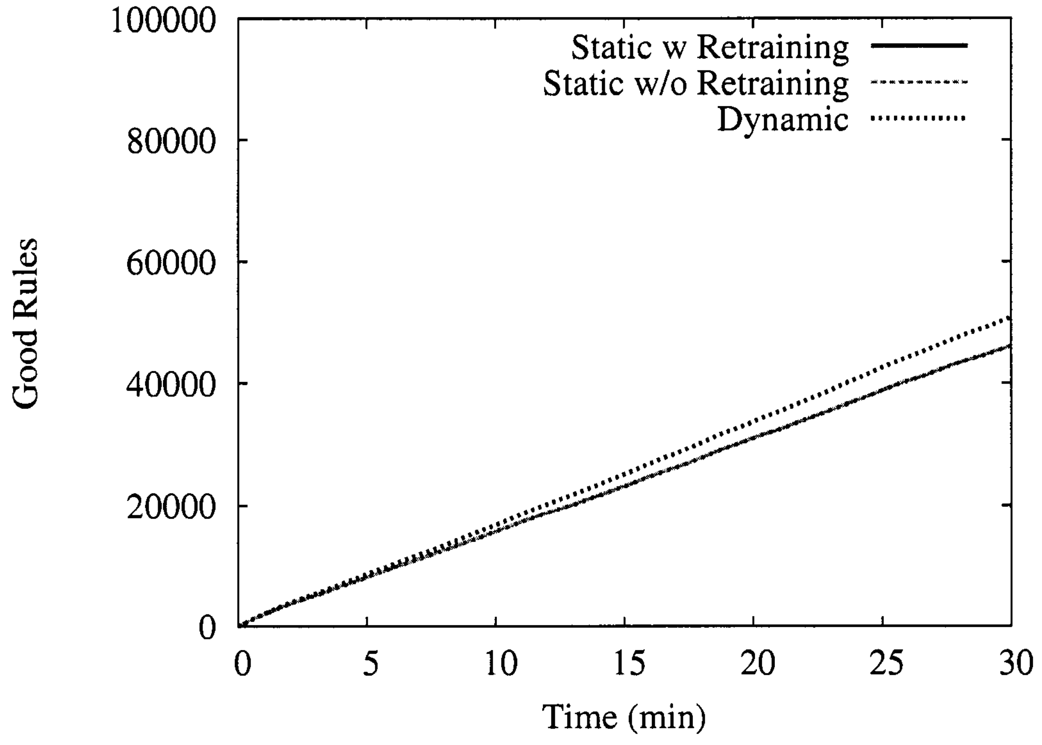

Figure 5.5: RUBiS Dynamic vs. Static Mining Rule Evolution

we notice that there are two retraining points in the run, one around the 9 minute mark, and one around the 20 minute one. By retraining we are able to refresh the rules, thus maintain a high number of efficient rules. However, with dynamic mining we get the best performance. The curve is linear, since the rules are updated on the fly: obsolete rules are removed, while new rules are added to the list.

Table 5.3, shows the rule accuracy for both workloads, comparing the dynamic and the static mining with retraining schemes. RUBiS has similar rule accuracy for all the schemes with context. For DBT-2, when performing dynamic mining we also improve the rule accuracy compared to the run with retraining.

5.3.2 Varying the Window Size

In the next set of experiment we ran the dynamic mining algorithm with various window sizes and analyzed the hit rate, rule evolution and rule parameters. We used the DBT-2

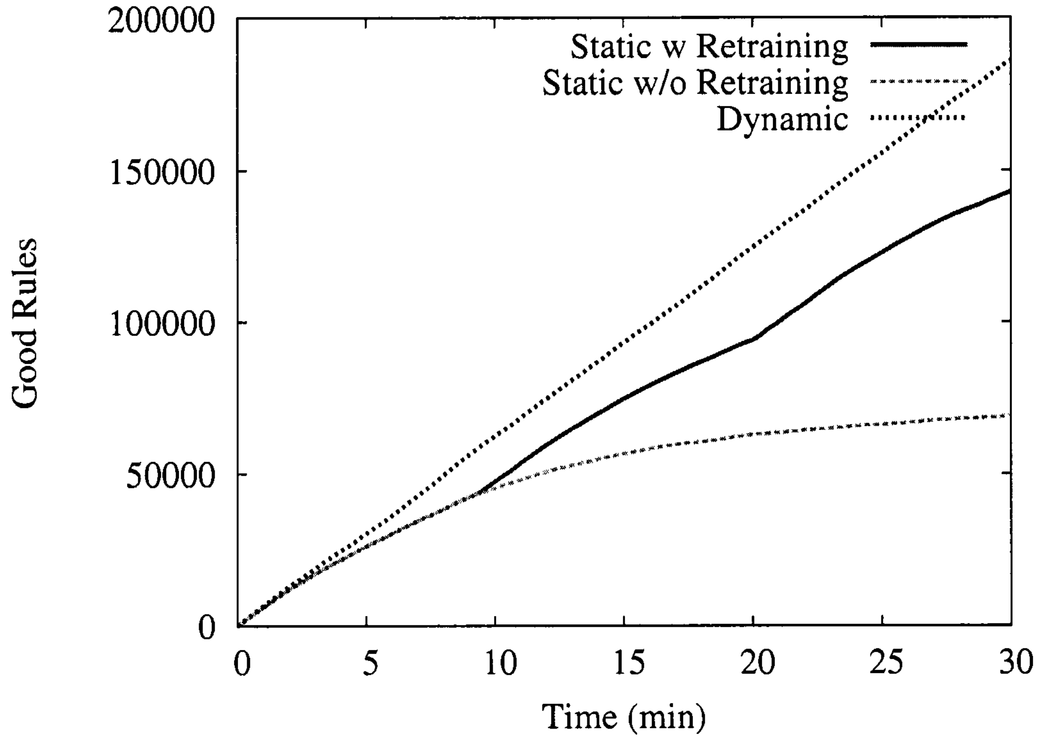

Figure 5.6: DBT-2 Dynamic vs. Static Mining Rule Evolution

benchmark, the transaction context, and window sizes from 20,000 to 80,000.

Figure 5.7 plots the hit rate for the different window sizes used. We see that even when using a smaller window size, such as 20K, we get a high increase in the hit rate. The results for window sizes of 60K and 80K are similar, thus it does not make sense to use a larger window size. This is because only rules from the last 60K context instances are valid in the future.

In Figure 5.8 we show the rule evolution for a cache size of 512MB and for the window sizes used in the mining algorithm. The results follow the hit rate ones. We see that all the curves are linear with an increasing slope as we use a larger window size. For window sizes of 60K and 80K the curves are similar.

The advantage of using a smaller window size comes in terms of the memory used to maintain the rules. In Table 5.4 we see that the number of rules generated goes higher as we increase the window size. However, even by maintaining only 1 million rules in

RUBiS	
Scheme	**Rule Accuracy**
No Context w Retraining	25%
Query Context w Retraining	49%
Query Context Dynamic	49%
Web Interaction Context w Retraining	49%
Web Interaction Context Dynamic	49%
DBT-2	
Scheme	**Rule Accuracy**
No Context w Retraining	67%
Query Context w Retraining	79%
Query Context Dynamic	85%
Transaction Context w Retraining	62%
Transaction Context Dynamic	70%

Table 5.3: Dynamic vs. Static Mining Rule Accuracy

memory we get an improvement of up to 11% in hit rate. With 3 million rules we get up to 20% higher hit rates in the storage cache.

In Table 5.5 we show the rule accuracy. As expected, the accuracy goes down with larger window sizes. This is because when using a window size of 80K the probability of maintaining rules that are obsolete is higher than with a window size of 20K.

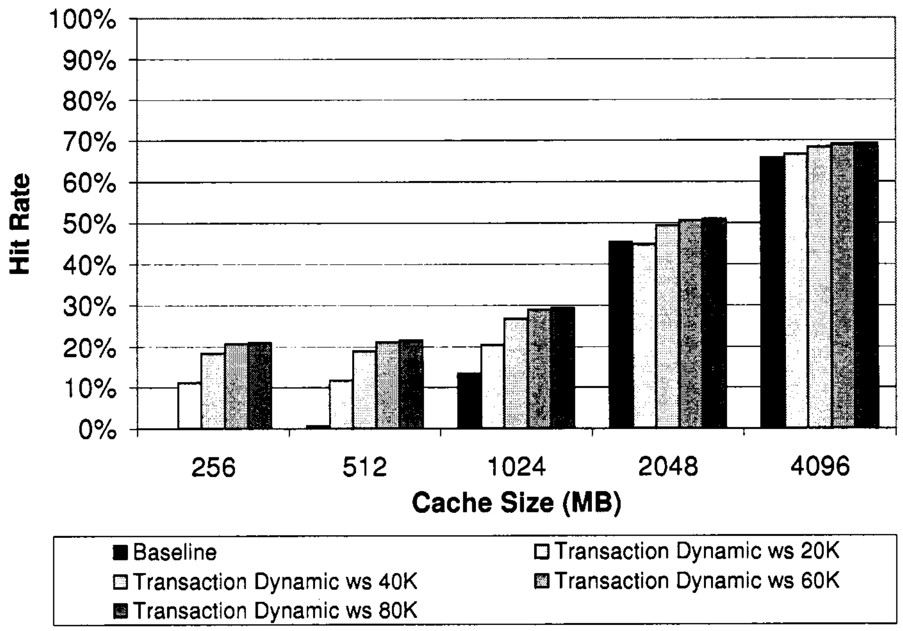

Figure 5.7: Hit Rate when Varying the Window Size

DBT-2	
Scheme	**Number of Rules**
Transaction Context Dynamic ws 20K	1,164,755
Transaction Context Dynamic ws 40K	2,243,970
Transaction Context Dynamic ws 60K	3,281,610
Transaction Context Dynamic ws 80K	4,308,628

Table 5.4: Rule Generation when Varying the Window Size

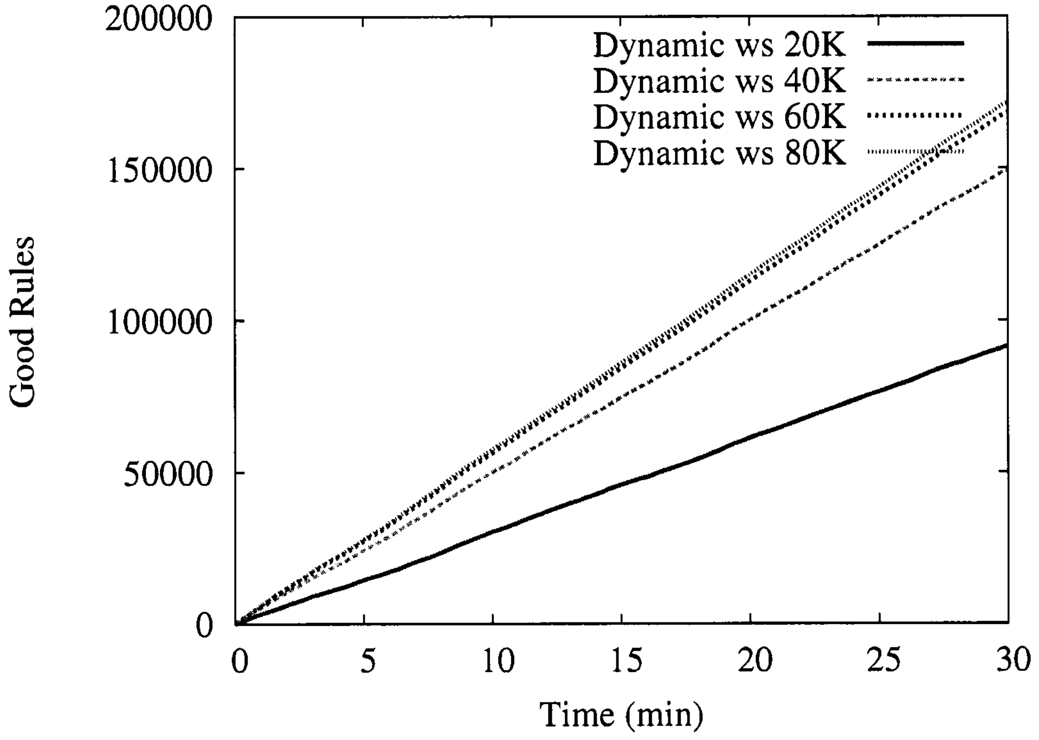

Figure 5.8: Rule Evolution when Varying the Window Size

DBT-2	
Scheme	**Rule Accuracy**
Transaction Context Dynamic ws 20K	82%
Transaction Context Dynamic ws 40K	77%
Transaction Context Dynamic ws 60K	73%
Transaction Context Dynamic ws 80K	70%

Table 5.5: Rule Accuracy when Varying the Window Size

Chapter 6

Related Work

Our research draws on related work in a variety of domains in storage systems and databases, including: inferring block correlations through "black-box" or "gray-box" approaches that do not require changing the storage interface, newly proposed storage interfaces that aim at pushing semantic knowledge from the storage clients into storage servers, high-level prefetching schemes that can take advantage of the logical knowledge, e.g. by analyzing the query strings, storage cache optimization techniques and adaptive cache management. In the following, we describe the related work in these areas and compare our approach with the most relevant related works in each area.

6.1 Inferring Block Correlations without Breaking the Interfaces

A recent work [18, 19] proposed a new frequent sequence mining algorithm to discover block correlations from streams of I/O accesses at the storage server. The authors try to correlate blocks fully transparently inside the storage server without changing the storage clients or the storage interface. Since it is a "black-box" approach, the technique is very appealing since it is general and targets any storage client. However, with a high concurrency degree in the storage clients, this algorithm can become ineffective. This is because the algorithm generates lots of false correlations while, at the same time loses the true ones. In our scheme based on static mining, we enhance this algorithm with the knowledge of contexts. Furthermore, we extend the context-guided algorithm with efficient re-training decisions, in order to refresh the block correlation rules that became obsolete due to data changes.

Related work in the file system area is similar to ours in that they expose certain aspects of the structure of the storage client e.g., the distinction between i-node and data blocks, to the storage manager indirectly through probing at the storage client [3, 31, 4].

Sivathanu et al. [30] apply the semantically-smart storage approach to databases. Through log snooping and changes to the DBMS to record a number of statistics, the storage system can provide cache exclusiveness and reliability for the database running on top. One drawback of this technique is that it is DBMS-specific, since the storage server has to have knowledge of the characteristics of all databases running on top of it. This may not be feasible in a data center for instance. Furthermore, their work is not applicable to prefetching since they cannot accurately determine multi-block correlations.

6.2 Changing the Interfaces to Bridge the Semantic Gap between Storage Clients and Servers

Type-safe disk is a newly proposed disk system that is aware of the pointer relationships between blocks in file systems, e.g. an inode block points to a data block [29]. Although the scheme is proposed in the context of file systems, it can be used in database systems as well. The authors propose enhancing the basic block-based storage interface with a number of primitives, such as `CREATE_PTR(Src, Dest)` that creates a pointer between the `Src` and `Dest` blocks. The pointers in this case are similar to the correlations in our scheme. In our work, we propose adding a context identifier to each I/O, which is a light change. Furthermore, the type-safe disks approach cannot efficiently identify multiblock correlations or correlations in high-granularity contexts, such as transactions. Having this would require implementing the mining algorithm in the storage client, which is infeasible. In our system, we target leveraging the processing capabilities of the storage system to perform the mining.

Another area of related work is the recently-standardized Object-based Storage Device interface [2] (OSD) and its applicability to database systems. Several studies [26, 25] investigate ways for providing the DBMS with more knowledge of the underlying storage characteristics. Other recent work in this area [27] performs a preliminary investigation on the usefulness of increasing semantic knowledge in storage servers by mapping database relations to objects. Our scheme can be implemented on top of the Object-based Storage Device Interface as well, by mapping objects to blocks and making use of the richer interface to pass the context identifier of the object. However, the new OSD interface requires design changes in the storage clients, besides implementation changes.

6.3 High Level Prefetching for Databases

Bowman et al. [8, 6] describe prefetching techniques leveraging dependencies between queries, which do not require modifications to interfaces. Specifically, they point out a nesting pattern where the results of the outer query are used as arguments to the inner query. The authors propose the Scalpel system where they use lateral derived tables to rewrite the outer and inner queries into a single merged query. In a follow-up paper [7], they apply a similar approach to a sequence of queries. Since we apply our methods at the storage level, our approach is complementary. In addition, we target multiple applications while the above papers target a single application.

Most database systems implement their own prefetching policy. MySQL, for instance, uses sequential read-ahead and random read-ahead to optimize its performance. In the same way, commercial database systems, such as SQL Server or DB2, can defer reading the data pages until all the row identifiers have been obtained from the index. This technique is usually called List Prefetching. Our technique is complementary to the prefetching scheme at the DBMS. In particular, in all of our experiments, MySQL has employed its usual prefetching scheme.

Our work is also related but orthogonal to other prefetching optimizations in database systems [23, 5].

6.4 Storage Cache Optimization Techniques

Several recent papers [17, 34, 9] explore techniques for establishing context communication, hence collaboration, between the database system, as the storage server client, and the storage server in order to improve caching efficiency. These papers propose to pass explicit hints from the database (client) cache to the storage cache. For example, these hints can indicate the reason behind a write block request to storage and whether a block is about to be evicted from the client cache and should be cached at the storage level [17], explicit demotions of blocks from the storage client to server cache [34], or the relative importance of requested blocks [9]. Similar to our own, these technique improve storage cache efficiency, through context communication. As opposed to our work, inserting the appropriate hints needs thorough understanding of the database system internals to distinguish the context surrounding each block I/O request. We use readily available information within the database system about preexisting contexts. These techniques, including our own, modify the interface between the storage client and server, by requiring that an additional identifier (representing the hint or our context tag, respectively) be passed to the storage server.

Related transparent techniques for storage cache optimization include inferring access patterns of the upper tier through observing characteristics of I/O requests. For example, in eviction-based prefetching [10], the storage cache detects whether a block has been evicted from the client cache by matching the in-memory address of a newly requested block with that of a block requested previously. The storage cache then issues prefetches for these blocks.

6.5 Adaptive Cache Management

The general area of adaptive cache management based on application patterns or query classes has been extensively studied in database systems and in file systems. For example the DBMIN algorithm [13] uses the knowledge of the various patterns of queries, i.e., sequential scans, index accesses, and clustered joins to allocate buffer pool memory efficiently and has been shown to be more efficient than the classic CLOCK algorithm [14].

In the file systems area, Choi et al. [12] propose a cache management scheme based on application or file level characterization of block references. Similar to our work, the authors observe that, in a multiprogramming system where multiple applications are running concurrently, meaningful access patterns can be obtained by leveraging the application or file context of a block access. The authors define a number of types of access patterns, e.g. sequential, and, at run-time, they classify the current access pattern at the application or file level. Furthermore, this classification is used in a per-application buffer allocation scheme. The scheme that we propose can be easily applied to file systems, by tagging each block request with a process, thread or file identifier.

Our approach relates as well to adaptive caching techniques based on learning, analyzed in the context of file systems [21, 22].

Chapter 7

Conclusions

In this thesis, we use a known data mining technique, frequent sequence mining, to correlate database accesses at the storage server. We evaluate I/O prefetching in the storage cache as an application of disk block correlations. We implement both a dynamic block correlation mining algorithm as well as a static block correlation mining one. For the static algorithm, we study schemes with and without correlation rule retraining. By monitoring rule efficiency we are able to make efficient re-training decisions and thus, update the rules that become obsolete due to data evolution.

Our technique is based on i) grouping block accesses according to high-level application contexts, such as queries or transactions, through lightweight instrumentation of the database system, and ii) training block prefetch rules and triggering prefetches according to these rules.

In our experimental evaluation, we perform logging of actual access patterns during runs for two applications, DBT-2, a TPC-C like workload and RUBiS, a bidding workload similar to e-Bay. We perform the mining at the storage level and measure the respective application hit rates in the storage cache. In our experiments we show that, by using block correlations, we improve the storage cache hit rate by up to 21% for DBT-2 and up to 47% for RUBiS, compared to the baseline. In addition, when training using the dynamic mining algorithm, we outperform the static mining-based approach in terms of higher hit rates and more accurate block correlation rules.

Bibliography

[1] IBM System Storage N7000 Modular Disk Storage System. *http://www-03.ibm.com/systems/storage/network/n7000/appliance/specification.html*, 2006.

[2] ANSI. Information Technology - SCSI Object-Based Storage Device Commands (OSD). Standard ANSI/INCITS 400-2004, 2004.

[3] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 43–56, Banff, Alberta, Canada, October 2001.

[4] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st International Symposium on Computer Architecture, ISCA*, pages 176–187, Munich, Germany, June 2004.

[5] Philip A. Bernstein, Shankar Pal, and David Shutt. Context-Based Prefetch for Implementing Objects on Relations. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 327–338, Edinburgh, Scotland, September 1999.

[6] Ivan T. Bowman and Kenneth Salem. Optimization of Query Streams Using Semantic Prefetching. *ACM Transactions on Database Systems*, 30(4):1056–1101, December 2005.

[7] Ivan T. Bowman and Kenneth Salem. Semantic Prefetching of Correlated Query Sequences. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE*, Istanbul, Turkey, April 2007.

[8] Ivan T. Bowman and Kenneth Salem. Optimization of Query Streams Using Semantic Prefetching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 179–190, Paris, France, June 2004.

[9] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 145–156, Banff, Alberta, Canada, June 2005.

[10] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Cache Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference, General Track*, pages 269–281, San Antonio, Texas, USA, June 2003.

[11] Yun Chi, Haixun Wang, Philip S. Yu, and Richard R. Muntz. Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window. In *Proceedings of the 4th IEEE International Conference on Data Mining, ICDM*, pages 59–66, Brighton, UK, November 2004.

[12] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. Towards Application/File-level Characterization of Block References: a Case for Fine-grained Buffer Management. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 286–295, 2000.

[13] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of 11th International Conference on Very Large Data Bases*, pages 127–141, Stockholm, Sweden, August 1985.

[14] F. J. Corbato. A Paging Experiment with the Multics System. *MIT Press*, pages 217–228, 1969.

[15] Robert T. Douglass, Mike Little, and Jared W. Smith. *Building Online Communities With Drupal, phpBB, and WordPress.* Apress, Berkeley, CA, USA, 2005.

[16] Jim Gray. The Next Database Revolution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–4, Paris, France, June 2004.

[17] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-Tier Cache Management Using Write Hints. In *Proceedings of the FAST '05 Conference on File and Storage Technologies*, San Francisco, California, December 2005.

[18] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *Proceedings of the FAST '04 Conference on File and Storage Technologies*, pages 173–186, San Francisco, California, USA, March 2004.

[19] Zhenmin Li, Zhifeng Chen, and Yuanyuan Zhou. Mining Block Correlations to Improve Storage Performance. *ACM Transactions on Storage*, 1(2):213–245, May 2005.

[20] Chih-Hsiang Lin, Ding-Ying Chiu, Yi-Hung Wu, and Arbee L. P. Chen. Mining Frequent Itemsets from Data Streams with a Time-Sensitive Sliding Window. In *Proceedings of the SIAM International Conference on Data Mining, SDM*, Newport Beach, California, USA, April 2005.

[21] Tara M. Madhyastha, Garth A. Gibson, and Christos Faloutsos. Informed Prefetching of Collective Input/Output Requests. In *Proceedings of the 1999 ACM/IEEE*

conference on Supercomputing: High Performance Networking and Computing, Portland, Oregon, USA, 1999.

[22] Tara M. Madhyastha and Daniel A. Reed. Learning to Classify Parallel Input/Output Access Patterns. *IEEE Transactions on Parallel and Distributed Systems*, 13(8):802–813, August 2002.

[23] Mark Palmer and Stanley B. Zdonik. Fido: A Cache That Learns to Fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, Barcelona, Catalonia, Spain, September 1991.

[24] Francois Raab. TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. 1993.

[25] Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Lachesis: Robust Database Storage Management Based on Device-specific Performance Characteristics. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 706–717, Berlin, Germany, September 2003.

[26] Jiri Schindler, John L. Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the FAST '02 Conference on File and Storage Technologies*, pages 259–274, Monterey, CA, January 2002.

[27] Steven W. Schlosser and Sami Iren. Database Storage Management with Object-based Storage Devices. In *Workshop on Data Management on New Hardware, DaMoN 2005*, Baltimore, Maryland, USA, June 2005.

[28] Kai Shen, Tao Yang, Lingkun Chu, JoAnne Holliday, Douglas A. Kuschner, and Huican Zhu. Neptune: Scalable Replication Management and Programming Sup-

port for Cluster-based Network Services. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems, USITS*, pages 197–208, San Francisco, California, USA, March 2001.

[29] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI*, Seattle, WA, USA, November 2006.

[30] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-Aware Semantically-Smart Storage. In *Proceedings of the FAST '05 Conference on File and Storage Technologies*, pages 239–252, San Francisco, California, December 2005.

[31] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the FAST '03 Conference on File and Storage Technologies*, San Francisco, California, USA, March 2003.

[32] Ramakrishnan Srikant and Rakesh Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proceedings of the 5th International Conference on Extending Database Technology, EDBT*, pages 3–17, Avignon, France, March 1996.

[33] Mark Wong, Jenny Zhang, Craig Thomas, Bryan Olmstead, and Cliff White. *Database Test 2 Architecture.* Open Source Development Lab, `http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_su%ite/osdl_dbt-2/dbt_2_architecture.pdf`, 0.4 edition, June 2002.

[34] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference, General Track*, pages 161–175, Monterey, California, USA, June 2002.