Query Affinity in Internet Applications

Minwen Ji, Edward W. Felten, Jaswinder Pal Singh and Mao Chen

Abstract

We investigate how to improve the price-performance of Internet application servers by caching their dynamic content in a cluster of cache servers. In particular, we study distribution strategies that automatically and dynamically partition and replicate data across individual cache servers and directs queries to the right servers, in order to maximize effective cache capacity and minimize synchronization cost. Despite the conflicts in partitioning dynamic content, we observe natural query affinity in a wide range of Internet applications, which could be exploited in distribution strategies. We have designed an affinity-based distribution (ABD) strategy for a cluster of cache servers. We evaluate the distribution strategy with a set of trace-based simulations. The results show that ABD reduces cache miss ratio by a factor of 1.7 to 9 over alternative strategies. The results also indicate that application servers, especially those that update their back-end databases, have a higher demand for good distribution strategies than static web servers.

1 Introduction

We shall start the discussion in this paper by considering how to improve the price-performance of Internet application servers, i.e. how to achieve high quality of service with minimal committed resources. In today's Internet data centers or server farms [6], the ability to guarantee the same service level agreement (SLA) with fewer server machines allows savings in power consumption, environmental control (e.g. cooling), administration effort, rack space as well as equipment cost.

Application servers are typically persistent processes that sit between web servers and databases and perform so-called "business logics". Many popular Internet sites, especially e-businesses [5], web portals [9] and search engines [7], provide value-added services as well as information to their clients in the form of *dynamic* content. Application servers generate dynamic content on demand by accessing and perhaps updating back-end databases and communicating with web servers using protocols such as Common Gateway Interface (CGI). In contrast, *static* content, such as HTML pages and images, is pre-existing in file systems and is loaded directly by web servers.

We investigate improving the price-performance of application servers by decoupling the request processing in application servers from the heavyweight query processing in databases. Traditional databases were not specifically designed for Internet workloads; their complexity and overhead are often depressing for Internet applications.

We consider interposing a cache server between application servers and back-end, on-disk databases. The cache server caches frequently accessed data from the on-disk databases and is optimized in buffer management, retrieval and indexing specifically for memory-resident data. Although caching and buffer management inside database systems have also been extensively studied [12] [2], traditional databases are optimized for disk I/O or client/server communications rather than for memory-resident operations. The cache server offers the same, standard interfaces as traditional, on-disk databases, such as indexed search, concurrency control and consistency guarantee; therefore, the use of the cache server is made transparent to application servers.

However, it is not unusual that the size of an on-disk database, e.g. tens of gigabytes to terabytes, exceed the physical memory capacity of a single commodity machine, e.g. hundreds of megabytes to a few gigabytes. A single cache server might limit the scalability of application servers due to thrashing or CPU saturation. Intuitively, a scalable caching system might be composed of a cluster of cache servers, each of which runs on a commodity computer and contributes to the aggregate memory capacity and processing power of the cluster as a whole.

We expect that a well-designed data/query distribution strategy for such a cluster of cache servers is necessary in order to achieve the projected price-performance advantage. The task of distribution is to partition and replicate data across individual cache servers and to direct queries to the right servers, while maintaining the consistency across servers at a low cost. The goal of this task is to maximize effective cache capacity and minimize synchronization cost. Such a task in the database community has often been performed manually and statically by database administrators. While the existing approach worked for on-disk databases, an automatic and dynamic approach is necessary for a cluster of cache servers because of the volatile nature of main memory and the rapid changes in contents and access patterns on the Internet.

Dynamic content can be accessed through multiple applications or by multiple attributes, and a single query can access multiple data items. These require the data to be partitioned across cache servers in different, often conflicting, ways. The partitioning conflicts result in data sharing across individual cache servers. If the data is read only, sharing causes data duplication or transmission across cache servers. If the data is written, e.g. when a browsing request implicitly causes an update to a customer preference database, one needs to pay synchronization cost for the data sharing. Therefore, data sharing needs to be reduced for both read-only data and write-shared data in order to improve the price-performance of a cluster of cache servers.

In the rest of the paper, we investigate scalable and cost- effective distribution strategies for a cluster of cache servers. First, we will show that a good partitioning of dynamic content does exist in certain applications despite the conflicts [Section 4]. Then we design a distribution strategy that takes advantage of this fact and compare it with alternative strategies that handle the conflicts to different degrees [Section 6]. Related work is cited in Section 7. Finally, we draw conclusions in Section 8.

2 Cluster-based Internet infrastructures

The discussion in the rest of this paper is based on the following assumptions about cluster-based Internet infrastructures. A number of *processing nodes* run web servers and application servers, and each web server or application server is capable of processing any HTTP or application-specific requests. The application servers store all their persistent data in a shared, on-disk database, which we call the *master database*. Redundancy may be applied inside the master database for high reliability and scalability. A cluster of cache servers is transparently situated between the application servers and the master database, caching partial or all data from the master database. Data accessed by a query will be *loaded* from the master database into a cache server before the query is executed in that server. Data will stay in the cache server until it is *evicted* by a cache replacement policy or *invalidated* by a synchronization protocol. The cache servers may or may not be physically co-located with the application servers. See Figure 1.

3 Challenges of dynamic content illustrated

As discussed in the previous section, the challenges in the distribution strategies are mainly raised by the conflicts across queries for dynamic content, which result in data sharing across nodes in the cluster. It can be best explained by examples.

In the first example, we consider an online bookstore like Amazon.com, where books can be queried by subject, by author or by ISBN. In an ideal case, the books are cached in a minimal number of cache servers, i.e. no data is stored redundantly; each query can be executed in one of the individual cache servers without data transmission from any others. Unfortunately, queries by different attributes require the books to be partitioned in different, possibly conflicting ways. For instance, if the books are assigned to individual cache servers by a random hash function on their ISBNs, none of the individual cache servers could guarantee to contain the complete results of any query by author or subject. Even if the books are partitioned in the ideal way, it is still hard to direct the queries to the right cache servers for execution without knowing the query results first.

In the second example, we consider two queries to a general search engine like Google, one containing the keyword "herbs" and the other "vitamins", and a highly ranked article on "healthy food". The search engine will typically return this article to both queries. Similarly to the first example, it is hard to direct the two queries to the cache server that caches this article. In fact, it is even hard to predict that the two queries should be executed in the same cache server without executing them first.

4 Observation on query affinity

Despite the variety of the contents and services that the Internet provides, we observe query affinity in a wide range of applications, including e-commerce, search engines, maps, directories, news, and digital libraries. By *query affinity* we mean the fact that there exists a way of dividing queries into groups where queries in the same group access the same or overlapped data sets (we call them *affined queries*) while queries in different groups access separate data sets. In addition, query affinity is a natural result of the content structures or access patterns of the applications, rather than a result of the physical data storage.

We hereby introduce two important sources of query affinity:

- **Containment**: data accessed by certain queries tends to contain data accessed by certain other

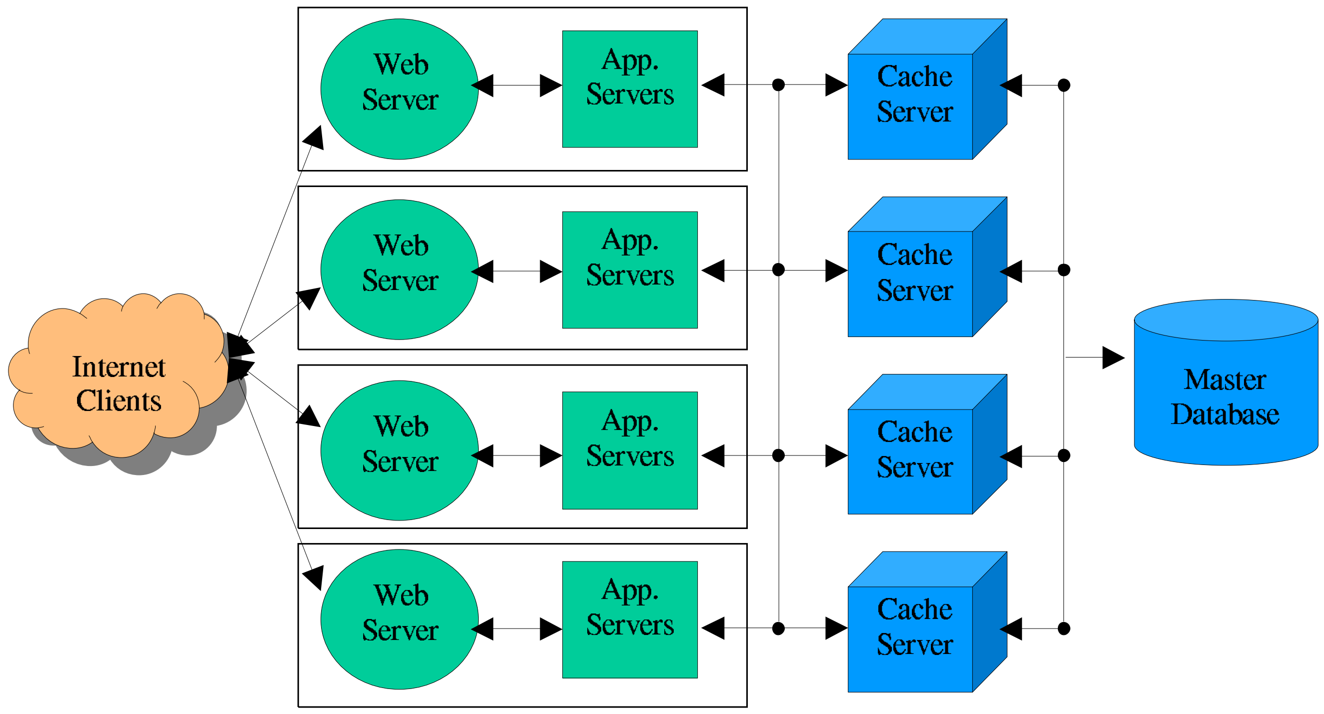

Figure 1: cluster-based Internet infrastructures.

queries. In the previous bookstore example, the books written by an author, e.g. Brian W. Kernighan, often belong to a few particular subjects, e.g. computer programming, because most authors have expertise knowledge in a limited number of subjects. In another example, such as MapQuest, the map of California contains the map of Palo Alto because California geographically contains Palo Alto. The containment relationship is transitive; therefore, there might exist chains of queries in an application where queries in the front contain queries in the back. In this case, data sets accessed by different containment chains are separate from each other.

- **Ranging**: in range searches, data items "close" to each other in a domain-specific sense are often accessed together in the same query while data items "far" apart are rarely accessed together. Examples include searches for restaurants by distance to a given location, searches for articles by a range of publishing dates, and searches for people by similar names.

Table 1 summarizes instances of query affinity in some popular Internet services.

5 Exploiting query affinity

The presence of query affinity indicates that a good distribution strategy could be found for a cluster of cache servers. If affined queries are directed to the same cache server, data sharing across servers can potentially be reduced. Queries on different servers are likely non-affined queries, and hence will access separate data sets by definition.

In order to direct a query to the same server as its affined queries as it comes in, we need the following inputs:

1. To which queries in the past this query is affined.
2. To which servers the affined queries have been directed in the past.

Based on the facts that affined queries access the same or overlapped data and that data accessed by recent queries is kept in the cache servers where the queries were executed, the two required inputs can be reduced to the following respectively:

1. What data the query will likely access.
2. In which servers some or all of the data is currently cached.

Services	Containment	Ranging
Book stores	Subject ⊃ author ⊃ ISBN	Books with close publishing dates
Auctions	Category ⊃ seller ⊃ item	Items with similar prices or locations
Maps	Country ⊃ state ⊃ city ⊃ zip code	Geographically close places
News	Category ⊃ sub category ⊃ article	Articles of related topics
Yellow Pages	Category ⊃ brand name ⊃ retailer	Geographically close businesses
White Pages	State ⊃ city ⊃ phone	People with similar names
Digital Libraries	Subject ⊃ journal	Papers in related areas
Search Engines	General keyword ⊃ specific keyword	Documents with similar keywords

Table 1: query affinity. The "containment" column shows the containment chains in the databases. The "ranging" column shows the "close" items in the databases.

By selecting the server that currently has in its cache the most data for the query, we can effectively distribute query and data across servers with reduced sharing.

This strategy benefits only those applications that exhibit query affinity, and the quantitative reduction in data sharing is a function of the *dimensions of query affinity*, which we define as the average number of separate groups a query is affined to. The ideal number of dimensions is 1. Larger dimensions result in more partitioning conflicts and hence less reduction in data sharing. In range searches, the conflicts exist in the situations where a data item is "close" to more than one separate set of data items. In the containment case, the conflicts may arise if the containment relation is not strict, i.e. the data accessed by a query spreads across the data sets of more than one containing queries. For example, the books written by Isaac Asimov belong in many different subjects.

Based on the observation above, we have designed an *affinity-based distribution* (*ABD*) system for a cluster of cache servers. The basic method in ABD is to divide the execution of each query into two stages and to use the result from the (local) first-stage execution to determine the (remote) destination of the second-stage execution. The first stage is computation-intensive; the function and data needed for the first-stage execution are replicated on each machine that hosts application servers. The second stage is data-intensive; data accessed during the second-stage execution is partitioned across cache servers. The first stage determines the set of data that the query will likely access, which is then used to determine the destination of the query in its second stage. The second stage completes the query execution and generates results. The intention of the two-stage execution is to determine the destination of each query with the knowledge of the data to be accessed.

The implementation of ABD is system-dependent, i.e. it varies for different types of master databases, such as relational databases, object-oriented databases, or file systems. The implementation details are outside the scope of this paper.

6 Simulations

In this section, we present the evaluation of ABD in comparison to alternative distribution strategies that handle the conflicts to different degrees. With a set of simulations, we study the impact of the following factors on the performance of various distribution strategies: applications, access patterns, human assistance, dimensions of affinity, memory sizes and cooperative caching. This study is analogous to the study on request distribution strategies for a cluster of web servers [11].

6.1 Setup

We use a modified version of the cluster simulator previously used for distribution strategies for a cluster of web server [11]. The original simulator models the scheduling of CPU queues, disk queues and incoming request queues as well as activities on the main memory cache in the server machines. It assumes that the entire data set is replicated on the local disks of all server machines and a subset of data is cached in the main memory cache of server machines where it is frequently accessed. A request is processed in the following steps: connection setup, disk reads (if needed), target data transmission, and connection teardown. Parameters such as memory size, CPU speed, disk speed, network speed and caching protocol are configurable. A detailed description of the original simulator can be found in [11].

The major modifications we make to the original simulator in order for the simulator to work for application servers and databases rather than web servers and file systems are following. For write-shared data, we model a multi-reader-single-writer locking protocol for cache consistency. Each lock-related operation is charged for a round-trip network latency. In addition to the steps of connection setup and teardown for each query, each accessed data item in the query is processed in the following steps: lock acquisition (if needed), disk reads (if needed), data processing or transmission, and writes (if needed). For written data, we assume an asynchronous cache write-through policy; that is, we charge the CPU overhead for sending data to disk, but not disk write time. Therefore, written data is immediately visible to successive reads. In fact, we do not charge the delay caused by synchronization in any lock-related operations. These assumptions are conservative with regard to the benefits of ABD because they lower the performance penalty of the events that ABD is designed to reduce, i.e. shared writes.

Table 2 shows the parameter settings common in all simulations reported in this paper.

Parameters	Values
Connection setup and teardown time per HTTP request	750
Initialization overhead per request to an application server	1 ms
Data processing or transmission time per 512 bytes	85
Disk transfer time per 512 bytes	50
Average disk seek time	10 ms
CPU overhead per 512 bytes accessed on disk	4
Memory page size	1 KB
Replication of hot data allowed	Yes
Cache replacement policy	LRU

Table 2: Common parameter setting. Data is cached at row page granularity, whichever is larger.

We compare the following five distribution strategies in each experiment:

1. **Weighted round-robin distribution (WRRD)**: Queries are distributed to cache servers in a round-robin fashion, weighted by the servers' loads. This is analogous to a front-end, connection-based distribution strategy for a cluster of web servers [4].

2. **Query-based distribution (QBD)**: Identical queries are directed to the same server, but distinct queries that return the same or overlapped data will not necessarily be directed to the same server. In practice, this strategy cannot make a good decision for queries that consist operations unrelated to selection conditions, such as an "ordered by" operation. However, such operations are not present in the simulations, which benefits this strategy. This strategy is analogous to a content-based distribution strategy for a cluster of web servers [11].

3. **Human-assisted QBD or enhanced QBD (EQBD)**: Domain-specific information is added to QBD by application programmers or system administrators. The additional information helps QBD select the most relevant attributes in queries for decision making. Therefore, queries on the same data by the same selected attributes will be directed to the same server. However, queries on the same data by different selected attributes will not necessarily be directed to the same server.

4. **Affinity-based distribution (ABD)**: Queries on the same or overlapped data, i.e. affined queries, will be directed to the same server regardless of the attributes or operations in the queries.

5. **An ideal case (Ideal)**: This is a multi-processor machine with hardware shared memory of the same capacity as its cluster counterpart. No data replication or transmission is needed since all processors have access to all data in memory. This is viewed as an ideal case for comparison purpose.

We choose two example applications for case studies, white pages [8] and auctions [5]. These two examples exhibit the two sources of query affinity We observe, i.e. ranging and containment, respectively.

All five distribution strategies achieve comparable and good load balance in the simulations, which confirm the previous results [11], and hence load balance is not discussed in detail below.

6.2 Case study 1: White pages

We extract the traces on a university white page service [8] from the access logs of the university web server in the year 1999, which include 809194 queries by names and 181719 queries by phone numbers, email addresses and/or departments. 23442 distinct names were queried in total. Since the results of the experiments depend on the actual data accessed by the queries as well as the queries themselves, and since the actual data is not contained in the access logs, we resend the queries on the distinct names to the web server that generated the traces, and store the returned data from the web server for use in the simulations. For queries that contain multiple names, We use the intersection of the results for each name as the results for the query, which is what We observe the original CGI program does. To avoid resending an excessively large number of requests to the web server, we exclude the queries by phone numbers, email addresses and/or departments. Due to the small size of the university, only 19534 valid, distinct people's data is accessed in total. Since We am interested in studying a large data set that does not fit in the memory of a single node, we augment the actual data set by a factor of 8. Assuming each person's data takes 1024 bytes, the augmented data set is less than 153 MB. We set the

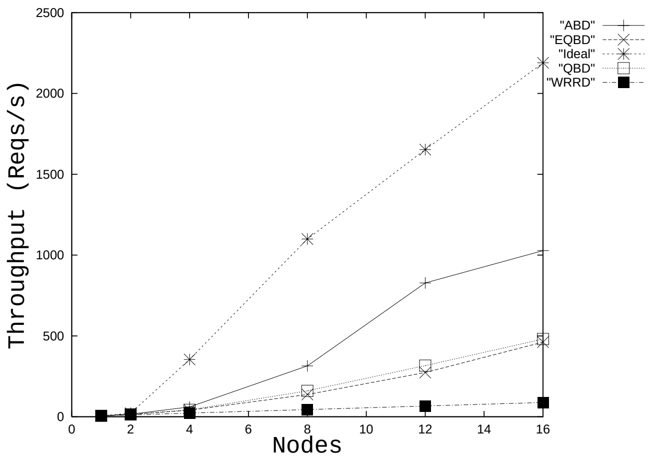

Figure 2: throughput of the white page traces

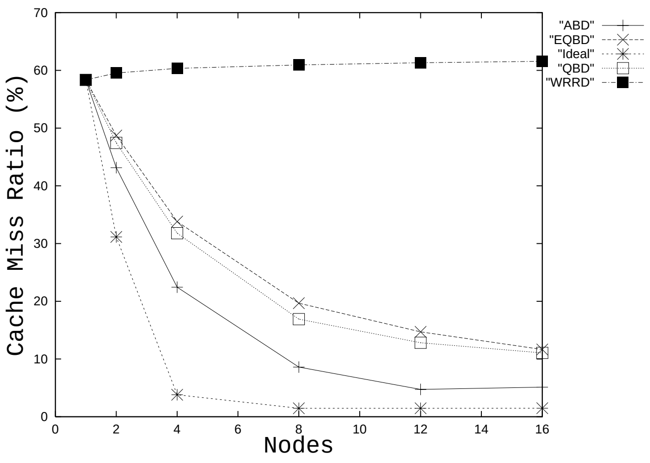

Figure 3: cache miss ratio (the number of cache misses divided by total accesses) of the white page traces

memory size of individual nodes to 16 MB to 32 MB in our experiments.

We run the five simulated distribution strategies with the white page traces. QBD and EQBD direct each query to a node based on the whole query and the first queried name, respectively. Figure 2 shows the throughput of the five distribution strategies on 1 through 16 nodes with 24 MB memory per node. Figure 3 shows the cache miss ratio, which explains the differences in throughput. (All ratios shown in this paper are the absolute number of events, e.g. cache misses, divided by the total number of data accesses in the simulation.) The results of WRRD match those for static content in a cluster of web servers: the cache miss ratio does not decrease as the cluster size increases, because the most accessed data tends to be replicated in all nodes. In QBD and EQBD, queries on the same names are directed to the same node; therefore, QBD and EQBD achieve locality to a certain degree, reduce cache miss ratio and improve throughput by a factor of 5 over WRRD with 16 nodes. QBD performs slightly better than EQBD because EQBD, using a single name in each query for distribution, experiences a small degree of load imbalance. ABD reduces cache miss ratio and improves throughput by a factor of 2 over QBD and EQBD and 9 over WRRD.

We also run the simulations with 16 MB and 32 MB memory per node, respectively. The results show that, as the memory to data ratio increases, the performance difference among the different distribution strategies decreases, which matches the results observed for static content in a cluster of web servers [11].

To summarize, read-only dynamic content behave largely in the same way as static content in a cluster of web servers under various distribution strategies except for the fact that there tends to be more data sharing for dynamic content; ABD reduces data sharing and improves throughput and scalability by directing queries on similar names, i.e. affined queries, to the same node.

6.3 Case study 2: Auctions

We downloaded the bid history of the first 50 completed items in 3212 categories from the well-known auction site eBay [5]. We extracted the following events with time stamps and relevant parameters from the bid histories: 117623 *SellItem* events (with the seller, category and item parameters), 231521 *BidItem* events (with the bidder and item parameters) and 117623 *CompleteItem* events (with the item parameter).

Since these events are only a subset of the actual events at eBay and represent only write accesses to the database, we synthetically add other events to the traces based on expected user behaviors. For each *SellItem*(seller, category, item) event, we generate a configurable number of BrowseCategor(category) events within half an hour before, *ViewItemsBySeller*(seller) events within half an hour after, *ViewItem* (item) events within half an hour after, and *ReviseItem*(item) events within two days after. For each *BidItem*(bidder, item) event, we generate a configurable number of *ViewItemsByCategory*(item.category) events within half an hour before, *ViewItem*(item) events within 15 minutes before, *ViewBidsByItem*(item) events within 5 minutes before and after, and *ViewBidsByBidder*(bidder) events within 5 minutes after. To each generated event, we assign a time stamp that is randomly chosen within the given time range. We sort all generated events together with the original events in ascending order of time stamps and use them as the input to the simulator.

In the traces used in the simulations, there are 698288 *ViewItemsByCategory* events, 698288 *ViewItem*

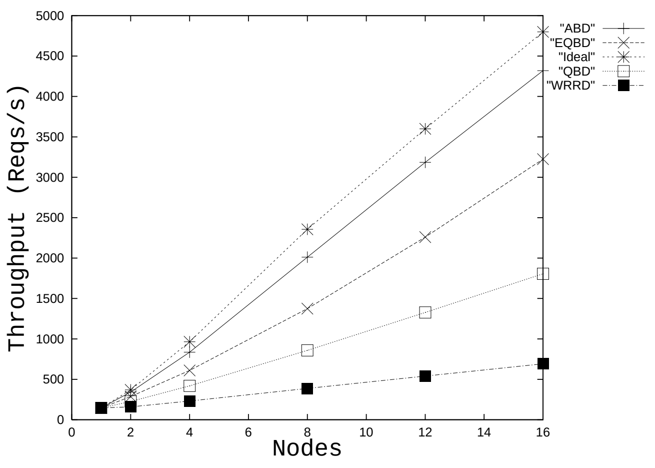

Figure 4: throughput of the auction traces

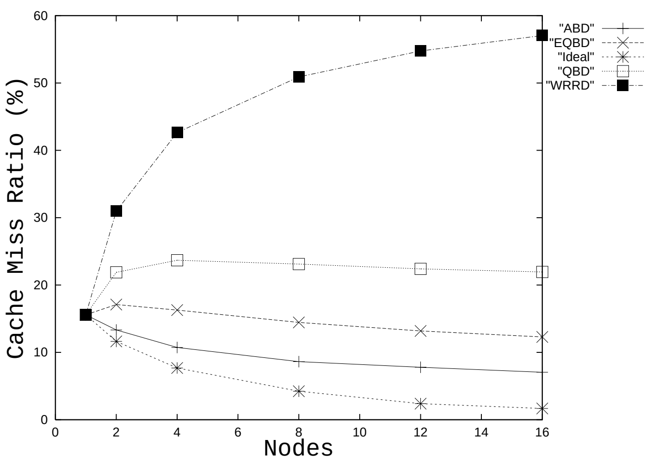

Figure 5: cache miss ratio (the number of cache misses divided by total accesses) of the auction traces

events, 117624 *CompleteItem* events, 231521 *BidItem* events, 115769 *ViewBidsByBidder* events, 463042 *ViewBidsByItem* events, 117623 *SellItem* events, 58729 *ViewItemsBySeller* events, and 11917 *ReviseItem* events. There are 2512801 events in total and 81% of them are reads.

The data set includes 117623 items in 3212 categories, offered by 42536 distinct sellers, and 231521 bids, made by 167752 distinct bidders. We assume that the size of each item is 4 KB and the size each of bid is 128 bytes. For simplicity, data items smaller than a page are padded to a page in cache (1 KB in the simulations). The memory footprint of all items and bids is roughly 686 MB. The memory space needed for the replicated keys, e.g. item ids and bidders, is roughly 21 MB per node. We choose 64 MB as the memory size per node so that the entire data set can fit in the memory in the best case in the simulations, i.e. a machine with 16 processors, 1 GB hardware shared memory and no replication.

We expect that the actual memory and data sizes at eBay are much larger than the sizes in the simulations, but the relative data to memory ratio in the simulations is reasonably realistic. It is reported that there are roughly 4 million items on sale at eBay everyday, meaning that it will take 16 GB to cache the items.

Figure 4 shows the throughput of the five distribution strategies on 1 through 16 nodes. Like in the white page case, the throughput is determined by the cache miss ratio, shown in Figure 5. Unlike the white page case, which has a read-only access pattern, the cache misses are caused both by memory pressure and by synchronization in this case. Figure 6 shows the cache eviction ratio as a measure of the memory pressure. Figure 7 shows the cache invalidation ratio as a measure of synchronization cost. The cache eviction ratio decreases as the cluster size increases except in WRRD, because the effective cache size increases with the cluster size in the other four cases. The cache invalidation ratio slightly increases with the cluster size in all but the ideal case because the number of nodes that write-share data increases. As a result, the overall cache miss ratio decreases at a slower speed than the cache eviction ratio except in the ideal case. ABD, EQBD and QBD achieve 90%, 67% and 38% of the ideal throughput with 16 nodes, respectively.

The throughput curves also suggest that, in order to achieve the same throughputs as WRRD, QBD and EQBD with 16 nodes, ABD requires only 4, 10 and 12 nodes, respectively.

It is known for static or read-only dynamic content that increasing memory size reduces the difference in distribution strategies (Section 5.6.2). We rerun the auction simulations with the memory size increased to 128 MB per node. The results show that there are no longer cache evictions in ABD, EQBD and QBD with 16 nodes, but the increased memory size does not help reduce cache invalidations. From 12 to 16 nodes, the cache eviction ratio is reduced from 5%, 10% and 17% to 0% while the overall throughput is improved by only 1%, 5% and 2% for ABD, EQBD and QBD, respectively.

To summarize, ABD improves the throughput and scalability of write-shared dynamic content because it reduces synchronization cost as well as memory pressure for write-shared data. The difference across distribution strategies for write-shared data is not as sensitive to memory size as that for static or read-only data because synchronization cost does not decrease as memory size increases.

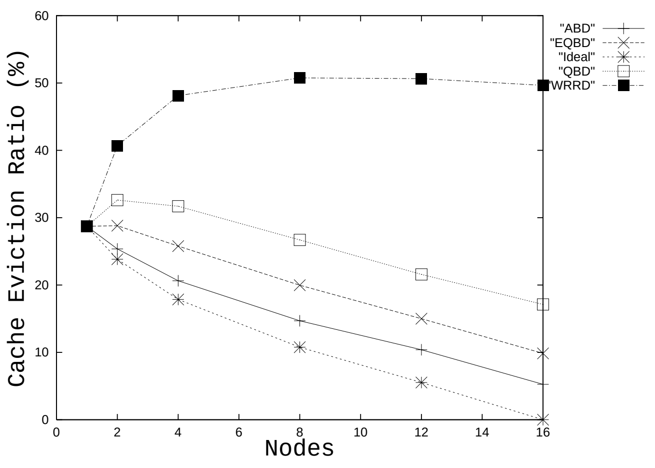

Figure 6: cache eviction ratio (the number of old data items evicted from cache to make room for new items, divided by total accesses) of the auction traces

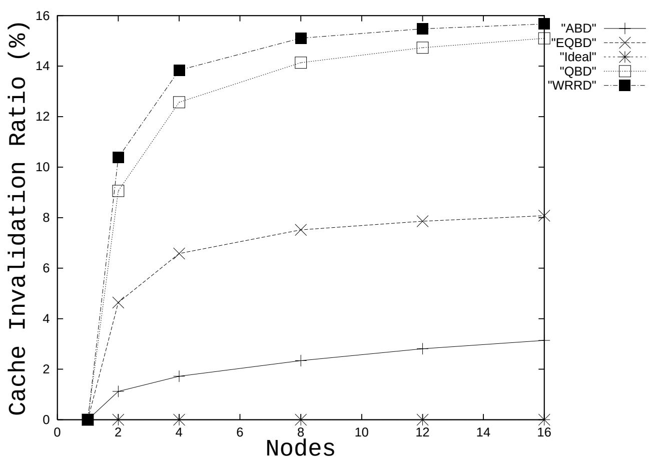

Figure 7: cache invalidation ratio (the number of data items removed from cache because the node no longer holds the read or write lock, divided by total accesses) of the auction traces

6.4 Dimensions of query affinity

The statistics on the white page traces show that on average each person's record is accessed by 6.8 distinct queries in EQBD and by 2.4 distinct queries in ABD. As a result, ABD reduces the average number of replicas per record from 6.8 to 2.4. The number 2.4 is in fact the dimensions of query affinity in this application. It results from the fact that each person has 2 to 3 names, i.e. the first name, the last name and probably the middle name, and queries with any of the names will access the person's record.

In the auction application, there also exists dimensions of query affinity that are larger than 1. For example, the items offered by the same seller could fall into more than one category. We examine the cache miss ratio for three queries, *ViewItem*, *ViewItemsBySeller* and *ViewBidsByBidder*, separately in ABD to study the impact of the multi-dimensional affinity on cache miss ratio. Table 3 shows the average and maximum numbers of containing queries and the cache miss ratio of the three queries with 16 nodes. The numbers of containing queries shown in the table are statistical results from the traces. The table shows that larger numbers of containing queries result in higher cache miss ratio.

Queries	Containing Queries (CQs)	Ave. Num. CQs	Max. Num. CQs	Cache Miss Ratio
ViewItem	ViewItems By-Category	1	1	11.10%
ViewItems BySeller	ViewItems By-Category	2.77	197	35.60%
ViewBids ByBidder	ViewBids ByItem	1.38	32	21.90%

Table 3: Impact of dimensions of query affinity on cache miss ratio. The third and fourth columns from the left show the average and maximun numbers of containing queries per query, respectively.

6.5 Cooperative caching

We study the impact of cooperative caching [3] [1] in both the white page case and the auction case. With "pull-based" cooperative caching, data can be transferred from a server's cache to another rather than be loaded from disks. The lock manager in the consistency protocol provides the locations of cached data for cooperative caching. In the simulator, each cache-to-cache data transfer is charged 0.5 ms network latency and 6 MB/s network transfer time. Without cooperative caching, each cache miss is charged 10 ms disk seek time and 9.8 MB/s disk transfer time.

Figure 8 and Figure 9 show the throughput of the white page traces and auction traces with cooperative caching, respectively. In the white page traces, the throughputs of QBD, EQBD and ABD are all significantly improved. In the auction traces, ABD is not improved much because it is already close to the ideal case without cooperative caching. Cooperative caching improves performance for two reasons: it replaces disk accesses with network accesses, and it helps make better cache replacement decisions due to the availability of global reference information. However, it is complementary to ABD in that it does not reduce data replication or write sharing. In other words, it does not reduce the

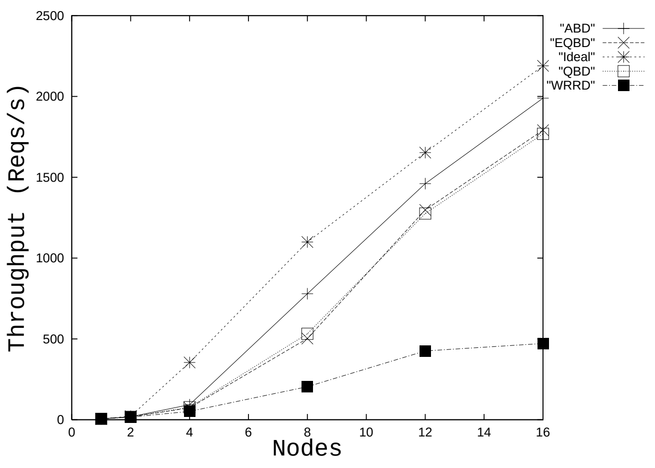

Figure 8: throughput of the white page traces with cooperative caching

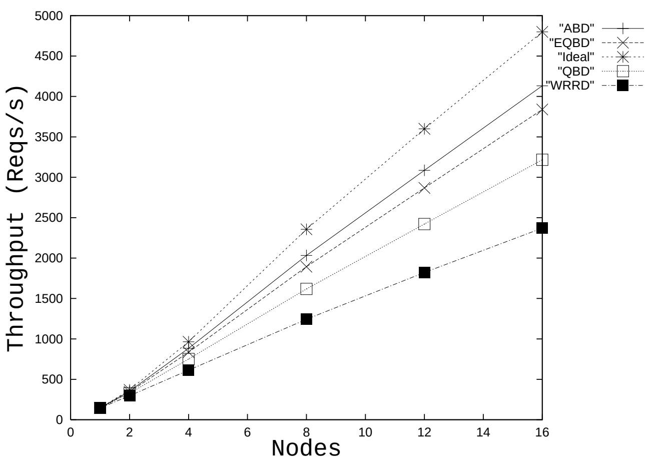

Figure 9: throughput of the auction traces with cooperative caching

number of costly events, but lowers their cost.

QBD benefits the most from cooperative caching and achieves 86% and 78% of the throughput of ABD with 16 nodes in the white page and auction cases, respectively. However, the results are based on conservative assumptions about the cost of cooperative caching and on the use of simplified queries in requests. In the simulator, we assume infinite network capacity, no synchronization delay and no bottleneck at the centralized lock manager for cooperative caching.

We have also studied another cooperative caching protocol, in which each data item is cached and accessed in a fixed node. It makes efficient use of cache space and avoids synchronization across nodes for write-shared data. However, it outperforms pull-based cooperative caching only under high memory pressure, e.g. with 1 to 4 nodes in the simulations, and suffers from lower local cache hit ratio than pull-based protocol as memory size increases.

Cooperative caching improves QBD and EQBD relative to ABD. However, in reality, the closing in the gap would likely be much smaller. Besides, many systems do not incorporate cooperative caching, while ABD is a general and easy-to-deploy strategy whether or not cooperative caching is present.

7 Related work

An alternative to caching raw data is to cache query results in web or proxy servers. Result caching has shown to reduce CPU cost and disk access involved in generating pages that do not change frequently and do not cause updates to the underlying database [10]. Application-specific annotation is usually needed for managing cached results due to the diversity in contents and operations [13]. Result caching has the following limitations: 1) Certain queries modify data and hence cannot be cached. 2) Certain applications need to be rewritten to explicitly keep cached results up to date. 3) There are often overlaps across different query results and cache space can be wasted on duplicate information.

Content-based request distribution strategies [11] [14] have been developed to effectively improve the performance and scalability of a cluster of web servers with static content. In the HACC project [15], there is an extension to content-based distribution for dynamic requests in *Lotus Domino*. It makes decision based on requested objects and actions in Lotus Domino, but does not address distribution of dynamic content in general. ABD extends the existing work by postponing the decision on query distribution till after the queries are processed and by exploiting natural affinity in a wide range of Internet applications.

Our main contributions are identifying the causes for data sharing in dynamic content servers, observing query affinity in a wide range of Internet applications, and reducing the sharing by exploiting query affinity.

8 Conclusion

We have studied data/query distribution strategies in cluster-based Internet application servers. We proposed a distribution strategy that exploits natural query affinity in content structures and access patterns of many popular Internet services. Such affinity is independent of the physical storage of data.

With trace-based simulations, we compare ABD to weighted round-robin distribution, basic query-based distribution, enhanced query-based distribution and an ideal case. The results show that ABD reduces cache miss ratio by a factor of 1.7 to 9 over alternative strategies, and that ABD outperforms alternative strategies by a factor of 1.3 to 9 and achieves up to 90comparison indicates that ABD requires only $\frac{1}{4}$ to $\frac{3}{4}$ of the resources in order to achieve the same throughput as the alternative strategies. The results also indicate that applications with dynamic content, especially write-shared content, have a higher demand for good distribution strategies than applications with static content, due to the tendency for more data sharing and the synchronization cost that cannot be reduced by simply increasing memory size.

References

[1] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, November 1994.

[2] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, Vol. 9 No.4, December 1984.

[3] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server dbms architectures. In *Proceedings of the 18th VLDB Conference*, August 1992.

[4] http://www.cisco.com. Cisco systems localdirector.

[5] http://www.ebay.com.

[6] http://www.exodus.com.

[7] http://www.google.com.

[8] http://www.princeton.edu/Siteware/puphf.shtml. University campus directory.

[9] http://www.yahoo.com.

[10] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems*, December 1997.

[11] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[12] O. Shmueli and A. Itai. Maintenance of views. *ACM SIGMOD Record*, Vol. 14 No. 2, 1984.

[13] B. Smith, A. Acharya, T. Yang, and H. Zhu. Exploiting result equivalence in caching dynamic web content. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[14] C. Yang and M. Luo. Efficient support for content-based routing in web server clusters. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[15] X. Zhang, M. Barientos, J. B. Chen, and M. Seltzer. Hacc: An architecture for cluster-based web servers. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.