MIS Quarterly

RESEARCH ARTICLE

A COST-BASED DATABASE REQUEST DISTRIBUTION TECHNIQUE FOR ONLINE E-COMMERCE APPLICATIONS[1]

Debra VanderMeer
College of Business, Florida International University,
Miami, FL 33199 U.S.A. {Debra.Vandermeer@fiu.edu}

Kaushik Dutta and Anindya Datta
Department of Information Systems, National University of Singapore,
SINGAPORE {duttak@nus.edu.sg} {datta@comp.nus.edu.sg}

E-commerce is growing to represent an increasing share of overall sales revenue, and online sales are expected to continue growing for the foreseeable future. This growth translates into increased activity on the supporting infrastructure, leading to a corresponding need to scale the infrastructure. This is difficult in an era of shrinking budgets and increasing functional requirements. Increasingly, IT managers are turning to virtualized cloud providers, drawn by the pay-for-use business model. As cloud computing becomes more popular, it is important for data center managers to accomplish more with fewer dollars (i.e., to increase the utilization of existing resources). Advanced request distribution techniques can help ensure both high utilization and smart request distribution, where requests are sent to the service resources best able to handle them. While such request distribution techniques have been applied to the web and application layers of the traditional online application architecture, request distribution techniques for the data layer have focused primarily on online transaction processing scenarios. However, online applications often have a significant read-intensive workload, where read operations constitute a significant percentage of workloads (up to 95 percent or higher). In this paper, we propose a cost-based database request distribution (C-DBRD) strategy, a policy to distribute requests, across a cluster of commercial, off-the-shelf databases, and discuss its implementation. We first develop the intuition behind our approach, and describe a high-level architecture for database request distribution. We then develop a theoretical model for database load computation, which we use to design a method for database request distribution and build a software implementation. Finally, following a design science methodology, we evaluate our artifacts through experimental evaluation. Our experiments, in the lab and in production-scale systems, show significant improvement of database layer resource utilization, demonstrating up to a 45 percent improvement over existing request distribution techniques.

Keywords: Database clusters, request distribution, task allocation, design research

Introduction

Since the advent of Internet-enabled e-commerce, online sales have attracted an increasing share of overall sales revenues. Online retail sales have grown from $155.2 billion in 2009 to $172.9 billion in 2010, with an expected 10 percent compound annual growth rate, projected to reach nearly $250 billion in 2014 (Mulpuru et al. 2010).

This growth translates into significant increases in online activity, which can be expected to result in a corresponding growth of activity on the underlying information technology

[1] Al Hever was the accepting senior editor for this paper. Samir Chatterjee served as the associate editor.

infrastructure. Supporting such growth organically (i.e., acquiring the necessary IT resources to support this commensurate growth in infrastructure) is challenging in the current era of austerity. Yet IT managers are expected to support this growth while continuing to improve the user experience with new features *and* with smaller budgets (Perez 2009).

Facing such pressures, the low up-front investment and pay-for-use pricing offered by cloud computing is enticing (Roudebush 2010). Here, *IT managers deploy their applications to virtualized platforms provided by third-party infrastructure companies or, potentially, an internal cloud infrastructure provider*. Each application is deployed to a set of virtualized servers on the cloud infrastructure, where the number of virtual servers allocated to the application varies with the application's workload: servers can be allocated as workload increases, and deallocated as workload decreases.

Gartner Research (2010) reports that the use of cloud computing is not only growing (with worldwide cloud services revenues of approximately $68 billion in 2010), but *the rate of growth of cloud service deployments is increasing*. As virtualization and cloud computing become more popular, data center managers and IT managers alike must accomplish more with fewer resources and support more applications with fewer dollars. Advanced request distribution techniques can help ensure both high utilization (Melerud 2010), to make sure that the capacity of existing resources is fully utilized before adding more resources, and smart request distribution, where requests are sent to the resources best able to service them.

The use of such request distribution techniques provides some major advantages that appeal to both data center managers as well as IT managers: (1) using existing resources in an optimal manner and (2) accomplishing more with fewer resources reduces operational costs, leading to lower costs of ownership for data center managers, allowing them to offer lower, more competitive costs to their customers (IT managers).

In this paper, we explore request distribution techniques for online e-commerce applications *focusing on the data layer*. In this context, we first describe a typical online application architecture, and then discuss request distribution needs within this architecture.

Online applications are typically organized in a three-tier architecture, as depicted in Figure 1.

For most online businesses, even for small- to medium-sized organizations, a single instance at each layer will not suffice, since each of these layers can experience workloads beyond the capacity of a single server or software resource. The best practice described by vendors across all three layers of the typical online application architecture is to *cluster* multiple identically configured instances of hardware and software resources at each layer, as depicted in Figure 2 (Cherkasova and Karlsson 2001; Schroeder et al. 2000).

As shown in Figure 2, there are three significant request distribution (RD) points. First, the web switch must distribute incoming requests across a cluster of web servers for HTTP processing. Second, these requests must be distributed across the application server cluster for the execution of application logic. Subsequently, a set of database requests emanate from the application server cluster that need to be distributed across the database cluster.

Let us now consider the different tiers shown in Figure 2 in the context of request distribution.

The objective of virtually every request distribution method in the first layer, the web layer, is load balancing (Cardellini et al. 2001). In fact, web layer load balancers, or web switches, constitute one of the largest and most successful market segments in the internet equipment space (think of Cisco, Juniper, and F5).

The next layer, the application layer, consists of a cluster of application servers. Request distribution in the application layers is a relatively new area of research (see Dutta et al. 2007).

The third layer, the data layer—the layer of interest in this paper—consists of a cluster of database servers. Existing work in this area focuses almost entirely on online transaction processing (OLTP) systems. However, there is an interesting feature of online multitiered applications, our target application area, that sets it apart from a general purpose OLTP system: application workloads in online application systems tend to be read-intensive. On on average, 80 to 95 percent of online database workloads consist of read requests. As we will discuss, and demonstrate, existing transaction routing strategies, designed for OLTP systems, while highly effective for update requests, do not perform well in distributing requests in these read-mostly scenarios. In fact, in certain substantial application domains, such as retail e-commerce, the read requests (roughly corresponding to browsing, while writes would approximately correspond to buying) could be as much as 95 percent of the total request workload (Gurley 2000).

In such scenarios, replication across cluster instances is the primary strategy for data availability (Rahm and Marek 1995). Indeed, best practices documentation from major database vendors supports the use of replication for read-heavy scenar-

Figure 1. Typical Multitiered Architecture for Online Applications

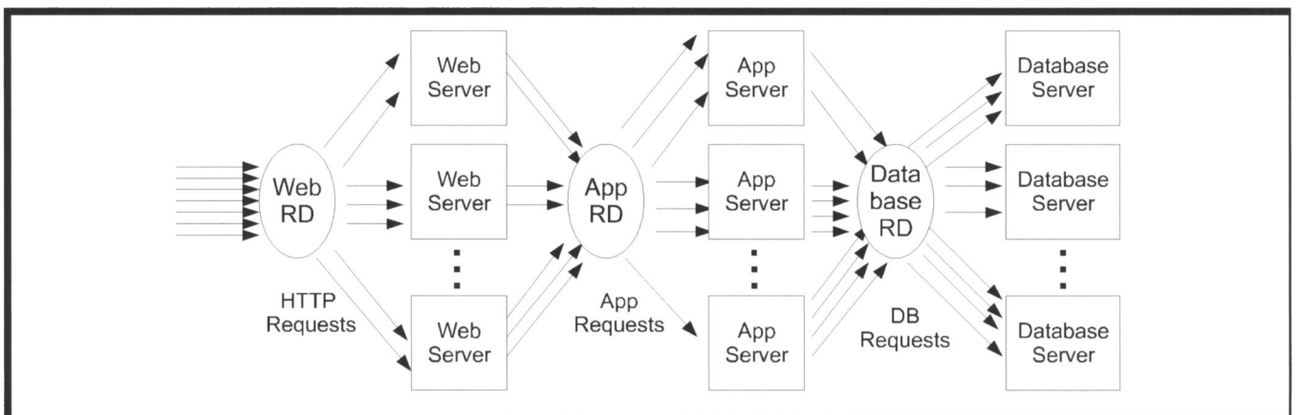

Figure 2. Clustered Multitiered Online Application Architecture

ios (Microsoft Inc. 2008). For instance, an e-commerce company might replicate product catalog data across multiple database servers, or a business analysis service such as Hoover's might replicate public-corporation financial data. This allows each database server to service requests for any data, while simultaneously removing the scalability limits of a single database server.

Current request distribution techniques for replicated database scenarios borrow heavily from techniques developed for the web and application layers. However, these techniques do not consider the fact that database requests can generate widely varying loads; they assume that each incremental request adds the same amount of load to a server. This is not the case for database requests, since two requests can impose very different processing requirements on a database. Thus, any request distribution technique for this layer must take into account the effect of varying processing loads across requests on overall database loads. Ideally, such a request distribution technique would route a request to a suitable database instance that can process it with the least amount of work. When applied across all requests, such a technique would be expected to reduce overall workloads across all instances, resulting in improved scalability.

The scale of workloads incident on online applications, where the arrival rate of database requests may be on the order of hundreds per second, adds to the challenge of request distribution on the data layer. Given the dynamic nature of database workloads noted above and the rates of request arrival, any request distribution technique must be lightweight, imposing little additional overhead in making request distribution decisions; such a technique should provide scalability benefits that are far greater than the cost of making distribution decisions.

In this paper, we propose a smart request distribution strategy designed to improve the scalability of the data layer in online applications by routing a request to a database instance that can process it with a minimal amount of work (see Figure 3), as compared to other instances in the cluster based on database workloads on each instance.

In this context, we address the following research questions in this work:

1. How can we model database workloads in online multitiered applications?

2. Based on this model, how can we design an effective request distribution mechanism for the data layer in online multi-tier applications?

3. How can we demonstrate the efficacy and utility of our mechanisms over existing mechanisms?

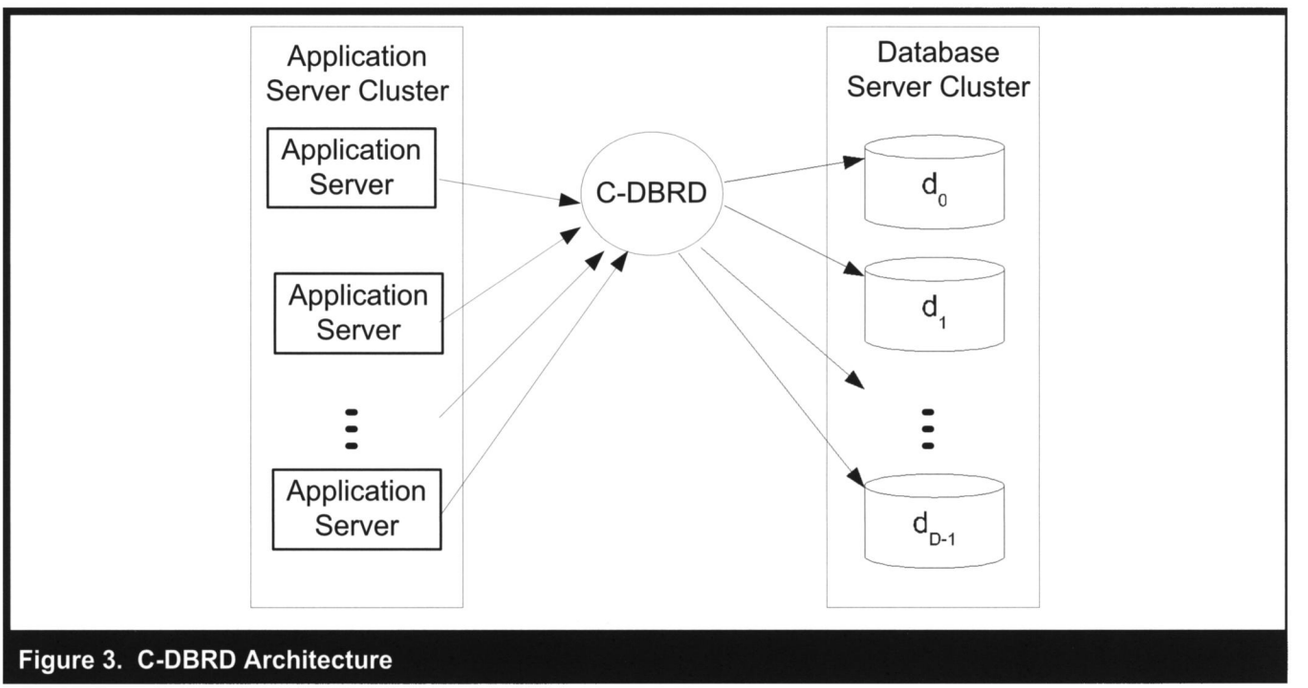

Figure 3. C-DBRD Architecture

Our method attempts to take advantage of data structures cached on the database (such caching is standard in industry, and is implemented by virtually every major database vendor) by routing each incoming database request to the database instance most likely to have useful cached data for the request.

We have implemented our strategy and evaluated it extensively experimentally. The results, reported in this paper, have been very encouraging, with improvements of up to 45 percent over existing distribution mechanisms in overall response time. In the next section, we provide a detailed overview of why existing request distribution strategies are inadequate.

Our work falls into the category of design science research (Hevner et al. 2004). In this vein, we create a set of artifacts, in this case describing a request distribution mechanism for data layers, aimed at improving IT practice. Specifically, we (1) propose a *theoretical model* of database workload that takes into account the effects of caching; (2) define a workable *method* for utilizing the model in practice; and (3) develop an implemented software *instantiation*, suitable for experimental evaluation. Finally, (4) we present the *results of a set of analytical studies, simulation experiments, and field experiments* to show the practical utility of our method, and test its properties under various operating conditions.

The remainder of this paper is organized as follows. We consider related work, and describe the managerial benefits of our method in the following section. Next, we provide an overview of, and describe the technical details of, our approach. We then evaluate the performance of our proposed approach both analytically and experimentally by comparing it with that of existing approaches designed for the web and application layers. To further illustrate the efficacy of our approach, we present a brief field experiment. In this field experiment, we compare the performance of our approach to that of an off-the-shelf database clustering solution in the context of a mid-size e-commerce vendor's application infrastructure. Finally, we discuss the practical benefits and potential risks of adopting our scheme from an IT manager's perspective, and conclude our paper.

Related Work

The heart of our strategy is a novel request distribution mechanism across the cluster of databases that comprise the data layer of a typical multitiered software application. In this section, we first consider the problem of request distribution in the data layer in the context of the broad research literature. We then discuss existing distribution strategies, and consider their utility in the data layer of multitiered online applications.

Why is RD relevant for the design science community? Design science researchers can bring to bear hitherto unemployed solution techniques in the RD problem space. The design science community has a successful track record in applying optimization techniques to complex dynamic problems in general, and to systems problems in particular. Examples of such problems include database processing (Bala and Martin 1997; Gopal et al. 1995, 2001; Krishnan et al. 2001; Segev and Fang 1991), networking (Kennington and Whitler 1999; Laguna 1998, and caching (Datta et al. 2003; Dutta et al. 2006; Hosanagar et al. 2005; Mookerjee and Tan 2002). *RD falls into the same class of problems; it is an optimization problem for which classical optimization techniques have not yet been applied.* The problem is rendered even richer due to the fact that a direct application of extant techniques is not enough; innovation is required in both modeling the RD problem, as well as in solving the models. We elaborate below.

Request distribution specifically falls into the area of *task assignment* (Mazzola and Neebe 1986). Here, the goal is to optimize the servicing of a given workload by appropriate assignment of tasks to cluster instances, virtually identical to the high-level goal of the RD problem. This problem has been addressed generally in the optimization community (Amini and Racer 1995; Haddadi and Ouzia 2004; Mazzola and Neebe 1986). This body of work proposes generic approaches to the generalized assignment problem, where an objective function describes the optimization goal, and a set of constraints describe the decision-making space. In many problem scenarios, we can leverage these general techniques to develop an optimal solution if we can model the problem domain appropriately. *The trouble is that these general optimization techniques assume problem characteristics that make it difficult to apply existing work directly in the case of request distribution.*

Traditional task assignment optimization techniques assume a static decision-making problem (i.e., given a set of tasks and resources, traditional approaches will make a single decision that allocates all tasks to specific resources). Optimization problems that arise in the RD scenario, however, are dynamic in nature. Here, the RD method must make a separate resource allocation decision for each incoming task (request), and each allocation decision modifies the workload of the resource assigned to the task. Thus, the RD solution approach cannot operate over a static problem frame; rather, it must operate over a dynamic problem frame. We note that some work has been published in the recent literature in dynamic task allocation problems (Spivey and Powell 2004); however, these problems assume that changes in the problem framework occur very slowly, on the order of a few times an hour. In contrast, an effective RD scheme must respond in real time to each request in a high request-rate scenario (potentially hundreds of requests per second), where each request changes the RD decision framework.

Approaching RD as a variant of the dynamic scheduling problem, techniques from the scheduling field (e.g., Colajanni et al. 1997; Menasce et al. 1995) might appear to be applicable here. While this is true at a high level, a straightforward application is difficult. Virtually all dynamic scheduling techniques (Tanenbaum 2001) presuppose some knowledge of either the task (e.g., duration, weight) or the resource (queue sizes, service times), or both. This assumption does not work in our case, because both the tasks and the resources are highly dynamic. Moreover, resources in our case are black boxes, providing only as much access as is allowed by query languages and published APIs. There are also classical studies of dynamic load sharing approaches (e.g., Chow and Kohler 1979; Wang and Morris 1985). These consider incoming work as purely computational from a CPU-intensive perspective. In contrast, database workloads are not only CPU-intensive; they are also memory- and I/O-intensive. This makes a straightforward application of these techniques in the database case impossible.

We next discuss the relationship of our problem to those tackled in the extensive literature on load balancing in distributed and parallel databases. There are two broad themes in this work. The first theme deals with load sharing in distributed databases. A fair amount of this work is not applicable to our problem domain, as most research in this area considers partitioned databases (recall that our focus here is on online applications using replicated database clusters). Rather than citing multiple papers, we refer the reader to the paper by Yu and Leff (1991), which provides an excellent summary of this work. In the work on distributing load across replicated databases, most work has concentrated on online transaction processing (OLTP) issues, for example, efficient synchronization maintenance and distributed concurrency control (e.g., lock control) issues (Yu and Leff 1991).

Interesting related work on request distribution in the data layer appears in Amza et al. (2003) and Zuikevičiūtė and Pedone (2008). Amza et al. suggest a request scheduling strategy for databases behind dynamic content applications, while Zuikevičiūtė and Pedone propose a generic load balancing technique for transactions to prevent lock contention. These authors also consider the distribution of write requests in replicated database clusters.

This work is complementary to our work in the sense that we assume that the write workload is handled using existing

schemes. Our focus here is the substantial read workload incident on online application databases. Although we cannot directly apply the work described above, much of this work shares a common idea: the exploitation (or avoidance) of reference locality to identify instances better suited to processing certain requests. In the context of this work, the term *reference locality* refers to the fact that different queries run on different database instances in the cluster will result in different data residing in memory in each instance. Here, a query might run faster on instance A as opposed to another instance B, if the data needed for the query already resides in memory on instance A, but not B. We carry this thread forward in our work.

We now move on to consider related work in the area of clustering. Clustering has been viewed and discussed in the context of all three layers, with specific solutions having been identified at each layer.

Clustering in the web layer is discussed in Cardellini et al. (2001), application layer clustering in Hwang and Jung (2002), and data layer clustering in Amza et al. (2003).

Interestingly, database clustering, which is of interest in the context of our work, has also been researched extensively, particularly in the late 1980s and early 1990s. Virtually all extant database clustering work has looked at the problem in the context of classical OLTP from two perspectives: (1) dynamically load balancing subtasks of a parallelized query processing algorithm across a shared-nothing multiprocessor framework (Hua et al. 1995; Rahm and Marek 1995), and (2) transaction routing in distributed database frameworks (Yu et al. 1987; Yu and Leff 1991). In particular, from the work in the second area, in which write/update statements are sent to the appropriate distributed databases for processing, as well as significant amounts of work done in the context of concurrency, locking, and associated synchronization maintenance techniques in distributed and parallel databases (Bernstein and Goodman 1981), there are well-known commercially viable solutions for distributing write requests to replicated data systems. The reader should note that the work in the first area, that of parallel query processing, is orthogonal to our work in this paper. This work considers queries that can be broken down into multiple subtasks and assigned to separate threads for processing. However, the reader might wonder why the work in the second area, deceptively similar-sounding to our problem (i.e., transaction routing sounds similar to request distribution), cannot be applied in our scenario. The reason is that, as noted in the "Introduction," read workloads could be as much as 95 percent of the total request workload (Gurley 2000) in these applications. This scenario is the focus of our work here.

One point of note is that these applications typically run in shared-resource enterprise data centers, where the data layer is composed of off-the-shelf database servers. Further, the primary method of distributing data across these databases is to replicate data across instances (Rahm and Marek 1995). (Cecchet et al. (2008) provide a good overview of database replication in commercial implementations, as well as techniques proposed in the academic literature.) The only other possible approach that avoids replication would utilize an alternative clustering framework (e.g., Oracle 10g RAC) or a similar framework available from other vendors. These types of frameworks use multiple database processes and shared storage. Such architectures are useful for write-heavy environments, but are very complex to configure and maintain. Thus, most web sites do not use them. (In a later section, we compare our approach to a clustering implementation provided by a major database vendor, and show that our approach significantly outperforms it for read-mostly workloads.) Since concurrency control for writes in replicated environments has been well-addressed in the literature (Bernstein and Goodman 1984; Thomas 1979), and is orthogonal to our work in this paper, we assume the presence of a distributed update mechanism such as that described by Plattner and Alonso (2004) to separate the read from the write workload and the work described by Bernstein and Goodman (1984) to handle the light write workload, and maintain concurrency across all copies of the database. Thus, the types of request distribution strategies of interest to us in this work are such that they can operate in the context of replicated databases.

Having considered the RD problem in the context of the broad literature, we now move on to consider specific request distribution work in the context of online applications.

There are a number of RD policies in use today. These policies fall into the three general categories: (1) *blind* request distribution, (2) *workload-aware* request distribution, and (3) *locality-aware* request distribution. We consider each in turn.

Blind policies, such as *random* and *round-robin* (RR) (Cardellini et al. 1999), distribute requests without regard to the content of the request or the workload state of the cluster instances. The random policy chooses a cluster instance based on a uniform distribution, while the RR policy distributes requests to each cluster instance in turn. These policies work well when all arriving tasks require the same amount of work (as in HTTP processing in web servers, for example) and all resources have equivalent capacities, since they will assign roughly equal workloads to each resource and likely ensure uniform quality of service across tasks.

Workload-aware policies attempt to account for differences in load incident on cluster instances due to the differences in work required to service assigned requests. These policies distribute requests based on a *least loaded* basis, using on a metric called *server load index* (Cardellini et al. 2001; Qin et al. 2009). Polices of this type consider measures of load on each cluster instance in decision making. The two most common variants of workload-aware policies are *lowest CPU* (LCPU) (Eager et al. 1986; Harchol-Balter and Downey 1997), where each incoming request is routed to the instance with the lowest percentage of CPU utilization, and *greatest free memory* (FMEM) (Zhang et al. 2000), which identifies the instance with the greatest percentage of free memory. Elnikety et al. (2007) describe a memory-aware request distribution technique similar to FMEM, where the proposed method attempts to estimate the size of the working set for a request, and route requests to maximize in-memory processing (i.e., to avoid multiple disk reads of the same data for a single request). RR, LCPU, and FMEM are the most common existing request distribution techniques in practice for replicated database clusters, and thus we will compare our method to these in our experiments.

Locality-aware policies consider the data residing on cluster instances in request distribution. Such policies fall into two classes: (1) client/session affinity strategies and (2) content affinity strategies.

Client/session affinity schemes (Cardellini et al. 2001; Hunt et al. 1998; Linux Virtual Server Project 2004), which are based on locality of session state with respect to the cluster instances, are widely used at the application layer. Respecting such locality in RD is critical for stateful applications, where a user's session state, which might include the contents of a shopping cart or partially completed airline reservation, resides on a single application instance in the cluster; sending a request to a cluster instance where the session state is not available creates a new session state on that instance. This work cannot be directly applied in the data layer because databases do not retain client-related data between requests.

Content affinity schemes attempt to take advantage of data residing in memory on cluster instances due to the processing of prior requests. In one of the first papers to apply content affinity in dynamic content generation scenarios, where the work required to process requests is not known *a priori*, Pai et al. (1998) propose the *locality-aware request distribution* (LARD) algorithm. The LARD strategy attempts to route tasks to exploit the locality among the working sets of received requests (e.g., cache sets on different web servers). Our aim is similar: we extend the notion of content affinity to the data layer in this work.

At a high level, application servers and database servers share the common property that their request workloads tend to be dynamic, with requests requiring widely varying computational loads to execute. If we look more closely, however, we note that there is a significant difference between application and database requests; specifically, application requests result in program invocations, while database requests arrive as structured statements. These structured statements are inherently more amenable to analysis than program invocations, providing more information to leverage when making request distribution decisions. This motivated us to consider the fact that targeted request distribution strategies in the data layer might work particularly well. In this paper, we explore the possibility of taking advantage of such cached data to distribute load more effectively across a database cluster.

In the next section, we describe the details of our method.

Approach and Technical Details

In this paper, we propose a *cost-based database request distribution* (C-DBRD) mechanism to distribute requests from the application layer to the database layer. Architecturally, the C-DBRD module is an application middleware component independent from the database layer, located between an application server cluster and a database cluster, as shown in Figure 3. The applications running in application servers submit database requests to the C-DBRD, which runs its decision logic and chooses the most appropriate database instance among the set of D instances ($d_0 \ldots d_{D-1}$) in the database cluster to process each request. We first describe the intuition behind the C-DBRD logic and then provide the details of its decision-making mechanism.

C-DBRD Intuition

The goal of C-DBRD logic is as follows: *route a task to a DB instance that can process it with the least amount of work.* While the computational work required to process a task depends on a number of factors, the most significant one is the "similarity" of the current task to prior jobs executed at that instance. It is well known that the *performance of a statement on a database instance is greatly dependent on which statements were run previously on it* (Oracle 2009). Modern database systems are cognizant of the often significant amounts of locality (i.e., commonality of accessed data objects) that exist between statements, especially when they are coming from applications, which rarely generate arbitrary *ad hoc* database statements. These databases optimize perfor-

mance by caching multiple internal data structures to take advantage of this locality. In other words, they cache the results of work done for recently completed jobs with a goal of reusing those results for future tasks. In most commercial DBMS implementations, both data (i.e., disk blocks containing relations) and metadata (data describing where and how data are stored) are cached. For example, Oracle 10g maintains two such caches to share objects between concurrent clients (Oracle 2009): (1) a shared pool that stores meta-data structures (e.g., intermediate query-processing related structures, parsing-related structures, SQL execution path calculations, and optimization calculations), and (2) a buffer cache that stores recently used disk blocks (including table spaces, data files, indexes, and other data structures). Other vendors may cache a slightly different set of data structures, but the general concept applies across all enterprise databases. This caching infrastructure can have a dramatic impact on the execution time of statements able to take advantage of cached structures. Having described the impact of locality and caching on task processing efficiency, we now proceed to outline the C-DBRD strategy.

At a high level, C-DBRD distributes requests by performing a *trade-off analysis* across two dimensions. First, it takes advantage of caching in the database instance by attempting to route a request to an instance where *similar requests have been executed in the recent past*, and can take advantage of data structures cached on the instance. Second, it attempts to *minimize load differences* across instances in the cluster.

We describe the request distribution problem generally as follows: At runtime, given a cluster of D database instances and upon receiving a statement s, the C-DBRD is responsible for choosing the appropriate database instance $d \in D$ to which s should be routed. As discussed previously, the choice of the appropriate database instance to process statement s will depend upon (1) the marginal load that will be caused by s upon this instance, and (2) the current total load upon the instance.

At this juncture we would like the reader to appreciate the fact that the load computation mechanism described below is a major contribution of this paper. This is because database load computation in particular, and dynamic system load computation in general, are important unsolved problems with wide applicability. We provide a short discussion below.

- Consider the problem of database load computation, addressed directly in this paper. This problem arises in a vast number of research and practice areas of database systems, such as transaction scheduling, dynamic data allocation, online data backup and recovery, and virtually any database problem that requires decision making at runtime. In particular, this is a major issue in the currently popular area of cloud computing (Birman et al. 2009), where data and task distribution is key to all runtime processing. In spite of the ubiquity of the problem, there exists no systematic (let alone analytically tractable) method to compute a metric expressing load on a running database (Ahmad et al. 2009). Our proposed method addresses this long-standing issue.

- In a broader context, load computation in unpredictable dynamic systems such as a computer system is a hard problem (Lipasti et al. 1996), but one whose solution would be of great value. The underlying estimation problem is similar to the database problem described above. For instance, the issues in estimating the load on a running computer system are very similar to transactions in a database: localities of task working sets are a primary determinant of the effort required to process them. Our mechanisms are extensible to some of these broader issues as well.

To realize the strategy outlined above, we propose, in this paper, a method to compute a metric which dynamically measures load on a database system, given a specific set of tasks incident on it.

We introduce the intuition behind our notion of database load next.

The load on a database instance d, denoted l^d, at a particular point in time is simply a measure of the amount of work that d is performing at that time. The amount of work that d is performing depends on the current set of statements executing in the database. Let us denote the quantification of the amount of work performed by a database system to process a statement s as C_s. The issue is how to compute C_s. While the database can produce an estimate of the statement cost (based on the query plan), such an estimate would likely suffer from high inaccuracy due to the effects of locality and caching, as discussed earlier. In other words, *it can cost more to process a given statement s on some database instance d_i than it might cost on another instance d_j, due to the effects of caching on each database instance.* To make this problem even more complex, the workload and cache state of each instance both change with each request assigned to it. To estimate the load on each database instance in a cluster, we need a model of database workloads that takes into account the effects of caching in the cost of processing statements. We were unable to find such a model in the current literature. We then require a method that implements the model as a decision-making method, workable for production IT scenarios. In this section,

Table 1. Table of Notation	
D	Set of database instances
d_i	Data instance in D
n	Number of statements for which the QHR maintains a list of data objects accessed for each database instance
s	Statement submitted for database processing
S	Set of incoming database statements
C_s	Cost of processing a statement s
b_i	Set of all tables accessed by statement s_i
h_i	Set of all indexes accessed by statement s_i
B_s	Set of all tables accessed by statements in S
H_s	Set of all indexes accessed by statements in S
$SF_{i,j}$	Similarity factor for statements s_i and s_j
SF^{d}_{s,s^d}	Similarity factor for a statement s compared to a set of statements S_d executing on a database instance d
δ	Network and client connection processing overhead for a statement s
γ	Processing load reduction due to caching
α	Effect of statement similarity in reducing database load
E^{d}_{s}	Effective load on instance d, if statement s were to be added to d
t^{y}_{s}	Time required to process the y^{th} execution of s on a give database instance

we propose such a model. We then propose a workable heuristic method of using our workload model to make data-layer request distribution decisions.

Table 1 summarizes the notation used through the remainder of this paper.

Modeling Database Workload

The starting point for our model is an estimate of the cost of processing a statement, C_s. While each database instance in the cluster will produce the same estimate of statement cost based on the query plan the database generates, this estimate does not take into account the effects of caching, which varies across all instances in the cluster. To quantify the database workload more realistically, we need to define a way to estimate C_s in the presence of caching, taking into account the cache state of each instance in the cluster.

Next, we need a model of total database workload that takes into account the costs of all statements executing on the instance, modulated by the effects of caching. We further note that each database vendor implements caching to a different degree, with the effect that a cluster running on a vendor's software that implements caching to a greater degree than another vendor will see a greater effect from caching. Our model needs to account for these differences across vendors.

In the remainder of this section, we develop two important theoretical foundations: (1) a model of the processing cost savings for a statement in the presence of caching and (2) a model of database workload that takes into account the effects of caching.

Modeling the Effects of Caching on Statement Processing Cost

We model the cost of processing a database statement in a cache-enabled scenario using two distinct notions: (1) *base statement cost*, which represents the expected cost of processing a statement without considering caching effects, and (2) *similarity factor*, which measures how similar two statements are in terms of the data objects required to process them.

Base Statement Cost: We use the notion of the cost of a statement s, denoted C_s, to represent the expected cost of processing s without considering the effects of cached data structures in the database. C_s is dependent upon the query plan (based on the database schema) for s, and is typically measured as the time required to execute the statement in an unloaded database system. The C_s computation in our implementation is based on the cost computation performed by the cost-based SQL optimizers in Oracle (Burleson 2009) and SQL Server (Microsoft Inc. 2011). Such costs are given as the estimated time (in milliseconds) the statement will take to execute in an unloaded database, as provided by the database query optimizer. Since query optimizers are known to give cost estimates that are not particularly accurate, we use the cost obtained from optimizer as a relative value to indicate relative load of a statement on a database instance.

Similarity Factor: Intuitively, if two similar statements run in sequence on the same database instance, it is likely that the second statement will be able to reuse work done on behalf of the first. Since C-DBRD attempts to identify an instance where the data objects needed to process a request are most likely to be in an instance's cache, we need a way to measure the extent to which two statements will access an overlapping set of data objects. We model this with the notion of *similarity factor* (SF). We define the *Similarity factor*, $SF_{i,j}$, as a number between [0, 1], which quantifies the similarity of a database statement s_i with another database statement s_j in terms of data object access. We note here that we are applying the notion of similarity to database statements for the purpose of identifying reusable read-oriented database objects. The notion of a similarity factor, however, is much more broadly useful than this specific application. Still within the realm of comparing database statements, we can use a similarity factor to estimate potential conflicts for two update statements. We can extend the utility of similarity factor further, beyond database statements, to apply it to any cache-enabled component (e.g., application servers or other software components) to estimate workload differences due to caching effects.

To understand how the similarity factor is computed for two database statements, we first present two axioms that describe the boundary cases for similarity factor.

Axiom 1: *If the two statements s_i and s_j are identical, then $SF_{i,j}$* = 1.

Intuitively, if s_i and s_j are equal, the database instances that process them will use the same query plan and access the same database structures in the course of processing. This occurs because query plan generation is deterministic: if two database instances are configured with the same version of a vendor's database software and the same database schema, then given the same query, these two database instances will generate the exact same query plan, referencing the same database structures (e.g., tables, indexes). Here, the $SF_{i,j} = 1$ tells our request distribution logic that s_i and s_j have high locality of reference, such that there is a high likelihood of cache utilization if they are processed by the same database instance.

Axiom 2: *If O_i is the set of database objects accessed by statement s_i, and O_j is the set of database objects accessed by statement s_j, and $O_i \cap O_j = \varnothing$, then $SF_{i,j} = 0$. That is, if there is no overlap in the set of database objects accessed by two statements s_i and s_j, then the similarity factor is* 0.

When $SF_{i,j} = 0$ for s_i and s_j, the two queries will access a completely different set of database structures from one another. In other words, they have no locality of reference in terms of the database structures they access and, therefore, present no opportunity to utilize cached data structures on a database instance.

The values $SF_{i,j} = 0$ and $SF_{i,j} = 1$ represent boundary conditions for $SF_{i,j}$ for s_i and s_j. In most cases, we would expect that the set of database structures accessed for s_i and s_j would only partially overlap, that is, that the value for $SF_{i,j}$ would be between 0 and 1, and should represent the extent to which there is potential to benefit from cached data structures if s_i and s_j are processed on the same database instance. Based on the above, we define $SF_{i,j}$ as

$$SF_{i,j} = \frac{|\{s_i\} \cap \{s_j\}| + |O_i \cap O_j|}{1 + |O_i|}$$

Here, the $|\{s_i\} + \{s_j\}|$ portion of the numerator of this expression indicates whether or not s_i and s_j are exactly the same; if so, then $|\{s_i\} \cap \{s_j\}| = 1$, otherwise $|\{s_i\} \cap \{s_j\}| = 0$.

In the above discussion, O represents a set of generic cached data objects. There are many different types of data objects that may be stored in memory, such as parsing-related structures, SQL execution path calculations, optimization calculations, low-level data blocks, table spaces, data files, indexes. For instance, results of intermediate join-processing steps (or other intermediate results) might be cached. For the purposes of simplicity of explanation, we consider only two types of database objects that are accessed by a statement s_i in our discussion here (we note that this can be extended to describe any data structures cached by the database): database tables and indices. We define b_i as the set of database tables accessed by the statement s_i, and h_i as the set of database indices accessed by the statement s_i. We note here that

some tables may be too large to store in cache. In such cases, it is possible to track access at the level of a disk block; however, considering this level of detail would unnecessarily complicate our explanation here. Based on our example set of data objects (i.e., tables and indices), we can redefine

$$SF_{i,j} = \frac{|\{s_i\} \cap \{s_j\}| + |b_i \cap b_j| + |h_i \cap h_j|}{1 + |b_i| + |h_i|}$$

To illustrate the notion of the similarity factor, let us consider an example. Consider two database statements as described below:

- s_1: Select * from EMPLOYEE WHERE EMPLOYEE.SSN = '888-88-8888'
- s_2: Select * from EMPLOYEE e, DEPARTMENT d, DEPARTMENT_EMPLOYEE de WHERE e.SSN = '888-88-8888' AND d.DEPT_ID = de.DEPT_ID AND de.EMP_ID = e.EMP_ID

Let us also assume that indices exist for EMPLOYEE.SSN, EMPLOYEE.EMP_ID, DEPARTMENT.DEPT_ID, DEPARTMENT_EMPLOYEE.DEPT_ID, and DEPARTMENT_EMPLOYEE.EMP_ID.

In this scenario, b_1 = {EMPLOYEE} and b_2 = {EMPLOYEE, DEPARTMENT, DEPARTMENT_EMPLOYEE}. Further, h_1 = {EMPLOYEE.SSN}, and h_2= {EMPLOYEE.SSN, DEPARTMENT.DEPT_ID, DEPARTMENT_EMPLOYEE. DEPT_ID, DEPARTMENT_EMPLOYEE.EMP_ID and EMPLOYEE, EMP_ID}.

Thus, $|b_1| = 1$, $|b_2| = 3$, $|h_2| = 1$, and $|h_2| = 5$. We also have $b_2 \cap b_1$ = {EMPLOYEE} and $h_2 \cap h_1$ = {EMPLOYEE.SSN}. Thus, $SF_{2,1} = 2/9 = 0.22$. If s_1 has executed in the database, it has accessed the table EMPLOYEE and the index EMPLOYEE.SSN, so these two database objects will be cached in the database. When s_2 comes to the database, then these two objects do not need to be fetched from storage again; only the following objects need to be retrieved: tables DEPARTMENT and DEPARTMENT_EMPLOYEE, and indices DEPARTMENT.DEPT_ID, DEPARTMENT_EMPLOYEE.DEPT_ID, DEPARTMENT_EMPLOYEE.EMP_ID, and EMPLOYEE.EMP_ID. This reduces the total workload on the database required to process s_2.

This reduction in percentage workload reduction is estimated by the similarity factor $SF_{2,1}$.

We note that $SF_{i,j}$ provides us only with a comparison between two queries, $s_i$and s_j. However, the contents of a database instance's cache represent the structures used for multiple recent queries. Thus, it is possible, for a database query s and a database instance d_k, that multiple recent queries on d_k could collectively have a high locality of reference with s, where each of several recent queries accessed a portion of the database structures needed for s. To recognize this, we need a way to measure the similarity of s to a set of database statements.

Next, we define the similarity factor of a database statement s with a set of database statements S (i.e., $SF_{s,S}$). Let us define $B_S = \cup_{j \in S} b_j$ (i.e., the set of all tables accessed by all statements in the set S). Similarly, we define $H_S = \cup_{j \in S} h_j$. From there we can define

$$SF_{s,S} = \frac{|S \cap \{s_j\}| + |B_S \cap b_s| + |H_s \cap h_s|}{1 + b_s| + |h_s|}$$

At a high level, the similarity factor metric comparing a newly arrived statement to the currently executing statement sets across all database instances in a cluster is used to direct a statement to a particular database instance in a cluster. We explain this computation using an example. Consider a statement s arriving at the C-DBRD. Upon receiving s, the C-DBRD determines (based on the query plan for the statement) that processing s will require a set of tables b_s and a set of indexes h_s. Now, let us consider a target instance d_j. Based on information stored regarding prior statements executed on each database instance, the C-DBRD knows (1) S_j, the unique set of the last n statements processed at d_j; (2) B_j, the unique set of tables accessed by the queries represented in S_j; and (3) H_j, the unique set of indexes accessed for these last n queries. In this case

$$SF^j_{s,S_j} = \frac{|S_j \cap \{s_j\}| + |B_j \cap b_s| + |H_j \cap h_j}{1 + |b_s| + |h_s|} \quad (1)$$

Effectively, Expression 1 models the processing cost savings for a statement in the presence of caching, given a set of queries previously run on an instance j. This fulfills the first part of our modeling requirements. Clearly, the value of n is important here; we discuss the factors impacting the value of this parameter at the end of this section, after describing our model of database workload and the details of our method.

Modeling Database Instance Workload

In this section, we model the estimated database workload due to a set of statements running in the database. We first present a set of axioms, which we use to form the basis of our estimated load computation.

Axiom 3: *If there is a set of database statements S running in a database instance, such that* $b_i \cap b_j = \varnothing$, $\forall i, j \in S$ *and* $h_i \cap h_j = \varnothing$, $\forall i, j \in Q$, *that is, there is no overlap in the objects accessed by any two statements in S, the total database load l on the database instance is computed as the sum of the estimated database load by each of the statement* $s \in S$, *that is,* $l = \Sigma_{s \in S} C_s$, *where each* C_s *is estimated as the statement cost from the database query analyzer.*

Axiom 4: *If a database statement* s_j *arrives at the database when another statement* s_i *is being executed, such that* $s_i = s_j$, *that is, the two statements are identical, then due to various levels of caching deployed in the database systems, the total load of the database will be* $C_i + \delta$, *where* $\delta << C_j$, *where* δ *represents the base overhead for processing the statement* s_j, *even if all needed data objects are drawn from cache. For example, these overheads might include network and client connection handling.*

Axiom 5: *If a database statement sj arrives at the database when another statement* s_i *is being executed, such that* $s_i = s_j$, *that is, the two statements are not identical, and* $O_i \cap O_j = \varnothing$, *that is, there is an overlap in the sets of objects accessed by the two statements, then due to various level of caching deployed in the database systems, the total load of the database will be* $C_i + \gamma C_j$, *where* γ *is the reduction factor of load due to caching on the database. We estimate* γ *to be* $\gamma = (1 - SF_{i,j})$.

So, if $SF_{i,j} = 0$, the total database load is $C_i + C_j$, and Axiom 3 follows.

If $SF_{i,j} = 1$, the total database load is C_i, which follows from Axiom 4, by ignoring δ. To include the overhead of network and client connection due to processing of statement s_j (i.e., δ), we modify the reduction factor as $\gamma = (1 - \alpha SF_{i,j})$, where $\alpha \in [0, 1]$ is a factor indicating the effect of similarity between s_i and s_j in reducing the total database load. Intuitively, α represents the expected level of cache utilization on a database instance (i.e., the amount of work avoided), as measured by processing execution time, through the use of cached data structures (note that we cannot expect caching to completely mitigate the work of processing a request). If $\alpha = 0$, the effect of similarity has zero impact on database load, and Axiom 1 will follow. If $\alpha = 1$ and $SF_{i,j} = 1$, $\delta = 0$.

In reality, δ can never be 0 and α can never be 1; caching cannot mitigate all of the work required to process a request. For instance, caching cannot mitigate the need for a database to process a request arriving over a network connection and determine which data objects are needed to process it.

In real terms, α will always fall somewhere between 0 and 1, tending more toward 1 as objects needed to process requests can be found in cache, and more toward 0 as fewer objects needed to process requests can be found in cache. The value of α is, therefore, dependent on how effectively the database has implemented caching. We describe how we estimate α, along with other important parameters for our method, at the end of the "C-DBRD Architecture and Method" section.

We now extend the above load computation to allow for more than one statement. We begin with the notion of load on a database instance before a new statement is added to its workload, denoted l^d. Here, the load on an instance is

$$l^d = \sum_{s \in S_d} \left(1 = \alpha SF^d_{s,S_d}\right) \times C_s \tag{2}$$

We now consider the scenario where we estimate the additional load that would occur on an instance if a new statement s were assigned to it, which we call the *effective load* of an instance, denoted E^d_s. Let us assume that a set of database statements S has executed in the database instance d such that the objects accessed by these statements are in the cache and the current load of the database d is l^d. If a new statement s arrives at d, the new load E^d_s on the database instance with this new statement will be

$$E^d_s = l^d + \left(1 - \alpha SF^d_{s,S_d}\right) C_s \tag{3}$$

Together, Expressions 2 and 3 represent a model of database workload in the presence of caching.

C-DBRD Architecture and Method

Now we are in a position to describe the actual C-DBRD algorithm. We begin our discussion with an architectural overview of the C-DRBD module, and then propose a specific method for request distribution decision-making in the context of database requests.

C-DBRD Architecture

Architecturally, the C-DBRD consists of task distribution logic and a set of data stores that house data useful for computing statement costs for incoming statements and estimating database workloads, as depicted in Figure 4. We describe each component next.

Statement History Repository (SHR): The SHR stores a configurable-length history of requests processed by the C-DBRD. More precisely, the SHR keeps a list of the *request statement*, *tables accessed*, and *indexes used* for the last *n*

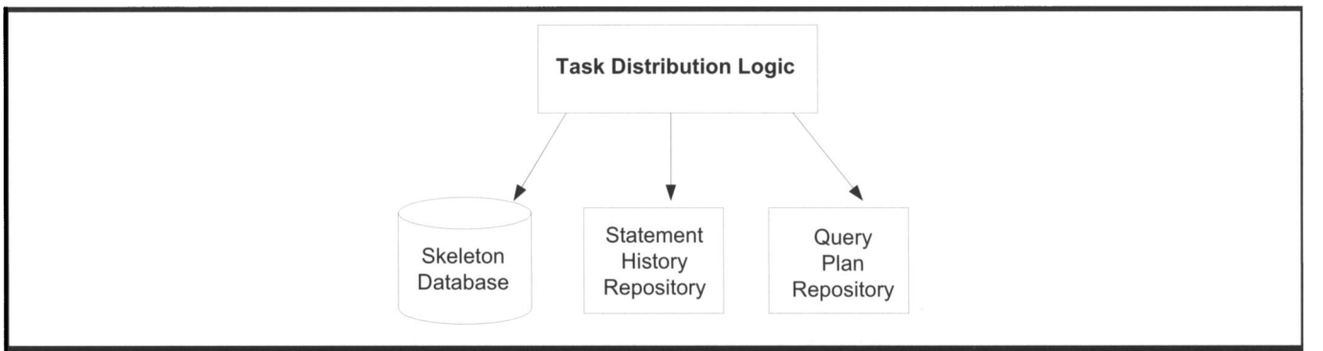

Figure 4. C-DBRD Modules

statements for each database instance d_i in the DB cluster. Since all requests for each database instance pass through the C-DBRD, maintaining a list of the most recent n statements for each database instance d_i is simply a matter of recording these statements as they pass through the request distribution logic. (We describe how we estimate n, along with other important parameters for our method, at the end of "C-DBRD Architecture and Method" section.

Query Plan Repository (QPR): The QPR stores, for each request processed by the C-DBRD, the associated *query plan* and the *statement cost*. The primary use of this is as a cache to improve the performance of request handling in the C-DBRD. As we will describe, the C-DBRD uses statement costs, estimated using a query plan, to make request distribution decisions. Instead of computing the cost for each incoming statement every time, it first checks to see whether the plan and cost already exist for a newly arrived statement in the QPR.

This type of caching is beneficial only when there is an expectation of reuse of cached items (i.e., when calls to the database will consist of a finite set of SQL statements). This is actually the case in the online e-commerce domain. To see why, let us consider how database statements are typically programmed in applications.

Statements may be statically coded (i.e., hard-coded into the application at design time). At runtime, variables are dynamically bound. An example of such a statement might be

```
SELECT * FROM CUSTOMER WHERE
CUSTOMER_ID = $customerid
```

Here, the structure of the query, including table and attribute names, is set, and only the customerid variable changes at runtime. The number of statically coded statements in an application is clearly a finite number; changes only occur when the database schema or the application itself is updated.

Statements may also be dynamically generated, where the structure of the statement itself is built on demand at runtime. Most dynamic statement generation in online e-commerce applications occurs in the context of object-relational mapping. Here, the programmer describes the data desired from the database as an arbitrarily complex set logic expression over sets of programmatic objects. At runtime, the object-relational framework—for example, Hibernate (JBoss Community 2011) or LINQ (Microsoft .NET Developer Center 2011)—follows an algorithm to convert the set logic expression to SQL. Given the same input expression, the algorithm will generate the same SQL statement (in contrast, two humans might convert the set logic expression into two different SQL statements). That is, each coded set logic expression in an application maps to one SQL statement structure at runtime (only bound variables change). Thus, the number of SQL statements dynamically generated in the context of object-relational frameworks is also finite.

It is possible to write code that dynamically generates SQL statements, but this typically occurs only in very limited and well-bounded and controlled circumstances due to the need for testability. To be able to build appropriate tests for such code, the programmer needs to be able to define the boundaries of the set of SQL statements that could potentially be generated by a code segment. The size of this set must also be finite.

Given that the set of statements submitted by online applications is typically finite, at steady state the hit ratio of the QPR cache turns out to be quite high (often over 90 percent, assuming sufficient space is allocated for QPR storage), so caching this information significantly reduces the cost of processing at the C-DBRD. If the needed query plan and statement cost are not found in the QPR, the C-DBRD requests them from the SDB.

Skeleton Database (SDB): The SDB stores skeleton instances of the various database schemas that are present in the database cluster under the control of the C-DBRD. It is used to compute query plans and estimated costs for the requests arriving at the C-DBRD from the application layer.

We note here that it would be possible to compute query plans on the production database instances in the cluster. However, that would create additional load on those instances, and would involve a remote call from the C-DBRD to a database instance for each needed query plan. In the interests of scalability and performance, therefore, we instead designed our method with the inclusion of the SDB. The logical structure of the SDB is the same as the production database instances in the cluster; it is created using the same DDL (data definition language) script that defined the actual production database instances in the cluster.

Task Distribution Logic (TDL): The TDL is the brain of the C-DBRD; it implements the decision logic using the information stored in the SDB, QHR, and QPR modules described above (as indicated by the arrows in Figure 4) to make request distribution decisions. We now present the details of the TDL. Since TDL is really the central task logic component of the C-DBRD, we shall simply use the term C-DBRD to refer to TDL through the remainder of the paper without loss of generality.

Having described the internals of the C-DBRD module, we now move on to discuss our proposed request distribution method.

C-DBRD Request Distribution Method

In our proposed request distribution method, we take a dynamic solution approach, where each statement that arrives from the application layer to the clustered data layer is assigned to one of the instances of the database in the data layer by the C-DBRD module. Intuitively, upon receiving a statement request, the C-DBRD computes, for each target instance under its control, the effective load denoted E_s^d, as the sum of existing load and the additional load that would be introduced by the new statement modulated by the similarity factor of this statement at the various target instances. Note that this algorithm assumes that the C-DBRD keeps track of the currently executing requests at each instance, which can be done trivially as all requests and responses flow through the C-DBRD.

Algorithm 1. C-DBRD Scheduling Algorithm

1. Input: A newly arrived statement s
2. Output: Database instance $d_s$where statement s will be forwarded.
3. Compute C_s from QPR and SDB
4. Compute current load $l^d$of each database instance d.
5. Compute effective load of E_s^d of each database instance with the addition of statement s
6. We select the database instance d such that
$$E_s^{d_q} < E_s^d, \forall d \in D$$
where D is the set of all instances in the cluster.
7. In the case of equivalent values of E_s^d, C-DBRD follows a round-robin distribution strategy.

In step 3, the algorithm estimates the cost of the statement s using the QPR and SDB. If the information for s is found in the QPR, the estimated cost is fetched directly from the QPR. If not, the C-DBRD uses SDB to compute the estimated cost using the SDB's plan generator.

In step 4, we compute the load of the presently executing statements in each database instance (as discussed in the context of Expression 1).

In step 5, we compute the estimated load of each database if the newly arrived statement s is added as discussed in the section on "Modeling Database Instance Workload" (in the context of Expression 3). Here again, we use α to consider the realistic state in database instances, where the cost of the second execution of a statement is not zero, nor is it exactly the estimated cost modified by similarity factor effects. (Recall that we use the parameter α to represent the extent to which an RDBMS implements caching, that is, the portion of data objects needed for a query *could potentially be found in cache*, based on the database implementation, and to account for the time required to perform request processing work that cannot be mitigated by using cached data.)

In step 6, the database instance with the minimum estimated load is chosen to process the newly arrived statement s. If the database instances are not homogeneous (i.e., they have different memory and CPU power), we assign a weight factor (W_d) (between 0 and 1) to each database instance (d) such that highest performing database server has the highest weight 1, and other instances' weights are estimated in proportion to it. The instance with minimum load across all database instances is computed based on the weighted load $W_d \times E_s^d$ for each database instance d.

Step 7 considers the case where the estimated load on each cluster member is equal prior to dispatching the incoming request (which would occur, for example, at system startup). Here, the algorithm uses a type of round-robin dispatching; it selects the "next" cluster member, based on the cluster member that received the immediately prior request, regardless of whether that prior request was dispatched based on round-robin scheduling, or based on estimated load on each cluster member.

If the workflow is carefully examined, it can easily be seen that the execution logic is lightweight: upon the receipt of every request, the C-DBRD performs lookups on the QPR and the QHR (both of which are cached in memory) and executes the algorithm above, which involves some simple computations. On rare occasions (in steady state) when there is a cache miss in the QPR, the database optimizer is invoked. To validate this claim, we demonstrate the lightweight nature of our method experimentally in the "Experimental Study" section.

Environmental and Input Parameters Affecting Load Estimates

The two parameters that are important for the C-DBRD scheduling algorithm are (1) n, the number of statements in history considered for computing similarity factor, and (2) α, the factor denoting the effect of similarity on database instances.

The value of n is chosen based on the size of memory reserved for caching in the database versus the database size. A higher value of this ratio indicates that a larger number of tables, indexes, and statements can be kept in the cache, indicating that n can have a higher value. In "Experimental Study" section, we will experimentally demonstrate the effect of n on overall performance of the database cluster.

As we discussed in section on "Modeling Database Instance Workload," the value of α is dependent on the extent to which a database vendor has implemented caching. Before presenting a procedure for estimating α for a database system, we present two definitions that will form the basis of our estimation.

Axiom 6: *If a particular database system does not have any caching implemented within it,* $\alpha = 0$.

Axiom 7: *If statement s takes* 0 *milliseconds for its second consecutive execution in a database instance,* $\alpha = 1$.

Let us assume that t_s^1 is the time it takes to execute a statement s for the first time in a database system where nothing else is executing. Let us also assume that t_s^2 is the time it takes to execute the same statement s in the second consecutive time in the same database system without anything else executing in the database. For a set of statements S, the α for the database system can be estimated as

$$\alpha = \left((1 - \frac{\sum_{s \in S} \frac{t_s^2}{t_s^1}}{|S|} \right)$$

Based on our experiments with Oracle 11g on Windows XP, we have found that Oracle 11g implements caching very effectively, and have found that $\alpha = 0.8$ is an effective value for α for this database.

An Analysis Comparing C-DBRD to Other Virtualization Methods

In this section, we compare the expected effectiveness of the C-DBRD method as compared to a round-robin (RR) approach, the most common database request distribution method in practice, and compare the time complexity of C-DBRD with other request distribution approaches.

Expected Effectiveness of C-DBRD

We are primarily interested in utilization of capacity as a measure of the effectiveness of a request distribution approach. We use processing time for a request as a proxy for capacity utilization, since a request distribution method that processes requests faster than another method will be able to support additional processing capacity as compared to the slower method and, therefore, will have better capacity utilization.

> **Hypothesis 1:** *The C-DBRD approach of load distribution in the clustered data layer will result in lower or equal loads on individual database instances compared to the round-robin approach.*

We illustrate this with an example. Let us assume that there are four statements s_1, s_2, s_3, and s_4 and each incur an equivalent cost C_s load on a database instance. Let us also assume that the set of database objects accessed by s_1 is B_1, the set of database objects accessed by s_2 is B_2, the set of database objects accessed by s_3 is B_2, and the set of database objects accessed by s_4 is B_1. Further, let us assume that there are two database instances d_1 and d_2.

Following a round-robin approach, s_1 and s_3 will be delegated to database d_1, and s_2 and s_4 will be delegated to database d_2.

So the total load on database d_1 and database d_2 will be $2C_s$. In contrast, in the C-DBRD approach s_1 and s_4 will be delegated to d_1, because both of them are accessing the set of database objects B_1. s_2 and s_3 will be delegated to database d_2, because both of them are accessing the set of database objects B_2. In this scenario, if $SF_{4,1}$ is the similarity between s_1 and s_4 and $SF_{3,2}$ is the similarity between s_2 and s_3, we have $SF_{4,1} > 0$ and $SF_{3,2} > 0$. So, considering the similarity factor, the total load of the database d_1 is $C_s + (1 - \alpha SF_{4,1})C_s$ and the total load of the database d_2 is $Cs + (1 - \alpha SF_{3,2})Cs$. So, the total load on each of d_1 and d_2 will be less than that of the round-robin approach. (We demonstrate this experimentally in the "Experimental Results" section, in the discussion surrounding Figures 5 and 6.)

> **Hypothesis 2:** *The C-DBRD approach and the round-robin approach will be equivalent if and only if all incoming database statements are the same; in all other cases, the C-DBRD approach will give better request distribution decisions than the round-robin approach.*

To illustrate, let us assume a set of database instances $D = \{d_1, d_2, \ldots, d_u\}$ and a sequence of incoming requests $s_1, s_2, \ldots, s_v$. If s_i is assigned to d_j, then round-robin scheduling will assign $s_i + 1$ to database instance $d_j + 1$, invariant of load on each database instance.

In contrast, in C-DBRD the instance selected for the statement $s_i + 1$ will be selected based on the estimated load of each database instance due to this $s_i + 1$, thus the selection set for the database instance of $s_i + 1$ is much broader in C-DBRD than in the case of round-robin, allowing C-DBRD to select the instance with the lowest expected cost. In cases where there is no expected cost-saving advantage in deviating from a round-robin approach, the C-DBRD will follow round-robin scheduling. Thus, we can expect the C-DBRD to provide better load distribution than the round-robin approach. In the worst case, C-DBRD will provide an equivalent quality of request distribution decisions, when compared to the round robin approach. (We demonstrate proof of this experimentally in the "Experimental Results" section, in the discussion surrounding Figures 5 and 6.)

Complexity Analysis

In this section, we compare the time complexity of our approach to the three most common request distribution techniques in practice: RR, LCPU, and FMEM. For each of these methods, we consider the cost of computing the request distribution decision; that is, when a request arrives, how much computational effort is required for each approach to determine which instance in a database cluster of $|D|$ instances should process the request? We consider each approach in turn. For round-robin, the cost of computing the target database instance is simply the cost of moving a pointer in an ordered list. Thus, for RR, the time complexity is constant, $O(1)$.

For LCPU, the database request distribution module needs to compute the CPU utilization for each database instance, and then search the list for the instance with the lowest CPU utilization. If t_{cpu} is the time required to get the CPU utilization for one database instance, the total time complexity for the request distribution decision for LCPU is $O(|D|t_{cpu} + log|D|)$.

For FMEM, the request distribution module must retrieve the free database buffer size from each database instance, and then search the list for the greatest percentage of free memory. If the time required to retrieve the memory data from each instance is t_{mem}, the total time complexity for FMEM is $O(|D|t_{mem} + log|D|)$.

We now consider the C-DBRD case. In our approach, we compare the incoming statement s to the expected contents of each database instance's cached tables and indexes, and select the instance with the highest similarity, in terms of cached tables and indexes, to the needs of the incoming query. Architecturally, this is supported by two caching structures in the C-DBRD module (1) a QPR cache of query plans and associated statement costs, sized to accommodate the set of unique statements generated by the application, and (2) an SHR cache of the most recent n statements processed by each instance, along with a list of the tables and indexes accessed by each statement.

Let us denote the time cost of computing a query plan for a new incoming statement s as t_s time, and the cost of computing the effective load E_s^d for a database instance d based on the addition of s as t_e.

At system startup, the QPR is empty, and the cost of computing the query plan for each unique s in the application is paid once by the SDB (as described in section "C-DBRD Architecture"), after which, for each subsequent arrival of s, the query plan is drawn from QPR cache. The QPR cache is a hash table, which stores three data items for each query: (1) an MD5 128 bit encoding for the SQL statement, (2) a 32 bit integer for the estimated execution cost to execute the query plan, and (3) a hashed value representing a list of tables and indices to be accessed by the SQL statement, which varies in size but rarely exceeds 2 KB. Based on these numbers, roughly 10,000 SQL statements and their details can be cached in the QPR hash table that requires only 2 MB of memory.

At steady state, virtually all query plans are available from cache, making t_s an $O(1)$ operation. Let us consider the time required to fill the cache. If there are about 10,000 SQL queries in an application and the throughput of the system is 40 requests per second (i.e., 40 SQL queries get executed every second), then, if all 10,000 SQL queries are executed sequentially, we estimate that after about 250 seconds, all queries and their associated query plan data will be in the QPR cache. After this point, there is no need to fetch the query plan from the SDB. Thus, the duration of the transient state (during which SDB interaction is required to fetch the query plan) is only a few minutes (in the range of 4 to 5 minutes).

If we assume a Zipfian distribution of user requests, where 80 percent of requests are for 20 percent of queries, the query plan data for the 20 percent of highly requested queries would be placed into cache within a few minutes, while the data for the 80 percent of less-frequently requested queries would be placed in cache upon the first request for each query. While the time to fill the cache would be longer than then the sequential case, the work done to fill the cache (i.e., the work of generating 10,000 query plans) is the same in both cases.

At steady state (i.e., when the QPR cache is full), the query plan for s is known (drawn from cache), and the computation of effective load E^d is a constant time arithmetic operation for each of $|D|$ database instances.

Based on the above, the time complexity for our C-DBRD approach is $O(t_s + |D|te + log|D|)$, where $O(log|D|)$ is the order of complexity required to find the database instance with minimum E^d_s.

Obviously, RR has the least overhead; however, we do not expect it to perform well as compared to other request distribution methods, since it does not measure server workload in any meaningful way. Our experimental results in the next section demonstrate this clearly.

Getting the CPU or memory information for LCPU or FMEM, respectively, requires a remote call to each database instance. In contrast, in the C-DBRD approach in steady state, when the query details are in the QPR cache, no remote calls are needed; all processing consists of local in-memory hash-table access and arithmetic computation. Thus, we can say that $t_e < t_{mem}$ and $t_e < t_{cpu}$.

We can conclude that *at steady state, the computational overhead for the C-DBRD request distribution module is greater than RR, but smaller than LCPU and FMEM.* Next, we will show a comparison of these overheads for all four cases.

Experimental Study

In this section, we explore the effectiveness of our proposed approach experimentally. In our experimental results, we report two measures of effectiveness:

- *Application Response Time:* We measure the average response time of requests submitted to a web application—based on the TPC-W benchmark specification (Transaction Processing Council 2009)—as experimental conditions vary. As in our analysis of expected effectiveness in the previous section, we consider statement processing time as a proxy measure for capacity utilization in comparing different database request distribution methods.

- *Application Throughput:* We measure the average number of requests the web application can process per second under different experimental conditions.

We are specifically interested in two broad questions: first, comparing the performance of C-DBRD to other database request distribution methods, and second, measuring the sensitivity of C-DBRD to some key parameters.

First, we explore the response time and throughput of our proposed C-DBRD approach with three other approaches, borrowed directly from web server and application server request distribution techniques, in use in industry today. Specifically, we consider the RR, LCPU, and FMEM approaches that we described in our discussion of related work.

Second, we explore the sensitivity of the C-DBRD approach along three important dimensions:

- *Performance and Resource Overhead Sensitivity as Cluster Size Increases:* We describe how both response time performance and CPU usage on the C-DBRD instance vary as the number of database instances in the cluster increases to show how the C-DBRD algorithm scales. These experiments also consider the possibility that larger cluster sizes may require more than one C-DBRD to scale well; here, as cluster sizes increase, the number of C-DBRD instances increases as well.

- *Response Time Sensitivity as Database Buffer Size Increases:* We consider how response time on database instances in the cluster varies as the database buffer size (i.e., the amount of available memory for caching) increases.

- *Response Time Sensitivity as Length of History (n) Increases:* We consider how response time on database

instances in the cluster varies as n, the number of queries C-DBRD considers, increases.

- *Comparison of Overheads of C-DBRD, RR, LCPU, and FMEM:* We consider the overheads associated with running the decision logic for all four cases, including a demonstration of how C-DBRD overhead decreases as it moves from a cache-empty state (representing system startup) to a steady-state scenario, with significant amounts of data stored in the QPR.

Experimental Platform

In our experimental setup, we developed a testbed environment using off-the-shelf software as the clustered environment for our experiments. We configured a set of four Apache 2.2 instances as the web server cluster. Eight instances of WebSphere v7 with JDK 1.6 in cluster mode serve as the application server layer. Three instances of Oracle database 11g running on Windows XP serve as the database cluster. All three database servers are identical. Each web server, application server, and database server instance runs on separate hardware, configured with a dual-core 2.4 GHz CPU and 2 GB RAM on a Windows XP operating system. All machines are connected through a switched 100 Mbps Ethernet LAN.

The C-DBRD, LCPU, FMEM, and RR algorithms are implemented in C++ based on the Oracle listener architecture. (Note that every modern client-server database system is based on a similar listener architecture, so this architecture generalizes well.) The database request distributor (which runs C-DBRD, LCPU, FMEM, or RR as the experimental design dictates) runs on a Windows XP machine with a dual-core 2.4 GHz CPU and 2 GB of RAM. We implemented C-DBRD to interface with both Oracle and SQL Server databases.

We used the TPC-W benchmark from the Transaction Processing Council, a standard web benchmark for e-commerce systems, for our experiments. This benchmark simulates a bookstore, and its database contains eight tables: customer, address, orders, order line, credit info, item, author, and country. The database size is determined by the number of items in the inventory and the size of the customer population. We used 100,000 items and 2.8 million customers in our experiments, resulting in a database of about 4 GB.

The TPC-W benchmark specifies 14 different interactions. Six of the interactions are read-only, while eight cause the database to be updated. The read-only interactions include access to the home page, listings of new products and best sellers, requests for product detail and two interactions involving searches. Update transactions include user registration, updates of the shopping cart, two order-placement transactions, and two for administrative tasks. Following the focus of this research, to concentrate on the read-only queries, the six read-only interactions specified in the TPC-W workload constitute 95 percent of the browsing mix; the eight update transactions make up the remaining 5 percent of the workload. The complexity of these interactions varies widely, with interactions taking between 20 and 700 milliseconds on an unloaded machine. The complexity of queries varies widely as well. In particular, the most heavyweight read queries are 50 times more expensive than the average read query.

We used Radview's WebLoad (Radview, Inc. 2009) tool to simulate user behavior. In our experiments, each simulated WebLoad user waits for a specified think time before initiating the next interaction. The simulated user session time and the think time are generated from a distribution specified by TPC-W.

Experimental Results

In this section, we present the results of our experiments. We first compare our method to other possible choices of database request distribution methods, and then we consider the sensitivity of the C-DBRD method along three important dimensions.

Comparing Database Request Distribution Methods as Workload Increases

Figures 5 and 6 demonstrate how the response time and throughput (respectively) of the TPC-W application vary as the number of simulated users is increased for the cases of C-DBRD, RR, LCPU, and FMEM.

In general, as the number of simultaneous users increases, the response time increases for all four cases. However, the rate of increase is different for each strategy. Overall, response time increases much more slowly with an increase in user load for C-DBRD as compared to all other cases. This is the result of taking advantage of query similarity. FMEM provides the next-best performance, with faster response times than both LCPU and RR. This is due to FMEM's strategy of providing as much free buffer memory as possible. RR, as a rudimentary technique, does not take advantage of any potential performance-improvement strategy, and thus does not perform as well as C-DBRD or FMEM.

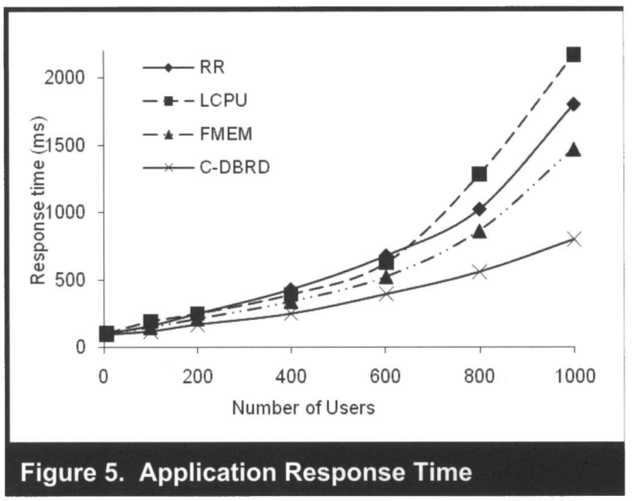

Figure 5. Application Response Time

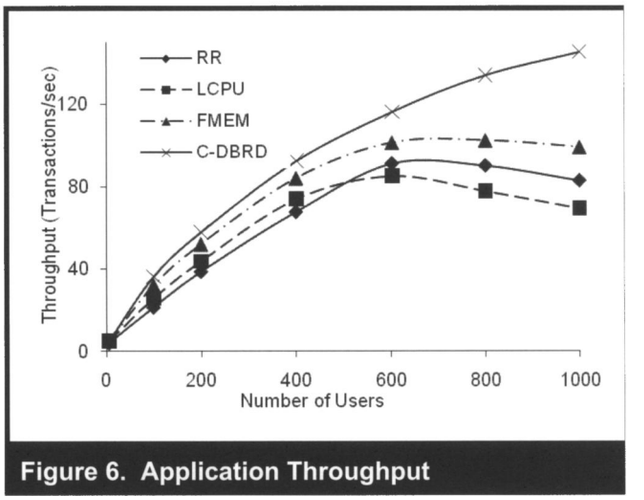

Figure 6. Application Throughput

The LCPU curve shows an interesting result: initially, it performs better than RR, but as user load increases, it provides response times higher than RR. This is because CPU represents only part of the performance impact of a query; database query processing is both CPU and I/O intensive. At low loads, when there is no resource constraint, LCPU performs slightly better than RR. However, as load increases, the curve for LCPU crosses that of RR, indicating worse performance than RR as load increases. This occurs because CPU is an indicator of only part of the processing requirement for executing queries. Here, as load increases, query processing becomes I/O-bound, leaving idle CPU cycles that do not actually indicate excess processing capacity, but rather blocked processes. At high loads, then, LCPU actually produces a highly non-optimal result, mistakenly dispatching statements to database instances that actually do not have any capacity to execute them.

To summarize, C-DBRD attempts to maximize the utilization of structures cached in database buffers. In doing so, it reduces both the CPU and I/O required for query processing. At high loads, it reduces the response time by 45 percent compared to its best-performing rival, the FMEM approach, indicating significant improvements in resource utilization with the C-DBRD method.

The throughput curves (Figure 6) for each scheme follow those of the response time experiments. Throughput is highest in case of C-DBRD—almost 50 percent more than that of its closest rival, the FMEM approach. Moreover, due to lower costs to execute queries in the C-DBRD approach, the C-DBRD does not reach its saturation point. In contrast, in cases of RR, LCPU, and FMEM, the cluster reaches saturation, where the throughput tends to decrease due to the effects of queuing on computational resources.

Sensitivity to Cluster Size

To demonstrate how our system performs with the number of database instances, we ran an experiment varying the number of database instances. Because a single C-DBRD system may not be sufficient to support the total database cluster load, we deployed multiple C-DBRD modules, configured to share common data through multicasting. Such multicasting for sharing data is a common architecture in web server and application layer cluster (Banerjee et al. 2002), with minimal CPU overhead, as we demonstrate in this experiment.

To demonstrate how our C-DBRD system scales with the number of database instances and multiple C-DBRD systems, we compare the average response time for three cases. Our baseline is the one C-DBRD system with three database instances (denoted 1-C-DBRD/3-DB). We compare our baseline with two cases: 2-C-DBRD/4-DB; and 3-C-DBRD/6-DB. We show the results of this experiment in Figure 7. Here, each cluster case is subjected to the same workload. Response times are highest for the 1-C-DBRD/3-DB cluster. Lower response times are provided in the case of the 2-C-DBRD/4-DB case; further reductions in response time are seen in the 3-C-DBRD/6-DB case. The reduced response times are a result of the availability of additional I/O and processing resources as the cluster size increases. Clearly, this indicates the horizontal scalability of our system, that is, how well our system handles additional workload as additional software resource units (C-DBRD instances) are made available, with the increase of both database instances and C-DBRD instances.

In Figure 8 we demonstrate the average CPU required for the C-DBRD module as the number of simultaneous users increases

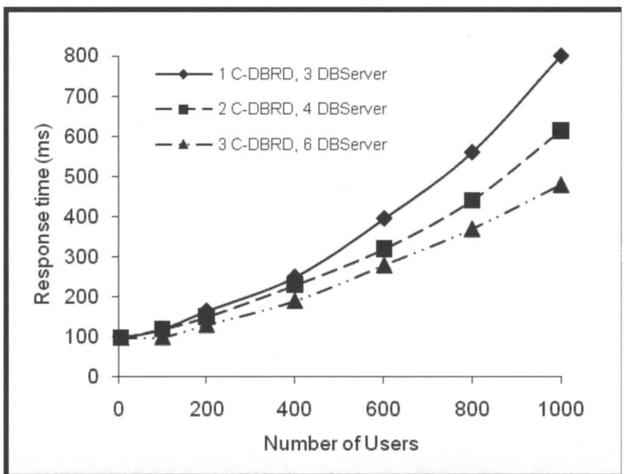

Figure 7. Sensitivity of Response Time to Cluster Size

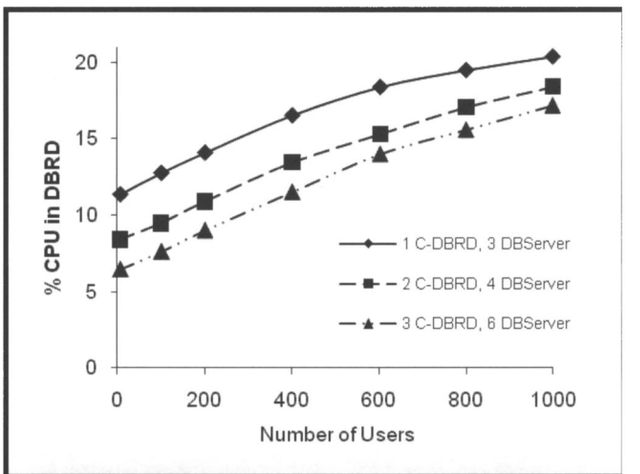

Figure 8. Sensitivity of CPU Utilization to Cluster Size

for three cluster size cases: 1-C-DBRD/3-DB; 2-C-DBRD/4-DB; and 3-C-DBRD/6-DB. Note that we do not expect such C-DBRD-to-cluster size ratios in real-life scenarios; our resource utilization experiments show that much larger cluster sizes can be supported per C-DBRD instance. We include additional C-DBRD instances here to show that multi-C-DBRD scenarios scale well.

For each case, CPU usage increases as user load increases. For all cases, however, the overall CPU requirement is very low, in the range of 10 to 20 percent even for 1,000 concurrent simulated users.

Comparatively, the CPU required for the 2-C-DBRD/4-DB instance case is lower than that of our baseline case of 1-C-DBRD/3-DB instance. Although we have increased the number of database instances in this case, thereby increasing the complexity of C-DBRD algorithm, the addition of another C-DBRD system has reduced the average CPU requirement in C-DBRD systems because the two C-DBRD modules are sharing the request load. By similar logic, the case of 3-C-DBRD/6-DB instances shows even lower CPU usage than that of the 2-C-DBRD/4-DB instance case.

Sensitivity to Database Buffer Size

Figure 9 shows how the end-to-end response time performance of the system varies for different buffer sizes. Recall that our database size is 4 GB and the buffer size for baseline experiment is 500 MB. In these experiments, we consider the database buffer sizes of 250 MB and 1 GB as well. In each case, we ran the same TPC-W workloads. Clearly, as the buffer size is reduced, the database instance needs additional I/O processing to execute queries, which increases the average response time of the system; response times are almost 50 percent higher at high loads than in the baseline case. Moreover, reducing buffer sizes effectively reduces the advantage of our C-DBRD algorithm over other algorithms by limiting the amount of space available to cache data structures. By similar logic, as the buffer size is increased, database instances require less I/O, leading to improved end-to-end response times in the system.

Sensitivity to Length of History Stored in the QPR

Figure 10 demonstrates how our algorithm is affected by varying n, the number of queries to consider in computing the similarity factor. The number of users in this experiment is kept constant at 600 with one C-DBRD and three database instances. As n is increased, generally the performance of the system improves, as demonstrated by the lower response times. However, increasing n shows diminishing returns. Beyond a certain value of n (in this experiment, for $n = 15$), increasing n actually increases the response time. This is because data structures (tables, indexes) related to older queries are deleted from database buffer to accommodate the structures of new queries. If we consider such older queries in computing similarity factors, a query may be sent to a database instance anticipating a cache hit, but the result is actually a cache miss.

We explain this in more detail with an example. Let us assume that two statements, q_1 and q_2, have executed in sequence

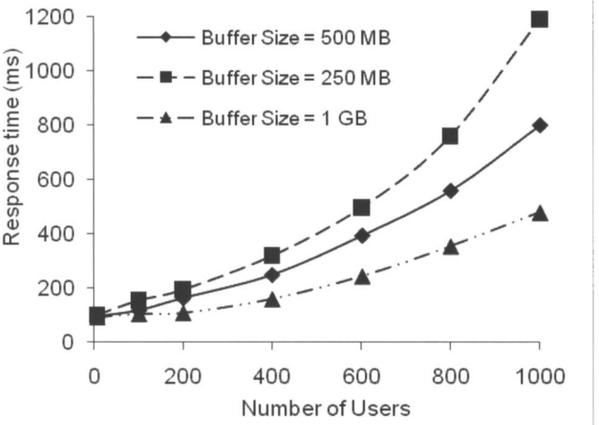

Figure 9. Sensitivity of Response Time to Database Buffer Size

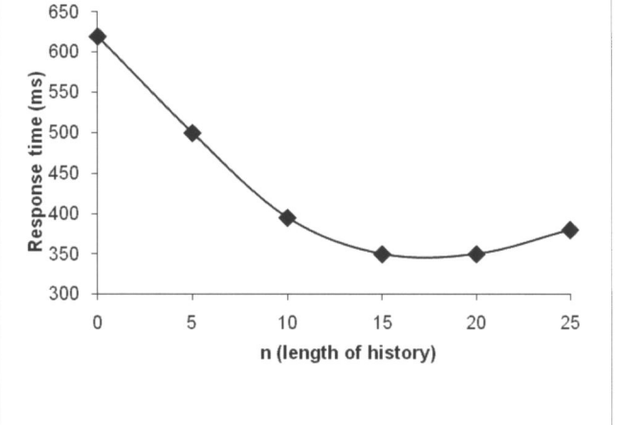

Figure 10. Sensitivity of Response Time to *n*, Length of History

in database instance d_1. Let us also assume there is a query, q_1' that is executed in the database instance d_2, and that there is a high similarity between q_1 and q_1'. If the database can cache only one statement at a time, then in the database instance d_1 the query q_2 will be cached, but not q_1. In database instance d_2, q_1' will be cached. Now, if the statement q_1 arrives again, if $n = 2$, the statement will be sent to the database instance d_1. However, in this case, the cached q_1 has already been evicted from cache, so the execution of the new q_1 cannot utilize cached data objects. However, if $n = 1$, the C-DBRD will find similarity with q_1' and will send it to the database instance d_2, where it can exploit the cache related to the query q_1. This simple example demonstrates how a high value of n can actually reduce the performance. (This is reflected in Figure 10 as well.)

Clearly, choosing a proper value of n is an important factor of our C-DBRD algorithm. In future research, we intend to extend our research work to dynamically estimate optimal values of n from the size of various structures and the buffer size.

Comparing the Overhead of C-DBRD, RR, LCPU, and FMEM

In this section, we compare the overhead of the C-DBRD, LCPU, and FMEM approaches. We devised an experiment using the TPC-W database and the experimental testbed described at the beginning of this section of the paper. Specifically, we created a list of 10,000 generated queries based on the TPC-W databases. The queries were generated by randomly selecting "fields" and "where" clauses based on the TPC-W schema. We built a Java-based client program that simulates 100 users by creating 100 parallel threads, where each thread randomly selects an SQL statement from the pregenerated 10,000 SQL statements and submits it to the database request distributor module for a request distribution decision to one of the three database instances behind the database request distributor. The database request distributor module can execute the RR, LCPU, FMEM, or C-DBRD module based on a configuration setting. We further introduced a simple timer command within the code of the database request distributor module, to measure the average time required to run the request distribution logic (including the time required to fetch needed information, such as SQL query plans, CPU loads, or memory usage from remote database nodes, as required by the respective algorithms). The average time for every 60-second interval is reported and plotted in Figure 11 for all of the four scenarios.

As is clear in Figure 11, the RR method demonstrates the lowest runtime for request distribution decision-making. (In some cases, RR showed a running time at a sub-millisecond level. These cases have been reported as execution times of 0 milliseconds in our experimental results.) This fast running time makes sense, since RR does no complex processing. The LCPU and FMEM methods each require three remote calls to the cluster databases to get the current CPU and memory utilization values, respectively. These operations have constant time complexity. In both the LCPU and FMEM cases, the average time throughout the experiment is around 50 milliseconds.

As discussed in the previous section, the overhead of the C-DBRD method depends on whether the QPR cache is filled or not. For the C-DBRD approach in these experiments, the QPR

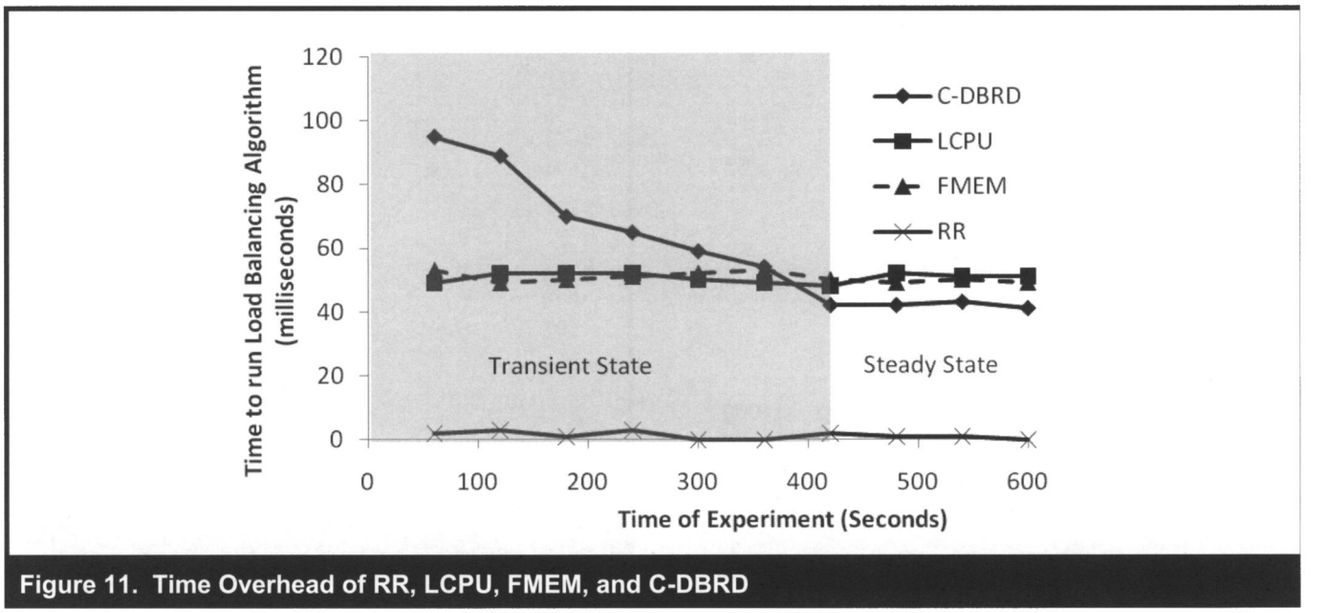

Figure 11. Time Overhead of RR, LCPU, FMEM, and C-DBRD

cache was initially empty. For each new incoming SQL statement not found in cache, the C-DBRD had to fetch the query plan from the SDB, after which the query plan was placed in the QPR cache and was available for subsequent calls for its associated query. In Figure 11, this initial cache-filling is demonstrated in higher decision-making times for C-DBRD in the early part of the experiment. However, as the time line of the experiment progresses, more and more of the needed query plans are available from the QPR cache, in which case there is no need for the C-DBRD algorithm to fetch the query plan data from the SDB. In these cases, the C-DBRD decision making involves accessing in-memory hash maps and computation as discussed in the previous section. Over the course of the experiment, this increasing availability of cached query plan information manifests as a decreasing average decision-making time for the C-DBRD case. At steady state (i.e., when the QPR cache is full), the average C-DBRD decision-making time is actually lower than that of the FMEM and LCPU cases. As discussed in the previous section, the FMEM and LCPU methods require remote calls, which involves network I/O, not CPU. In contrast, although the C-DBRD method runs a more complex algorithm then LCPU and FMEM, at steady state all the information it needs is available to the local process; no out-of-process calls or remote server calls are required. This results in slightly lower decision-making times for the C-DBRD case as compared to the FMEM and LCPU cases, when C-DBRD is running in a steady-state condition.

To summarize, *while the C-DBRD method is in a transient state, the FMEM and LCPU cases provide lower decision-making time overheads. However, when the C-DBRD is at steady state, the C-DBRD case provides lower decision-making overheads than FMEM and LCPU.*

Field Experiment

In this section, we compare the C-DBRD approach for distributing read requests with SQL Server 2005 clustering (Microsoft Inc. 2009) and RR request distribution in the staging environment of an online floral retailer. The web site, which is coded in ASP.NET 2.0, normally runs on six SQL Server database servers running on Windows Server 2003, each with a dual-core 2.3 GHz processor with 2 GB of RAM.

We first consider the C-DBRD use case. Here, the product catalog is replicated across four standalone SQL Server databases, as shown in Figure 12. The loads for browsing the product catalog are distributed to these database servers using C-DBRD request distribution. Specifically, we installed two C-DBRDs in front of the four product catalog database servers (one C-DBRD is used as primary; the other is configured as a fail-over node). The other two of the six SQL Server databases are configured as a cluster with shared storage, where both the instances are active and ready to receive transactional load, and the transactional loads are distributed by an SQL Server connection pool. These two database servers are used to store transactional data (i.e., purchase records). The four replicated catalog databases are updated at night (when load is very low) based on daily transaction records and inventory. We denote this case as "C-DBRD."

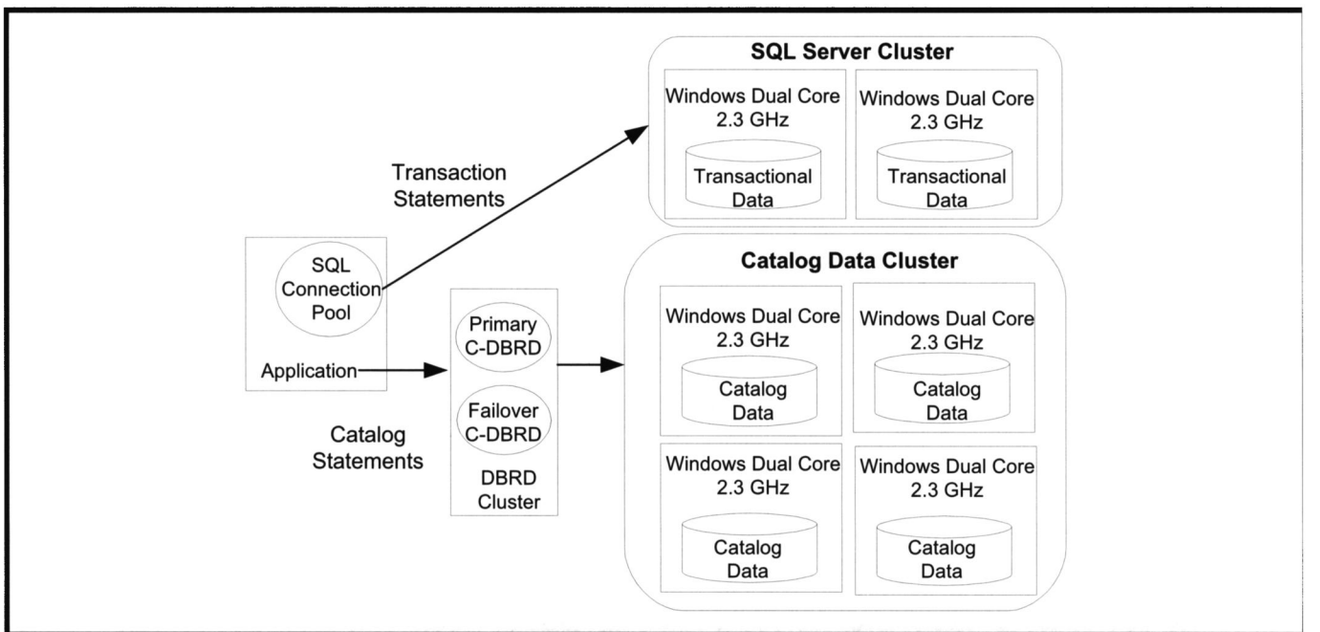

Figure 12. C-DBRD Experimental Architecture

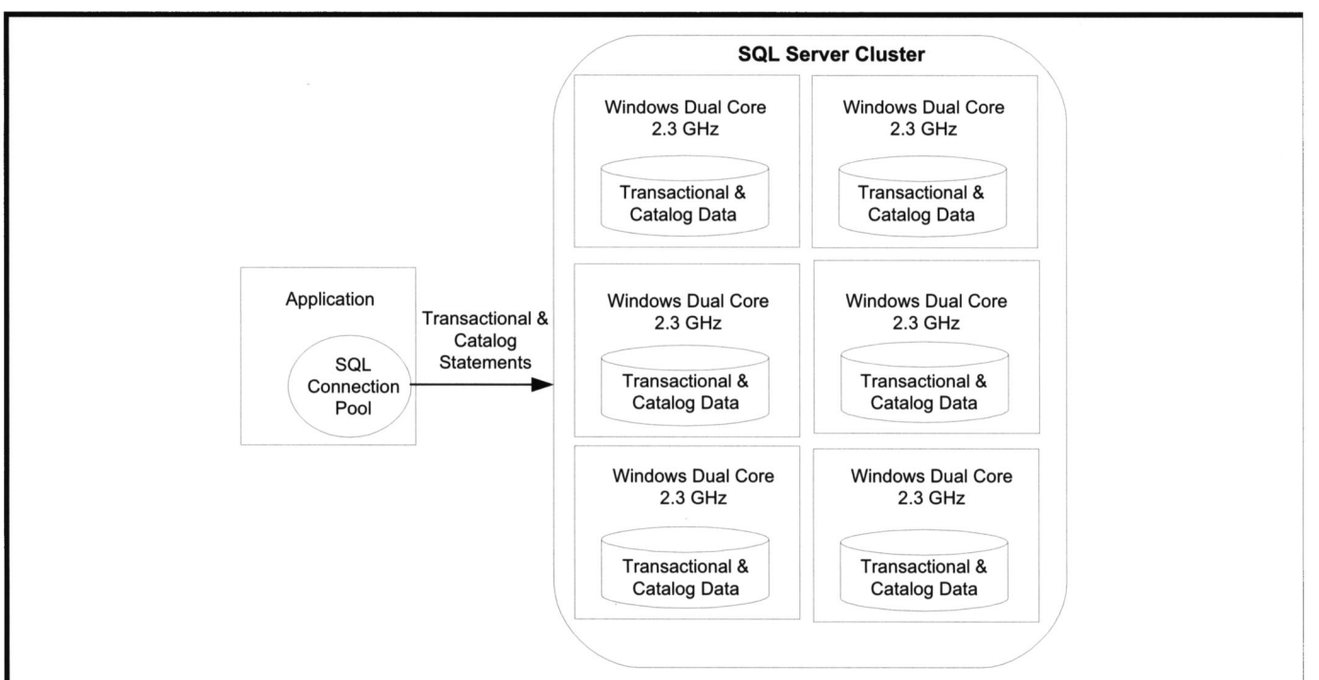

Figure 13. SQL Server Experimental Architecture

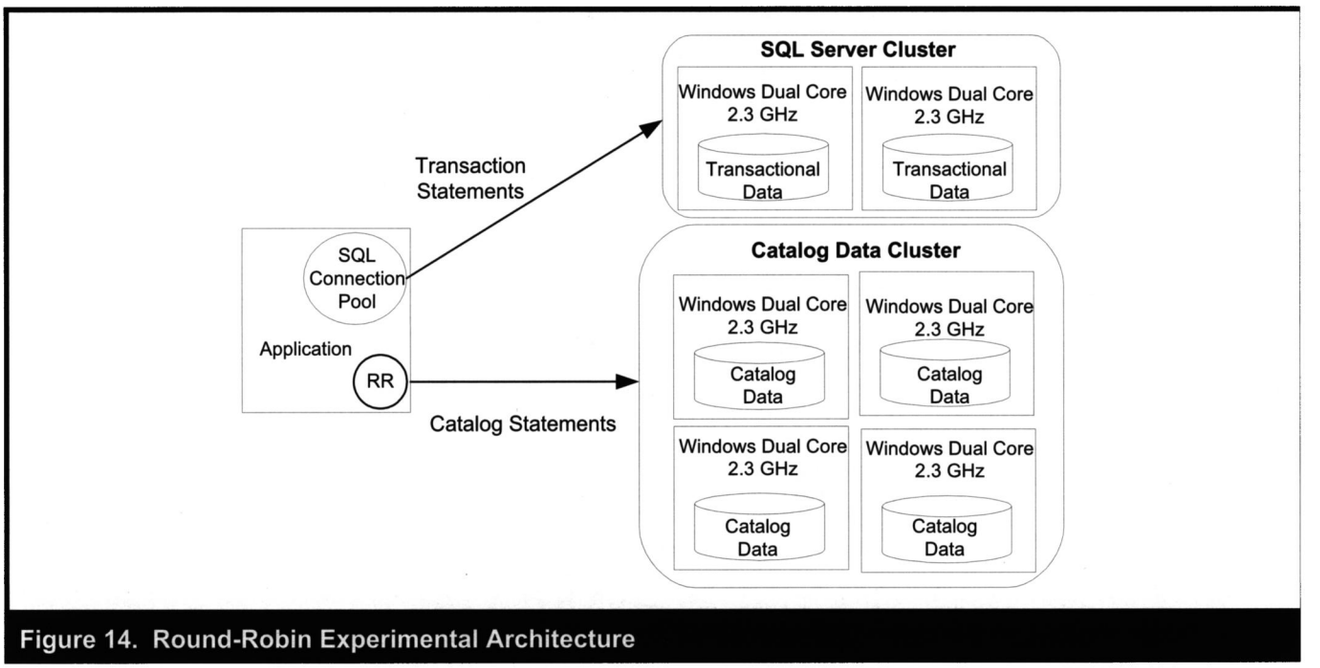

Figure 14. Round-Robin Experimental Architecture

We next describe the SQL Server clustering use case, shown in Figure 13. In this scenario, all six SQL Server instances are configured in an SQL Server cluster using shared storage (i.e., six RDBMS software instances all access the same data tables via shared storage). In this case, all six SQL Servers have access to both the product catalog database and the transactional data, which means that both read and write workloads for all six database software instances access the same data tables. The loads across all of these six SQL Server cluster instances are distributed by the SQL Server connection pool mechanism. We denote this case as "SQL Cluster."

Finally, we describe the RR case. In this case, the six SQL Server databases are configured identically to the C-DBRD case (i.e., four databases are configured for product catalog loads and two are configured for transactional loads). The only difference is in the request distribution mechanism for the product catalog loads. Here, as shown in Figure 14, the loads for browsing the product catalog are distributed to these database servers by the SQL Server connection pool using round-robin distribution. We denote this case as "RR."

The loads are simulated using LoadRunner (Mercury, Inc. 2009), and based on a day's worth of application server access log data to simulate user behavior on the site. The number of simulated users is controlled by LoadRunner. At each simulated user count of interest, we note the throughput (transactions per second) and response time (milliseconds) through the LoadRunner console. We plot the response time and throughput as the load is increased from five simultaneous users to 1,000 simultaneous users in Figures 15 and 16, respectively, for the C-DBRD, SQL Cluster, and RR cases.

Figure 15 shows how the three system configurations compare as the load is increased. Initially, C-DBRD, SQL-Cluster, and RR show virtually equivalent performance, with marginal improvement in the C-DBRD case. However, after 600 simultaneous users, the response time in the SQL-Cluster case increases rapidly, creating a difference of about 40 percent at 1,000 users. The RR case revealed similar, but less acute, increases in response time at the same load level.

In the simulated user behavior, we observed that about 87 percent of actions are related to product catalog browsing (reads) and 13 percent are related to transactions (writes). For the SQL-Cluster case, at high loads, the overhead caused by updates from purchases caused significant overheads on the database, seriously hampering the performance for browsing behavior. In contrast, in the C-DBRD case, due to the separation of catalog and transactional databases, browsing performance is not hampered. Further, the number of database servers in the cluster for transactional data is reduced to two in case of C-DBRD, as opposed to six in the case of SQL-Cluster. This further reduces the synchronization overhead of updates in the transactional database.

For the RR case, the increase in response time over the C-DBRD case is due to the fact that the RR case does not take

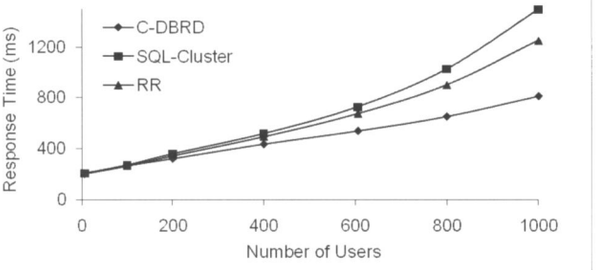

Figure 15. Field Experiment Response Time

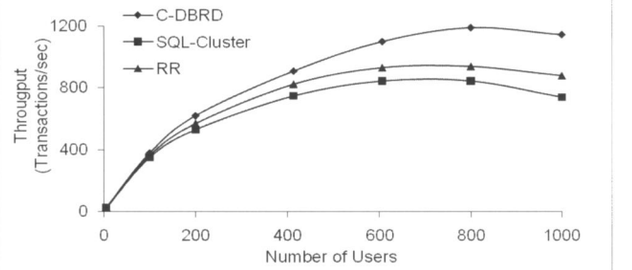

Figure 16. Field Experiment Throughput

into account the potential cached data in the database instances. These results are closely correlated to our experimental results, described earlier.

Discussion and Conclusion

In this paper, we present an effective request distribution strategy for improving resource utilization in the data layer in multitiered online enterprise applications. Specifically, we addressed three main research questions:

(1) How can we model database workloads in online multitiered applications?

(2) Based on this model, how can we design an effective request distribution mechanism for the data layer in online multitiered applications?

(3) Given such a request distribution mechanism, how does it perform against existing request distribution techniques?

Intuitively, our approach makes use of reference locality in the database cluster, by attempting to route requests to the server instance most likely to have the data resources needed for the query in its cache. As in the application layer, locality-based request distribution will improve the performance of the data layer. However, due to the nature of the data structures accessed in a database request, a cost model to quantify the potential impact of content affinity-based request distribution needed to be developed. To support this, we developed an analytical model describing database workloads from a cost perspective with two aims: (1) to model the expected cost savings for processing a statement in the presence of caching, and (2) to model the workload of a database instance, based on the statement cost savings model.

We developed an analytical comparison of the C-DBRD and RR approaches, and showed that we can expect C-DBRD to better distribute requests for improved resource utilization than RR. At worst, we expect C-DBRD to provide equivalent request distribution compared to RR when caching cannot be leveraged for improved resource utilization. We further compare the expected overhead of computing routing decisions for C-DBRD, RR, LCPU, and FMEM. We found that both LCPU and FMEM can both be expected to incur higher overheads than C-DBRD when the C-DBRD-enabled system is in steady state. RR incurs virtually no computational overhead in making request distribution decisions (i.e., we expect C-DBRD to incur higher overheads for request distribution decision making than RR). However, we expect that the improvements in statement response times using C-DBRD to overcome the differences in request distribution decision-making overhead.

We compared our approach to existing distribution strategies employed in the web server and application server layers, to extant request distribution mechanisms in the research literature, as well as those available commercially, and found that our approach performed significantly better than the other methods tested. In an experiment comparing response time and throughput performance, where improvements in these measures serve to represent the additional capacity that becomes available with the use of our method, we found that the C-DBRD method provides substantial improvements in performance at the highest workload tested, on the order of 45 percent, in comparison to the FMEM method. Futher, C-DBRD showed a 55 percent improvement over the round-robin approach, and a 63 percent improvement over the lowest CPU approach.

In a field experiment, we further tested our method in a corporate production-staging environment of a mid-size e-commerce site, comparing our method to a clustered SQL

Server configuration using shared storage, as well as the baseline RR case. Here, we found that our method provided a 40 percent improvement over the SQL Server clustering case and a 30 percent improvement over the RR case. These performance gains represent real additional capacity on database instance (i.e., existing resources can handle substantially higher workloads using our method). Considering this, our approach promises significant real-life value, in particular for read-mostly scenarios typical in large e-commerce retail applications.

We note a few limitations to take into account when assessing these results. As we mentioned in the first two sections of this paper, our focus here is on replicated databases in low-transaction scenarios; our methods are not appropriate for transaction-heavy environments. Further, we note that our results may not generalize as well for applications that generate *ad hoc* queries on the fly as a substantial portion of their workloads; we would expect lower similarity between queries, and therefore fewer opportunities to take advantage of cached data on the instances of a cluster.

IT managers may, rightfully, question the risk of adding a request distribution layer, both in terms of the possibility of C-DBRD as a single point of failure, and as a performance bottleneck itself. C-DBRD can be configured for high availability, with multiple C-DBRD instances in the request distribution layer (one primary, one for failover), or clustered (where each instance distributes requests) when a single C-DBRD instance cannot handle the full request load. Our experiments show that, even at very high workloads with multiple C-DBRD instances, C-DBRD CPU usage was very low—at most 20 percent across all cases. Since the C-DBRD method is primarily processing-oriented, there is clearly room for further workload. If a single C-DBRD instance proves unequal to the task, multiple C-DBRD instances can be configured to handle the load. Thus, there is little risk in implementing C-DBRD-based request distribution for the data layer.

When should an IT manager consider a change in request distribution technique? We are certainly not advocating changes in a working system using an existing request distribution technique: If it is not broken, there is no need to fix it. However, IT application environments and application portfolios are very dynamic, and rarely grow smaller. IT managers face a constant challenge in finding resources to support the growth of usage for existing applications and new initiatives coming online. When faced with the choice to add new infrastructure to support growth or a new application, our request distribution method offers the IT manager the opportunity to improve the efficiency of existing resources (up to 45 percent response time improvements, based on our experimental results), rather than simply installing additional infrastructure. Whether the resources made available through improved efficiency are used to support additional growth of existing applications or new initiatives, there is a clear financial benefit with the use of our method.

Once the implementation decision is made, the IT manager will need to configure a value for n, the length of history to be maintained in the QPR. Our experiments indicate that the choice of a good value of n is important in terms of C-DBRD performance. Therefore, we present some guidelines below that will be helpful in this regard.

- If the database buffer size is very small, the caching has zero impact on the C-DBRD database instance selection and the optimal value of n is 0.
- If the database buffer size is equal to or more than the total database size, the buffer can hold all database objects without any eviction, so in this scenario IT managers can select a very high value for optimal n.
- The higher the ratio of database buffer size versus the total database size, the higher the optimal value of n.

In practice, a good value for n will be dependent on the application(s) and the expected workload. An IT manager can choose a value for n by running the experiment, described in the section "Sensitivity to Length of History Stored in the QPR," in a staging environment with the application set of interest, and load testing scripts that implement expected workloads. Running a set of such experiments with increasing values for n will yield the data needed for a graph similar to Figure 10. When the response time starts to increase with a larger value of n, the optimal value for n is clear.

We believe that we have made a strong case for IT managers to consider request distribution at the data layer using C-DBRD, based on the additional capacity available using this method. Our results demonstrate substantial potential gains in resource capacity utilization for read-mostly environments in practice, since database resources will be able to handle significantly larger workloads with our method in place than without it.

References

Ahmad, M., Aboulnaga, A., and Babu, S. 2009. "Query Interactions in Database Workloads," in *Proceedings of the Second International Workshop on Testing Database Systems*, Providence, RI (http://www.cs. duke.edu/~shivnath/papers/dbtest09i.pdf).

Amini, M. M., and Racer, M. 1995. "A Hybrid Heuristic for the Generalized Assignment Problem," *European Journal of Operational Research* (87:2), pp. 343-348.

Amza, C., Cox, A. L., and Zwaenepoel, W. 2003. "Conflict-Aware Scheduling for Dynamic Content Applications," in *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems*, Seattle, WA.

Bala, M., and Martin, K. 1997. "A Mathematical Programming Approach to Data Base Normalization," *INFORMS Journal on Computing* (9:1), pp. 1-14.

Banerjee, S., Bhattacherjee, B., and Kommareddy, C. 2002. "Scalable Application Layer Multicast," in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Pittsburgh, PA, August 19-23, pp. 205-217.

Bernstein, P. A., and Goodman, N. 1981. "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* (13:2), pp. 185-221.

Bernstein, P. A., and Goodman, N. 1984. "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems* (9:4), pp. 596-615.

Birman, K., Chockler, G., and van Renesse, R. 2009. "Toward a Cloud Computing Research Agenda," *ACM SIGACT News* (40:2), pp. 68-80.

Burleson, D. K. 2009. "Cost Control: Inside the Oracle Optimizer," Burleson Consulting (http://www.dba-oracle.com/topica/cost_control_Oracle_Optimizer.htm; accessed December 1, 2009).

Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. 2001. "The State of the Art in Locally Distributed Web-Server Systems," IBM Research Division Technical Report RC22209 (W0110-048), October.

Cardellini, V., Colajanna, M., and Yu, P. S. 1999. "Dynamic Load Balancing on Web-Server Systems," *IEEE Internet Computing* (3:3), pp. 28-39.

Cecchet, E., Candea, G., and Ailamaki, A. 2008. "Middleware-Based Database Replication: The Gaps Between Theory and Practice," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*,Vancouver, BC, Canada, June 9-12, pp. 739-752.

Cherkasova, L.., and Karlsson, M. 2001. "Scalable Web Server Cluster Design with Workload-Aware Request Distribution Strategy WARD," in *Proceedings of the Third International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, San Juan, CA, June 21-22, pp. 212-221.

Chow, Y. C., and Kohler, W. H. 1979. "Models for Dynamic Load Balancing in a Heterogeneous System," *IEEE Transactions on Computing* (C-28:5), pp. 354-361.

Colajanni, M., Yu, P. S., and Dias, D. M. 1997. "Scheduling Algorithms for Distributed Web Servers," in *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, MD, May 27-30, pp. 169-176.

Datta, A., Dutta, K., Thomas, H., and VanderMeer, D. 2003. "World Wide Wait: A Study of Internet Scalability and Cache-Based Approaches to Alleviate It," *Management Science* (49:10), pp. 1435-1444.

Dutta, K., Datta, A., VanderMeer, D., Thomas, H., and Ramamritham, K. 2007. "ReDAL: An Efficient and Practical Request Distribution Technique for Application Server Clusters," *IEEE Transactions on Parallel and Distributed Systems* (18:11), pp. 1516-1528.

Dutta, K., Soni, S., Naraimhan, S., and Datta, A. 2006. "Optimization in Object Caching," *INFORMS Journal on Computing* (18:2), pp. 243-254.

Eager, D. L., Lazowska, E. D., and Zahorjan, J. 1986. "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering* (12:5), pp. 662-675.

Elnikety, S., Dropsho, S., and Zwaenepoel, W. 2007. "Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases," *ACM SIGOPS Operating Systems Review* (41:3), pp. 399-412.

Gartner Research. 2010. "Gartner Says Worldwide Cloud Services Market to Surpass $68 Billion in 2010," Press Release, (http://www.gartner.com/it/page.jsp?id=1389313; accessed January15, 2011).

Gopal, R. D., Ramesh, R., and Zionts, S. 1995. "Access Path Optimization in Relational Joins," *ORSA Journal on Computing* (7:3), pp. 257-268.

Gopal, R. D., Ramesh, R., and Zionts, S. 2001. "Cascade Graphs: Design, Analysis and Algorithms for Relational Joins," *INFORMS Journal on Computing* (13:1), pp. 2-28.

Gurley, J. W. 2000. "The One Internet Metric that Really Matters," *Fortune*, March 6 (http://money.cnn.com/magazines/fortune/fortune_archive/2000/03/06/275195/index.htm).

Haddadi, S., and Ouzia, H. 2004. "Effective Algorithm and Heuristic for the Generalized Assignment Problem," *European Journal of Operational Research* (153:1), pp. 184-190.

Harchol-Balter, M., and Downey, A. B. 1997. "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *ACM Transactions on Computer Systems* (15:3), pp. 253-285.

Hevner, A. R., March, S. T., Park, J., and Ram, S. 2004. "Design Science in Information Systems Research," *MIS Quarterly* (28:1), pp. 75-105.

Hosanagar, K., Krishnan, R., Chuang, J., and Choudhary, V. 2005. "Pricing and Resource Allocation in Caching Services with Multiple Levels of Quality of Service," *Management Science* (51:12), pp. 1844-1859.

Hua, K. A., Lee, C., and Hua, C. M. 1995. "Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning," *IEEE Transactions on Knowledge and Data Engineering* (7:6), pp. 968-983.

Hunt, G., Goldszmidt, G., King, R., and Mukherjee, R. 1998. "Network Dispatcher: A Connection Router for Scalable Internet Services," *Computer Networks* (30:1-7), pp. 347-357.

Hwang, S., and Jung, N. 2002. "Dynamic Scheduling of Web Server Cluster," in *Proceedings of the 9th International Conference on Parallel and Distributed Systems*, Chungli, Taiwan, December 17-20, p. 563.

JBoss Community. 2011. "Hibernate: Relational Persistence for Java and .Net" (http://www.hibernate.org/; accessed June 23, 2011).

Kennington, J. L., and Whitler, J. E. 1999. "An Efficient Decomposition Algorithm to Optimize Spare Capacity in a Telecommunications Nework," *INFORMS Journal on Computing* (11:2), pp. 149-160.

Krishnan, R., Li, X., Steier, D., and Zhao, L. 2011. "On Heterogeneous Database Retrieval: A Cognitively Guided Approach," *Information Systems Research* (12:3), pp. 286-301.

Laguna, M. 1998. "Applying Robust Optimization to Capacity Expansion of One Location in Telecommunications with Demand Uncertainty," *Management Science* (44:11, Part 2), pp. S101-S110.

Linux Virtual Server Project. 2004. "Linux Virtual Server" (http://www.linuxvirtualserver.org/).

Lipasti, M. H., Wilkerson, C. B., and Shen, J. P. 1996. "Value Locality and Load Value Prediction," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, October 1-5, pp. 138-147.

Mazzola, J. B., and Neebe, A. W. 1986. "Resource-Constrained Assignment Scheduling," *Operations Research* (34:4), pp. 560-572.

Melerud, P. 2010. "Server Load Balancing: Not Just for the Big Guy Anymore" Tech News World: Data Management (http://www.technewsworld.com/story/69767.html; accessed January 15, 2011).

Menasce, D. A., Saha, D., da Silva Porto, S. C., Almeida, V. A. F., and Tripathi, S. K. 1995. "Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architecture," *Journal of Parallel and Distributed Computing* (28:1), pp. 1-18.

Mercury, Inc. 2009. "Load Testing Software: Automated Software Performance Testing" (http://www.mercury.com/us/products/performance-center/loadrunner/; accessed December 1, 2009).

Microsoft, Inc. 2008. "Improving Scalability: SQL Server 2008 r2" (http://msdn.microsoft.com/en-us/library/ms151322.aspx; accessed January 5, 2011).

Microsoft Inc. 2009. "SQL Server 2005 Product Overview" (http://www.microsoft.com/sqlserver/2005/en/us/default.aspx; accessed December 1, 2009).

Microsoft, Inc. 2011. "Overview of SQL Query Analyzer" (http://msdn.microsoft.com/en-us/library/aa216945(v=SQL.80).aspx; accessed January 15, 2011).

Microsoft .NET Developer Center. 2011. "Getting Started with LINQ in Visual Basic" (http://msdn.microsoft.com/library/bb397910.aspx; accessed June 23, 2011).

Mookerjee, V. S., and Tan, Y. 2002. "Analysis of a Least Recently Used Cache Management Policy for Web Browsers," *Operations Research* (50:2), pp. 345-357.

Mulpuru, S., Hult, P., Freeman Evans, P., Sehgal, V., and McGowan, B. 2010. "US Online Retail Forecast, 2009 to 2014: Online Retail Hangs Tough for 11% Growth in a Challenging Economy," Technical Report, Forrester Research, Cambridge, MA.

Oracle. 2009. "Oracle 11g Documentation" (http://www.oracle.com/technology/documentation/ database11gr1.html; accessed December 1, 2009).

Pai, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. "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*, San Jose, CA, October 3-7.

Perez, J. C. 2009. "Gartner Offers IT Tips to Stretch E-Commerce Budget," PC World, May 11 (http://www.pcworld.com/businesscenter/article/164680/gartner_offers_it_tips_to_stretch_ecommerce_budget.html; accessed January 15, 2011).

Plattner, C., and Alonso, G. 2004. "Ganymed: Scalable Replication for Transactional Web Applications," in *Proceedings of the Fifth ACM/IFIP/USENIX International Conference on Middleware*, Toronto, Ontario, Canada, October 18-22.

Qin, X., Jiang, H., Manzanares, A., Ruan, X., and Yin, S. "Dynamic Load Balancing for I/O-Intensive Applications on Clusters," *ACM Transactions on Storage* (5:3), pp. 1-38.

Radview, Inc. 2009. "Webload Load Testing Software" (http://www.radview.com; accessed December 1, 2009).

Rahm, E., and Marek, R. "Dynamic Multi-Resource Load Balancing in Parallel Database Systems," in *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, pp. 395-406.

Roudebush, B. 2010. "Deploying a Secure Ecommerce Application in the Cloud," Ecommerce Developer, Grand Junction, CO, June 25, (http://developer.practicalecommerce.com/articles/2019-Deploying-a-Secure-Ecommerce-Application-in-the-Cloud; accessed January 15, 2011).

Schroeder, T., Goddard, S., and Ramamurthy, B. 2000. "Scalable Web Server Clustering Technologies," *IEEE Network* (14:3), pp. 38-45.

Segev, A., and Fang, W. 1991. "Optimal Update Policies for Distributed Materialized Views," *Management Science* (37:7), pp. 851-870.

Spivey, M. Z., and Powell, W. B. "The Dynamic Assignment Problem," *Transportation Science* (38:4), pp. 399-419.

Tanenbaum, A. 2001. *Modern Operating Systems*, Upper Saddle River, NJ: Prentice Hall.

Thomas, R. H. 1979. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems* (4:2), pp. 180-209.

Transaction Processing Council. 2009. "TPC Benchmarks" (http://www.tpc.org; accessed December 1, 2009).

Wang, Y. T., and Morris, R. J. T. 1985. "Load Sharing in Distributed Systems," *IEEE Transactions on Computing* (C-34:3), pp. 204-217.

Yu, P. S., Balsamo, S., and Lee, Y. H. 1987. "Dynamic Transaction Routing in Distributed Database Systems," *IEEE Transactions on Software Engineering* (14:9), pp. 1307-1318.

Yu, P. S., and Leff, A. 1991. "On Robust Transaction Routing and Load Sharing," *ACM Transactions on Database Systems* (16:3), pp. 476-512.

Zhang, X., Xiao, L., and Qu, Y. 2000. "Improving Distributed Workload Performance by Sharing Both CPU and Memory Resources," in *Proceedings of the 20th International Conference on Distributed Computing Systems*, Taipei, Taiwan, April 10-13, p. 233.

Zuikevičiūtė, V., and Pedone, F. 2008. "Conflict-Aware Load-Balancing Techniques for Database Replication," in *Proceedings of the 23rd ACM Symposium on Applied Computing*, Fortaleza, Ceará, Brazil, March 16-20, pp. 2169-2173.

About the Authors

Debra VanderMeer is an assistant professor in the Department of Decision Sciences and Information Systems in the College of Business at Florida International University. Her research interests involve applying concepts from computer science and information systems to real-world problems; her work is published widely in these fields. She holds a Ph.D. from the Georgia Institute of Technology.

Kaushik Dutta (Ph.D., Georgia Institute of Technology) is an associate professor at the School of Computing, National University of Singapore. Dr. Dutta has extensive industry experience in leading the engineering and development of commercial solutions in the area of caching, business process monitoring, and text processing/mining. He has published over 40 peer-reviewed journal and conference papers. His publications on middleware caching are some of the highly cited papers on caching. Previously, Dr. Dutta was on the faculty of Florida International University.

Anindya Datta (Ph.D., University of Maryland) is currently an associate professor in the Department of Information Systems at the National University of Singapore. Dr. Datta is a serial entrepreneur backed by tier 1 venture capitalists. His research has formed the basis of state of the art commercial solutions in database and Internet systems. He has published over 60 papers in both journals and conferences. Dr. Datta has served on the faculty at the University of Arizona and Georgia Institute of Technology.