Empirical Analysis of Database Server Scalability Using an N-tier Benchmark With Read-intensive Workload

Simon Malkowski
CERCS
Georgia Institute of Technology
266 Ferst Drive
30332-0765 Atlanta, USA
zmon@cc.gatech.edu

Deepal Jayasinghe
CERCS
Georgia Institute of Technology
266 Ferst Drive
30332-0765 Atlanta, USA
deepal@cc.gatech.edu

Markus Hedwig
Chair of Information Systems
Albert-Ludwigs-University
Platz der Alten Synagoge
79805 Freiburg, Germany
markus.hedwig@is.uni-freiburg.de

Junhee Park
CERCS
Georgia Institute of Technology
266 Ferst Drive
30332-0765 Atlanta, USA
jhpark@cc.gatech.edu

Yasuhiko Kanemasa
Fujitsu Laboratories Ltd.
1-1, Kamikodanaka 4-chome
Nakahara-ku
Kawasaki 211-8588, Japan
kanemasa@jp.fujitsu.com

Calton Pu
CERCS
Georgia Institute of Technology
266 Ferst Drive
30332-0765 Atlanta, USA
calton@cc.gatech.edu

ABSTRACT

The performance evaluation of database servers in N-tier applications is a serious challenge due to requirements such as non-stationary complex workloads and global consistency management when replicating database servers. We conducted an experimental evaluation of database server scalability and bottleneck identification in N-tier applications using the RUBBoS benchmark. Our experiments are comprised of a full scale-out mesh with up to nine database servers and three application servers. Additionally, the four-tier system was run in a variety of configurations, including two database management systems (MySQL and PostgreSQL), two hardware node types (normal and low-cost), and two database replication techniques (C-JDBC and MySQL Cluster). In this paper we present the analysis of results generated with a read-intensive interaction pattern (browse-only workload) in the client emulator. These empirical data can be divided into two kinds. First, for a relatively small number of servers, we find simple hardware resource bottlenecks. Consequently, system throughput increases with an increasing number of database (and application) servers. Second, when sufficient hardware resources are available, non-obvious database related bottlenecks have been found that limit system throughput. While the first kind of bottlenecks shows that there are similarities between database and application/web server scalability, the second kind of bottlenecks shows that database servers have significantly higher sophistication and complexity that require in-depth evaluation and analysis.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*; C.4 [**Performance of Systems**]: Performance attributes; H.2.4 [**Database Management**]: Systems—*distributed databases*

General Terms

Measurement, Performance, Experimentation, Verification.

Keywords

Bottleneck, Database replication, Distributed systems, Middleware, N-tier applications, RUBBoS.

1. INTRODUCTION

Scalability of N-tier applications in general and database servers in particular present serious challenges to both academia and industry. This is largely due to requirements such as non-stationary workloads and dependencies created by requests that are passed between web servers, application servers, and database servers. Some of these challenges have been previously referred to as "gaps between theory and practice" [10] in database replication. We attempt to start bridging these gaps with a large-scale experimental evaluation of database server scalability using a N-tier application benchmark (RUBBoS [6]). For our study we have collected more than 500GB of data in over 6,000 experiments. These experiments cover scale-out scenarios with up to nine database servers and three application servers. The configurations were varied using two relational database management systems (MySQL and PostgreSQL), two database replication techniques (C-JDBC and MySQL Cluster), workloads ranging from 1,000 to 13,000 concurrent users, and two different hardware node types.

The initial analysis of our extensive data produced several interesting findings. First, we documented detailed node-level bottleneck migration patterns among the database, application, and clustering middleware tiers when workload

(a) Software setup.

Function	Software
Web server	Apache 2.0.54
Application server	Apache Tomcat 5.5.17
Cluster middleware	C-JDBC 2.0.2
Database server	MySQL 5.0.51a PostgreSQL 8.3.1 MySQL Cluster 6.2.15
Operating system	GNU/Linux Redhat FC4 Kernel 2.6.12
System monitor	Systat 7.0.2

(b) Hardware node setup.

Type	Components		
Normal	Processor	Xeon 3GHz	64-bit
	Memory	2GB	
	Network	6 x 1Gbps	
	Disk	2 x 146GB	10,000rpm
Low-cost	Processor	PIII 600Mhz	32-bit
	Memory	256MB	
	Network	5 x 100Mbps	
	Disk	13GB	7,200rpm

(c) Sample C-JDBC topology (1/2/1/2L).

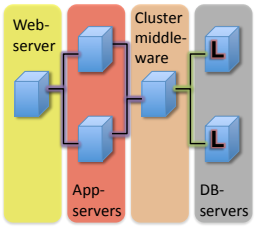


Table 1: Details of the experimental setup on the Emulab cluster.

and number of servers increased gradually. These bottlenecks were correlated with the overall system performance and used to explain the observed characteristics. Second, the explicit comparison of performance differences between varying system configurations yielded concrete configuration planning insights, which have escaped analytical approaches so far. Third, the identification of non-obvious bottlenecks showed "migratory bottleneck phenomena" that can arise in real systems and result in surprising performance effects.

The paper makes two main contributions. First, using an automated experiment creation and management system, we collected a significant amount of data on database server performance in N-tier applications. We present an experimental evaluation of database server scalability in a read-intensive scenario (i.e., setup, description, GBs of trace data, and runtime statistics). Second, our analysis of this data set shows interesting bottlenecks and their characteristics as mentioned above, which are potentially widely applicable. The detailed bottleneck migration patterns and explicit performance comparisons, for example, can be useful for datacenter administrators that manage N-tier systems by providing measured results on concrete N-tier system performance and resource utilization.

The remainder of this paper is structured as follows. Section 2 outlines experimental setup and methods. Section 3 presents the analysis of our data on database scale-out through database replication. In Section 4 we detail an interesting scenario that shows how our methodology can be used to uncover unexpected bottlenecks with surprising performance effects. Related work is summarized in Section 5, and Section 6 concludes the paper.

2. EXPERIMENTAL SETTING

2.1 Benchmark Applications

Among N-tier application benchmarks, RUBBoS has been used in numerous research efforts due to its real production system significance. Readers familiar with this benchmark can skip to Table 1(a), which outlines the concrete choices of software components used in our experiments.

RUBBoS [6] is an N-tier e-commerce system modeled on bulletin board news sites similar to Slashdot. The benchmark can be implemented as three-tier (web server, application server, and database server) or four-tier (addition of clustering middleware such as C-JDBC) system. The benchmark places high load on the database tier. The workload consists of 24 different interactions (involving all tiers) such as register user, view story, and post comments. The benchmark includes two kinds of workload modes: browse-only and read/write interaction mixes. In this paper we solely use the browse-only workload for our experiments.

Each of our experiment trial consists of three periods: ramp-up, run, and ramp-down. In our experiments, the trials consist of an 8-minute ramp-up, a 12-minute run period, and a 30-second ramp-down. Performance measurements (e.g., CPU or network bandwidth utilization) are taken during the run period using Linux accounting log utilities (i.e., Sysstat) with a granularity of one second.

2.2 Database Replication Techniques

In this subsection we briefly introduce the two different database replication techniques that we have used in our experiments. Please refer to the cited sources for a comprehensive introduction.

C-JDBC [11], is an open source database cluster middleware, which provides a Java application access to a cluster of databases transparently through JDBC. The database can be distributed and replicated among several nodes. C-JDBC balances the queries among these nodes. C-JDBC also handles node failures and provides support for check-pointing and hot recovery.

MySQL Cluster [4] is a real-time open source transactional database designed for fast, always-on access to data under high throughput conditions. MySQL Cluster utilizes a "shared nothing" architecture, which does not require any additional infrastructure and is designed to provide 99.999% data availability. In our experiment we used "in-memory" version of MySQL Cluster, but that can be configured to use disk-based data as well. MySQL Cluster uses the NDBCLUSTER storage engine to enable running several nodes with MySQL servers in parallel.

2.3 Hardware Setup

The experiments used in this paper were run in the Emulab testbed [1] with two types of servers. Table 1(b) contains a summary of the hardware used in our experiments. Normal and low-cost nodes were connected over 1,000 Mbps and 100 Mbps links, respectively. The experiments were carried out by allocating each server to a dedicated physical node. In the initial setting all components were "normal" hardware nodes. As an alternative, database servers were also hosted on low-cost machines. Such hardware typically entails a compromise between cost advantage and performance

Data metrics	Normal MySQL	Low-cost MySQL	Normal PostgreSQL	MySQL Cluster	All	All in paper
# experiments	871	1,352	1,053	416	6,318	1,638 (26%)
# nodes	15,769	18,382	17,693	6656	91,598	31,590 (34%)
# data pts.	459.7M	713.4M	555.7M	223.7M	3,334.1M	864.35M (26%)
Data size	82.8 GB	129.3 GB	70.6 GB	40.2 GB	525.6 GB	139.1 GB (26%)

Table 2: Data set size and complexity for RUBBoS experiments.

loss, which may be hard to resolve without actual empirical data.

We use a four-digit notation #W/#A/#C/#D to denote the number of web servers, application servers, clustering middleware servers, and database servers. The database management system type is either "M" or "P" for MySQL or PostgreSQL, respectively. If the server node type is low-cost, the configuration is marked with an additional "L". The notation is slightly different for MySQL Cluster. If MySQL Cluster is used, the third number (i.e., "C") denotes the number of MySQL servers and the fourth number (i.e., "D") denotes the number of data nodes. A sample topology of an C-JDBC experiment with two client nodes, one web server, two application servers, one clustering middleware server, two low-cost database servers, and non-specified database management system (i.e., 1/2/1/2L) is shown in Table 1(c). It should be noted that we solely use one web servers in all our experiments because almost all RUBBoS content consist of dynamic web pages, which do not stress the web tier.

2.4 Experimental Infrastructure

We have run a very high number of experiments over a wide range of configurations and workloads. Table 2 shows the magnitude and complexity of a typical RUBBoS experimentation cycle. Solely for the analysis shown in this paper, we have run 1,638 experiments, which produced around 139GB of data. For each of the three hardware types and database management system configurations, the total experimental data output averaged at around 277,841,000 one second granularity system metric data points (e.g., network bandwidth and memory utilization) in addition to higher-level monitoring data (e.g., response times and throughput). In the course of conducting our experiments, we have used (both concurrently and consecutively) 91,598 testbed hardware nodes. Our data warehouse is filled with around 3,334,100,000 system metric data points. In order to investigate system behavior as low as SQL query level, we have modified (if necessary) the original source code of all software components to record detailed accounting logs. Because an empirical analysis of the experimental results showed that detailed logging can affect overall performance up to 8.5 percent, we additionally implemented effective sampling algorithms to minimize the logging performance impact. We should mention that all system performance measurements in this paper (i.e., throughput and response time) were collected without such detailed logging turned on. The latter was solely used for specific scenario analysis.

Although the deployment, configuration, execution, and analysis scripts contain a high degree of similarity, the differences among them are subtle and important due to the dependencies among the varying parameters. Maintaining these scripts by hand is a notoriously expensive and error-prone process. To enable experimentation at this scale, we employed an experimental infrastructure created for the Elba project [3] to automate system configuration management, particularly in the context of N-tier system staging. The Elba approach divides each automated staging iteration into steps such as converting policies into resource assignments, automated code generation, benchmark execution, and analysis of results. The automated analysis of results for bottleneck detection is outlined briefly below.

3. SCALE-OUT EXPERIMENTS

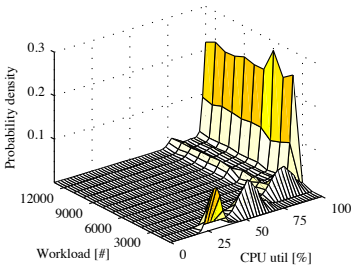
In the following we present RUBBoS scale-out experiments that increase the number of database (and application) servers gradually to find all primary bottlenecks for each configuration. In Subsections 3.1 and 3.2 we show C-JDBC experiments with MySQL and PostgreSQL on normal server nodes, respectively. Subsection 3.3 evaluates system characteristics when using low-cost nodes for the deployment of C-JDBC MySQL servers. An important property of such low-cost nodes (see Table 1(b)) is that they impose resource constraints, which especially highlight the scalability characteristics of the database server. In Section 3.4 we first evaluate the general characteristics of MySQL Cluster and additionally discuss system behavior with and without database partitioning. In Subsection 3.5 we explicitly compare the choices of database management system and hardware type for the C-JDBC experiments to show both commonalities and differences between the four investigated scenarios.

We should point out that due to smart memory management components (e.g., buffer manager) in typical database management systems (including MySQL and PostgreSQL), the database server uses adaptive approaches to fully utilize all available physical memory, so "raw" memory bottlenecks such as high paging activity are avoided. In this study, we analyze bottlenecks directly observable at the operating system level such as high CPU or disk utilization. The evaluation of impact of memory constraints on database servers is a topic for future research.

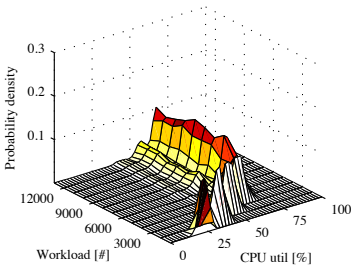
3.1 C-JDBC MySQL on Normal Servers

For this subsection we collected data from C-JDBC RUBBoS scale-out experiments with MySQL database servers and normal hardware. The experiments start from 1/1/1/1M and go to 1/3/1/9M. For each configuration, we increase the workload (i.e., number of concurrent users) from 1,000 up to 13,000 in steps of 1,000. The collected experimental data include overall response, throughput and other system statistics.

Classic bottleneck analysis employs average values to investigate resource saturation states. Such results are then used to support claims concerning service-level performance, throughput characteristics, and hardware demands. However, average numbers can mask the variations introduced by non-stationarity in workloads, which are an inherent characteristic of N-tier systems. We address this statistical threat through kernel density estimation to display the actual dis-



(a) Density graph of 1/1/1/1M (CPU saturated).



(b) Density graph of 1/1/1/2M (CPU not saturated).

Figure 1: MySQL database server CPU utilization for two C-JDBC configurations on normal nodes.

tribution of resource utilization values. Densities intuitively indicate the percentage of time each resource spent at a given utilization level. Dominant peaks at high percentiles of resource utilization (e.g., above 95 percent utilization) imply that the resource spent a significant fraction of time in a fully saturated state.

Figures 1(a) and 1(b) show the density (z-axis) of the database server CPU utilization (x-axis) as the workload grew from 1,000 to 13,000 users (y-axis). The two graphs illustrate the CPU utilization at the first database server for the 1/1/1/1M and 1/1/1/2M configurations, respectively. Both utilization density graphs start from a near-normal shape at low load (1,000 users), and the locations of their inflection points corresponds to the inflection points of the respective system throughput graphs. While the graph in Figure 1(b) does not reach critical utilization values, the CPU utilization in the 1/1/1/1M configuration (Figure 1(a)) saturates at a workload of 4,000 users. Past that workload the majority of values lay in the top five percent utilization interval. This analysis of experimental data supports the hypothesis that the database CPU is the system bottleneck only in the first of the two examined cases.

Due to the amount of collected data, we use two kinds of simplified graphs (e.g., Figures 2(a) and 3(a)) to highlight performance changes and bottleneck shifting. Figure 3(a) shows the highest achievable throughput (Z-axis) for experiments with varying configurations. We can again recognize the three groups in Figure 3(a). The first group consists of configurations 1/1/1/1M, 1/2/1/1M, and 1/3/1/1M, which show similar maximum throughput for one database server (X-axis). The second group consists of all remaining configurations with one application server (Y-axis). These configurations have a similar maximum throughput level, which is significantly higher than the throughput level of the first group. The third group has the highest maximum throughput level and consists of the other 16 configurations (i.e, at least two application servers and two database servers).

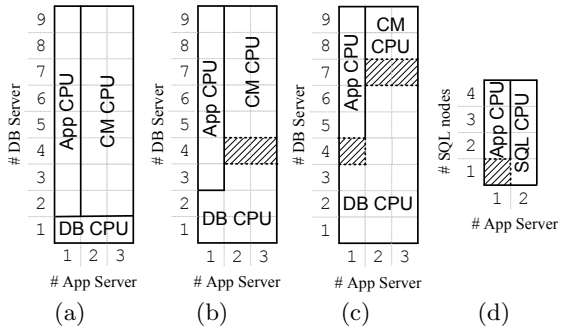


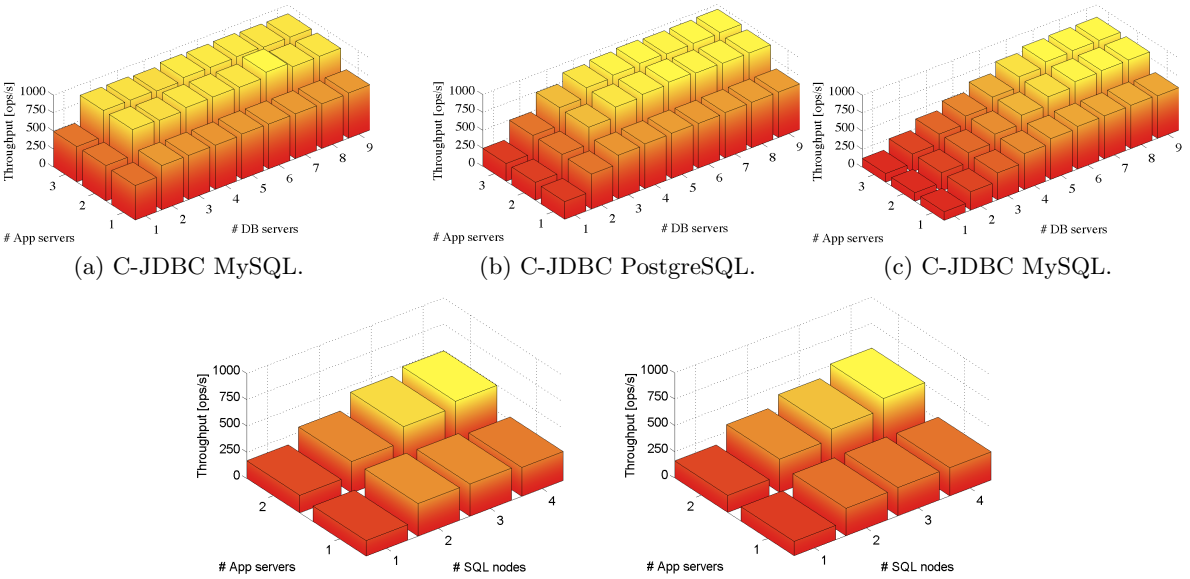
Figure 2: Bottleneckmaps: (a) MySQL, C-JDBC, normal nodes; (b) PostgreSQL, C-JDBC, normal nodes; (c) MySQL, C-JDBC, low-cost nodes; (d) MySQL Cluster, normal nodes.

Figure 3(a) can also be used to illustrates some of the typical bottlenecks observed in the RUBBoS benchmark under browse-only workload. The increase in throughput when adding a database server (from the first group to the second group) indicates that the database server was the bottleneck for the first group. Similarly, the throughput increase when adding an application server (from the second group to the third group) indicates the application server was the bottleneck for the second group. The lack of throughput increase within the third group indicates a bottleneck elsewhere (not in the database or application tiers) or a more complex bottleneck scenario. These observations and our analysis of all density graphs are summarized in Figure 2(a), which shows three regions corresponding to the three groups. The bottleneck in the third group is attributed to the C-JDBC (clustering middleware) server CPU.

In general, the RUBBoS benchmark places high load on the database server in the browse-only mode. In these scale-out experiments, we have chosen full replication of database servers so any one of them can answer every query. Full data replication provides the best support for read-only workload, but it requires consistency management with the introduction of updates in a read/write mixed workload. However, due to spacing constraints the latter is beyond the scope of this paper.

3.2 C-JDBC PostgreSQL on Normal Servers

We have performed experiments with similar setup as in Subsection 3.1 with the PostgreSQL DBMS. The summary of the results is contained in Figures 2(b) and 3(b). An explicit comparison of the results of both DBMS studies is subject of Section 3.5. Figure 3(b) shows that PostgreSQL and MySQL share some similar throughput characteristics (e.g., the three groups from Figure 3(a)) but different scalability characteristics. The first group contains all configurations with up to three database servers except the 1/1/1/3P case. The group is the result of a database server CPU bottleneck. The second group results from an application server CPU bottleneck and contains all configurations with a single application server and between three and nine database servers. The third group is caused by the clustering middle-



(a) C-JDBC MySQL. (b) C-JDBC PostgreSQL. (c) C-JDBC MySQL.

(d) MySQL Cluster with 2 data nodes. (e) MySQL Cluster with 4 data nodes.

Figure 3: Maximum system throughput for RUBBoS systems with: (a)-(b) normal nodes; (c) low-cost nodes; and (d)-(e) normal nodes in the database tier.

ware CPU bottleneck for more than three database servers and more than one application server. Interestingly, for four database servers, the two configurations with more than one application server can be attributed to both groups because they simultaneously saturate the CPUs in the two dernier tiers. The complete bottleneck analysis is summarized in Figure 2(b).

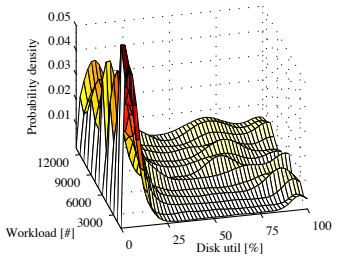
3.3 C-JDBC MySQL on Low-cost Servers

In the following we analyze data from RUBBoS scale-out experiments with MySQL on low-cost hardware. The summary of our experimental results (i.e., maximum throughput and bottleneck behavior) is shown in Figures 2(c) an 3(c). The maximum throughput of low replication configurations can simply be increased by replicating the backend as shown in Figure 3(c). Consistent with our previous discussion, this is caused by a database CPU bottleneck in all configurations with less than four database servers, and the bottlenecks can be resolved by adding a server node to the database tier. However, the performance gain of additional database servers with just one application server diminishes significantly in the 1/1/1/4ML configuration because the primary bottleneck starts to migrate to the application server tier. For all higher database replication states (i.e., at least five database servers and only one application server) the maximum throughput remains stable. For all configurations with at least two application servers, the system performance is scalable up to seven database nodes. After that point the primary bottleneck starts shifting from the backend tier to the clustering middleware server. Once the CPU of the C-JDBC server determines the system throughput, the addition of database and application servers has no significant performance effect anymore.

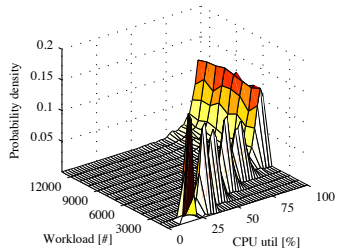
3.4 MySQL Cluster on Normal Servers

In order to explicitly investigate the effects of different replication technology approaches, we have run a similar setup of experiments as discussed in Subection 3.1 with MySQL Cluster in lieu of C-JDBC. First, we have run the experiments without partitioning the database, which implies two data nodes[1] and one management node, while increasing the number of MySQL servers from one to four and varying between one or two application servers. The summary of the results is contained in Figures 2(d) and 3(d). As illustrated in Figure 3(d), the maximum throughput that can be achieved is seemingly low compared to the same number of nodes when using C-JDBC MySQL. An intuitive assumption is that the system is bottlenecked by the number of data nodes. Therefore, we have partitioned the database into two parts using the MySQL Cluster default partitioning mechanism and repeated the set of experiments. The performance summary is shown Figure 3(e). However, the comparison of the data in Figures 3(d) and 3(e) reveals that increasing the number of data nodes does not significantly increase the maximum throughput. Consequently, the empirical analysis shows that the system is not bottlenecked in the data nodes when exposed to browse-only workload. The explanation for this behavior is that in MySQL Cluster, MySQL server acts as a light weight server invoking little or no disk IO. In contrast, the data nodes act as the backend data store and handle all the disk IO and data synchronization. Nonetheless, for browse-only workload the SQL nodes are significantly more stressed than the data nodes because there is no writing or synchronization overhead. As a result, the MySQL servers do not have to wait for the data nodes even under heavy load. This performance characteristic is

[1]MySQL Cluster has a restriction that the number of data nodes can not be increased without spiting the dataset.



(a) 1/2/1/9ML first database disk utilization density (mostly not saturated).



(b) Cluster middleware CPU utilization density (saturated after 6000 users).

Figure 4: Analysis of C-JDBC 1/2/1/9ML (8000 users) to demonstrate the influence of long SQLs on MySQL performance.

summarized in Figure 2(d), which is the correct bottleneck map for both cases with and without data partitioning with two and four data nodes, respectively.

3.5 C-JDBC Configuration Comparison

The previously discussed C-JDBC performance figures showed that the overall system reaches a similar maximum throughput level of around 900 interactions per second for both MySQL and PostgreSQL. However, bottleneck and performance characteristics are very different between the two DBMSs for lower replication numbers. Figures 3(a) and 3(b) depict the fraction of MySQL throughput that was achieved in the PostgreSQL experiments. PostgreSQL is only able to match around 50 percent of the MySQL throughput when the backend is not replicated. When more database nodes are added the throughput ratio grows (i.e., the performance of PostgrSQL improves relatively). This is caused by the MySQL system becoming bottlenecked outside the database tier for all configurations with more than one database node. Figures 3(a) and 3(b) indicate that after adding the fourth database, the PostgreSQL system even gains a slight performance advantage of around five percent. In general, these figures clearly illustrate the consequences of the clustering middleware (i.e., C-JDBC) bottleneck and how its saturation limits the entire system regardless of the two DBMSs.

Next, we investigate a different hardware choice commonly known from practice. Figures 3(a) and 3(c) allow the performance comparison of normal and low-cost nodes in the C-JDBC MySQL scenarios. For low replication numbers, the low-cost nodes are not able to match the performance of the normal nodes. In case of a single database node, the performance ratio can be as low as 30 percent. However, similarly to the previous comparison, a rapid relative performance improvement is observable when adding more backend nodes. With more than five database servers in the one application server case and more than seven databases in the other cases, the low-cost nodes come within a five percent range of the normal hardware performance. These performance characteristics can again be explained with the underlying bottleneck behavior. More generally, if the same primary bottleneck determines the system throughput, the performance implications of different submodules may diminish significantly.

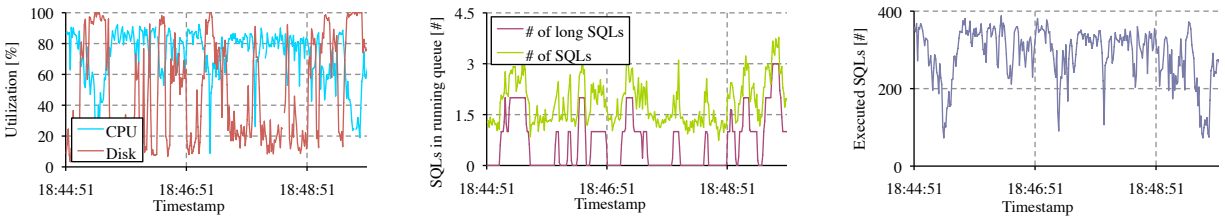
In essence, these results explicitly reveal the performance differential that is necessary to asses the utility of one hardware choice over the other. When combined with cost estimates of these configurations, these data can directly be applied in configuration planning, which is a good use-case example for our approach. In fact, these results also document the threat of assuming a single-class workload in performance models. Under such an assumption some of the observed characteristics (e.g., maximum throughput delta) are not explicable. Therefore, the results presented in this section can typically not be obtained analytically.

4. MIGRATORY BOTTLENECKS FOR LONG READ-QUERIES

Implementation specific bottlenecks may be temporarily-limited and are caused by certain design and configuration choices, which makes database scale-out performance a highly non-linear function of the number of servers. In our experiments we have analyzed a number of implementation and configuration specifics that cause a small number of queries to have five to six orders of magnitude longer execution times. However, due to the space constraints of this article, we solely include one of them in the following. This case demonstrates that these problematic queries have a highly significant effect on overall system performance despite their extremely low frequency. Such queries were difficult to identify since they are similar to normal queries both syntactically and semantically. Additionally, the described phenomena only crystallize if the primary system bottleneck moves to a different tier in higher replication states. This further emphasizes the benefits of our approach and the necessity of large-scale experimental research.

Here, we analyze "heavy read queries" in the 1/2/1/9ML C-JDBC configuration and find overly long read queries on MySQL low-cost nodes causing a problematic load-balancing effect in the clustering middleware. Note that such phenomena are impossible to identify when resource utilization is aggregated in average numbers.

The density graphs in Figures 4(a) and 4(b) show how the database CPU bottleneck migrates away when hardware nodes are added to the backend. Figure 4(a) shows the database CPU utilization density in the 1/1/1/2ML case, and the resource is clearly saturated for the majority of the experiment time. Figure 4(b), on the other hand, depicts the database CPU utilization with additional hardware (i.e., 1/2/1/9ML), and the dominant utilization level has dropped to around 80 percent. Additionally, the graph has become noticeably flattened with a fat lower tail, which implies that there are significant variations in the CPU utilization during the experiment runtime. An examination of the disk utilization in the 1/2/1/9ML configuration (Figure 5(a)) reveals that there are noticeable periods of full resource saturation



(a) Resource utilization. (b) SQLs in running queue. (c) Executed SQLs.

Figure 5: Analysis of first database in 1/2/1/9ML C-JDBC (8000 users) to demonstrate the influence of long SQLs on MySQL performance.

(i.e., graph is ragged at high percentiles). Meanwhile, the primary system bottleneck is located in the clustering middleware CPU (Figure 4(b)). The underlying reason for these observations is that certain RUBBoS queries fail to be optimized under specific conditions. These queries are search queries on the biggest table ("old_comments") in RUBBoS, and there are two conditions, under which MySQL fails to optimize them. First, the length of the search keyword for the column "subject" exceeds ten characters due to a faulty indexing schema in RUBBoS, which only uses the first ten characters to create the index. Second, the specified offset is more than zero. If a search query satisfies either of these conditions, MySQL fails to optimize it. The following is an example of a query that satisfies both conditions and causes the described phenomenon in on our environment.

```
SELECT
   old_comments.id, old_comments.story_id,
   old_comments.subject, old_comments.date,
   users.nickname
FROM old_comments, users
WHERE
   subject LIKE 'independently%' AND
   old_comments.writer = users.id
ORDER BY date DESC
LIMIT 25 OFFSET 25;
```

If a query cannot be optimized, the entire table is retrieved for brute force processing. In the case of low-cost nodes, this leads to extremely long response time because the table is retrieved from disk. In Figure 5(a) the effect of these queries is identifiable in disk I/O intervals at 100 percent bandwidth utilization. However, despite the intuitive assumption that such disk I/O is the direct cause for the correlated drop in CPU utilization, we find the actual reason for low overall throughput to lay in the load-balancing policy of C-JDBC. The latter allocates load to the database servers according to the number of running SQLs on each node. If a certain server is assumed loaded, the load-balancer moves to the next higher replica number. (This strategy usually achieves higher performance than round-robin.) When "heavy read queries" arrive at the database management system, the long disk access times cause the average number of SQLs in the running queue to increase (see Figure 5(b)). Although this does not imply that the CPU is overloaded, the load-balancing algorithm in the clustering middleware distributes the load to other databases. Consequently, the number of executed SQLs drops rapidly (Figure 5(c)), which is strongly correlated with the CPU consumption in Figure 5(a).

One conclusion from these observations is that load-balancing of multiple databases is a difficult problem when more than one resource, such as the CPU and the disk, are saturated. A sophisticated algorithm is necessary for a load-balancer to handle more than one independent resources efficiently, especially in the case of non-stationary request streams. Monitoring the number of active SQLs is not sufficient to resolve such more complex bottleneck scenarios.

It is important to realize that this characteristic behavior can be directly attributed to the load-balancing in the clustering middleware and consequently only occurs if the database is not the primary system bottleneck. In the presented scenario, this means that the C-JDBC node's CPU is the primary bottleneck (Figure 4(b)), and the database servers (predominantly) operate under unsaturated conditions at around 80 percent CPU utilization (Figure 4(b)).

5. RELATED WORK

Cloud computing, data center provisioning, server consolidation, and virtualization have become ubiquitous terminology in the times of ever-growing complexity in large-scale computer systems. However, database replication continues to fall short of real world user needs, which leads to continuously emerging new solutions and real world database clusters to be small (i.e., less than 5 replicas) [10]. In this paper we address this shortcoming explicitly. This work extends our previous approaches to empirical system analysis and bottleneck detection. Initially, we have analyzed our data for complex bottleneck phenomena. We have introduced terminology that explicitly distinguishes single-bottlenecks and multi-bottlenecks [19]. We have also investigates different ways of evaluating the economical implications of large-scale empirical system analysis. With the help of this investigation we have developed a configuration planning tool (i.e., CloudXplor) that is able to facilitate iterative and interactive configuration planning in clouds [18].

The work that is closest to this paper follows a similar observational approach to system analysis and evaluation [17]. While the context and framing of the two publications are closely related, the contributions are fundamentally different. More concretely, the evaluated datasets are completely disjunct. Where this paper focuses solely on read-only workload, our previous work [17] was exclusively concerned with workload that exhibits read/write access patterns to the database. Consequently, the findings, conclusions, and contributions are independent of each other.

There exist many popular solutions to database replica-

tion (i.e., IBM DB2 DataPropagator, replication in Microsoft SQL Server, Sybase Replication Server, Oracle Streams, replication in MySQL). Among others there are academic prototypes that use master-slave replication such as Ganymed [21] and Slony-I [2]. In this work we focus on multi-master architectures, which are used in commercial products such as MySQL Cluster and DB2 Integrated Cluster. Academic multi-master prototypes are C-JDBC [11], which is used in this work or Tashkent [13]. Further efforts focus on adaptive middleware to achieve performance adaptation in case of workload variations [20].

The evaluation of database replication techniques often falls short of real representability. There are two common approaches to evaluation with benchmarks. Performance is either tested through microbenchmarks or through web-based distributed benchmark applications such as TPC-W, RUBiS, and RUBBoS [8]. These benchmark applications are modeled on real applications (e.g., Slashdot or Ebay.com), offering real production system significance. Therefore this approach has found a growing group of advocates ([11–13, 21, 22]).

6. CONCLUSION

We used techniques and tools developed for automated experiment management, including automatically generated scripts for running experiments and collecting measurement data, to study the RUBBoS N-tier application benchmark with read-intensive workload. The data collected over a wide range of settings and system configurations was analyzed using statistical tools such as kernel density estimation, identifying hardware resource bottlenecks, and producing detailed bottleneck migration patterns as the number of servers and workload increase gradually. When there are sufficient hardware nodes, non-obvious bottlenecks were found through careful hypothesis generation, followed by confirmation through data analysis. An example is the finding of temporary bottlenecks caused by PostgreSQL's differentiated processing of SQL queries with more than nine times self-joins, which seems to take several orders of magnitude more CPU.

Our results suggest that further study (e.g., through refined analysis of our data or new experimental data) may bring new understanding of database server scale-out in N-tier applications. In addition, techniques used in the "tuning" of database management systems to improve performance may be useful in the identification of internal bottlenecks that can take a critical role in achieving service level objectives in mission critical applications.

Our current work involves a similar analysis as presented in this paper on data generated with read/write workload. Our midterm goal is to compare the presented middleware-based replication strategy to DBMS-internal clustering.

7. ACKNOWLEDGMENTS

This research has been partially funded by National Science Foundation grants ENG/EEC-0335622, CISE/CNS-0646430, CISE/CNS-0716484, AFOSR grant FA9550-06-1-0201, NIH grant U54 RR 024380-01, IBM, Hewlett-Packard, Wipro Technologies, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

8. REFERENCES

[1] Emulab - Network Emulation Testbed. `http://www.emulab.net`.

[2] Slony-I. `http://www.slony.info`.

[3] The Elba project. `www.cc.gatech.edu/systems/projects/Elba`.

[4] MySQL Cluster. `http://www.mysql.com/products/database/cluster`.

[5] RUBBoS: Bulletin board benchmark. `http://jmob.objectweb.org/rubbos.html`.

[6] RUBBoS: Bulletin board benchmark. `http://jmob.objectweb.org/rubbos.html`.

[7] C. Alexopoulos. Statistical analysis of simulation output: state of the art. In *WSC '07*.

[8] C. Amza, A. Ch, A. L. Cox, et al. Specification and implementation of dynamic web site benchmarks. In *In 5th IEEE Workshop on Workload Characterization*, 2002.

[9] G. Balbo and G. Serazzi. Asymptotic analysis of multiclass closed queueing networks: multiple bottlenecks. *Perform. Eval.*, 30(3):115–152, 1997.

[10] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *SIGMOD '08*.

[11] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX '04*.

[12] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Partial replication: Achieving scalability in redundant arrays of inexpensive databases. *Lecture Notes in Computer Science*, 3144:58–70, July 2004.

[13] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. *SIGOPS Oper. Syst. Rev.*, 40(4):117–130, 2006.

[14] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1996.

[15] B. E. Hanson. Bandwidth selection for nonparametric distribution estimation. `http://www.ssc.wisc.edu/~bhansen/papers/wp.htm`.

[16] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley & Sons, Inc., New York, NY, USA, 1991.

[17] S. Malkowski, M. Hedwig, D. Jayasinghe, et al. A New Perspective on Experimental Analysis of N-tier Systems: Evaluating Database Scalability, Multi-bottlenecks, and Economical Operation. In *CollaborateCom '09*.

[18] S. Malkowski, M. Hedwig, D. Jayasinghe, C. Pu, et al. CloudXplor: A Tool for Configuration Planning in Clouds Based on Empirical Data. In *SAC '10*.

[19] S. Malkowski, M. Hedwig, and C. Pu. Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks. In *IISWC '09*.

[20] J. M. Milan-Franco, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme. Adaptive middleware for data replication. In *Middleware '04*.

[21] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[22] C. Pu, A. Sahai, J. Parekh, et al. An observation-based approach to performance characterization of distributed n-tier applications. In *IISWC '07*, September 2007.