Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases

Sameh Elnikety Steven Dropsho Willy Zwaenepoel

School of Computer and Communication Sciences
EPFL
Switzerland

ABSTRACT

We present a *memory-aware* load balancing (MALB) technique to dispatch transactions to replicas in a replicated database. Our MALB algorithm exploits knowledge of the working sets of transactions to assign them to replicas in such a way that they execute in main memory, thereby reducing disk I/O. In support of MALB, we introduce a method to estimate the size and the contents of transaction working sets. We also present an optimization called *update filtering* that reduces the overhead of update propagation between replicas.

We show that MALB greatly improves performance over other load balancing techniques – such as round robin, least connections, and locality-aware request distribution (LARD) – that do not use explicit information on how transactions use memory. In particular, LARD demonstrates good performance for read-only static content Web workloads, but it gives performance inferior to MALB for database replication as it does not efficiently handle large requests. MALB combined with update filtering further boosts performance over LARD.

We build a prototype replicated system, called Tashkent+, with which we demonstrate that our MALB and update filtering techniques improve performance of the TPC-W and RUBiS benchmarks. In particular, in a 16-replica cluster and using the ordering mix of TPC-W, MALB doubles the throughput over least connections and improves throughput 52% over LARD. MALB with update filtering further improves throughput to triple that of least connections and more than double that of LARD. Our techniques exhibit super-linear speedup; the throughput of the 16-replica cluster is 37 times the peak throughput of a standalone database due to better use of the cluster's memory.

Categories and Subject Descriptors

H.2.4 [**Systems**] – *distributed databases, concurrency.*

General Terms

Measurement, Performance, Design.

Keywords

Database replication, Load balancing.

1. INTRODUCTION

Database replication on a cluster of servers is a cost-effective approach for scaling databases. Recent research prototypes [ACZ03-2, DS06, EDP06, LKPJ05, PA04, PJKA05, ZP06] show promise for high scalability using tens of database replicas.

With conventional database replication on a cluster, client requests are intercepted by a load balancer which hides the replicated nature of the cluster and distributes transactions to the database replicas using a load balancing strategy such as round robin or least active connections. These strategies balance the load well, but, as we will show, they can introduce memory contention, which causes the system to perform poorly.

To reduce memory contention in static content clustered Web servers, content-aware load balancing techniques have been introduced, such as LARD (Locality-Aware Request Distribution) [PAB+98]. LARD biases dispatching of requests to replicas at which the same requests were last served, in the expectation that content recently served is still memory resident.

LARD has proven very effective for read-only static content Web workloads, which consist of mostly small files. As we will show in this paper, LARD can cause poor performance for workloads, such as database transaction workloads, in which requests with large working sets occur more frequently. Furthermore, having been designed for read-only workloads, LARD has no provision for efficiently handling updates.

In this paper we introduce the notion of memory-aware load balancing. A memory-aware load balancer explicitly uses information about the size and the contents of the working set of transactions to assign them to replicas in such a way that they execute in memory. A memory-aware load balancer faces two challenges: how to estimate the working set size and contents of transactions, and how to use this information to favor in-memory execution.

Many memory-aware load balancing algorithms are possible, differing in the way they meet these challenges. In Section 2 we present one such algorithm, MALB-SC, and demonstrate that it has good performance on database workloads resulting from e-commerce applications. Such workloads are memory-intensive. We assume the database application has a fixed-set of parameterized *transaction types*. Many e-commerce database applications map to this model to enforce the business logic; ad-hoc access to the database is restricted and the database is accessed through a pre-defined set of interactions.

While the working set of an arbitrary process is difficult to predict, the working sets of database transactions are dominated by the tables and indices needed for processing. By discovering the tables and indices referenced, and how they are accessed from the database query execution plan, MALB-SC estimates

the working set size *and* contents for different transaction types. Using the working set estimates, MALB-SC creates transaction groups whose combined working sets fit in main memory so they can efficiently share replicas. Roughly speaking, each transaction group is assigned a sufficient number of replicas to handle its load.

For updates, we present *update filtering* to reduce the overhead of update propagation in the replicated system. MALB-SC dynamically assigns each replica a subset of the transaction types. If the system configuration and workload characteristics are stable, the replica can drop tables not needed for its subset of transaction types. As a result, the replica does not receive updates for those tables.

We build Tashkent+, a prototype write-anywhere database, replicated on a LAN cluster. We demonstrate performance improvements using the TPC-W and RUBiS benchmarks across a range of workloads. In particular, running the update-intensive ordering mix of the TPC-W benchmark on a 16-replica cluster, our experiments show that MALB-SC *doubles* (105% improvement) the throughput compared to least connections and provides 52% more throughput than LARD. MALB-SC with update filtering improves throughput to triple (202% improvement) that of the least connections — another full factor of improvement over just MALB-SC alone — and more than doubles (126% improvement) that of LARD. Thus, with the ordering mix, our techniques show a super-linear speedup of 37x over a single system due to better use of the cluster's memory.

The contributions of this paper are the following:

1. We identify and explain why LARD algorithms do not handle large requests well. We introduce memory-aware load balancing, and we demonstrate that it handles such requests better.
2. Memory-aware techniques require working set information. We propose a number of methods to estimate the working set size and contents of database queries.
3. We compare different degrees of how aggressively to pack transaction types on the replicas. We show that memory-aware techniques are effective even when the working set information is approximate, but that there is a danger in being overly aggressive in packing different transaction types on to the same replica.
4. We propose the *update filtering* optimization in which update propagation is selectively filtered to greatly reduce the consistency load on the system and improve scalability.
5. We implement all of the above techniques, and show their effects using the TPC-W and RUBiS benchmarks on a 16-replica cluster.

The rest of the paper is organized as follows. Section 2 contains a detailed description of the MALB-SC algorithm. In Section 3 we present update filtering. In Section 4 we discuss the implementation of Tashkent+ and the experimental environment. In Section 5 we present our experimental evaluation. In Section 6 we discuss related work. We present our conclusions in Section 7.

2. MALB-SC ALGORITHM

2.1 Overview

Here we present an instance of a memory-aware load balancing algorithm we call MALB-SC. Other implementations and improvements are certainly possible, however, our implementation details many of the issues any memory-aware algorithm must address.

The properties of a good memory-aware load balancing algorithm are the following :

1. Dispatch transactions to replicas such that they fit together in memory, avoiding memory contention.
2. Dynamically allocate more replicas to transaction types that require more resources.
3. Respond to changes in the workload by rebalancing replica allocation accordingly.

Our implementation, MALB-SC, uses estimates of the working sets in order to create transaction groups such that each group fits in the main memory of a database replica. Then, MALB-SC dynamically allocates replicas to transaction groups.

The state of the database is continuously monitored to create up-to-date estimates of the working sets using queries on metadata for the tables. If changes in the working sets (i.e., growth/shrinkage) require re-grouping the transactions, new transaction groups are formed. Throughout, replica allocation continuously adjusts to the groupings and the needs of the workload mix. Demanding groups get more replicas, while less demanding groups give up excess replicas or even merge with other lightly loaded groups to improve the efficiency of the overall system. Next, we discuss how to estimate the working sets, followed by creating transaction groups and dynamic allocation of replicas.

2.2 Estimating Working Set Information

We use the execution plan as well as metadata from the database to generate the working set estimate for each transaction type. The load balancer requests from the database the *execution plan* of the transaction type. The execution plan contains the tables and indices used and how the database accesses them. The load balancer uses this information to compute estimates on the transaction's use of memory, i.e., its *working set*.

The following four categories of information represent an increasing level of detail about transactions.

Transaction Type. The application provides the transaction type to the load balancer when it requests a connection to start a new transaction. Thus, when a request arrives, the load balancer knows its type and uses the type information in its dispatching decisions.

Working Set Size. The working set for processing a transaction is dominated by the tables and indices referenced. Therefore, a practical estimate on the working set size is the sum of the sizes of the tables and indices referenced in the query execution plan.

Working Set Content. Knowing which tables and indices are referenced is important when two or more transaction types execute on a replica. The working sets of two transaction types overlap when that they access common tables or indices. Shared content is not double counted when estimating the combined working set of transaction types.

Working Set Access Pattern. When a table or index is linearly scanned, all its pages are brought to memory. In contrast, each random access to a table likely only touches a handful of pages. The working set differs considerably between the two access styles. Taking the access pattern from the query execution plan into account provides better estimates on the working set size

for a transaction instance. However, if many instances of a transaction type with different parameters execute on a replica, then the aggregate effect of many random accesses may result in accessing all pages. We explore this topic in our experiments.

2.3 Creating Transaction Groups

With the working set information, we use a bin packing heuristic to group transaction types so that their combined working sets fit into available memory. We investigate the following range of methods that use progressively more information to construct groups.

Method MALB-S (Size only). We choose the well-known *Best Fit Decreasing (BFD)* [L99] bin packing algorithm using only the size of the working sets, not their contents. In BFD, the largest objects (i.e., biggest transaction types) are greedily fit first into the bins (i.e., memory) with the smaller objects fit in last. Overlap of the working sets is not considered here. For example, if transaction type T1 uses tables A and B, and T2 uses tables B and C, then the estimate of the memory needed to pack T1 and T2 together is (|A| + 2 |B| + |C|).

Method MALB-SC (Size+Content). A more accurate estimate is to avoid recounting shared content (e.g., shared tables). In the prior example, T1 and T2 share table B. When considering content in the estimate, the size shrinks to (|A| + |B| + |C|). The BFD algorithm is modified to account for overlap. A transaction type is added to the bin for which (1) the non-overlap component fits in the available free space and (2) there is maximal overlap. Because overlap reduces the sum of combining working sets, this method packs groups more efficiently than MALB-S and produces a better estimate on a group's aggregate working set size.

Method MALB-SCAP (Size+Content+Access Pattern). Using all the referenced tables and indices as in MALB-S and MALB-SC likely over-estimates the working set. We consider a lower bound estimate made up of only the heavily used tables and indices in the query plan. We define the heavily used tables as those that are linearly scanned. The packing algorithm is the same as for MALB-SC but the input is just the list of scanned tables and indices for each transaction type and their sizes. In general, fewer groups are generated. The working set *may* be under-estimated which can result in over-packing of transaction types into a single group (i.e., the group's memory needs may exceed the memory capacity of the replicas).

Overflow Transactions. Transaction types whose working set estimates are larger than main memory are considered *overflow transaction types*. Each overflow transaction type is assigned its own group.

2.4 Replica Allocation

Load balancing requires some method of estimating the load on each replica. The common technique of using outstanding connections is too coarse when the complexity of transactions varies greatly across the mix. In our system, the load balancer continuously receives replica load information on the CPU and the disk I/O channel utilization from lightweight daemons running on each of the replicas. The key features of the re-allocation process using this information are the following:

Group Load Calculation. The load balancer calculates the load for each transaction group by averaging the (smoothed) CPU and disk utilizations of all replicas assigned to that group. For example, if a group is assigned three replicas having (CPU, disk) utilizations (in %) of (45, 10), (40, 8), and (53, 9), the load balancer would summarize the load as (46, 9) for the group.

Comparing Loads. To compare loads between groups — some of which may be CPU-bound and others that may be I/O-bound — we use *MAX(CPU, disk)* as the load function to indicate the utilization of the bottleneck resource. Thus, we consider I/O-bound transactions and CPU-bound transactions as equally significant in their ability to limit the throughput of the system. Other weighted functions could be used, but we find that the simple *MAX* function works well.

Replica Allocation. We allocate additional replicas to the most loaded group from the least loaded group. Instead of using the current load statistics to determine the least loaded group, the load balancer calculates via simple linear extrapolation what the *future* load of each group would be if one of its replicas were removed. In the prior illustration of three replicas having an average load factor of 46, if one replica were removed the estimate for the future average load would be 46 x 3/2 = 69, i.e., the same total load but distributed over two replicas.

Using an estimate of the future load accommodates naturally the higher sensitivity to re-allocations of groups having just a few replicas. For example, consider two groups, one with two replicas and an average utilization factor of 20 and another with six replicas and an average utilization factor of 25. The future average load of each group if one replica were removed would be 40 and 30, respectively. Thus, it is better to re-allocate one from the group with six replicas rather than the from the group of two replicas even though the current load factor is lower in the latter.

Finally, because load measures are somewhat noisy, we add hysteresis by restricting re-allocations unless the most loaded group (to which a replica would be added) has a utilization of at least 1.25 times that of the (future) least loaded group (from which the replica would be taken).

Fast Re-allocation. If the workload characteristics change dramatically, re-allocating only one replica at a time could result in poor performance while the system slowly reconfigures. Faster convergence on reconfiguration is done by solving a simple set of simultaneous equations on the *estimate* of the total resource needs of each group. We estimate a group's total resource needs by multiplying its average utilization by the number of replicas allocated to it. For example, a system with two groups, a group M having three replicas at 70% utilization and another group N with seven replicas at 10% utilization would have the balance equations:

$$(0.70 \times 3) / m = (0.10 \times 7) / n$$

$$m + n = 10$$

The solution is m=7.5 and n=2.5 which, if rounded conservatively, is m=7 and n=3. Thus, the allocation changes quickly by, at once, removing 4 replicas from group N and giving them to group M. Future adjustments fine tune the allocation, but the bulk of the resources are correctly deployed quickly to minimize the time operating in a less efficient configuration during the transition.

Merging Low Utilization Transaction Groups. It is possible that the load of a transaction group is so low that even a single replica allocated to it is underused. Such groups should not tie up a replica and keep it drastically under-utilized. However, sharing the replica with another group is likely to generate a higher degree of memory contention, precisely the event MALB-SC is designed to avoid. If there are two such transaction groups, each lightly utilizing its single replica, then we assign one replica to the two groups and reclaim a replica for allocation elsewhere. By restricting sharing to groups that

substantially under-utilize their *single* replicas, MALB-SC minimizes the chance that memory contention creates a bottleneck in the system.

If significant memory contention occurs such that the newly shared replica becomes the most loaded system then dynamic re-allocation is triggered. In this case, instead of allocating another replica, the two transaction groups are split and a separate machine is allocated to each group. This method stops the memory contention, if that indeed were occurring.

3. UPDATE FILTERING

Update filtering is an optimization to reduce the overhead of reflecting updates system-wide. In a replicated database, updates must be reflected at all copies in the system to maintain consistency. While the aggressiveness on when and how these updates are propagated varies depending on the consistency model being used, work must be done eventually at all replicas to update all copies of changed items. This update propagation overhead to change all copies is a fundamental scalability bottleneck. Update filtering targets this overhead.

The MALB-SC algorithm partitions transaction groups across replicas. Under stable workload characteristics, this partitioning can be made permanent by the load balancer. Since each replica receives only a subset of the transaction types, any tables not used at a replica can be dropped or allowed to go out-of-date. Updates to these unused tables do not have to be processed by the replica, i.e., their remote updates can be filtered.

Recovery of a replica is not affected by update filtering. If a replica crashes and later restarts, standard recovery is used. For example, the database can be restored from other copies in the cluster or by the persistent log at the certifier [EDP06]. We limit the discussion here to ensuring availability in the presence of update filtering, where the system ensures a minimum level of redundancy for every transaction type *and* table in the system.

There are two constraints that must be met by the load balancer to ensure a target level of availability, in contrast to a replicated system without update filtering. The first constraint is to ensure *transaction type availability*, i.e., each transaction type must have a minimum number of replicas upon which it can run. Even if the transaction's group requires only a single replica for performance reasons, the load balancer ensures that additional replicas have up-to-date state to run the transaction if needed. For every transaction group, the number of replicas serving its transactions at runtime is dictated by the load.

The second constraint is to ensure *table availability* in the presence of update filtering. Ensuring enough copies of tables and indices are available is a separate concern from transaction type availability. However, table availability is automatically addressed if transaction type availability is provided.

4. IMPLEMENTATION

We have built a prototype system called Tashkent+ that embodies the main ideas in this paper. Tashkent+ is an extension of our Tashkent replicated database system [EDP06]. We give an overview of transaction processing in Tashkent and then we outline the changes implemented for Tashkent+. Finally, we describe the experimental environment.

4.1 Overview of Tashkent

Tashkent [EDP06] is a high-performance replicated database system. Tashkent represents the base system to which we compare the benefits of MALB-SC and update filtering.

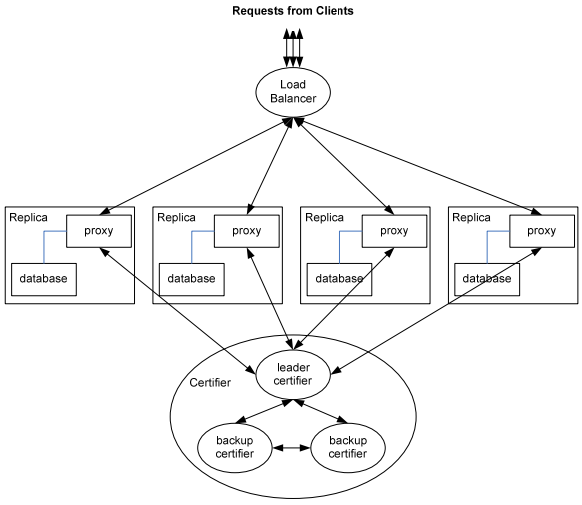

Figure 1: Tashkent replicated database design

Tashkent uses *generalized snapshot isolation* (GSI) [EPZ05] for concurrency control. GSI extends the well-known snapshot isolation (SI) protocol to replicated databases. The design consists of two main logical components both of which are replicated: (1) the database replica and (2) the certifier. Replicas are symmetric in that any replica can process any client request, whether read or write.

Under the GSI concurrency protocol, replicas process read-only transactions entirely locally. When a replica receives an update transaction it executes it locally, except the commit operation which requires certification to detect write-write conflicts. Replicas communicate only with the certifier component, not directly with each other. The certifier certifies update transactions from all replicas and gives them a global commit order.

The Tashkent design is pure replication middleware. As shown in Figure 1, attached to each replica is a *transparent proxy* that intercepts requests to perform the replication functionality. The proxy appears as the database to clients, and appears as a client to the database. The proxies and certifiers constitute the *replication middleware*. The proxy performs admission control to prevent bursts from overloading the database using the Gatekeeper algorithm [ENTZ04].

In Tashkent, database replicas are replicated mainly for performance, whereas the certifier is replicated mainly for availability. In this study, we simply assume a separate certifier component which is replicated, though our conclusions apply to other configurations. For example, the certifier component could be implemented via an atomic broadcast mechanism incorporated into the proxy at every replica [LKPJ05, WK05].

Data consistency is maintained across replicas by propagating modifications at a replica to all other replicas using writesets. A *writeset* is the core information required to reflect the effects of an update transaction's changes [KA00].

Processing read-only transactions is straightforward. Each read-only transaction is processed against an assigned snapshot. Processing update transactions is illustrated in Figure 2 and includes the following steps.

Certification. When an update transaction attempts to commit, the proxy at the replica sends to the certifier a request to certify the writeset. The certifier processes the writeset to detect write-

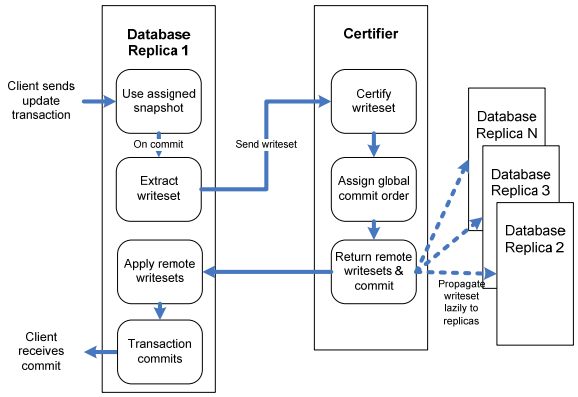

Figure 2: Processing of update transactions in Tashkent

write conflicts by comparing table and field identifiers for matches against writesets from recently committed update transactions. Successfully certified writesets (i.e., without conflicts) are recorded in a persistent log, thus creating a global order. The state of any replica is always a consistent prefix of the certifier's log.

Responding to replicas. The certifier responds to the replica with the result of the requested certification test and *any remote writesets from intervening update transactions at other replicas.* The middleware proxy at the replica applies the remote writesets to the database before it commits the local update. This broadcast of writesets to all replicas is essential to maintaining consistency. Applying the writesets is a fundamental scalability limit since all updates system-wide must be processed by *every* replica.

Update propagation. Updates are propagated as a side effect of certification requests. In addition, Tashkent uses two mechanisms to trigger update propagation. First, the proxy pulls new updates periodically (every 500 msec) from the certifier if the replica has not issued certification requests recently. Second, the certifier sends short notification messages to replicas that are behind (e.g., if a replica missed 25 commits) to prod them to make a request for updates.

Durability. The Tashkent design employs the novel method of combining durability and ordering in the middleware [EDP06]. Doing so avoids a serious disk I/O bottleneck that arises when durability is performed in the databases but commit ordering is decided in the middleware. When ordering and durability are not combined, the bottleneck occurs because the certifier determines the global order of commits, but proxies must commit update transactions and remote writesets serially to ensure the same order is followed at each replica.

Since Tashkent unites ordering and durability, its database replicas have a very efficient I/O subsystem: replicas do not need to issue an fsync() call to commit local or remote update transactions (durability is guaranteed in the middleware). This makes Tashkent a challenging environment for our study since MALB-SC and update filtering improve performance by reducing stress on the disk I/O channel. Since the Tashkent design already makes very efficient use of its disk I/O channel, we expect the MALB-SC and update filtering techniques to be of even greater benefit if durability and ordering are not united.

4.2 Tashkent+ Implementation

4.2.1 Load Balancer Implementation

The load balancer is implemented in Java as a JDBC driver. It contains the different load balancing algorithms as well as support for update filtering. The load balancer is light-weight. Forwarding all database requests takes less than 5% of the CPU in our experiments.

Fault-Tolerance. We use a Primary-Backup scheme for availability. The load balancer has only soft state that can be reconstructed from the replicas. When the primary load balancer fails, clients fail over to the backup load balancer and all active transactions are aborted and retried. The backup load balancer starts by querying the replicas to re-construct its soft state.

Consistency Guarantees. Tashkent+ provides Generalized Snapshot Isolation (GSI) [EPZ05] with Session Consistency [DS04]. All workloads discussed in this paper run serializably under GSI [EPZ05, F05, FLO+96].

4.2.2 Obtaining Working Set Information

Here we describe the specific mechanisms the load balancer uses to determine transaction types and the details of the working sets when using the PostgreSQL [PG] database:

1. The application provides the transaction type with each request for a JDBC database connection. However, this can be automated with static analysis or dynamic sniffing at the load balancer [ENTZ04, BCD+06].
2. The load balancer retrieves the database schema to find all tables and their associated indices.
3. For each table or index, its size in pages is determined by the PostgreSQL query: "SELECT relpages FROM pg_class WHERE relname='<tablename>' ". Each page is 8KB.
4. To retrieve the transaction execution plan, the load balancer sends an instance of each transaction type to PostgreSQL prefixed with the EXPLAIN command. PostgreSQL returns the execution plan of each query. The load balancer processes the plan and records all tables and indices accessed as well as how they are accessed.

4.2.3 Update Filtering Implementation

When update filtering is enabled, the load balancer sends to each proxy the list of tables for which the replica receives remote writesets. The proxy stores this information in the database and only forwards the writesets for those tables to the replica.

We disable dynamic replica allocation when update filtering is enabled. Therefore, in this paper update filtering is used only when the workload characteristics are stable. In future work, we plan to study strategies that enable the load balancer to combine dynamic replica allocation with update filtering.

4.3 Baseline Load Balancing Algorithms

We use the following two load balancing algorithms as a baseline for comparison with MALB-SC and update filtering.

LeastConnections: LeastConnections uses no information about the transaction type. The number of outstanding requests at a replica is used as a measure for balancing load. LeastConnections is a form of weighted round robin.

LARD: The algorithm knows only the transaction type and dispatches a transaction to a replica where instances of the same

transaction type have recently run [PAB+98, ZBCS99]. It has no information about the working set, neither its size nor its contents. Rather, the technique relies on locality-aware request distribution to re-use data in memory from a prior executions of transactions of the same type.

4.4 Experimental Environment

The replicated database system is the Tashkent+ prototype described in Section 4.2 with the load balancer modified to implement the various policies. Our primary performance metric is throughput, which is the number of transactions completed per second. We measure the performance of a single standalone database and determine the number of clients needed to generate 85% of the peak throughput. In the following experiments, we use that number of clients per replica to load the system.

System specification. Each machine in our cluster runs the 2.6.11 Linux kernel on a single Intel Xeon 2.4GHz CPU with 1GB ECC SDRAM, and a 120GB 7200pm disk drive. The machines are connected through a switched 1Gbps Ethernet LAN. We monitor the system load with a modified version of the Mercury server management system [HCG+06]. For the certifier, we use a leader and two backups for fault tolerance. We use the PostgreSQL 8.0.3 database configured to run transactions at the snapshot isolation level (which is the strictest isolation level in PostgreSQL and called the *"serializable transaction isolation level"*). Unless otherwise specified, there are 16 database replicas in the system.

The amount of available memory used as input to the bin packing algorithm is reduced by 70 MB to account for the memory usage of the operating system, PostgreSQL processes, the proxy processes, and monitoring daemons as well as for processing remote writesets during update propagation.

TPC-W Benchmark. TPC-W is a benchmark from the Transaction Processing Council [TPC] designed to evaluate e-commerce systems. It implements an on-line bookstore and has three workload mixes that differ in the relative frequency of each of the transaction types. We report results from all three mixes using an open source implementation of TPC-W [ACC+02]. The *ordering mix* workload has 50% updates, the *shopping mix* workload has 20% updates, and the *browsing mix* workload has 5% updates. In update propagation, the average writeset size is 275 bytes.

In Section 5.6 we explore a range of the problem space by varying the size of the database, the transaction mix, and the available memory at each replica. The goal is to highlight how system parameters interplay with our techniques. We focus on TPC-W and scale the database via its EBS parameter and experiment with a small database of 100 EBS (0.7 GB), a medium database of 300 EBS (1.8 GB), and a large database of 500 EBS (2.9 GB) for each of the three TPC-W mixes. In addition, we vary memory as 256 MB, 512 MB, and 1024 MB. To restrict memory use, at each replica we run a side process that locks down a specified number of pages via the system call *mlock()* before we run experiments. For example, to run with 756 MB of available memory, we lock out 256 MB of the available 1 GB RAM. For the initial discussion, we use the middle point in our configuration space: middle database size (1.8 GB) and middle memory size (512 MB).

RUBiS Benchmark. RUBiS [OW] is a benchmark from Rice University that emulates an on-line auction site modeled after eBay. In our experiments, the RUBiS database has 10,000 active items, 1M users, 500,000 old items and is 2.2 GB. There are two workload mixes, a *browsing mix* that is read-only and a *bidding mix* having 15% updates. The bidding mix is the main mix in RUBiS. We implemented a transactional version of the benchmark with primary key indices as the original benchmark does not support transactions [ACC+02]. The average writeset size is 272 bytes.

5. PERFORMANCE EVALUATION

5.1 Objectives

We perform an experimental evaluation using the Tashkent+ prototype to answer the following questions:

- What are the benefits of using working set information in load balancing?
- How do the different methods of constructing transaction groups by packing working sets into available memory differ?
- Does replica allocation have to be dynamic? How well does the load balancer cope with significant changes in the workload mix?
- What is the benefit of update filtering?
- Under which environments are MALB-SC and update filtering effective?

5.2 Exploiting Working set Information

The throughput of least connections (LeastConnections) versus MALB-SC (size+contents overlap) is shown in Figure 3. The throughput of LeastConnections is 37 tps with a response time of 2.2 sec. For perspective, the standalone database has a peak throughput of 3 tps with a response time of 2.6 sec. Thus, on 16 replicas, LeastConnections scales well to 12 times the performance of a single system.

MALB-SC *doubles* the throughput of LeastConnections to 76 tps with a response time of 0.81 sec. Thus, MALB-SC has a super-linear speedup of 25 over a single system. Super-linear speedup is observed, among other circumstances, when systems transition from being disk-bound to being memory-bound, reflecting more efficient use of the aggregate memory of the cluster.

The locality-aware algorithm LARD supports 50 tps throughput (with 1.4 sec response time), an improvement over LeastConnections, but MALB-SC is still 52% higher in performance. The performance differences can be explained by looking at the amount of disk I/O in the three systems, shown in Table 1. LeastConnections performs on average 84 KB of disk I/O per transaction, of which 72 KB is for reading data and 12 KB are writes from updates. LARD reduces read activity to 57 KB, but MALB-SC creates the greatest reduction to 20 KB. The reduced read disk activity means there is less memory contention in MALB-SC. Table 2 shows the groupings settled on by MALB-SC, as well as the number of replicas assigned to each group in this experiment.

For RUBiS in Figure 4, we see the same pattern, with MALB-SC out performing both LeastConnections and LARD by 39% and 26%, respectively. MALB-SC reduces the amount of disk I/O from reads significantly, as shown in Table 3. From the groupings in Table 4, we see that the transaction *AboutMe* is a demanding transaction that receives 9 of the 16 replicas. *AboutMe* is a large, frequent transaction that reads from almost all the tables in the database.

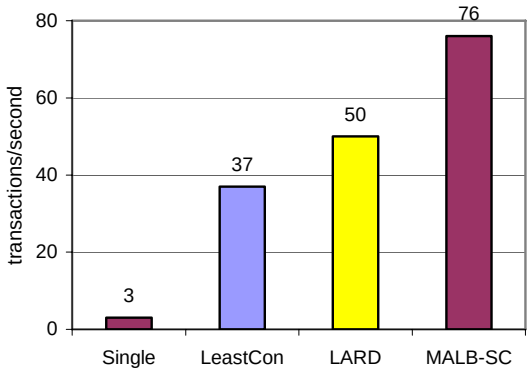

Figure 3: TPC-W comparison of methods. MidDB 1.8GB, RAM 512 MB, 16 replicas, Ordering (Single is standalone single database)

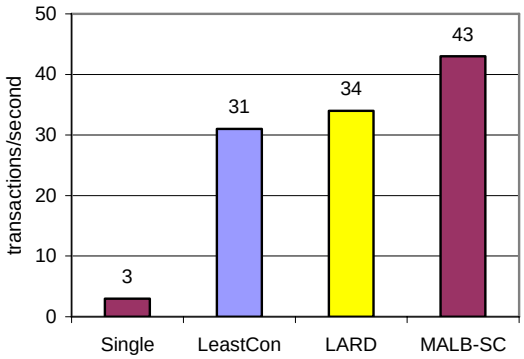

Figure 4: RUBiS comparison of methods. DB 2.2GB, RAM 512 MB, 16 replicas, Bidding (Single is standalone single database)

Table 1: TPC-W Average Disk I/O per Transaction

Method	Write	Read	Read Fraction to LeastConnections
LeastConnections	12 KB	72 KB	1.00
LARD	12 KB	57 KB	0.79
MALB-SC	12 KB	20 KB	0.28

Table 3: RUBiS Average Disk I/O per Transaction

Method	Write	Read	Read Fraction to LeastConnections
LeastConnections	11 KB	162 KB	1.00
LARD	11 KB	149 KB	0.92
MALB-SC	11 KB	111 KB	0.69

Table 2: TPC-W MALB-SC groupings

Transaction types	Replicas
[BestSeller]	2
[AdminRespo]	4
[BuyConfirm]	7
[BuyRequest, ShopinCart]	1
[ExecSearch, OrderDispl, OrderInqur, ProducDet]	1
[HomeAction, NewProduct, SearchRequ, AdmiRqust]	1

Table 4: RUBiS MALB-SC groupings

Transaction types	Replicas
[AboutMe]	9
[PutBid, StoreComment, ViewBidHistory, ViewUserInfo]	4
[Auth, BrowseCategories, BrowseRegions, BuyNow, PutComment, RegisterUser, SearchItemsByRegion, StoreBuyNow]	1
[RegisterItem, SearchItemsByCategory, StoreBid, viewItem]	2

MALB-SC versus LARD. The performance difference between MALB-SC and LARD deserves special discussion. Without specific working set information, LARD does not perform as well as MALB-SC, which does use working set information. The key insight into this difference is in how LARD allocates replicas to transactions.

When a large transaction is assigned to a replica, every time it runs it displaces the pages for other transaction types. Under LARD, when a large transaction is *frequent* it competes heavily with the other transactions for memory space creating contention and slowing down *all* transactions on the replica due to competition for the disk I/O channel. *The replica becomes less productive*. This contention results in longer response times and more open connections which signals to the LARD load balancer that the replica is heavily loaded (it is, but not with productive work). This triggers LARD to allocate another replica to the large transaction, creating memory contention on yet another machine and getting little if any additional throughput gains. The process continues until all machines are heavily utilized and, thus, "turns off" the LARD algorithm from making further allocations. LARD stabilizes at a sub-optimal configuration, with a lot of memory contention.

The above scenario does not occur in LARD if the workload is static content consisting mostly of small files. With small files, no single request can systematically wipe out large portions of the memory. If LARD overloads a machine, it occurs gradually with the many small requests providing fine-grained feedback for LARD to reconsider its allocations. In contrast, a large transaction is coarse-grained in that the memory contention it creates does not arise gradually, but instantaneously and pronounced.

In contrast, MALB-SC successfully manages large transactions by isolating their effects on other requests. The performance impact of making a bad dispatch decision on large transactions is high. MALB-SC succeeds by avoiding the contention cases. Having the working set information allows proper grouping of requests to share resources amicably while avoiding memory contention.

For the rest of the results we shall focus on TPC-W as it has greater variety in its transaction mixes, including a wider range of update activity. We return to RUBiS in Section 5.6.

5.3 Constructing Transaction Groups

We contrast the three methods for utilizing working set information to form transaction groups. MALB-S (size only) generates 7 groups. MALB-SC (working set size and contents) generates 6 groups, since shared tables are not double-counted. MALB-SCAP (working set size, contents, access pattern) generates 4 groups when using only heavily scanned items as a lower estimate of the working set. MALB-S and MALB-SC are more likely to over-estimate the true size of a working set, while MALB-SCAP is more likely to under-estimate the size.

Figure 5 shows the performance of the system when using the different transaction groupings (as well as LeastConnections and LARD for reference). MALB-SCAP, MALB-S, and MALB-SC give 57, 73, 76 tps respectively. All have higher throughputs than that of LeastConnections and LARD.

Even though MALB-SCAP performs better than LARD and LeastConnections, it does not perform as well as the two other methods, MALB-S and MALB-SC. MALB-SCAP generates more disk read I/O than both MALB-S and MALB-SC indicating some over-packing of transaction groups.

This result may seem surprising, since MALB-SCAP uses more detailed information to more precisely estimate the working sets, while MALB-SC uses less information and therefore tends to over-estimate. The reason for MALB-SCAP's inferior performance is that the penalty for under-estimation is high. Under-estimation and the resulting over-packing of transaction types in a single transaction group leads to more disk I/O, the cost of which dwarfs any gain achieved by tighter packing. A further consideration is that the estimation is inherently approximate, as demonstrated next.

Experimental Working Set Measurement. We measure the working set of all transaction types experimentally by dedicating transaction types to a single machine and adjusting the amount of free memory until the amount of disk I/O spiked. We compare the measured working set size to the lower estimate (as used in MALB-SCAP) and upper estimate (as used in MALB-SC).

For many transaction types the estimates differ little between MALB-SC and MALB-SCAP. For example, for the *BestSeller* transaction the estimates are 608 and 610 MB, respectively. These estimates agree well with the measured working set sizes that range between 600 and 650MB. For other transactions, however, the difference between the estimates of MALB-SC and MALB-SCAP are substantial. For example, for the *OrderDisplay* transaction MALB-SCAP estimates the working set size to be 1MB, while MALB-SC arrives at 1600MB, because it makes random accesses to nearly every table but scans only one small one. Its true working set size is between 400 and 450 MB. Using the MALB-SCAP estimate would suggest that *OrderDisplay* can be packed with any group, but its true size of over 400 MB means that its working set actually consumes most of the available 512 MB of main memory. Therefore, it should be packed separately in a single group.

Merging Groups. To compensate for occasional under-packing resulting from conservative estimates in MALB-S and MALB-SC, transaction groups that under-utilize their replicas are merged. To measure the effect of merging, we disable it and measure the resulting throughput. The throughput of MALB-S decreases from 73 to 66 tps, and the throughput of MALB-SC decreases from 76 to 70 tps. Thus, in these experiments merging transaction groups on under-utilized replicas compensates for having many groups, of which some have infrequent requests.

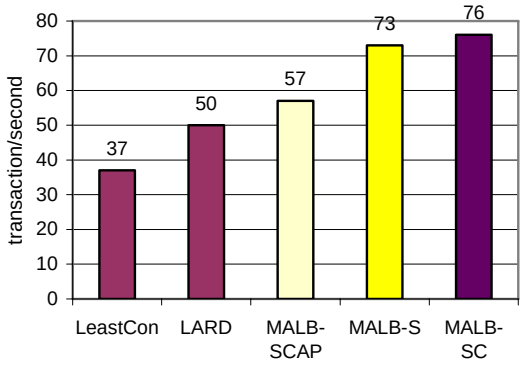

Figure 5: Throughput of grouping methods. MidDB 1.8GB, RAM 512 MB, 16 replicas, Ordering

Summary. We conclude that being conservative in estimating the working set is safer because it avoids memory contention. Over-estimating the working sets can result in many transaction groups of which some may not be able to load a dedicated replica. Merging groups which have only a single under-utilized replica mitigates having too many transaction groups. Furthermore, in the event of memory contention, the MALB-SC algorithm prioritizes the undoing of merging before allocating additional replicas. Thus, memory contention of merged groups may occur in MALB-SC but in a controlled fashion such that it can also be undone.

5.4 Dynamic Reconfiguration

Should the workload change, the system must adapt to the new mix of requests. In Figure 6, we change the workload from the shopping mix to the browsing mix and then back to the shopping mix. We use these two mixes since the distribution of replicas across the groups differ the most. For this experiment, we run the shopping mix for 2000 seconds, then switch to the browsing mix for 2000 seconds, and then switch back to the shopping mix for another 2000 seconds. The baseline performance for the shopping mix is 76 tps. The browsing mix has a lower baseline performance of 45 tps. We should expect the system to match these baseline throughputs in the time intervals during which the corrseponding workload is in effect.

The lighter curve is the number of transactions executed per 30 second interval. The darker curve is the moving average of a 150 second window. The system tunes itself to the shopping mix and then adjusts to the expected performance for the browsing mix. The system responds quickly. The slope between transitions is due to the lag to detect the change in workload and from averaging over the interval.

The thick bottom line (19 tps) is the throughput of running the browsing mix with the best static configuration for the shopping mix. This is the throughput that would occur if the load balancing could not adapt to the workload. With such a static algorithm, the throughput of MALB under the browsing mix (19 tps) is *lower* than the throughput of LeastConnections under the browsing mix (37 tps). Using the wrong static configuration

leads to unbalanced replicas; some are overloaded and others are underloaded. Therefore, a dynamic algorithm is necessary to get the benefits of MALB when the workload changes.

5.5 Effectiveness of Update Filtering

We enable update filtering after the system stabilizes the configuration of the database cluster. The results are shown in Figure 7. For the ordering mix with 50% of its transactions being updates, MALB-SC with update filtering (MALB-SC+UpdateFiltering) has a throughput of 113 tps which is 47% above the performance of just MALB-SC, and 202% above that of LeastConnections. The average response time for MALB-SC+UpdateFiltering is 0.349 sec. MALB-SC+UpdateFiltering shows super-linear speedup, with a throughput of 37 times the peak throughput of a standalone database.

Table 5 lists the same disk I/O information as Table 1, but with an additional entry for "MALB-SC+UpdateFiltering". The amount of data written to disk has dropped 25% to 9 KB from 12 KB (at a higher throughput rate) due to filtering. The relatively large performance boost comes from update filtering removing competition for the disk channel on replicas with transactions that need data from disk. The slight reduction in read activity is because filtering updates also reduces memory pressure somewhat.

The magnitude of the performance improvement resulting from update filtering may seem surprising. The writesets average 275 bytes each, so 76 tps in the ordering mix (50% updates) generates 38 writesets per second, or less than 10K bytes per second. A paltry 10 KB/sec should not add much stress to the disk I/O channel, but it appears to do so because update filtering improves performance by 50% over just MALB-SC. However, Table 5 reveals that the total amount of data actually written to disk by MALB-SC is 76 tps * 12 KB per transaction, or 912 KB/sec. Thus, the small updates being propagated are apparently creating significant disk activity.

What is happening is that the 10 KB of updates are actually distributed throughout the database and touch many pages. Since a database page (8KB in our system) must be written completely to disk whether one byte is dirty or all 8KB are dirty, the randomness of the writes greatly impacts the amount of disk I/O activity generated.

5.6 Different Database and Memory Sizes

The relative sizes of the database and memory, as well as the transaction workload mix, impact the dynamics of the MALB-SC algorithm. We explore this space in TPC-W and vary the memory size, the database size, and the workload mixes.

Figure 10 has nine plots representing 81 experiments using different combinations. The top row is LargeDB of 500 EBS (2.9 GB); the middle row is MidDB of 300 EBS (1.8 GB) database; and the bottom row is SmallDB of 100 EBS (0.7 GB). The workload mixes vary across the columns, the first being the ordering mix (50% updates), the second the shopping mix (20% updates), and the third the browsing mix (5% updates). Within each chart, the memory is varied, at values of 256 MB, 512 MB, and 1024 MB. The configuration space covered is summarized graphically in Figure 9. Finally, the three techniques compared are LeastConnections, MALB-SC, and MALB-SC with update filtering (MALB-SC+UpdateFiltering). LeastConnections is used as a base comparison to highlight when a simple load balancing approach suffices. For context, the results of Figure 7 map to the middle set of bars in chart MidDB-Ordering (middle row, first column).

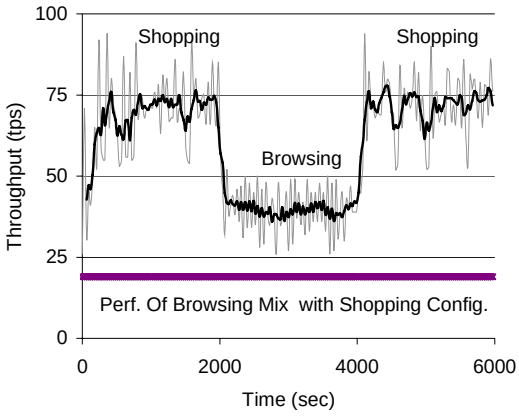

Figure 6: Dynamic reconfiguration: TPC-W workload mix switches from Shopping to Browsing to Shopping. X-axis is time (sec). Y-axis is tps. MidDB 1.8GB, RAM 512 MB, 16 replicas.

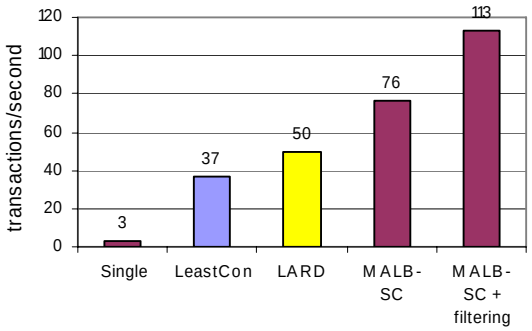

Figure 7: TPC-W throughput of MALB-SC+UpdateFiltering. MidDB 1.8GB, RAM 512 MB, 16 replicas, Ordering.

Table 5: TPC-W Average Disk I/O per Transaction

Method	Write	Read
LeastConnections	12 KB	72 KB
LARD	12 KB	57 KB
MALB-SC	12 KB	20 KB
MALB-SC + UpdateFiltering	9 KB	18 KB

The chart LargeDB-Ordering at 256 MB shows there is little benefit with MALB-SC when the database is large and the memory is small. This corresponds in Figure 9 to the region where the working set is too big for memory. Under these conditions, the transactions run from disk, regardless of the partitioning (or grouping) of transactions. However, as memory is increased to 1 GB the benefits of partitioning become pronounced, improving performance from 39 tps to 110 tps, and further to 147 tps with update filtering. The benefit of update filtering decreases in charts LargeDB-Shopping and LargeDB-Browsing because of the lower update rates; there are fewer updates to filter.

Charts SmallDB-Ordering, SmallDB-Shopping, and SmallDB-Browsing at 1 GB RAM show the other extreme, a small

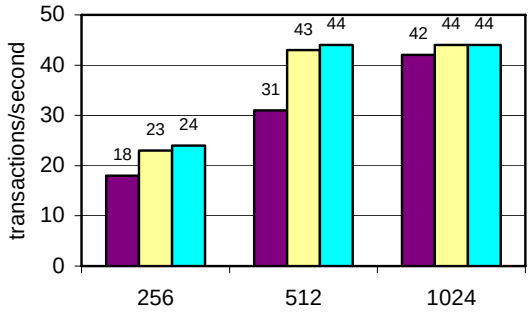

Figure 8: RUBiS bidding mix with update filtering. DB 2.2 GB, RAM 256, 512, 1024 MB, 16 replicas

database with large memory. Here, even LeastConnections performs well on all mixes since the full database fits entirely into memory. MALB-SC and update filtering help when memory is small (256 MB) as MALB-SC fits the working sets in the replicas' main memories.

The middle column showing the shopping mix displays a nice range of behavior as the database size is increased. In chart SmallDB-Shopping, 256 MB appears to be just a bit too small for the working sets and MALB-SC helps. For the MidDB-Shopping, both the 256 MB and 512 MB configurations benefit. For LargeDB-Shopping, all three memory sizes see benefits from MALB-SC. The 20% update rate is not high enough for update filtering to have much effect.

One graph for the RUBiS benchmark running the bidding mix is shown in Figure 8. MALB-SC helps improve performance below 1 GB of memory, but the working sets fit in 1 GB so LeastConnections performs as well as MALB-SC. Update filtering shows little help as the 15% update rate of the bidding mix is too low for filtering to be of advantage.

In summary, MALB-SC and update filtering improve performance significantly when working sets of transaction groups fit into the available memory, but the combined sum across all transaction groups exceeds available memory. If the memory is too small or too large, then MALB-SC is of little help. However, even if there is no additional benefit from MALB-SC, in our experiments the MALB-SC algorithm still generates configurations whose performance is at least as high as LeastConnections.

5.6.1 Update Filtering

Here we discuss in more detail the effects of using update filtering to reduce the system-wide overhead of update propagation. We focus on the ordering mix with 50% updates, i.e., the leftmost column.

From the three graphs of this column, the pattern that emerges is that when MALB-SC enhances performance then update filtering tends to also add significant improvements. However, when MALB-SC adds little benefit then update filtering does not appear to help much either.

MALB-SC improves performance by reducing the amount of data "pulled" from disk. In contrast, update filtering helps by reducing the amount of data "pushed" to disk and competing with reads for disk I/O. Both techniques reduce disk channel activity.

If the database is "small" and the memory "large" such that the database essentially fits in memory, then there is little data being "pulled" from disk. Thus, there is little activity for MALB to reduce. This also implies (1) there is plenty of buffer space to hold dirty database pages and (2) there is plenty of spare disk channel bandwidth for the write back of dirty pages from update propagation. Hence, update filtering offers little benefit.

Conversely, if the database is "large" and memory is "small", such that all transactions generate disk I/O, then MALB is unable to reduce memory contention. The system is I/O-bound, throughput is relatively low and, thus, update activity is also low, so filtering can only have minimal impact.

6. RELATED WORK

We focus on related work in two areas: replicated databases and load balancing in server systems.

6.1 Replicated Databases

Many database systems use a front-end to perform request scheduling and load balancing [ACZ03-1, ACZ03-2, PA04, ZP06]. In general, they use load balancing algorithms that do not exploit working set information. For example, conflict-aware scheduling [ACZ03-1] is content-aware in that the application provides to the scheduler the tables that are accessed so requests are scheduled to copies that are up-to-date. The emphasis is on correctness; no provision is made for ensuring working sets fit in memory. In distributed versioning [ACZ03-2], the scheduler increases concurrency to scale the performance of replicated databases. Another form of conflict-aware scheduling [ZP06] is used to reduce aborts, and consequently increase system performance. Again, working set size and memory contention issues are not addressed.

Partial replication [RT04] partitions data across replicas. In contrast, Tashkent+ is essentially fully replicated design and MALB-SC partitions transaction types, not data, across replicas. Update filtering reduces the overhead of update propagation, which is also reduced in partial replication. In Tashkent+, every transaction is fulfilled by the replica to which it is dispatched. Complex queries in partial replication may require distribution across many replicas simultaneously.

The P*TIME system [CS04] is not a replicated database but it uses a shared memory design emphasizing parallel disk I/O channels for writes and fast recovery. Our work reduces disk I/O through better memory use and is intended for memory-intensive rather than update-intensive workloads.

6.2 Load Balancing

Here we discuss three systems LARD [PAB+98], HACC [ZBCS99], and FLEX [CP00]. LARD uses locality-aware load balancing for serving static web pages. FLEX and HACC use a notion of the size of requests in load balancing. In both, the size of the *returned content* from the set of instances seen thus far is parsed from the log files and used as the estimate for the working set. The authors of FLEX term this *content-aware* load balancing. However, when serving dynamic content, the size of the returned data is not a good measure of the total memory needed to process requests (in TPC-W the difference can be many orders of magnitude). Estimating working set size as in Tashkent+ provides (1) a safe estimate and (2) enables assigning an efficient configuration once a single *instance* for each request *type* occurs.

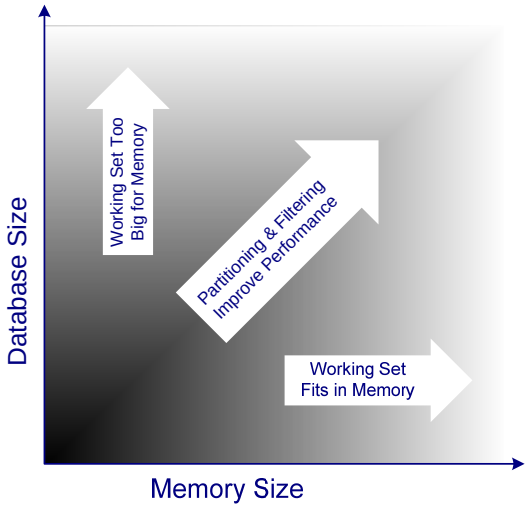

Figure 9 : The space of database size versus memory size

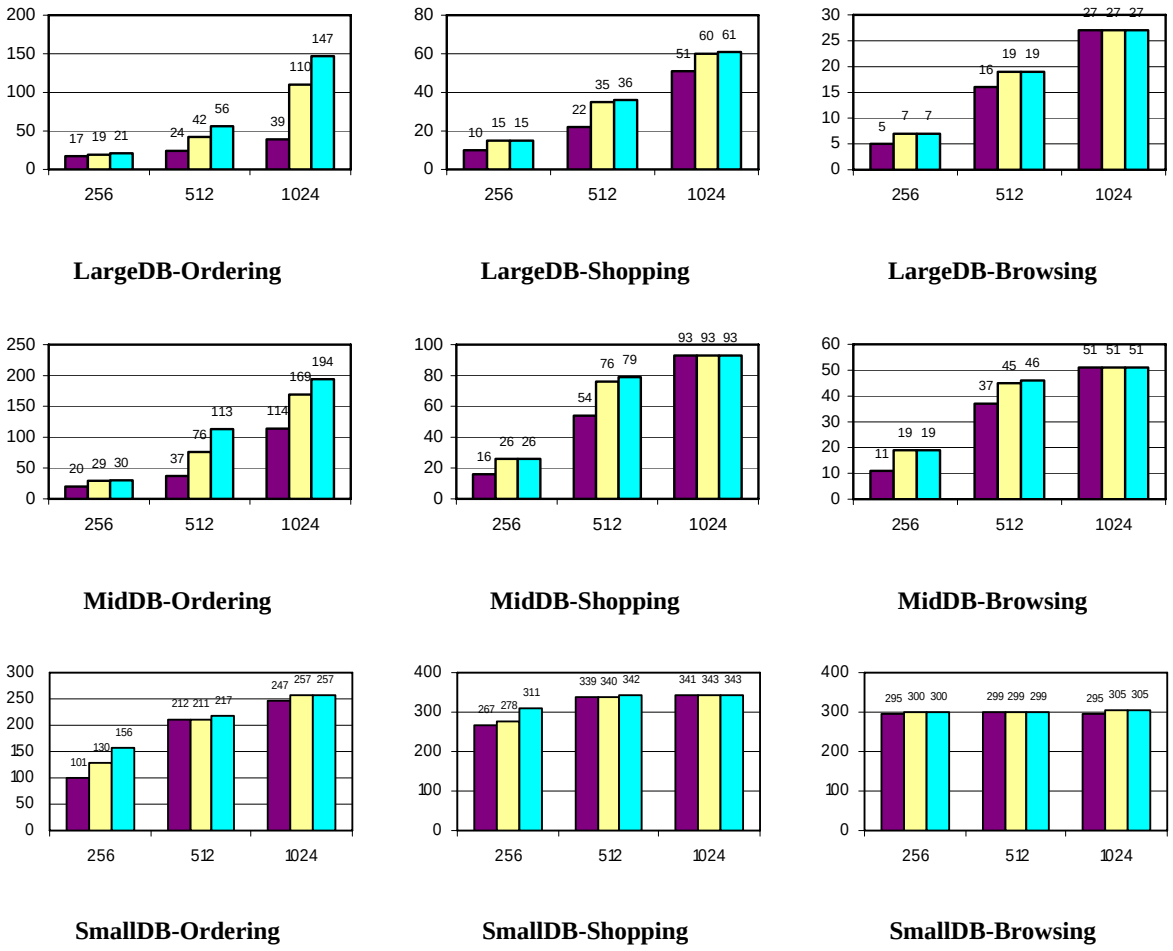

Figure 10: TPC-W throughput graphs for different configurations. Y-axis is throughput (tps). Each graph has three groups of three bars. The groups are RAM sizes (256, 512, 1024 MB). The three bars are LeastConnections, MALB-SC, and MALB-SC + UpdateFiltering.

Our contributions include the identification of problems for locality-aware techniques when there are frequent requests with large working sets and how to estimate the working sets of database transactions (dominated by tables and indices rather than simple files). LARD, HACC and FLEX were not designed for database workloads; none of them addresses updates in the workload. In other domains, working set information has been used dispatch independent jobs in a computing cluster [BB97].

In decision support systems, an analytical model [KS00] of the memory access behavior has been developed. The model is shown to be effective, but requires a number of parameters that will not be readily available to a load balancer. Our working set estimation techniques are simpler yet still effective in MALB-SC.

7. CONCLUSIONS

This paper presents memory-aware load balancing (MALB) for replicated databases. MALB uses information on transaction working sets to dispatch transactions to replicas in a manner that reduces memory contention. We show that without using additional information on how transactions use memory, existing load balancing techniques cannot prevent memory contention when there are frequent requests with large working sets as in memory-intensive database workloads. We present a method to estimate the size and content of transaction working sets from the query execution plan and metadata from the database. Using information on memory use, our MALB-SC algorithm significantly improves performance over other load balancing methods.

The MALB-SC algorithm distributes transaction types among replicas in a way that we exploit for a complementary optimization called update filtering. The update filtering algorithm specializes each replica to its workload: The replica applies those updates to the tables and indexes needed to service its workload, and *filters out* updates to unused tables. Update filtering reduces resources needed to propagate the effects of update transactions, and therefore further boosts performance.

We build a prototype system, called Tashkent+, to demonstrate the benefits of MALB and update filtering. On a cluster of 16 database replicas using the ordering mix of TPC-W benchmark, MALB-SC doubles system throughput compared to least connections and provides 52% improvement over LARD. Adding update filtering further improves throughput to triple that of least connections and more than double that of LARD.

MALB-SC and update filtering exhibit super-linear speedup; the throughput of the 16 replica cluster is 37 times the peak throughput of a standalone database.

8. ACKNOWLEDGMENTS

This research was partially supported by the Swiss National Science Foundation grant number 200021-107824 and by an equipment grant from Hasler Foundation. We thank Gustavo Alonso, Cristiana Amza, Ming-Yee Iu, and the anonymous reviewers for their many constructive comments.

9. REFERENCES

[ACZ03-1] Cristiana Amza, Alan Cox, and Willy Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems, March 2003.

[ACZ03-2] C. Amza, A. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-end Databases of Dynamic Content Web Sites. In ACM/IFIP/Usenix International Middleware Conference, June 2003.

[ACC+02] C. Amza, E. Cecchet, A. Chanda, Alan L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. WWC-5: The 5th Annual IEEE Workshop on Workload Characterization, Austin, Texas, USA, November 2002.

[BB97] A. Barak and A. Braverman, Memory Ushering in a Scalable Computing Cluster, In Proceedings of the IEEE Third International Conference on Algorithms and Architecture for Parallel Processing, Melbourne, 1997.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In Proceedings of the SIGMOD International Conference on Management of Data, May 1995.

[BCD+06] S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, and W. Zwaenepoel. Caching Dynamic Web Content: Designing and Analysing an Aspect-oriented Solution. In Proceedings of Middleware 2006, Melbourne, Australia, November 2006.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[CP00] L. Cherkasova and S. Ponnekanti: Optimizing a "Content-Aware" Load Balancing Strategy for Shared Web Hosting Service. In the Proceedings of the 8th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), San Francisco, CA, USA, August 2000.

[CS04] Sang Kyun Cha, Changbin Song. P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. In Proceedings of 30nd International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada, August 2004.

[DS04] Khuzaima Daudjee, Kenneth Salem. Lazy Database Replication with Ordering Guarantees. In Proceedings of the 20th International Conference on Data Engineering (ICDE 2004), Boston, MA, USA, March 2004.

[DS06] Khuzaima Daudjee, Kenneth Salem. Lazy Database Replication with Snapshot Isolation. . In Proceedings of 32nd International Conference on Very Large Data Bases (VLDB 2006), Seoul, Korea, September 2006.

[EDP06] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In the Proceedings of EuroSys, Leuven, Belgium, April 2006.

[ENTZ04] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In the Proceedings of the 13th International World Wide Web Conference (WWW2004), New York, NY, USA, May 2004.

[EPZ05] Elnikety, S., F. Pedone, and W. Zwaenepoel, Database Replication Using Generalized Snapshot Isolation. IEEE Symposium on Reliable Distributed Systems (SRDS 2005), Orlando, Florida, October 2005.

[F05] Alan Fekete. Allocating Isolation Levels to Transactions. ACM Sigmod, Baltimore, Maryland, June 2005.

[F1-99] Alan Fekete. Serialisability and Snapshot Isolation. In Proceedings of the Australian Database Conference, pages 201–210, Auckland, New Zealand, January 1999.

[F2-99] L. Frank. Evaluation of the Basic Remote Backup and Replication Methods for High Availability Databases. Software Practice and Experience, 29:1339–1353, 1999.

[FLO+96] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. In proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pages 173–182, June 1996.

[GD03] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun. Iyengar. Application Specific Data Replication for Edge Services. In Proceedings of the Twelfth International Conference on the World Wide Web, pages 449–460. ACM Press, 2003.

[GHOD96] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal (Canada), June 1996.

[HCG+06] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, R. Bianchini. Mercury and Freon: Temperature Emulation and Management for Server Systems. In Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS'06), San Jose, CA, October, 2006.

[J95] K. Jacobs. Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7. Technical report number A33745, Oracle Corporation, Redwoord City, CA, July 1995.

[KA00] Bettina Kemme and Gustavo Alonso. Don't be Lazy, Be Consistent: Postgres-R, a New Way to Implement Database Replication. In Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000), Cairo, Egypt, September 2000.

[KA98] Bettina Kemme and Gustavo Alonso. A Suite of Database Replication Protocols Based on Group Communication Primitives. In Proceedings 18th International Conference on Distributed Computing Systems (ICDCS), Amsterdam, The Netherlands, May 1998.

[KS00] M. Karlsson, and P. Stenström. An Analytical Model of the Working-Set Sizes in Decision-Support Systems. In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '00). Santa Clara, CA, USA, June 2000.

[L98] L. Lamport. The Part-time Parliament. ACM Transactions on Computer Systems, 16(2):133-169, May 1998.

[L99] Andrea Lodi. Algorithms for Two-Dimensional Bin Packing and Assignment Problems. PhD thesis. Uiversita Degli Studi di Bologna. 1999.

[LKPJ05] Y. Lin, B. Kemme, M. Patiño-Martínez, R. Jiménez-Peris. Middleware Based Data Replication Providing Snapshot Isolation. ACM International Conference on Management of Data (SIGMOD), Baltimore, Maryland, June 2005.

[OW] ObjectWeb Consortium. Rice University bidding system. http://rubis.objectweb.org/.

[Ora97] Data Concurrency and Consistency, Oracle8 Concepts, Release 8.0: Chapter 23. Technical report, Oracle Corporation, 1997.

[P86] Christos Papadimitriou. The Theory of Database Concurrency Control. Computer Science Press, 1986.

[PA04] Christian Plattner, Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 2004.

[PAB+98] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1998.

[PG] PostgreSQL, SQL Compliant, Open Source Object-Relational Database Management System. http://www.postgresql.org/.

[PJKA05] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, G. Alonso. Consistent Database Replication at the Middleware Level. ACM Transactions on Computer Systems (TOCS). Volume 23, No. 4, 2005, pp 1-49.

[PL91] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. SIGMOD Record (ACM Special Interest Group on Management of Data), 20(2):377–386, June 1991.

[RS04] Robbert van Renesse, and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. Sixth Symposium on Operating Systems Design and Implementation (OSDI '04). USENIX Association, (San Francisco, California, December 2004).

[RT04] M. Ronström and L. Thalmann. MySQL Cluster Architecture Overview. MySQL Technical White Paper, 2004.

[Sch90] F. B. SCHNEIDER. Implementing fault-tolerant services using the state machine approach: a tutorial. In ACM Computing Surveys. 22 (4):299–319, December 1990.

[TPC] Transaction Processing Performance Council – http://www.tpc.org/.

[WK05] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In proceedings of International Conference on Data Engineering (ICDE), April 2005.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000), Taipei, Taiwan, April 2000.

[ZBCS99] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. HACC: An Architecture for Cluster-Based Web Servers. In Proceedings of the 3rd USENIX Windows NT Symposium, Seattle, WA, July 1999.

[ZP06] V. Zuikeviciute and F. Pedone. Conflict-Aware Load-Balancing Techniques for Database Replication. USI Technical Report, 2006.