Service-Oriented Data Denormalization for Scalable Web Applications

Zhou Wei*
Tsinghua University
Beijing, China
zhouw@few.vu.nl

Jiang Dejun*
Tsinghua University
Beijing, China
jiangdj@few.vu.nl

Guillaume Pierre
Vrije Universiteit
Amsterdam, The Netherlands
gpierre@cs.vu.nl

Chi-Hung Chi
Tsinghua University
Beijing, China
chichihung@mail.tsinghua.edu.cn

Maarten van Steen
Vrije Universiteit
Amsterdam, The Netherlands
steen@cs.vu.nl

ABSTRACT

Many techniques have been proposed to scale web applications. However, the data interdependencies between the database queries and transactions issued by the applications limit their efficiency. We claim that major scalability improvements can be gained by restructuring the web application data into multiple independent data services with exclusive access to their private data store. While this restructuring does not provide performance gains by itself, the implied simplification of each database workload allows a much more efficient use of classical techniques. We illustrate the data denormalization process on three benchmark applications: TPC-W, RUBiS and RUBBoS. We deploy the resulting service-oriented implementation of TPC-W across an 85-node cluster and show that restructuring its data can provide at least an order of magnitude improvement in the maximum sustainable throughput compared to master-slave database replication, while preserving strong consistency and transactional properties.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of systems**]: Design studies; H.3.4 [**Information Storage and Retrieval**]: Systems and Software.

General Terms

Performance.

Keywords

Scalability, Web applications, data denormalization.

1. INTRODUCTION

The world-wide web has taken an important place in everyday's life. Many businesses rely on it to provide their customers with immediate access to information. However, to retain a large number of customers, it is important to guarantee a reasonable access performance regardless of the request load that is addressed to the system. Web application hosting systems therefore need the ability to scale their capacity according to business needs. Many research efforts have been made to provide scalable infrastructures for static content. However, scaling web applications that dynamically generate content remains a challenge.

Web applications are commonly used to generate dynamic content that is personalized to individual clients. Each user's request triggers the execution of specific business logic, which issues any number of queries to an underlying database before returning a result to the client. Queries can simply extract information from the database, but UDI queries will also Update, Delete or Insert information to the database. Queries may also be grouped into database transactions.

Many techniques have been proposed to improve the scalability of Web applications. Scaling application-specific computation is relatively easy as requests can be distributed across any number of independent application servers running identical code. Similarly, one can reduce the network bottleneck between the application and database servers [33]. The main challenge, however, is to scale access to application data. Besides classical techniques such as master-slave database replication, new techniques exploit knowledge of the application data access behavior. Database query caching relies on high temporal locality, and uses prior knowledge of data overlap between different query templates to efficiently implement invalidations [2, 4, 21, 30]. A query template is a parametrized SQL query whose parameter values are passed to the system at runtime. Partial replication techniques use similar information to reduce the data replication degree and limit the cost of database updates [14, 29]. However, we observe that these techniques work best under very simple workloads composed only of a few different query templates. When the number of templates grows, an increasing number of constraints reduces their efficiency: database caching mechanisms need to invalidate more cached queries upon each update to maintain consistency, and partial replication is increasingly limited in the possible choices of functionally correct data placements.

In this paper, we claim that scalable Web applications should not be built along the traditional monolithic three-tier architecture. Instead, restructuring the application data into independent data services, each of which having exclusive access to its private data store, allows one to reduce the workload complexity of each of the services. While this restructuring by itself does not lead to any performance improvements, it does allow for a more effective applica-

* Zhou Wei and Jiang Dejun also work at Vrije Universiteit Amsterdam. This work is supported by the China National Science Foundation Project #90604028 and 863 project #2007AA01- Z122.

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

tion of the aforementioned optimization techniques, thus leading to significantly better scalability. Importantly, this does not imply any loss in terms of transactional or consistency properties.

Restructuring a monolithic Web application composed of Web pages that address queries to a single database into a group of independent Web services querying each other requires to rethink the data structure for improved performance – a process sometimes named *denormalization*. Future Web applications should preferably be designed from the start along a service-oriented architecture. However, for existing monolithic applications we show how one can denormalize the data into data services.

To demonstrate the effectiveness of our proposal, we study three web application benchmarks: TPC-W [32], RUBiS [3] and RUBBoS [26]. We show how these applications can be restructured into multiple independent data services, each with a very simple data access pattern. We then focus on the UDI-intensive data services from TPC-W and RUBiS to show how one can host them in a scalable fashion. For RUBBos, this is almost trivial. Finally, we study the scalability of TPC-W, the most challenging of the three benchmarks, and demonstrate that the maximum sustainable throughput grows linearly with the quantity of hosting resources used. We were thus able to scale TPC-W by an order of magnitude more than traditional systems.

This paper is structured as follows. Section 2 presents related work. Then, Section 3 details our system model and the issues that we need to face. Section 4 presents the process of data denormalization, while Section 5 shows how individual data services can be scaled. Section 6 presents performance evaluations. Finally, Section 7 concludes.

2. RELATED WORK

In the past years, many techniques have been proposed to improve the scalability of web applications. The simplest one is edge-server computing where requests are distributed among several edge servers running the same code [9, 24]. Although this technique is very effective at scaling the computation part of the applications, the main challenge is to scale the access to the application data.

Replication is a common technique to improve the throughput of a RDBMS. Many RDBMS replication solutions aim at replicating data across multiple servers within a cluster [5, 18, 23, 25]. Database replication allows one to distribute read queries among the replicas. However, in these solutions, all UDI queries must first be executed at a master database, then propagated and re-executed at all other "slave" databases using 2-phase commit or snapshot isolation mechanisms. A few commercial database systems such as Oracle allow one to optimize the re-execution of UDI queries at the slaves by transferring a log of the execution at the master. However, these techniques do not improve the maximum throughput as they require a single master server to execute all UDI queries. The throughput of the master server then determines the total system's throughput.

As extensively discussed in [31], a number of techniques have been developed specifically to scale Web applications. Most of these techniques exploit the fact that Web applications issue a fixed number of query templates to the database. Several systems propose to cache the result of database queries at the edge servers. Static knowledge of conflicts among different query templates allows one to invalidate caches efficiently when a database update occurs. However, such systems work best under workloads with high query locality, few UDI queries, and few dependencies between query templates. Furthermore, the efficiency of caches does not grow linearly with the quantities of resources assigned to it, so caches alone cannot provide arbitrary scalability.

Another approach based on query templates relies on partial database replication, where not all tables get replicated to all database servers [14]. This allows one to reduce the number of UDI queries that must be issued to each database server. However, although this technique allows one to improve the system throughput, its efficiency is constrained by the templates that query multiple tables simultaneously (because of join queries or database transactions). In contrast, our proposal also relies to a certain extent on partial data replication, but at a much finer granularity which allows one to reduce the data placement constraints.

In [11], the authors propose an edge computing infrastructure where the application programmers can choose the best suited data replication and distribution strategies for the different parts of application data. By carefully reducing the consistency requirements and selecting the replication strategies, this approach can yield considerable gains in performance and availability. However, it requires that the application programmers have significant expertise in domains such as fault-tolerance and weak data consistency. In contrast, we strive to present a systematic method to denormalize and restructure data, with no implied loss in consistency or transactional properties. Note that, although we only focus on performance issues in this paper, improving availability can be realized by applying classical techniques to each individual service.

Several recent academic and industrial research efforts have focused on the design of specialized data structures for scalable and highly available services [6, 10, 13]. These systems usually provide a simple key-based put/get interface and focus on scalability and availability properties rather than rich functionality or transactional properties. These design choices mean that these systems implicitly or explicitly assume that an application has been decomposed into separate entities with very simple data access patterns. In contrast, this paper demonstrates how one can design a Web application along such a service-oriented architecture with simplified workload.

Data fragmentation techniques have been commonly used in the design of distributed relational database systems [15, 19, 20, 22]. In these works, tables are partitioned either vertically or horizontally into smaller fragments. Partitioning schemes are determined according to a workload analysis in order to optimize access time. However, these techniques do not fundamentally change the structure of the data, which limits their efficiency. Furthermore, changes in the workload require to constantly re-evaluate the data fragmentation scheme [17]. We consider that dynamic environments such as Web applications would make such an approach impractical. In contrast, we propose a one-time modification in the application data structure. Further workload fluctuations can be handled by scaling each service independently according to its own load.

Even though data denormalization is largely applied to improve the performance of individual databases, few research efforts have been made to systematically study them [27, 28]. Data denormalization often creates data redundancy by adding extra fields to existing tables so that expensive join queries can be rewritten into simpler queries. This approach implicitly assumes the existence of a single database, whose performance must be optimized. In contrast, we apply similar denormalization techniques in order to scale the application throughput in a multi-server system. Denormalization in our case allows one to distribute UDI queries among different data services, and therefore to reduce the negative effects of UDIs on the performance of replicated databases.

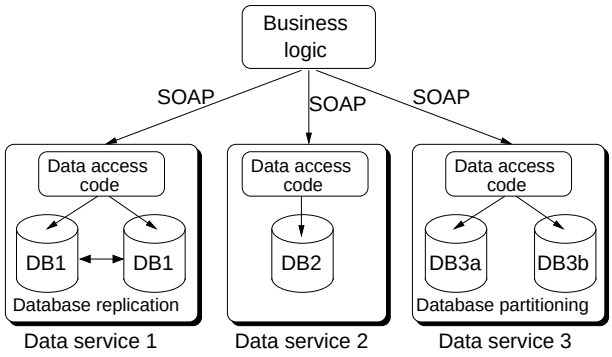


Figure 1: System model

3. SYSTEM MODEL

3.1 Goal

The idea behind our work is that the data access pattern of traditional monolithic Web applications is often too complex to be efficiently handled by a single technique. Indeed, proposed techniques work best under specific simple access patterns. Data replication performs best with workloads containing few or no UDI; query caching requires high temporal locality and not too many UDIs; partial replication or even data partitioning demand that queries do not span multiple partitions.

We claim that major gains in scalability can be obtained by restructuring Web application data into a collection of independent data services, where each service has exclusive access to its private data store. While such restructuring does not provide any performance improvement by itself, it considerably simplifies the data access pattern generated by each service. This allows one to apply appropriate scaling techniques to each service.

Figure 1 shows the system model of a Web application after restructuring. Instead of being hosted in a single database, the application data are split into three separate databases DB1, DB2 and DB3. Each database is encapsulated into a data service which exports a service interface to the application business logic. Each data service and its database can then be hosted independently using the technique that suits it best according to its own data access pattern. Here, DB1 is replicated across two database servers, DB2 is hosted by only one server, while DB3 has been further partitioned into DB3a and DB3b. Note that splitting the application data into independent services also improves separation of concerns: details about the internal hosting architecture of a data service are irrelevant to the rest of the application.

3.2 Data denormalization constraints

Denormalizing an application's data into independent data services requires deep changes to the structure of the data. For example, a table containing fields $\langle key, attr1, attr2\rangle$ and queried by templates "`SELECT key FROM Table where attr1=?`" and "`SELECT key FROM Table where attr2=?`" may be split into two tables $\langle key, attr1\rangle$ and $\langle key, attr2\rangle$, which may belong to two different data services.

However, not all tables can be split arbitrarily. In practice, data accessed by different queries often overlap, which constrains the denormalization. We identify two types of constraints: transactions and query data overlap.

Although database transactions are known as an adversary to performance, they sometimes cannot be avoided. An example is a checkout operation in an e-commerce application where a product order and the corresponding payment should be executed atomically. ACID requirements provide a strong motivation for maintaining all data accessed by one transaction inside a single database, and therefore inside a single data service. Splitting such data into multiple services would impose executing distributed transactions across multiple services, for example, using protocols such as 2-phase commit. We expect that this would negate the performance gains of the data decomposition.

Another source of constraints is created by ordinary queries executed outside transactions. Similar to constraints created by transactions, it seems logical to cluster the data accessed by each query. However, in most cases the overlap of different queries would lead to creating a single data service. Instead, we can apply two other transformations. First, certain complex database queries can be rewritten into multiple, simpler queries. Doing this reduces the data inter-dependency and allows better data restructuring. Second, data dependencies induced by overlapping queries can also be reduced by replicating certain data to multiple services. However, this implies a trade-off between the gains of splitting the data into more services and the costs of replicating update queries to these data over multiple services.

3.3 Scaling individual data services

In all our experiments, we noticed that the services resulting from data denormalization maintain extremely simple data structures and are queried by very few query templates. Such a simple workload considerably simplifies the task of hosting services in a scalable fashion. For example, some data services receive very few or even no UDI queries at all. Such services can therefore benefit from massive caching or replication. On the other hand, some other services concentrate on large number of UDI queries, often grouped together inside transactions. Such services are clearly harder to scale. However, they at least benefit from the fact that they receive less queries than the database of a monolithic application would. Additionally, we show in Section 5.1 that such services can often be *partitioned* so that UDI queries are distributed across multiple database servers.

4. DATA DENORMALIZATION

Service-oriented data denormalization exploits the fact that UDI queries and transactions often access only a part of the columns of a table. Decomposing such tables into multiple smaller ones helps distributing UDI queries and transactions to more data services, and thereby simplifies their workload. As discussed in Section 3, two main constraints must be taken into account when denormalizing an application's data. First, one should split the data into the largest possible number of services, such that no transaction or UDI query in the workload spans multiple services. Second, one must make sure that read queries can continue to operate over the then partitioned data.

4.1 Denormalization and transactions

As discussed in previous sections, we need to cluster the data into services such that no transaction overlaps multiple data services. To this end, we first mark which data columns are accessed by each transaction. Then, simple clustering techniques can be applied to decompose the data into the largest possible number of independent data services.

We distinguish three types of "transactions" that must be taken into account here. First, real database transactions require ACID properties. This means that all the data they access must be accessed atomically and must be placed into the same service. One exception to this rule is formed by data columns that are never up-

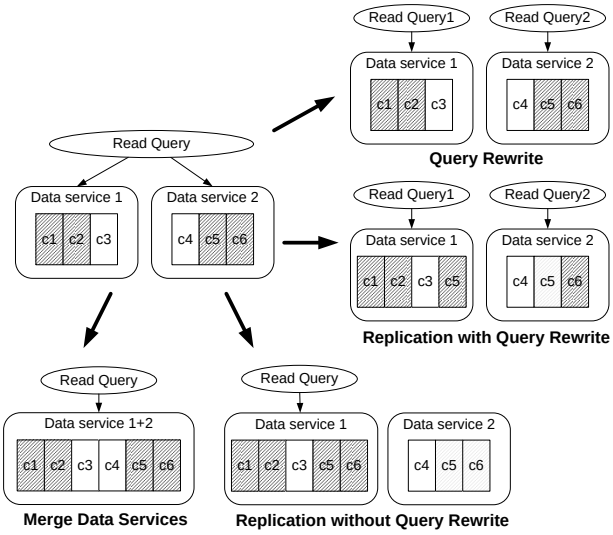


Figure 2: Different denormalization techniques for read queries

dated, neither by the transaction in question nor by any other query in the workload. An example is the table that matches zipcodes to local names. Such read-only data does not need to be placed in the same data service, and can be abstracted as a separate data service.

The second type of transaction is a so-called "atomic set," where only the Atomicity property of a normal transaction is necessary. Atomic sets appear, for example, in TPC-W, where a query that reads the content of a shopping cart and the one that adds another element must be executed atomically [34]. For such atomic sets, only the columns that are updated must be local to the same data service to be able to provide atomicity. Columns that are only read by the atomic set can reside outside the service, as they are not concerned by the atomicity property[1].

Finally, UDI queries that are not part of a transaction must be executed atomically, and therefore must be considered as an atomic set composed of a single query.

Once one has marked each transaction, UDI query and atomic set with the data columns that should be kept in a single service, simple clustering techniques can provide the first step of decomposition of the database columns into services. However, this step is not functional, as it accommodates only the needs of transactions and UDI queries. To become functional, one must further update this data model to take read queries into consideration.

4.2 Denormalization and read queries

Clearly, one can consider read queries similarly to UDI queries and transactions, and cluster data services further such that no read query overlaps multiple services. However, applying this method would increase the constraints to the data decomposition and lead to coarse-grain data services, possibly with a single data service for the whole application.

Instead, as shown in Figure 2, two different operations can be applied. First, certain read queries can be rewritten into a series of multiple sub-queries, where each sub-query can execute in one data service. For example, in TPC-W, the `CUSTOMER` and `ORDER` tables are located in different data services, whereas the following query spans both tables with a join operation: "`SELECT o_id FROM customer, orders WHERE customer.c_id = orders.o_c_id AND c_uname = ?`". However, this query can be easily rewritten into two sub-queries that access only one table: i) "`SELECT c_id FROM customer WHERE c_uname = ?`"; and ii) "`SELECT o_id FROM orders WHERE o_c_id=?`". The returned result of the first query is used as input for the second one and the final result is returned by the second query.

Another transformation often applied in traditional database denormalization techniques consists of replicating data from certain database tables to other tables. This allows one to transform join queries into simpler queries. Note that traditional denormalization applies this technique to optimize the efficiency of query execution within a single database whereas we apply this technique to be able to split the data into independent data services. For example, the following query accesses two tables in two different data services: "`SELECT item.i_id,item.i_title FROM item,order_line WHERE item.i_id = order_line.ol_i_id AND item.subject=? LIMIT 50`". Replicating column `i_subject` from table `ITEM` to the other data service allows one to transform this query and target a single data service. The only constraint is that any update to the `i_subject` column must be applied at both data services, preferably within a (distributed) transaction. This scheme is therefore applicable only in cases where the data to be replicated are rarely updated.

To conclude, complex query rewriting should be the preferred option if the semantics of the query allows it. Otherwise, column replication may be applied if the replicated data are never or seldom updated. In last resort, when neither query rewriting nor column replication is possible, merging the concerned data services is always correct, yet at the cost of coarse-grain data services.

4.3 Case studies

To illustrate the effectiveness of our data denormalization process, we applied it to three standard Web applications: TPC-W, RUBiS and RUBBoS.

4.3.1 TPC-W

TPC-W is an industry standard e-commerce benchmark that models an online bookstore similar to `Amazon.com` [32]. Its database contains 10 tables that are queried by 6 transactions, 2 atomic sets, 6 UDI queries that are not part of a transaction, and 27 read-only queries.

First, the transactions and atomic sets of the TPC-W workload impose the creation of four sets of transactions whose targeted data do not overlap. The first set contains transaction `Purchase`, and the two atomic sets `Docart` and `Getcart`; the second set contains the `Adminconfirm` transaction, the third set contains only the `Updaterelated` transaction. Finally, the last set contains `Addnewcustomer`, `Refreshsession` and `Enteraddress`. This means for example that the original `ITEM` table from TPC-W must be split into five tables: `ITEM_STOCK` contains the primary key `i_id` and the column `i_stock`; table `ITEM_RELATED` contains `i_id` and `i_related1-5`; table `ITEM_DYNAMIC` contains `i_id`, `i_cost`, `i_thumbnail`, `i_image` and `i_pub_date`; the last table contains all the read-only columns of table `ITEM`.

The result of the first denormalization step is composed of five data services: a *Financial* data service contains tables `ORDERS`, `ORDER_ENTRY`, `CC_XACTS`, `SHOPPING_CART`, `SHOPPING_CART_ENTRY` and `ITEM_STOCK`; data service

[1]In the case of actual database transactions, these data columns must reside inside the data service to be able to provide the Isolation part of ACID properties.

Data service	Data Tables(included columns)	Requests
Financial	ORDERS ORDER_ENTRY CC_XACTS I_STOCK(i_stock) SHOPPING_CART SHOPPING_CART_ENTRY	getLastestOrderInfo, createEmptyCart, addItem, refreshCart, resetCartTime, getCartInfo, getBesterIDs, computeRelatedItems, purchase
Customer	CUSTOMER ADDRESS COUNTRY	getAddress, setAddress, getCustomerID, getCustomerName, getPassword, getCustomerInfo, login, addNewCustomer, refreshSession
Item_dynamic	ITEM_DYNAMIC(i_cost i_pub_date i_subject i_image i_thumbnail)	getItemDynamicInfo, getLatestItems, setItemDynamicInfo
Item_basic	ITEM_BASIC(i_title i_subject) Author	getItemBasicInfo, searchByAuthor, searchByTitle, searchBySubject
Item_related	ITEM_RELATED(i_related1-5)	getRelatedItems, setItemRelated
Item_publisher	ITEM_PUBLISHER(i_publisher)	getPublishers
Item_detail	ITEM_DETAIL(i_srp i_backing)	getItemDetails
Item_other	ITEM_OTHER(i_isbn i_page i_dimensions i_desc i_avail)	getItemOtherInfo

Table 1: Data services of the denormalized TPC-W

Item_related takes care of items that are related to each other, with table `ITEM_RELATED`; data service *Item_dynamic* takes care of the fields of table `ITEM` that are likely to be updated by means of table `ITEM_DYNAMIC`; finally, data service "Customer" contains customer-related information with tables `CUSTOMER`, `ADDRESS` and `COUNTRY`. The remaining tables from TPC-W are effectively read-only and are clustered into a single data service. This read-only data service can remain untouched, but for the sake of the explanation we split it further during the second denormalization step.

The second step of denormalization takes the remaining read queries into account. We observe that most read queries can either be executed into a single data service, or be rewritten. One read query cannot be decomposed: it fetches the list of the best-selling 50 books that belong to a specified subject. However, the list of book subjects `i_subject` is read-only in TPC-W, so we replicate it to the *Financial* data service for this query[2]; `i_subject` is also replicated to the *Item_dynamic* data service for a query that obtains the list of latest 50 books of a specified subject.

The remaining read-only data columns can be further decomposed according to the query workload. For example, the "Search" web page only accesses data from columns `i_title`, `i_subject` and table `AUTHOR`. We can thus encapsulate them together as the *Item_basic* service. We similarly created three more read-only data services.

The final result is shown in Table 1. An important remark is that, although denormalization takes only data access patterns into account, each resulting data service has clear semantics and can be easily named. This result is in line with observations from [12], where examples of real-world data services are discussed.

4.3.2 RUBBoS

RUBBoS is a bulletin-board benchmark modeled after `slashdot.org` [26]. It consists of 8 tables requested by 9 UDI queries and 30 read-only queries. RUBBoS does not contain any transactions. Six tables incur UDI workload, while the other two are read-only. Furthermore, all UDI queries access only one table. It is therefore easy at the end of the first denormalization step to encapsulate each table incurring UDI queries into a separate data service.

[2]Note that we cannot simply move this column into the *Financial* service, as it is also accessed in combination with other read-only tables.

Data Service	Tables	Transactions
User	USERS[*U*] COMMENTS[*C*]	Storecomment(*U*,*C*) Registeruser(*U*)
Auction	ITEMS[*I*] BUY_NOW[*N*] BIDS[*B*]	Storebuynow(*I*,*N*) Registeritem(*I*) Storebid(*I*,*B*)
Categories	CATEGORIES	-
Regions	REGIONS	-

Table 2: Data services of RUBiS

All read queries can be executed in only one table except two queries which span two tables: one can be rewritten into two simpler queries; the other one requires to replicate selected items from `OLD_STORIES` into the `USERS` table. The `OLD_STORIES` table, however, is read-only so no extra cost is incurred from such replication. Finally, the two read-only tables are encapsulated as separate data services.

RUBBoS can therefore be considered as a very easy case for data denormalization.

4.3.3 RUBiS

RUBiS is an auction site benchmark modeled after `eBay.com` [3]. It contains 7 tables requested by 5 update transactions. Except for the read-only tables `REGIONS` and `CATEGORIES`, the other five tables are all updated by INSERT queries, which means that they cannot be easily split. This means that the granularity at which we can operate is the table. The transactions impose the creation of two data services: the "Users" data service contains tables `USERS` and `COMMENTS`, while the "Auction" data service contains tables `BUY_NOW`, `BIDS` and `ITEMS`. The final result of data denormalization is shown in Table 2.

RUBiS is a difficult scenario for denormalization because none of its tables can be split following the rules described in Section 4.1. We note that in such worst-case scenario, denormalization is actually equivalent to the way GlobeTP [14] would have hosted the application. We will show however in the next section that scaling the resulting data services is relatively easy.

5. SCALING INDIVIDUAL DATA SERVICES

In all cases we examined, the workload of each individual data service can be easily characterized. Some services incur either

```
1 Insert into ORDER with o_id=id;
2 Insert into CC_XACTS with cx_o_id=id;
3 foreach item i within the order do
4 |   Insert into ORDER_ENTRY with ol_o_id=id, ol_i_id=i;
5 |   Update I_STOCK set i_stock=i_stock-qty(i) where i_id=i;
6 end
7 Update SHOPPING_CART where sc_id=id;
8 Delete from SHOPPING_CART_ENTRY where scl_sc_id=id;
```

Algorithm 1: The purchase transaction

read-only or read-dominant workload. These services can be scaled up by classical database replication or caching techniques [31]. Other services incur many more UDI queries, and deserve more attention as standard replication techniques are unlikely to provide major performance gains. Furthermore, update-intensive services also often incur transactions, which makes the scaling process more difficult. Instead, partial replication or data partitioning techniques should be used so that update queries can be *distributed* among multiple servers. We discuss two representative examples from TPC-W and RUBiS and show how they can be scaled up using relatively simple techniques.

5.1 Scaling the financial service of TPC-W

The denormalized TPC-W contains one update-intensive service: the *Financial* service. This service incurs a database update each time a client updates its shopping cart or does a purchase. However, all tables from this service, except one, are indexed by a shopping cart ID and all queries span exactly one shopping cart. This suggests that, instead of replicating the data, one can *partition* them according to their shopping cart ID.

The *Financial* data service receives two types of updates: updates on a shopping cart, and purchase transactions. The first one accesses tables `SHOPPING_CART` and `SHOPPING_CART-_ENTRY`. Table `SHOPPING_CART` contains the description of a whole shopping cart, while `SHOPPING_CART_ENTRY` contains the details of one entry of the shopping cart. If we are to partition these data across multiple servers, then one should keep a shopping cart and all its entries at the same server.

The second kind of update received by the *Financial* service is the `Purchase` transaction. We present this transaction in Algorithm 1. Similar to the `Updatecart` query, the `Purchase` transaction requires that the order made from a given shopping cart is created at the same server that already hosts the shopping cart and its entries. This allows one to run the transaction within a single server of the *Financial* service rather than facing the cost of a distributed transaction across replicated servers.

One exception to this easy data partitioning scheme is the `ITEM_STOCK` table, in which any element can potentially be referred to by any shopping cart entry. One simple solution would be to replicate the `ITEM_STOCK` table across all servers that host the *Financial* service. However, this would require to run the `Purchase` transaction across all these servers. Instead, we create an `ITEM_STOCK` table in each server of the *Financial* service in which all item details are identical except the available stock which is *divided* by the number of servers. This means that each server is allocated a part of the stock that it can sell without synchronizing with other servers. Only when the stock available at one server is empty, does it need to execute a distributed transaction to re-distribute the available stock.

The *Financial* service receives two more read queries that access data across multiple data clusters. These queries retrieve respectively the 3333 and 10,000 latest orders from tables `ORDERS` and `ORDER_ENTRY` in order to obtain either the list of best-selling items or the items most related to a given other item. We implement these queries in a similar way to distributed databases. Each query is first issued at each server. The results are then merged into a single result set, and the relevant number of most recent orders is re-selected from the merged results.

In our implementation, we wanted to balance the load imposed by different shopping carts across all servers of the *Financial* service. We therefore marked each row of tables `SHOPPING_CART`, `SHOPPING_CART_ENTRY` and `ORDERS` with a key equal to the shopping cart ID. We then hash this ID to $H = (7id + 4)\%M$ (where M is the number of servers) to determine which server H should be responsible for that row. Our experiments show that this very simple hash function balances the load effectively in terms of data storage size and computational load.

This example shows that, even for relatively complex data services, the fact that each service has simple semantics and receives few different queries allows one to apply application-specific solutions. The resulting relative complexity of the service implementation, however, remains transparent to other parts of the application, which only need to invoke a simple service interface.

5.2 Scaling RUBiS

The denormalized RUBiS implementation contains two update-intensive services: "Auction" and "User." Similar to the previous example, most queries address a single auction or user by their respective IDs. We were thus able to partition the data rows between multiple servers. A few read-only queries span multiple auctions or users, but we could easily rewrite them such that individual queries would be issued at every server, before their results can be merged.

6. PERFORMANCE EVALUATION

As we have seen, RUBBoS and RUBiS are relatively simple to host using our denormalization technique. RUBBoS can be decomposed into several rarely updated data services. On the other hand, RUBiS requires coarser-grain update-intensive services, but they can be scaled relatively easily. We present here performance evaluations of TPC-W, which we consider as the most challenging of the three applications.

Our evaluations assume that the application load remains roughly constant, and focus on the scalability of denormalized applications. To support the fluctuating workloads that one should expect in real deployments, a variety of techniques exist to dictate when and how extra servers should be added or removed from each individual data service of our implementations [1, 7, 35].

We compare three implementations of TPC-W. "OTW" represents the unmodified original TPC-W implementation. We then compare its performance to "DTW", which represents the denormalized TPC-W where no particular measure has been taken to scale up individual services. Finally, "STW" (scalable TPC-W) represents the denormalized TPC-W with scalability techniques enabled. All three implementations are based on the Java implementation of TPC-W from the University of Wisconsin [16]. For performance reasons we implemented the data services as servlets rather than SOAP-based Web services.

We first study the performance of OTW and DTW to investigate the costs and benefits of data denormalization with no scalability techniques being introduced. We then study how replication and data partitioning techniques allow us to scale individual data services of TPC-W. Finally, we deploy the three implementations on an 85-node cluster and compare their scalability in terms of throughput.

6.1 Experimental setup

All experiments are performed on the DAS-3, an 85-node Linux-based server cluster [8]. Each machine in the cluster has a dual-CPU / dual-core 2.4 GHz AMD Opteron DP 280, 4 GB of memory and a 250 GB IDE hard drive. Nodes are connected to each other with a gigabit LAN such that the network latency between the servers is negligible. We use Tomcat v5.5.20 as application servers, PostgreSQL v8.1.8 as database servers, and Pound 2.2 as load balancers to distribute HTTP requests among multiple application servers.

Before each experiment, we populate the databases with 86,400 customer records and 10,000 item records. Other tables are scaled according to the benchmark requirements. The client workload is generated by Emulated Browsers (EBs). We use the number of EBs to measure the client workload. The workload model incorporates a think time parameter to control the amount of time an EB waits between receiving a response and issuing the next request. According to the TPC-W specification, think times are randomly distributed with exponential distribution and average value 7 seconds.

TPC-W defines three standard workloads: the browsing, shopping and ordering mixes, which generate 5%, 20% and 50% update interactions respectively. Unless otherwise specified, our experiments rely on the shopping mix.

6.2 Costs and benefits of denormalization

The major difference between a monolithic Web application and its denormalized counterpart is that the second one is able to distribute its UDI workload across multiple machines. Even though such an operation implies a performance drop when hosting the application on a single machine, it improves the overall system scalability when more machines are used. In this section, we focus on the costs and benefits of data denormalization when no special measure is taken to scale the denormalized TPC-W.

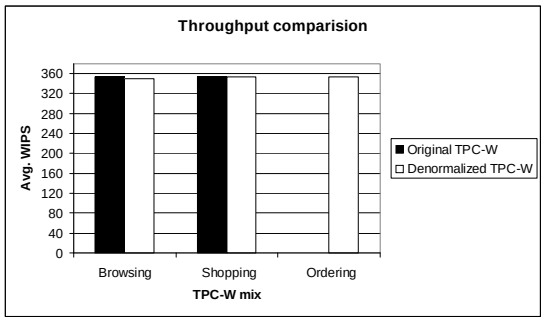
We exercise the OTW and DTW implementations using 2500 EBs, under each of the three standard workload mixes. Both systems are deployed over one application server and 8 database servers. In the case of OTW, the database servers are replicated using the standard PostgreSQL master-slave mechanism. DTW is deployed such that each data service is hosted on a separate database server.

We measure the system performance in terms of WIRT (Web Interaction Response Time) as well as WIPS (Web Interactions Per Second). According to the TPC-W specification, we defined an SLA in terms of the 90th percentile of response times for each type of Web interaction: namely, 90% of web interactions of each type must complete under 500 ms. The only exception is the "Admin confirm" request type, which does not have an SLA requirement. This request is issued only by the system administrator, and therefore does not influence the client-perceived performance of the system.

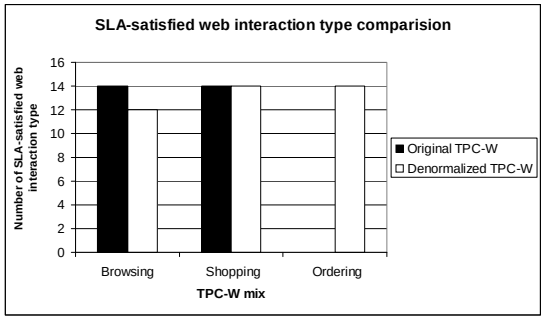
Figure 3 shows the performance of the different systems under each workload. Figure 3(a) shows the achieved system throughput, whereas Figure 3(b) shows the number of query types for which the SLA was respected.

The browsing mix contains very few UDI queries. Both implementations sustain roughly the same throughput. However, the denormalized TPC-W fails to meet its SLA for two out of the 14 interaction types. This is due to the fact that the concerned interactions heavily rely on queries that are rewritten to target multiple, different data services. These calls are issued sequentially, which explains why the corresponding request types incur higher latency.

At the other extreme, the ordering mix contains the highest fraction of UDI queries. Here, DTW sustains a high throughput and respects all its SLAs, while OTW simply crashes because of overload. This is due to the fact that DTW *distributes* its UDI queries across all database servers while OTW *replicates* them to all servers. Finally, the shopping mix constitutes a middle case where both implementations behave equally good.



(a) Average throughput comparison



(b) SLA-satisfied web interaction type number comparison

Figure 3: Throughput and performance comparison between original TPC-W and denormalized TPC-W. Note that the Ordering mix for the original TPC-W overloaded and subsequently crashed the application.

We conclude that data denormalization improves the performance of UDI queries at the cost of a performance degradation of rewritten read queries. We note, however, that the extra cost of read queries does not depend on the number of server machines, whereas the performance gain of UDI queries is proportional to the size of the system. This suggests that the denormalized implementation is more scalable that the monolithic one, as we will show in the next sections.

6.3 Scalability of individual data services

We now turn to study the scalability of each data service individually. We study the maximum throughput that one can apply to each service when using a given number of machines, such that the SLA is respected.

Since we now focus on individual services rather than the whole application, we need to redefine the SLA for each individual data service. As one application-level interaction generates on average five data service requests, we roughly translated the interaction-level SLA into a service-level SLA that requires 90% of service requests to be processed within 100 ms. The *Financial* service is significantly more demanding than other services, since about 10% of its requests take more than 100 ms irrespectively of the workload. We therefore relax its SLA and demand that only 80% of queries return within 100 ms.

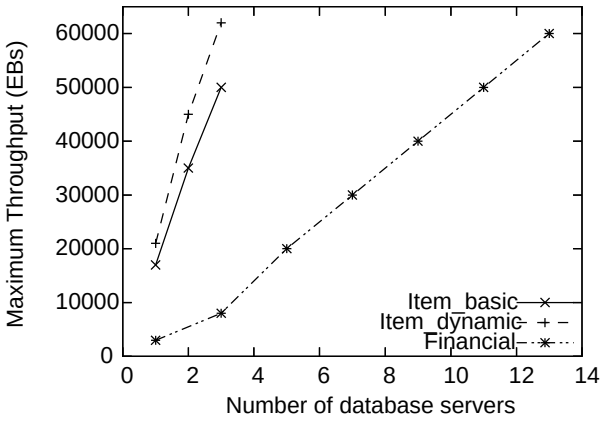


Figure 4: Scalability of individual TPC-W services

We measure the maximum throughput of each data service by increasing the number of EBs until the service does not respect its SLA any more. To generate flexible reproducible workloads for each data service, we first ran the TPC-W benchmark several times under relatively low load (1000 EBs) and collected the logs of the invocation of data service interfaces. We obtained 72 query logs, each representing the workload of 1000 EBs for a duration of 30 minutes. We can thus generate any desired workload, from 1000 EBs to 72,000 EBs step by 1000 EBs, by replaying the right number of elementary log files across one or more client machines concurrently.

Figure 4 shows the throughput scalability of three representative data services from the scalable TPC-W. The *Item_basic* data service is read-only. It is therefore trivial to increase its throughput by adding database replicas. Similarly, the *Item_dynamic* service receives relatively few UDI queries, and can be scaled by simple master-slave replication.

On the other hand, the *Financial* service incurs many database transactions and UDI queries, which implies that simple database replication will not produce major throughput improvements. We see, however, that the implementation discussed in Section 5.1 exhibits a linear growth of its throughput as the number of database servers increases.

To conclude, we were able to scale all data services to a level where they could sustain a load of 50,000 EBs. Different services have different resource requirements to reach this level, with the *Item_basic*, *Item_dynamic* and *Financial* services requiring 3, 3, and 13 database servers, respectively.

We believe that all the data services can easily be scaled further. We stopped at that point as 50,000 EBs is the maximum throughput that our TPC-W implementation reaches when we use the entire DAS-3 cluster for hosting the complete application.

6.4 Scalability of the entire TPC-W

We conclude this performance evaluation by comparing the throughput scalability of the OTW, DTW and STW implementations of TPC-W. Similar to the previous experiment, we exercised each system configuration with increasing numbers of EBs until the SLA was violated. In this experiment, we use the application-level definition of the SLA as described in Section 6.2.

Figure 5(a) compares the scalability of OTW, DTW and STW when using between 2 and 70 server machines. In all cases we started by using one application server and one database server. We then added database server machines to the configurations. In OTW, extra database servers were added as replicas of the monolithic application state. In DTW, we start with one database server for all services, and eventually reach a configuration with one database server per service. In STW, we allocated the resources as depicted in Figure 5(b). Note that in all cases, we deliberately over-allocated the number of application servers and client machines to make sure that the performance bottleneck remains at the database servers.

When using very few servers, OTW slightly outperforms DTW and STW. With increasing number of servers, OTW can be scaled up until about 6000 EBs when using 8 servers. However, when further adding servers, the throughput decreases. In this case, the performance improvement created by extra database replicas is counterbalanced by the extra costs that the master incurs to maintain consistency.

As no individual scaling techniques are applied to DTW, it can be scaled up to at most 8 database servers (one database server per service). The maximum throughput of DTW is around 3500 EBs. Note that this is only about half of the maximum achievable throughput of OTW. This is due to the extra costs brought by data denormalization, in particular the rewritten queries. Adding more database servers per service using database replication would not improve the throughput, as most of the workload is concentrated in the *Financial* service.

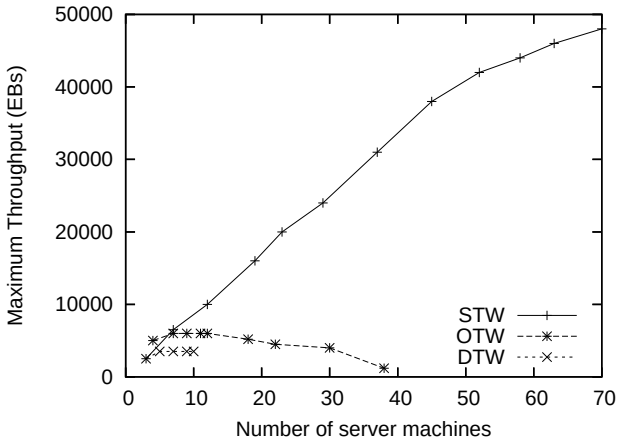
Finally, STW shows near linear scalability. It reaches a maximum throughput of 48,000 EBs when using 70 server machines (11 database servers for the *Financial* service, 12 database servers for the other services, 33 application servers and 14 load balancers). Taking into account the 14 client machines necessary to generate a sufficient workload, this configuration uses the entire DAS-3 cluster. The maximum throughput of STW at that point is approximately 8 times that of OTW, and 10 times that of a single database server.

We note that the STW throughput curve seems to start stabilizing around 50 server machines and 40,000 EBs. This is not a sign that we reached the maximum achievable throughput of STW. The explanation is that, as illustrated in Figure 4, 40,000 EBs is the point where many small services start violating their SLA with two database servers, and need a third database server. In our implementation each database server is used for a single service, which means that several extra database servers must be assigned to the small data services to move from 40,000 EBs to 50,000 EBs. We expect that using more resources the curve would grow faster again up to the point where the small data services need four servers.

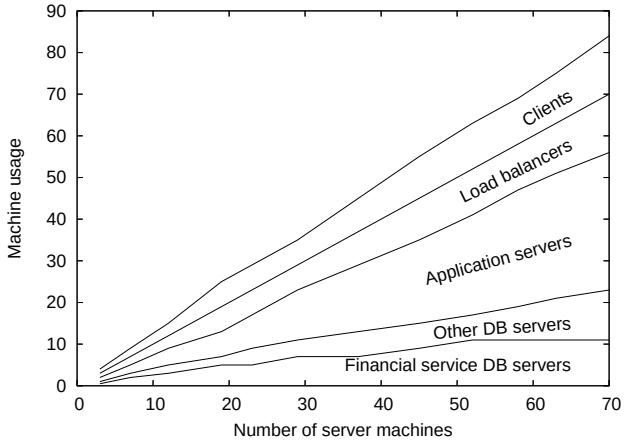
7. CONCLUSION

Most approaches toward scalable hosting of Web applications consider the application code and data structure as constants, and propose middleware layers to improve performance transparently to the application. This paper takes a different stand and demonstrates that major scalability improvements can be gained by allowing one to denormalize an application's data into independent services. While such restructuring introduces extra costs, it considerably simplifies the query access pattern that each service receives, and allows for a much more efficient use of classical scalability techniques. We applied this methodology to three standard benchmark applications and showed that it allows TPC-W, the most challenging of the three, to scale by at least an order of magnitude compared to master-slave database replication. Importantly, data denormalization does not imply any loss in terms of consistency or transactional properties. This aspect makes our approach unique compared to, for example, [11].

In our experience, designing the data denormalization of an application from its original data structure and query templates takes



(a) Maximum system throughput



(b) Allocation of machine resources for STW

Figure 5: Scalability of TPC-W hosting infrastructure

only a few hours. On the other hand, the work required for the actual implementation of the required changes highly depends on the complexity of each data service.

Data denormalization exploits the fact that an application's queries and transactions usually target few data columns. This, combined with classical database denormalization techniques such as query rewriting and column replication, allows us to cluster the data into disjoint data services. Although this property was verified in all applications that we examined, one cannot exclude the possible existence of applications with sufficient data overlap to prevent any service-oriented denormalization. This may be the case of transaction-intensive applications, whose ACID properties would impose very coarse-grained data clustering. It is a well-known fact that database transactions in a distributed environment imply important performance loss, so one should carefully ponder whether transactions are necessary or not.

The fact that denormalization is steered by prior knowledge of the application's query templates means that any update in the application code may require to restructure the data to accommodate new query templates. However, the fact that all data services resulting from denormalization have clear semantics makes us believe that extra application features could be implemented without the need to redefine data services and their semantics. One can also imagine to fully automate denormalization such that any necessary change in the data structure could be applied transparently to the application, using a proxy layer to translate the original application query templates into their data service-specific counterparts. We leave such improvements for future work.

8. REFERENCES

[1] B. Abrahao, V. Almeida, J. Almeida, A. Zhang, D. Beyer, and F. Safai. Self-adaptive SLA-driven capacity management for internet services. In *Proc. NOMS*, Apr. 2006.

[2] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proc. ICDE*, Mar. 2003.

[3] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *Proc. Intl. Workshop on Workload Characterization*, Nov. 2002.

[4] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18, June 2004.

[5] E. Cecchet. C-JDBC: a middleware framework for database clustering. *Data Engineering*, 27(2):19–26, June 2004.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. OSDI*, Nov. 2006.

[7] I. Cunha, J. Almeida, V. Almeida, and M. dos Santos. Self-adaptive capacity management for multi-tier virtualized environments. In *Proc. Intl. Symposium on Integrated Network Management*, May 2007.

[8] DAS3: The Distributed ASCI Supercomputer 3. `http://www.cs.vu.nl/das3/`.

[9] A. Davis, J. Parikh, and W. E. Weihl. Edge computing: Extending enterprise applications to the edge of the internet. In *Proc. WWW*, May 2004.

[10] G. DeCandia, D. Hastorum, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, Oct. 2007.

[11] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proc. WWW*, May 2003.

[12] J. Gray. A conversation with Werner Vogels. *ACM Queue*, 4(4):14–22, May 2006.

[13] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. OSDI*, 2000.

[14] T. Groothuyse, S. Sivasubramanian, and G. Pierre. GlobeTP: Template-based database replication for scalable web applications. In *Proc. WWW*, May 2007.

[15] Y. Huang and J. Chen. Fragment allocation in distributed database design. *Information Science and Engineering*, 17(3):491–506, May 2001.

[16] Java TPC-W implementation distribution. `http://www.ece.wisc.edu/~pharm/tpcw.shtml`.

[17] L. Kazerouni and K. Karlapalem. Stepwise redesign of distributed relational databases. Technical Report

HKUST-CS97-12, Hong Kong Univ. of Science and Technology, Dept. of Computer Science, Sept. 1997.

[18] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. VLDB*, Sept. 2000.

[19] S. Navathe, K. Karlapalem, and M.Ra. A mixed fragmentation methodology for initial distributed database design. *Computer and Software Engineering*, 3(4), 1995.

[20] S. Navathe and M. Ra. Vertical partitioning for database design: a graphical algorithm. *SIGMOD Records*, 18(2):440–450, 1989.

[21] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. Maggs, and T. Mowry. A scalability service for dynamic web applications. In *Proc. Conf. on Innovative Data Systems Research*, Jan. 2005.

[22] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, Feb. 1999.

[23] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proc. Middleware*, Oct. 2004.

[24] M. Rabinovich, Z. Xiao, and A. Agarwal. Computing on the edge: A platform for replicating internet applications. In *Proc. Intl. Workshop on Web Content Caching and Distribution*, Sept. 2003.

[25] M. Ronstrom and L. Thalmann. MySQL cluster architecture overview. MySQL Technical White Paper, Apr. 2004.

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

[27] G. L. Sanders and S. K. Shin. Denormalization effects on performance of RDBMS. In *Proc. HICSS*, Jan. 2001.

[28] S. K. Shin and G. L. Sanders. Denormalization strategies for data retrieval from data warehouses. *Decision Support Systems*, 42(1):267–282, Oct. 2006.

[29] S. Sivasubramanian, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In *Proc. WWW*, May 2005.

[30] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands, June 2006.

[31] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1):60–66, January-February 2007.

[32] W. D. Smith. TPC-W: Benchmarking an ecommerce solution. White paper, Transaction Processing Performance Council.

[33] N. Tolia and M. Satyanarayanan. Consistency-preserving caching of dynamic database content. In *Proc. WWW*, Nov. 2006.

[34] TPC-W frequently asked questions, question 2.10: "Why was the concept of atomic set of operations added and what are its requirements?", Aug. 1999.

[35] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Agile, dynamic capacity provisioning for multi-tier internet applications. In *Proc. ICAC*, June 2005.