Data Management Issues in Supporting Large-Scale Web Services

Philip A. Bernstein, Nishant Dani, Badriddine Khessib, Ramesh Manne, David Shutt
Microsoft Corporation
Redmond, WA, 98052-6399
{phil.bernstein, nishant.dani, badriddine.khessib, ramesh.manne, david.shutt}@microsoft.com

Abstract

This paper discusses technical problems that arise in supporting large-scale 24×7 web services based on experience at MSN with Windows Live^{TM} services. Issues covered include multi-tier architecture, costs of commodity vs. premium servers, managing replicas, managing sessions, use of materialized views, and controlling checkpointing. We finish with some possible research directions.

1 Introduction

Large-scale web services supported by fee-for-service or advertising revenue are heavy users of database technology. Such services include email, instant messaging, social networking, blogs, photo sharing, maps, shopping, and classifieds. They also include search services, though these typically use custom indexing engines, not relational database systems.

By large-scale, we mean systems with millions of users and thousands of servers. For example, Windows Live Messenger has thousands of servers and tens of millions of simultaneous online connections. It processes hundreds of millions of logins per day and has over 250 million unique users per month. Each login translates into queries to backend database servers for authentication and authorization. Users send billions of messages per day. The message transfer itself takes place independently of the database. Similarly large numbers have been reported for Hotmail® [1, 2].

These types of services are big business. The trailing 12-month revenue of Google, Windows Live, and Yahoo! combined is over $16B/year, most of which comes from such services. This annual revenue is comparable to the size of the market for database software products.

In general, these services require a lot of computing power, which is expensive to acquire and operate. A high-end commodity server to run a database system application currently costs upwards of $20K. A typical such system has 1-4 processors, 2-24GB of main memory, and 12-24 SCSI (Small Computer System Interface) disk drives. The size and technology of the disk system is a strong determinant of overall system price. For example, in August 2006 the best price/performance reported on TPC-C uses a Dell server with one Intel dual-core processor, 24GB of RAM, and 90 SAS (serial-attached SCSI) disk drives [3]. The disk subsystem cost is approximately $45K out of a total system cost of $64.5K. If a RAID configuration is needed for high availability, then the costs rise even higher. Multiplying that by the number of servers required to run a large-scale web service, we see that the system can cost tens or hundreds of millions of dollars.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

In addition to hardware cost, there is the cost of a building to house the system, power to run it, and system managers to keep it running. Minimizing these costs is critical to a company's ability to offer the service at competitive fees-for-service or advertising rates and still earn a profit. In this paper, we explore the sources of these costs and optimizations to reduce them.

2 System Architecture

To enable scalability across a large number of machines, most back-end systems supporting a large-scale web service at Microsoft use a fairly classical multi-tier design. Users submit transactions, which can be queries or updates, usually via a web browser. Each transaction request is received by a web server, which routes the request to a mid-tier server, which in turn calls database servers. In some cases, such as Windows Live Messenger, a client process communicates using SOAP directly to a mid-tier server with no web server in between.

Usually, the web servers, mid-tier servers and database servers run on different machines. Machines running web servers and mid-tier servers have limited storage requirements and therefore can run on less expensive machines than database servers.

Since databases are quite large, they are partitioned across multiple servers. For good performance, most transactions need to execute entirely on one partition. Therefore, each transaction needs an application-supplied parameter that the mid-tier can map to the database partition that can execute the transaction. Windows Live services use three different partitioning schemes, based on hashing, value ranges, or table-look-up. Hash-based and range-based partitioning are commonly used in database systems, especially when the servers do not share disks. In a table-look-up-based scheme, each parameter value, such as a user-id, is explicitly mapped to a particular partition. The table-look-up scheme offers several advantages over the other two. First, it gives fine-grained control over load balancing. For example, when a new user-id is assigned, the mid-tier can assign it to a lightly-loaded server. Second, it can offer unique features for each user. For example, users who pay an annual fee receive special services not available to users who opt for free services. And third, it enables the system to minimize the time a user is affected by a schema upgrade by upgrading users one-by-one. By contrast, with hash-based and range-based partitioning, while a partition is being upgraded, the application would not know whether to access a user's data in the partition using the old or new format. Thus, all of the users in a partition would be inaccessible while the upgrade was in progress. Of course, these benefits come at a cost, namely, a database of <parameter-value, partition-id> pairs. For large services, the database may store hundreds of millions of parameter values. A case study of the table-look-up scheme appears in [5].

Each database partition is replicated for high availability. If a database server fails, its query load is absorbed by the replicas of the failed server's primary database partitions. If a database server needs to be taken off-line (e.g., because it is failing too frequently), its queue of committed updates must be drained to its replicas, so that its query and update load can be absorbed by its replicas.

3 Using Commodity Servers

The main sources of expense in running a large-scale service are hardware, data center infrastructure, power, networking, and system managers. Given the cost of premium servers as described in Section 1, one approach to reducing the expense is to move from such servers that cost (say) $40K to commodity servers that cost (say) $4K. This is surely beneficial, but it must be accompanied by attention to the other factors: the data center and people requirements. Some of these expenses are proportional to the number of machines and hence increase with a larger number of cheaper machines. The increase is usually sublinear and may be logarithmic for well-managed systems with excellent system management software. In this section, we discuss the factors that affect total cost and walk through a hypothetical calculation. Our goal is to give a feeling for the importance of certain factors and for the ratios involved. Our numbers and ratios depend on many factors, so they are merely rough estimates.

The data center is a major cost. Some of the cost factors are floor space (i.e., the building), communications, power, and cooling. Often these factors are in conflict. For example, a rural location might offer inexpensive real estate, but might have higher constructions costs and require new fiber for sufficient communications bandwidth. The server configuration can be designed to minimize its footprint and hence its real estate and construction cost. Power is an expense that can be controlled by moving the data center to a cheaper source of power, by using better cooling techniques, or by configuring servers that require less power. The latter usually lowers cooling requirements too.

Power is not only a cost but also a constraint. A large server installation can require tens of megawatts of power, which limits the locations where the installation can be built. It is therefore not surprising that Google, Microsoft, and Yahoo! have recently been building data centers near large hydroelectric facilities [4].

On the other hand, many installations have been set up in locations with excellent network connectivity. But these are often in locations where power is expensive. The move to rural locations with plentiful inexpensive power may be in part a response to the industry deciding to trade off power costs for networking costs.

System management is a major expense. System managers are responsible for monitoring load, diagnosing malfunctioning components, replacing failed components, and upgrading software to new releases. Job skills range from technicians who swap boxes to third-tier support engineers who diagnose subtle problems of performance, reliability, etc.

Given these considerations, let's do a hypothetical cost-benefit analysis of moving from premium to commodity machines. Consider a write-heavy application that is currently using premium four-way multiprocessor machines including RAID storage, and replicating each database partition for high availability. Using such premium machines, server costs dominate at about 50% of the total expense. About 25% of the cost is floor space, and less than 10% is people cost.

Getting a comparable level of reliability on commodity machines entails replicating each data partition on, let's say, four machines. Given that the commodity machines use slower disks, we might need as many as eight of them to replace each premium machine. This 8:1 ratio gives us more processor and memory capacity than we had before. But it might nevertheless be needed to get enough I/O capacity, especially if there's insufficient locality of reference to benefit from more caching.

Some networking costs are proportional to the number of ports required. Since we increase the number of machines, we potentially increase the number of network ports that are needed. Avoiding this increase requires some adjustment in the hardware configuration so that each rack has its own switch, which in turn connects to the main switch connected to the Internet.

The biggest effect of moving to commodity servers is people cost. Given the much larger number of servers and their lower reliability, there are many more system failures that require attention. Therefore, in our hypothetical cost analysis, system management explodes to 50% of total cost, which actually increases by 45% over the premium server configuration. The cost breakdown is 25% for servers, 15% for floor space, and 50% people.

Clearly, the solution is to automate many of the system management tasks, such as detecting faults, logging them, and analyzing logs to identify servers that should be reimaged or taken out of service and replaced. With the right tools, people costs can be brought down to the same level as with the premium server configuration. The tools have some cost to develop and maintain, but for large configurations this is small relative to the savings in systems management. As a result, total cost is now 25% lower than the premium configuration.

Another metric that addresses the same issue is the number of machines a system manager can handle. Suppose the fully loaded cost of a system manager is $200K/yr. To get the 25% lower total cost, we need system management cost to be about 25% of the server cost. If servers are depreciated over 3 years, that means a system manager must be able to handle $2.4M worth of machines, or about 600 machines. If a typical machine fails once a month, that means a system manager needs to handle 20 failures per day. Clearly, substantial automation is needed.

4 Performance Challenges and Solutions

Good performance is important both for customer satisfaction and to reduce hardware costs. In a large-scale application, the importance of many standard issues is magnified, such as the size of each attribute, the choice of indexes, disk layout, and the size of main memory. In this section we discuss some performance issues that are less widely known but also very important.

Replication: Each server holds primary copies of some partitions and replicas of other partitions. When a database server D fails, the database servers holding D's replicas need enough spare processor and I/O capacity to handle D's load in addition to their own. This extra capacity is part of the cost of high availability. To reduce this cost, each database partition that is being replicated is further partitioned into sub-partitions. Each sub-partition is replicated to a different machine. This reduces the headroom required on each machine to handle the additional load due to a failed primary partition. Thus, the more partitions the better. However, the sub-partitioning has a cost, namely, the fixed cost of I/O, memory and processing that the replication system consumes to support the additional replicas. For each replica, the primay has some partition-specific control information, an additional storage area to store commands to be forwarded to the replica, and an agent to do this work. These costs are independent of the number of updates. For an application with an update intensive workload, four sub-partitions has proven to be a good compromise between saving the cost of too much headroom and incurring the cost of sub-partitioning during normal operation. If the update workload is light, then less of the database server's spare capacity is needed for processing updates, so more of it can be used to handle a larger number of sessions. Hence, a larger number of partitions can be used in this case.

Replicas might also be useful to pick up some of the query load when the primary is overloaded and the machines that host its replicas are not. However, it is worthwhile to implement such a load-balancing mechanism only if such a skewed load is common. Since an important goal of partitioning tables is to spread the load evenly across partitions, most applications do not experience sustained skewed load. Hence, in most cases there is not much value in using replicas to handle queries except in failover situations.

Some applications have fan-out queries that access all partitions. As the workload increases, one can scale-out such an application by adding more servers and placing fewer partitions on each server. Although this increases the number of sub-queries required to execute a query, each sub-query has less work to do and there is more processing power to run them, so the overall system capacity is increased. If each server has just one partition, then one cannot scale out any further this way. So the next step is to split partitions into sub-partitions and place each sub-partition on a separate server, which enables more servers to be added. At some point, spreading the load in this way does more harm than good, since it may replace sequential I/O (to access a large partition) by random I/O (to access several sub-partitions albeit on different servers). In addition, there's a fixed overhead to run a query on each server and return its results, which increases with the number of servers. If the data is static or if reading data that is slightly out of date is acceptable, then other solutions may be preferable.

Session Management: In the multi-tier architecture described in Section 2, each mid-tier server needs a database session to each database partition that it accesses. These sessions consume resources. In our experience, a high-end commodity machine (as in Section 1) running Microsoft SQL Server^{TM} can handle up to several thousand database connections. As the number of connections increases much beyond that, the server can become unsteady or even unresponsive.

For example, suppose each database server has d databases and there are x mid-tier servers. If each mid-tier server has a session to every database, then each database server has $x \times d$ sessions. If a large scale web service has a thousand mid-tier servers with 4 databases per server, then each database server's session load is too high.

One way to avoid this problem is to partition the mid-tier servers so that each partition has sessions to only a small subset of the database servers. This greatly reduces the number of sessions to each database server. To do this, an application-supplied parameter is needed to direct each transaction request to a particular mid-tier partition, not just to a particular database server as described in Section 2. Front-ends should randomly select a particular mid-tier server in a partition, to spread the load evenly among them.

Suppose that a mid-tier server creates additional sessions to a database server that seems sluggish. This enables the mid-tier server to have more active database requests and hence continue to handle its load despite that database server's sluggishness. If the sluggishness is temporary and the extra sessions are dropped when they're no longer needed, then this might be a good strategy. However, if the sluggishness lasts long enough that all x mid-tier servers experience it, then they all create more sessions to that database server, let's say n sessions per database, which implies $x \times d \times n$ sessions per database server. Even after partitioning the mid-tier servers, the number of mid-tier servers per database server (i.e., x) may be in the hundreds. So again each database server may have more sessions than it can handle. The same problem can arise after a database server failure, where the load on that server is transferred to replicas. At this point we have exceeded the capacity for which the system was designed, so we must throttle back the incoming traffic.

One approach to avoid this is to have mid-tier servers control the database server load. Each has only a small fixed number of connections (e.g., 2) to the database servers it needs. The mid-tier server can then manage a queue of requests that pass through these connections. If the mid-tier server detects a major degradation of a database server's response time, it lets requests back up on its queue to prevent the database server from thrashing.

Materialized Views: In many Windows Live applications, data entry transactions and queries need different keys for indexing and clustering the data they access. For example, suppose the Users table is partitioned by user-id. Users have multiple credentials, such as email address and fingerprint encoding. They also have some other attributes, such as date of birth, country, gender, and preferences. If we often search for user by a credential, such as email address, then we need an index that maps email address to user-id. If we use the database system's indexing mechanism to do this, we would need to have it maintain an index with each partition. Since it regards each partition as a separate table, this doesn't help much; to access an email address, we would need to probe the index in every partition. Rather, we want the index to be partitioned by email address, so that given an email address we can access the unique index partition that stores it. We can do this by creating a table with two columns, email address and user-id, which is partitioned by email address. It still requires two database accesses to retrive user information, first to find the user-id for an email address and then to access the Users table. But this is the same disk access cost as would be incurred if the database system implemented this itself. The consistency of this table is maintained outside the database system. If the data isn't too critical, we can just choose a known order to write the information and live with the lack of transactional consistency guarantees. Otherwise, we can take the performance hit and wrap it in a distributed transaction. Most consumer services will opt for higher performance.

This works well when the index has low fanout (e.g., one record for each key value in the previous case). For larger fanouts, a materialized view may be needed. For example, an entertainment application typically captures movie reviews for each user. Since each user wants to be able to see all of his or her reviews, the table of reviews is partitioned by user-id. Users also want to read all the reviews of a given movie. Using the above approach, we can create a table that maps movie name to the user-id's of users who have written reviews. For each movie, there may be many such users, which are scattered across many partitions of the movie reviews table. Thus, many disk accesses will be required to retrieve them. To avoid this expense, we can have a second copy of the movie reviews table that is partitioned by movie. This is a materialized view where all of the reviews for a given movie are clustered together. Other examples are classified ads where ads are clustered by advertiser for data entry and by product category for querying, and shopping where products are clustered by vendor for data entry and by product category for querying.

Reducing checkpoint overhead: A database checkpoint flushes dirty buffers to disk, to reduce recovery time after a failure and to permit log truncation to avoid log overflow. Checkpoints can consume a lot of I/O bandwidth, so it is important to control their frequency and duration.

There are four times that Microsoft SQL Server initiates a checkpoint: (1) as a result of a DBA command such as backup and restore, removing a log file, taking the database online/offline, turning off database mirroring, etc.; (2) when the log becomes too long to allow recovery within the "recovery interval," which is the maximum

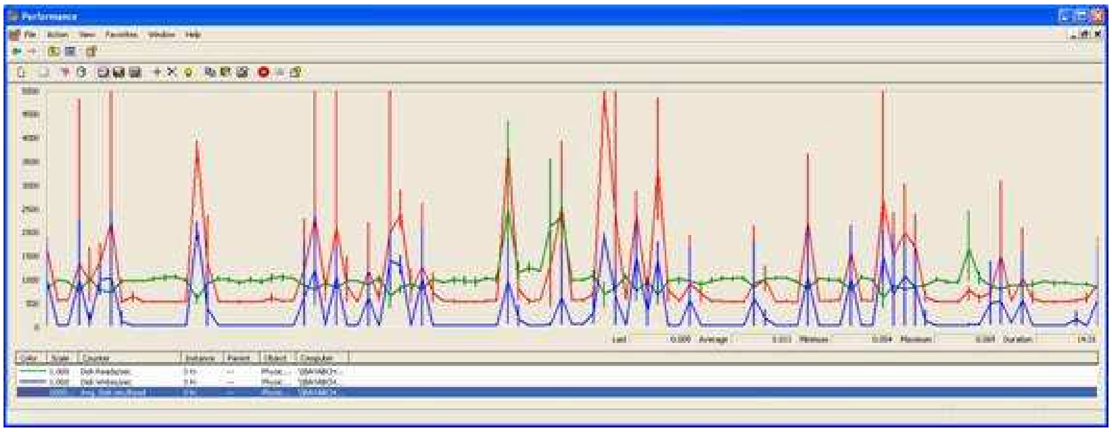

Figure 1: I/O operations on the database before reducing the number of checkpoints

recovery time specified by the database administrator (DBA); (3) when it is in “automatic log truncation mode” and the log is 70% full; and (4) when a DBA procedure calls the checkpoint command, which specifies a checkpoint duration, the amount of time for the checkpoint to complete. The latter enables SQL Server to calculate the number of writes per second required to flush all of the dirty pages in cache.

Many applications require frequent checkpoints, such as one per minute, to ensure fast recovery from failure and hence high availability. However, we found that in many applications taking checkpoints every half-hour is good enough, for three reasons. First, there is no risk of overflowing the log because log backups are taken every half-hour on the primary databases, after which the log is truncated. Second, as explained earlier, we ensure high availability by using replication, so fast log-based recovery is not crucial. Third, the log file generated in a half-hour interval is very small (approximately 20MB). If each log record is 200 bytes, then the log references 100K database pages. At 100 random I/Os per second per disk and, say, 20 disks, the recovery time is about a minute. So even if log-based recovery is needed, we expect it to be fast.

During a quick experiment on a production server, we set the recovery interval to a very large value. This eliminated automatic checkpoints on the primary databases except those triggered by log backup (which happens every half-hour). We still saw some automatic checkpoints on the replicas, which we believe are due to the small log files allocated to those databases and hence could be eliminated. In Figure 1 the red line (i.e., with highest peaks) and green line (i.e., with highest plateau and fewest peaks) show the writes/second and reads/second (respectively) without the checkpoint reduction. Figure 2 shows them with the checkpoint reduction. Notice the reduction in the number of spikes, which represents the I/O bottleneck as a result of checkpoints being generated.

Reducing checkpoints to once per half-hour risks having more dirty buffers to flush per checkpoint. However, in the above experiment, the number of buffers flushed per checkpoint did not increase when the inter-checkpoint interval increased from 1 to 30 minutes, for two reasons. First, in between checkpoints, a lazy writer is writing some of the dirty pages to make clean buffers available in response to memory pressure. An 8GB machine using 8KB/page has nearly 1M database pages in RAM. At 1000 writes/second, the lazy writer will write 1.8M pages in 30 minutes, which is almost twice the number of pages in RAM. So infrequently updated pages will certainly be flushed during that period to free up space for reads and will not require flushing at checkpoint time. The second reason is that during the half-hour, hot pages are written many times but only need to be flushed once at checkpoint time, so the large number of writes to these hot pages do not add to the checkpoint load.

A second checkpoint optimization is to spread each checkpoint’s writes over a longer time period, to avoid overwhelming the I/O subsystem. One approach is time-based. We assign a checkpoint duration that paces checkpoint I/Os to finish exactly within our half-hour time limit. For example, if a server has 8 databases, each database checkpoint should finish within 30/8 = 3.75 minutes. If each checkpoint writes about 15,000 buffers,

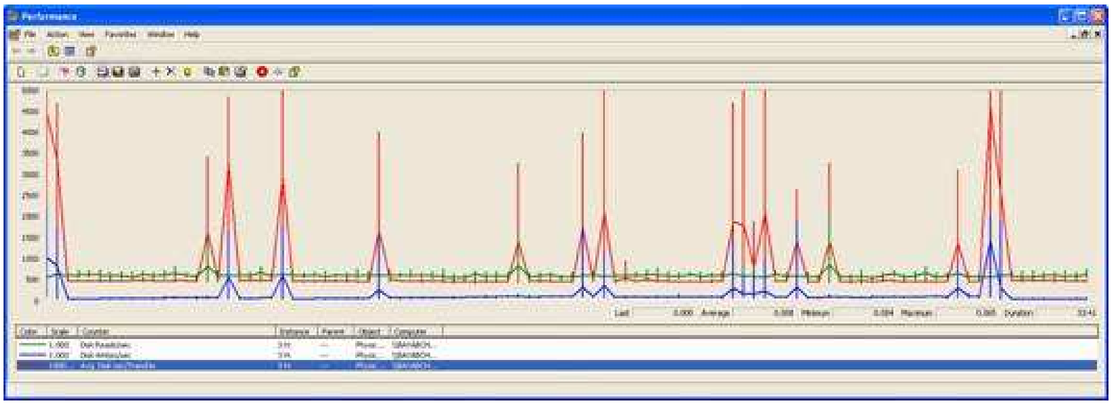

Figure 2: I/O operations on the database after reducing the number of checkpoints

then it will issue 67 writes/sec.

An alternative approach could be quota-based, where the user assigns a maximum I/O rate for each checkpoint and the database system ensures that this quota is not exceeded. When a checkpoint reaches the quota, it stops writing for a short time to avoid using too much I/O bandwidth. This guarantees that a checkpoint will not degrade the I/O rate available to applications below the given threshold, even when the checkpoint load is high. The cost is no upper limit on the checkpoint duration.

5 Research Approaches and Opportunities

The design of large-scale on-line systems has undergone continual change since their introduction in the late 1960's for airline reservation. The growing popularity of large-scale web services has changed things once again. These systems need to scale out to very large databases and user populations, often with hundreds of millions of users. They need to offer a mixture of transaction processing, document storage, and search functionality. They need to make the best use of the latest technology, inexpensive disk capacity, main memory, and flash memory. In particular, they need to cope with the changing hardware performance ratios, notably that disk capacity is increasing much faster than bandwidth and operations per second. And given the rapidly declining cost of hardware, they need to optimize for other major cost factors, notably system management. These factors present a challenge to designers of large-scale web services. They also present opportunities to researchers to discover ways to improve performance, availability, manageability and ease of programming, and to reduce cost.

References

[1] "A Conversation with Phil Smoot," *ACM Queue* 3, 10 (Dec 2005), pp. 16-24.

[2] Shahine, O., "Inside Hotmail," http://channel9.msdn.com/ShowPost.aspx?PostID=39016, August 21, 2005.

[3] Transaction Processing Council, www.tpc.org.

[4] www.datacenterknowledge.com

[5] Xiong, M. B. Goldstein, and C. Auger, "Scaling Out SQL Server," *Dell Power Solutions*, Aug. 2005, http://www.dell.com/downloads/global/power/ps3q05-20050100-Auger.pdf.