TDS Performance and Capacity Evaluation

Prepared for: SMARTER BALANCED

Fairway Technologies, Inc.
April 27, 2016

Table of Contents

Executive Summary

Fairway executed a series of load tests against an environment set up and configured following the recommendations specified in the Smarter Balanced Hosting Requirements Guide. Using database servers supported by top-tier storage services provided by Amazon Web Services (including provisioned input/output per second, which Amazon has improved since the deployment guide was released), Fairway could achieve 30,000 concurrent students and 1,500 test administrators ("proctors").

To support more than 30,000 concurrent students, the Smarter Balanced Hosting Requirements Guide recommends creating multiple TDS environments. One Test Delivery System (TDS) environment would be deployed per 30,000 concurrent students[1], with student data partitioned across several environments.

By means of these scalability test exercises, Fairway determined that reliance upon the TDS database server is the primary point of contention preventing the TDS from scaling beyond 30,000 concurrent students. We discovered many examples where the application did not conform to industry best practices. Specifically, issues related to unnecessary or overwhelmingly frequent database queries, minimal caching and temporary tables used as data structures, can all have a significant impact on performance between the database and application servers.

To alleviate the database contention issues, Fairway recommends moving to a MySQL database cluster, aggressively caching static/infrequently changing data, reducing the number of queries against the database server by consolidating queries to retrieve data and replacing temporary tables with data structures to manipulate data in memory on the application server.

NOTE: The TDS version used in the scalability environment was retrieved from the BitBucket source code repository and complied on December 14th, 2015.

[1] AIR, Smarter Balanced Hosting Requirements Guide, pg. 16

Scalability Testing

Scalability Requirements

The Smarter Balanced Hosting Requirements Guide indicates a single installation of the TDS will support 20,000 concurrent students. The guide goes on to state the AWS input/output characteristics (IOPS)[2] as a potential performance bottleneck, citing "one concurrent student can be supported per IOPS"[3]. Therefore, AIR recommends that one instance of the TDS be set up for every 20,000 students[4]. To support more than 20,000 concurrent students, multiple instances of the TDS need to be created. Multiple TDS environments will incur additional overhead; student registration must be partitioned across multiple TDS environments by some agreed upon criteria[5].

Since the release of the Smarter Balanced Hosting Requirements Guide, AWS has made some improvements to their service. Amazon RDS instances configured for Provisioned IOPS can now support a range of "1,000 - 30,000 IOPS"[6]. Previously, the maximum was 20,000 IOPS. To verify the one concurrent student to one IOPS unit recommendation, Fairway configured an RDS instance with 30,000 IOPS.

Considerations

Understanding the database server as a potential performance bottleneck, Fairway investigated possible performance tuning options for database servers hosted on AWS. Server CPU, RAM and network capacity configurations were considered. Additionally, Fairway investigated storage types for supporting input/output (I/O) operations. To measure the performance characteristics of two storage subsystems offered by AWS, Fairway captured two sets of results during the scalability testing effort.

AWS Storage Types

AWS offers several storage subsystems to support varying performance requirements. Described below are two popular storage subsystems used for AWS instances:

- **Provisioned IOPS:** "a storage type that delivers fast, predictable, and consistent throughput performance"[7]

[2] AWS Documentation – I/O Characteristics
[3] AIR, Smarter Balanced Hosting Requirements Guide, pg. 16
[4] AIR, Smarter Balanced Hosting Requirements Guide, pg. 16
[5] AIR, Smarter Balanced Hosting Requirements Guide, pg. 16
[6] Storage for Amazon RDS – Amazon Provisioned IOPS Storage to Improve Performance
[7] Storage for Amazon RDS – Amazon RDS Provisioned IOPS Storage to Improve Performance

- **General purpose SSDs:** "deliver single-digit millisecond latencies, with a base performance of 3 IOPS per Gigabyte (GB) and the ability to burst to 3,000 IOPS for extended periods of time up to a maximum of 10,000 [provisioned IOPS]"[8]

One RDS server was created using the provisioned IOPS storage subsystem. Another database server was created using general purpose solid state drives (SSD), a significantly cheaper storage mechanism. The details of the RDS server used for each test effort can be found in TDS Database Server Configuration Notes section (pg. 10).

Results

Below are descriptions of some major operations measured during the scalability test execution:

- **Proctor - Login:** A Test Administrator (TA) logs into the Proctor applications.
- **Proctor - Start session:** A TA begins a test session; many students connect to a session to take their assessments.
- **Proctor - Stop session:** A TA ends a test session.
- **Student - Log in and load assessment:** The student credentials are verified, all assessments available in the session are fetched and the student chooses the appropriate assessment
- **Student - Start assessment:** Student confirms they want to start the assessment, first set of questions are fetched and displayed.
- **Student - Get page content:** Fetch question assets (images, audio, video, etc.) for display.
- **Student - Get question group:** Get the next set of questions for the assessment.
- **Student - Review and submit assessment:** Student has finished assessment and can submit for scoring.

Database Server with Provisioned IOPS

Fairway conducted scalability tests using an RDS server with a three terabyte Provisioned IOPS storage system, allowing the server to be configured for 30,000 Provisioned IOPS. With the TDS database server configured in this manner, Fairway was able to achieve 30,000 concurrent students and 1,500 proctors. Performance results are shown in Table 1 (pg. 6). This performance is consistent with the one concurrent student to one IOPS assessment stated in the Smarter Balanced Hosting Requirements Guide.

[8] Storage for Amazon RDS – Types of Storage

Database Server with General Purpose SSD

Fairway also conducted scalability testing against a database server using general purpose SSD as its storage subsystem. After executing the scalability test multiple times against the same environment, Fairway was unable to achieve more than 20,000 concurrent students and 1,000 proctors. Any attempt to execute a scalability test with more than 20,000 concurrent students resulted in a large number of errors and very long response times.

Shown below in Table 1 are the results of some key operations that occur when students are taking proctored assessments. All times shown below are in milliseconds.

User Category	Operation	General Purpose SSD Average Time (in milliseconds)	Provisioned IOPS Average Time (in milliseconds)
Proctor	Login	1,490	798
Proctor	Start session	103	53
Proctor	Stop session	364	179
Student	Login and load assessment	27,476	4,948
Student	Start assessment	21,133	6,139
Student	Get page content	740	369
Student	Get question group	14,427	3,876
Student	Review and submit assessment	7,129	1,433

Table 1: Scalability test results

Environment

Load Testing Suite

Fairway developed a performance load test suite that simulates a large load of concurrent students and proctors. This performance load test suite can be updated/expanded to test well beyond the recommended specifications described in the Smarter Balanced Hosting Requirements Guide for a single environment.

Fairway conducted these tests using a 10-minute ramp-up time window for all proctors logging in and beginning test sessions, and a 20-minute ramp-up time for all students taking the test session. In this case the ramp-up time is the time window between the first and last student log in with the load distributed evenly throughout the window.

Fairway used the Apache JMeter open-source load testing framework for authoring and running the performance tests. Figure 1 shows the JMeter scalability testing environment.

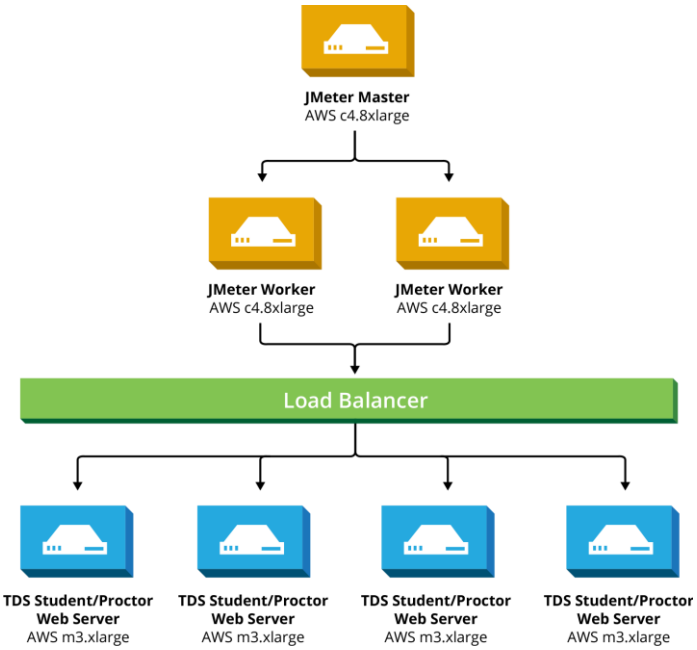

Figure 1: Apache JMeter distributed mode

Fairway used two JMeter server nodes (referred to as "workers") in a standard configuration to run tests simulating up to 60,000 concurrent students. Each worker node reports metrics back to the master so that a final comprehensive report can be created. The detailed report includes network latency, throughput, and errors for each action taken within the test.

TDS Servers

Fairway used AWS to host the servers and applications for the scalability testing environment. Table 2 (pg. 8) displays the specifications for the servers that comprise the scalability/load testing environment.

Application/Component	Quantity	AWS Instance	Notes
OpenDJ	1	c4.8xlarge vCPUs: 4 RAM: 60 GB	
OpenAM	1	m4.xlarge vCPUs: 4 RAM: 16 GB	
ProgMan Database	1	m3.medium vCPUs: 1 RAM: 3.75 GB	
ProgMan Application	1	m3.medium vCPUs: 1 RAM: 3.75 GB	-XX:+UseConcMarkSweepGC\ -Xms512m -Xmx2048m\ -XX:PermSize=512m\ -XX:MaxPermSize=1512m\
Permissions Database	1	m3.medium vCPUs: 1 RAM: 3.75 GB	
Permissions Application	1	m3.xlarge vCPUs: 4 RAM: 15 GB	-XX:+UseConcMarkSweepGC\ -Xms1024m -Xmx10240m\ -XX:PermSize=512m\ -XX:MaxPermSize=2048m\
ART Database	1	m3. xlarge vCPUs: 4 RAM: 15 GB	
ART Application	1	m3. xlarge vCPUs: 4 RAM: 15 GB	-XX:+UseConcMarkSweepGC\ -Xms2048m -Xmx10240m\ -XX:PermSize=1024m\ -XX:MaxPermSize=2048m\
TDS Database	1	db.m4.4xlarge vCPUs: 16 RAM: 64 GB	Amazon RDS instance with 200K tests taken; six types of tests
TDS Application	4	m3.xlarge vCPUs: 4 RAM: 15 GB	-XX:+UseConcMarkSweepGC\ -Xms5120m -Xmx28672m\ -XX:PermSize=512m\ -XX:MaxPermSize=2048m\ Hosts the Student, Proctor and Scoring applications; load balanced

Table 2: AWS instances for TDS scalability testing environment

Table 3 shows the AWS servers that host JMeter and the test suite created by Fairway. These servers are used to execute the load test and provide the detailed reports needed to determine the performance.

Application/Component	Quantity	AWS Instance Size	Notes
Load Test Master	1	c4.8xlarge vCPUs: 36 RAM: 60 GB	None
Load Test Worker	2	c4.8xlarge vCPUs: 36 RAM: 60 GB	None

Table 3: AWS Instances hosting Apache JMeter

Environment Configuration Notes

All AWS instances cited above:

- Use the Ubuntu 14.04 LTS 64-bit operating system
- Use General Purpose SSDs for storage
- Are shared tenancy instances
- Host TDS applications built based on code retrieved from the BitBucket code repositories current as of December 14, 2015

The TDS application instances cited above are placed behind a single load balancer. The load balancer is configured for sticky sessions using the `LBCookieStickinessPolicy` with an expiration period of 3600 seconds (1 hour).

TDS Database Server Configuration Notes

Fairway conducted the scalability testing against multiple configurations of the TDS database server, configured as follows:

Provisioned IOPS RDS Server

- db.m4.4xlarge instance type (16 vCPUs, 64 GB RAM)
- 3 TB provisioned IOPS storage
- 30,000 provisioned IOPS
- MySQL configuration settings based on configuration settings provided by AIR (Appendix A, pg. 23)

General Purpose SSD RDS Server

- db.m4.4xlarge instance type (16 vCPUs, 64 GB RAM)
- 100 GB general purpose SSD storage
- MySQL configuration settings based on configuration settings provided by AIR (Appendix A, pg. 23)

Environment Configuration Discrepancies

For conducting the scalability test described in this report, Fairway made every effort to follow the recommendations specified in the Smarter Balanced Hosting Requirements Guide. Even so, Fairway understands the AWS instances used in the scalability/load test environment may not necessarily conform precisely with the recommendations put forth by Smarter Balanced. When constructing the scalability/load test environment, Fairway consulted a variety of sources to determine appropriate AWS instance types:

- The Smarter Balanced Hosting Requirements document
- Files committed to the administrative_release source control repository:
    - `machine_configs.txt` (found here)
    - `machines_and_types.xlsx` (similar but not identical to `machine_configs.txt`, found here)
- Data evaluated by running the load test suite against the AWS instances in the scalability test environment. Specifically, reviewing the reports and error logs produced by JMeter and monitoring performance metrics provided by AWS.

The Smarter Balanced Hosting Requirements document recommends AWS instances of a particular size for baseline and on-demand web servers that host the TDS applications: “m1.xlarge (64-bit, 4 vCPUs,

15 GB RAM)"[9]. The machine specifications described in `machine_configs.txt` and `machines_and_types.xlsx` offer yet another perspective on AWS instance recommendations. Fairway opted to follow the specifications proposed by the Smarter Balanced Hosting Requirements Guide, on the assumption that this is the document most system administrators would follow when deploying a new TDS installation.

Scalability Test Execution

Fairway configured the scalability testing suite to simulate large number of concurrent students taking assessments in sessions created by 1,000 proctors. Fairway is aware that the requirements specify a 10:1 student/proctor ratio, but we were advised by Smarter Balanced that a 20:1 student to proctor ratio is a more viable real-world scenario.

The first step in the scalability test script is for proctors to log in and create test sessions. These operations occur over a 10-minute period, representing the scenario of assessment administrators logging in and starting sessions over a period of time rather than all at once.

After the test sessions are created, the scalability test script simulates students logging in to a test session and taking an assessment. The test script is set up in such a way that some students will be logging into the system while others are answering questions in their assessments.

The scalability test suite runs until all simulated students have completed and submitted the assessments. After the assessments are submitted and the test sessions have been closed, the JMeter log file is analyzed using the Apache JMeter user interface.

Test Conditions

Table 4 summarizes/re-states the input used to run the scalability tests described in this report:

Criteria	RDS Server with General Purpose SSD Storage Input	RDS Server with Provisioned IOPS Storage Input
Number of Students	20,000	30,000
Number of Proctors	1,000	1,500
Source code version	Code from sbacoss/release from December 14th, 2015	

Table 4: Scalability test input

[9] AIR, Smarter Balanced Hosting Requirements Guide, p. 17

Web Server Performance

Figure 2 shows the activity from the four web servers (each an m3.xlarge instance [4 vCPUs, 15 GB RAM]) as the scalability test ran for 30,000 students with an RDS server using a provisioned OPS storage system:

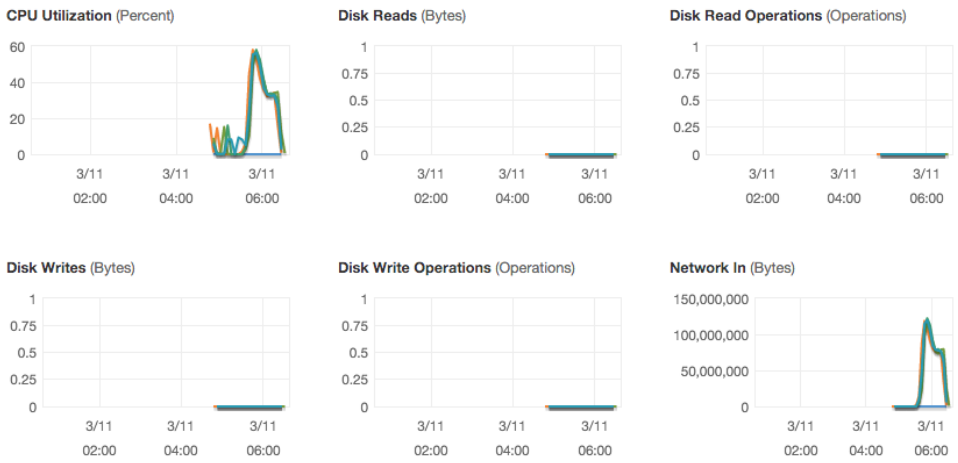

Figure 2: TDS web server utilization during scalability test run for 30,000 students against an RDS server using 3 TB provisioned IOPS storage

Figure 3 shows the activity on the four web servers (each an m3.xlarge instance [4 vCPUs, 15 GB RAM]) as the scalability test was running for 20,000 students with an RDS server using a general purpose SSD storage system:

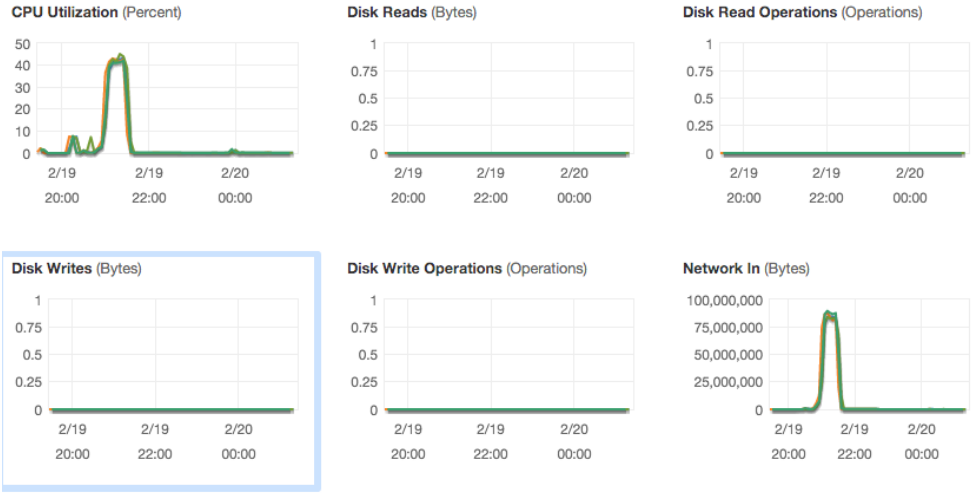

Figure 3: TDS web server utilization during scalability test run for 20,000 students against an RDS server using 100GB general purpose SSD storage

For Figures 2 and 3 (pg. 12), CPU utilization is nominal; none of the four web servers ever exceeded 60% utilization. The additional CPU utilization during the provisioned IOPS is due to the increased workload (20,000 students for Figure 3 vs. 30,000 students for Figure 2). Disk activity is non-existent. There is a fair amount of network activity, which is expected due to the following:

- Students taking assessments are actively and frequently communicating with the server (e.g. fetching assessment questions and posting answers)
- Proctors are periodically polling the database server to get the status of the students in their session

Like the CPU utilization variance, the variation in network utilization between Figures 2 and 3 can be explained by the additional workload handled by the web servers in Figure 2.

Database Server Performance

Provisioned IOPS

Figure 4 shows the activity on the database server (db.m4.4xlarge [16 vCPUs, 64 GB RAM]) using 30,000 provisioned IOPS as the scalability test was running:

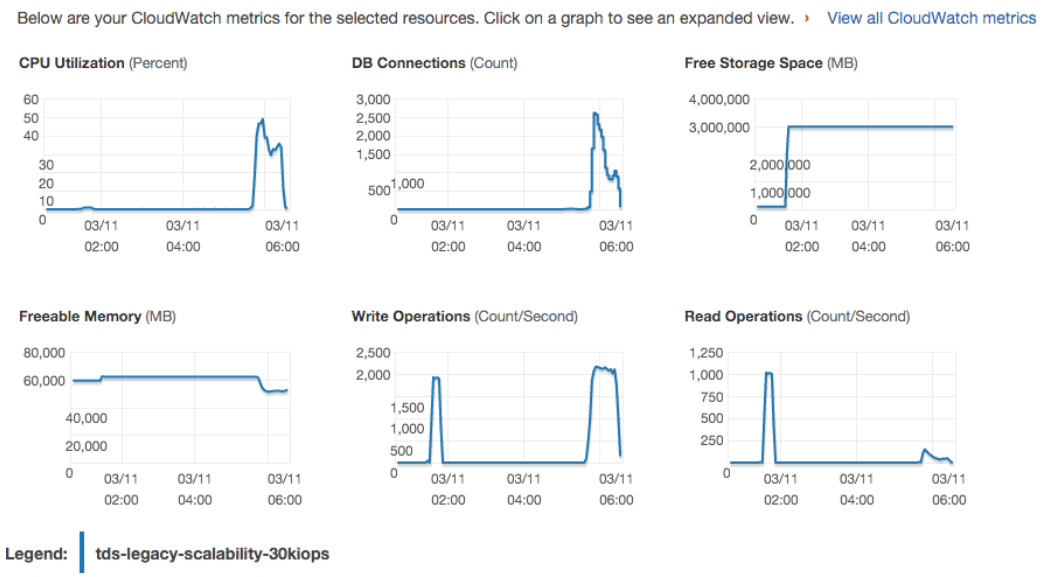

Figure 2: RDS server utilization with 3 TB provisioned IOPS storage during scalability test using 30,000 student workload

General Purpose SSDs

Figure 5 shows the activity on the database server (db.m4.4xlarge [16 vCPUs, 64 GB RAM]) using general purpose SSDs as the scalability test was running:

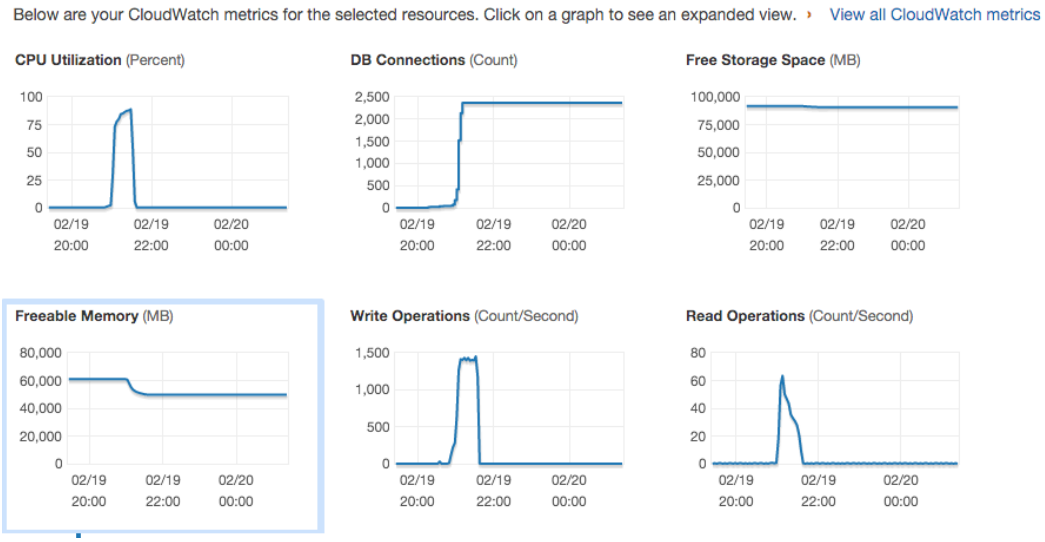

Figure 5: RDS server utilization with 100 GB general purpose SSD storage during a scalability test using a 20,000 student workload

There is a significant amount of CPU utilization on the RDS server using general purpose SSD storage: around 85% at its peak. The RDS server using provisioned IOPS showed less CPU utilization during the test: around 50% at its highest. Both severs maintained a large number of open database connections. Available memory and disk operations are not a concern; those subsystems were able to handle the workload without issue.

Scalability Issues

Database Reliance

The TDS application is heavily dependent on its database. Fairway has identified three main issues that cause the database server to be a performance bottleneck.

- **Queries:** Within a single method call, the application executes many queries against the database
- **Temporary Tables as Data Structures**: In some cases, data is stored in temporary database tables instead of a data structure on the application server
- **No Caching of Static Data:** During every method call, data that does not change frequently is retrieved from the database server instead of an in-memory cache

Queries

In many cases, the TDS application executes queries against the database server repeatedly for different pieces of data. In some of these cases, the same table is repeatedly queried, retrieving different columns to satisfy the request.

Shown below in Figure 6 is a sequence diagram describing the interactions when a student starts a test. Note that Figure 6 demonstrates a small sub-section of activity that occurs during this process and there are many other areas that repeatedly query the same database tables for additional data.

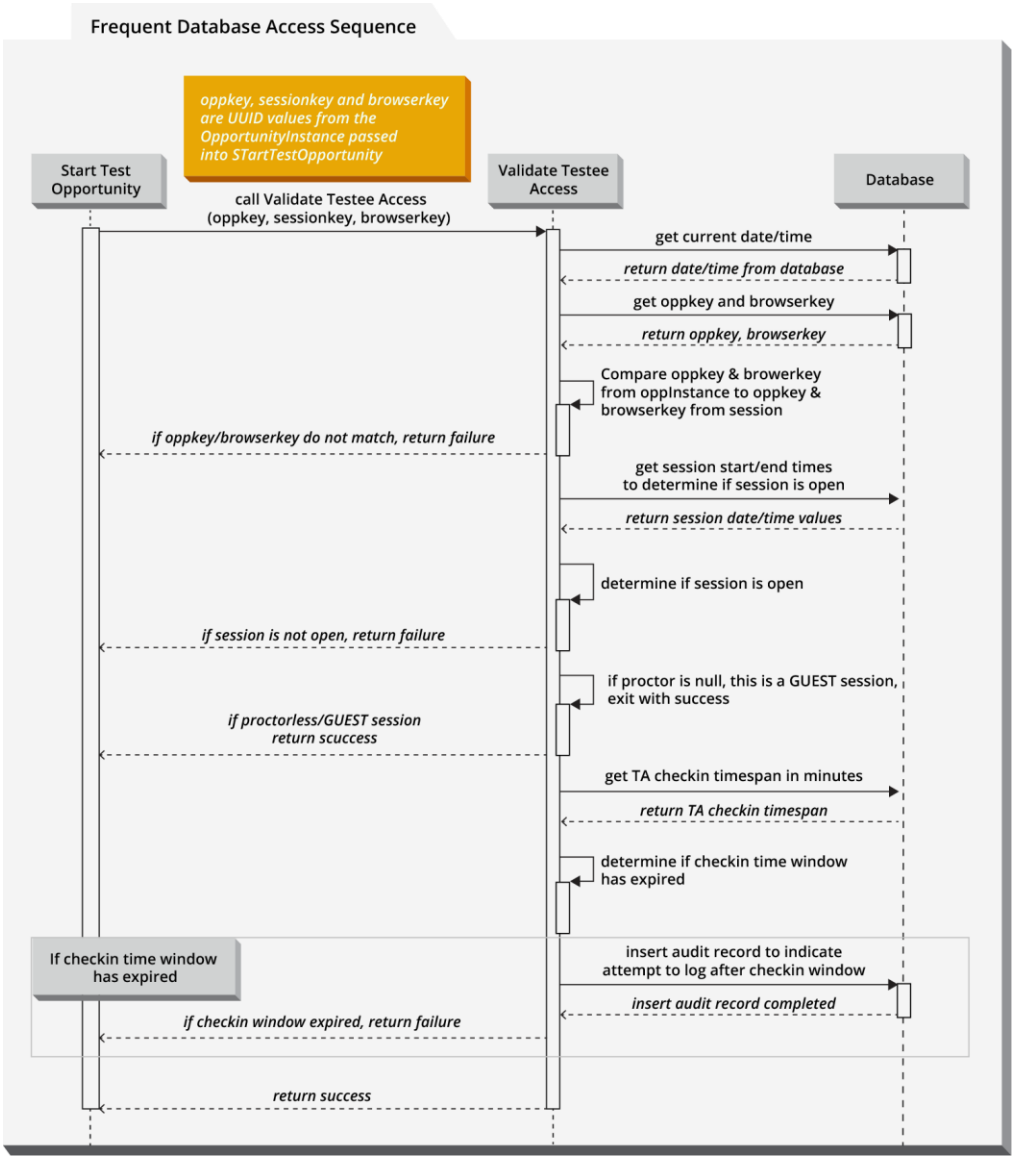

Figure 6: Frequent database access

In the case displayed in above, five distinct requests are made to the database. Consolidating queries will minimize the number of requests to the database, alleviating some of the database server's workload and allowing it to serve other requests.

Figure 7 shows an optimized version of the method shown above. A simple refactoring fetches all the assessment opportunity and assessment session data required throughout its execution and stores it in simple data structures. After refactoring, a single database call is conditionally made, based on whether or not the validation was successful.

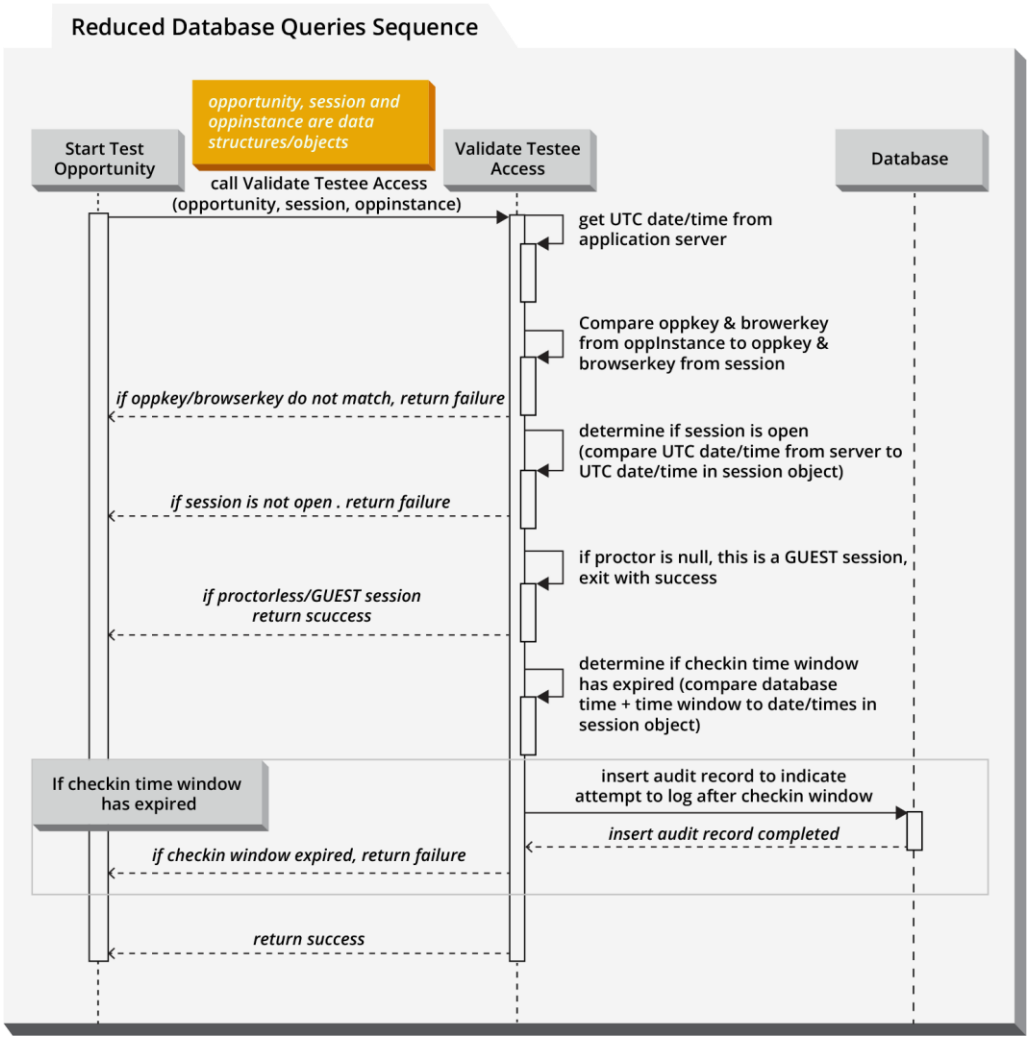

Figure 7: Reduced database queries

Temporary Tables as Data Structures

There are many cases where temporary tables are used in lieu of data structures. This design decision means any time the application needs the data stored in the temporary table, it must make a request to the database server. Interacting with a temporary table involves a network call from the application server to the database server. While MySQL temporary tables are typically stored in memory, if the table becomes too large it will be written to disk. Even if most of the temporary tables created in the application will be very small, thus never written to disk, a large number of temporary tables can still cause performance issues on the database server by:

- Creating a large memory footprint that consumes resources otherwise used for MySQL's buffer pool[10] and query cache[11]
- Increasing the number of requests that must be managed by the database

While the design decision to use temporary tables may be well-intentioned, the cost may outweigh the benefit. The idea of affecting many records that need to be changed/updated in a single set-based operation via SQL is attractive. As mentioned above, the temporary table is created on the database server, meaning any time the TDS application server wants to manipulate the records, the following must occur:

- A SQL query must be constructed
- The application must open a connection or request a connection from the pool of available connections
- Execute the query against the database server
- Release the connection to the database server or return it to the connection pool

The network overhead of all the operations cited above is more expensive than using a conventional `for` loop to iterate over a collection and make the necessary modifications. Use of a Java library like Guava[12] that provides a fluent interface for performing operations against a collection would also be suggested.

No Caching of Static Data

Like many other applications, TDS relies heavily on configuration values. The vast majority of TDS's configuration is stored in database tables (typically within a configuration database). These values rarely change after TDS has been deployed.

Even though the configuration values do not change frequently (if at all), the values are never cached. Instead, they are retrieved from the database with every method call. Referring to Figure 4 above, the TA check-in time value does not vary from student to student – every student receives the same value. Even though the value will always be the same, the application always fetches the value from the database. While it is very likely that the value is cached in the database server's buffer pool or query cache, fetching this single piece of data still requires a network roundtrip from the application to the database.

[10] MySQL Documentation – The InnoDB Buffer Pool
[11] MySQL Documentation – The MySQL Query Cache
[12] Google Guava documentation

Service Components

Service Communication/Network Usage

As mentioned above in the Database Reliance section (pg. 14), the service components frequently interact with the data layer. This "chatty" interface results in a large amount of network traffic between the service layer and data layer, requiring powerful servers to manage any significant load.

For example, assume the Student application server is hosted on an m3.xlarge AWS instance. If 100,000 students attempt to log in simultaneously, the auto-scaling feature must spin up enough servers to handle the work load. This could result in a large number of AWS instances that see minimal CPU and memory usage, but have been created to handle the network traffic.

During Fairway's scalability testing, the TDS application server performance was closely monitored. The web servers hosting the Student and Proctor applications never saw CPU utilization above 50%, disk and memory usage were trivial. There was a large amount of network traffic to and from the application servers; expected behavior given the design of the application.

Recommendations

This section describes Fairway's recommendations to alleviate/mitigate the items cited in the Scalability Issues section (pg. 14) of this document.

Relational Database Scalability

For TDS environments intending to support at most 20,000 concurrent students, a single MySQL database server configured for 20,000 IOPS should be sufficient[13]. If the server is of sufficient capacity (following the specifications made in the Smarter Balanced Hosting Requirements Guide[14]), a single database server should satisfy all the read and write requests sent to it from the TDS application servers.

To support more than 20,000 concurrent students while maintaining a relational database, a MySQL Cluster[15] configuration is more appropriate. MySQL Cluster gives MySQL the ability to scale out for supporting read and write operations. The MySQL Cluster software allows for automatic sharding of data across many data nodes. Unlike some other databases that offer sharding, MySQL's automatic sharding feature does not sacrifice conventional JOIN operations while retaining Atomicity, Consistency, Isolation and Durability (ACID)[16] guarantees.

[13] Using general purpose SSDs may be sufficient; load testing using the expected number of concurrent users should be conducted to determine the most appropriate storage type.
[14] AIR, Smarter Balanced Hosting Requirements Guide, pg. 15
[15] MySQL Documentation – MySQL Cluster
[16] ACID definition

Additional data nodes can easily be added to an existing cluster. These nodes can be added while the cluster is online, using a "rolling restart" methodology to ensure that parts of the cluster are always available to serve data while the new data nodes are registered within the cluster.

MySQL NoSQL Interface

For additional performance gains, both MySQL and MySQL Cluster offer an interface that is tightly integrated with memcached[17] (an open-source distributed caching system). Referred to as the NoSQL Interface, this allows access to the InnoDB storage engine without having to first transform the SQL. The memcached plugin is easy to install and register/configure within MySQL. Installation and configuration of the memcached MySQL plugin can be added to the existing deployment scripts. Alternately, instructions can be provided to install and configure the memcached plugin after the MySQL database server has been set up.

There are some considerations when determining whether the NoSQL interface is the proper solution:

- Data is only committed to the InnoDB engine after 32 operations have been performed against the NoSQL Interface (i.e. memcached) protocol. This means those operations are only visible via query after 32 operations.
- Tables that the NoSQL Interface interacts with must be registered in the `innodb_memcache.containers` table. This means there will be additional SQL scripts to execute during deployment.

MySQL Cluster Alternatives

An alternative to the MySQL Cluster installation/configuration is application-level sharding. This strategy is similar to what is suggested in the Smarter Balanced Hosting Requirements Guide for supporting more than 20,000 concurrent students. As stated in the Results section (pg. 5) of this document, the Hosting Requirements Guide effectively suggests sharding student data across multiple TDS environments[18].

The MySQL Router can simplify fetching and storing data from multiple standalone MySQL servers. A sharding strategy will be carefully considered and extensively tested to ensure even distribution across multiple database servers.

Application-Level Sharding Concerns

Application-level sharding can introduce an additional level of complexity. An effective sharding strategy (i.e. a strategy that evenly distributes data) can be difficult to identify and implement. If one of the standalone servers goes offline for any reason, some data might not be available. To handle this scenario, redundant servers must be put in place and synchronized often to minimize data loss. MySQL Cluster software handles sharding, data replication between nodes and implementing fault tolerance.

[17] memcached.org
[18] AIR, Smarter Balanced Hosting Requirements Guide, pg. 16

Due to the risks associated with application-level sharding, Fairway recommends avoiding this design unless all other alternatives prove untenable.

Caching

Data which changes infrequently or is complex to generate can be stored in a cache for quick retrieval. Aggressively caching static data has several benefits:

- **Alleviate the workload on the database server.** With the application server providing static data from cache, the database server will be freed up to respond to other requests.
- **Reduce overall network traffic.** The application can now receive data from cache, eliminating the need to make a network call to the database server for data.

Client-Side Caching

In addition to caching data on the server side, TDS may benefit from storing some data elements on the client side. For example, AngularJS[19] (a client-side Javascript framework used for extending/enhancing browser-based user interfaces) provides a mechanism[20] for caching requests. Additionally, most modern browsers support a local storage cache, which can be used to store data that is frequently used but does not often change. Leveraging client-side caching can eliminate network calls to the server, thus alleviating the application server's workload.

Consolidate Database Queries

Where possible, methods should fetch all the data they need to complete their execution in a single query. Frequent queries to the database for additional data can cause performance issues (additional connections from the connection pool, additional I/O operations and query processing).

Replace Temporary Tables with Data Structures

Using proper data structures instead of temporary tables will reduce the amount of communication between the application and database servers. Operations against data structures will occur in memory, eliminating the need to make a call to the database server in order to interact with the temporary table. Eliminating the use of temporary tables will free up resources on the database server to conduct other operations.

Synchronize All Servers to Same Time Zone

To eliminate date/time discrepancies, all servers in the environment should be set to use the same time zone. Ideally, the time zone should be set to UTC. Server clocks are already synchronized using

[19] AngularJS documentation
[20] AngularJS Documentation – cacheFactory.Cache

`ntp` (Network Time Protocol, a protocol for clock synchronization between computers)[21]. In the database, date and time values should be stored using the same time zone (ideally UTC). Synchronizing all servers and data to use the same time zone offers a few benefits:

- No need to query the database for the current date/time
- Server date/time is consistent within the environment

Component Communication

Communication Between Services

Where possible, reduce the amount of inter-service calls between components (i.e. reduce "chatty" interactions). This recommendation also applies to communication between services and the data layer. Rather than making many small requests, consider making fewer requests that return all required data in a single response.

Communication with Data Layer

The Communication Between Services recommendation extends to interaction between application components and the data layer. Instead of issuing many small queries (especially against the same table many times) for different data elements, execute fewer larger queries to fetch all the required data from the specified table(s) at once.

Eventual Consistency

For applications that have high throughput, leveraging eventual consistency[22] can improve application scalability. Rather than having a method call wait on synchronizing the data store (i.e. wait on the response of an insert/update/delete against the database), a technique like event sourcing[23] can be used to record the data modification as an event that is written to a log. When data/application state needs to be read, the log is replayed, returning the current state of the data.

Content Delivery Network

For delivering static files (e.g. CSS, javascript), a content delivery network (CDN) can reduce load on the application server. Storing some content on a CDN can reduce the overall workload on an application server that is serving up client-side content. Setting up a CDN in AWS or Microsoft Azure is a simple process; both service providers allow the website to be the origin server, meaning the TDS deployment process will be the same.

[21] ntp Documentation
[22] Eventual consistency definition
[23] Martin Fowler, Event Sourcing

NOTE: Care should be taken when considering files that are candidates for delivery by CDN. If the file contains proprietary algorithms or other sensitive data, it might not be a good candidate for serving from the CDN.

Application State

If server-side session state is required, the application must continually have its requests routed to a particular server (referred to as "client affinity"). In a load-balanced environment, client affinity behavior is typically supported by a "sticky session" feature. Enabling sticky sessions can result in uneven load on some application servers. In addition, if the application server is taken out of the load balancer for any reason (e.g. an unexpected outage), the session state for the clients it was supporting will no longer be available. Loss of server-side application state could cause unexpected behavior in the clients that had affinity for the affected server.

Ideally the TDS application will be stateless, meaning the application's request can be routed to and handled by any server behind the load balancer, therefore not requiring any session state information to be stored.

Client Side Assets

In addition to responding to requests for data, the application servers must also respond to the browser's request for assets (e.g. CSS files, JavaScript files, images, etc.). As mentioned above, moving some or all of these assets to a CDN can be beneficial in alleviating the application server's workload. Additionally, we recommend moving away from the YUI JavaScript framework, bundling and minifying JavaScript and CSS files, and implementing server-side compression.

Smaller JavaScript Footprint

The TDS applications rely on the YUI JavaScript framework. In addition to no longer being supported by Yahoo!, the YUI framework is a large client-side library. We recommend transitioning to a modern JavaScript framework that has a much smaller footprint than YUI.

Server-Side Compression

Enabling server-side compression can reduce overall bandwidth usage. Many popular servers support compression (Apache HTTP Server, nginx, Microsoft IIS), as do all browsers. Gzip is a common, well-known compression library that is readily supported by the servers mentioned previously. XML, due to its repetition (e.g. begin and end tags have largely the same text) compress very well. JSON can also be compressed, but the benefit will be slightly less noticeable because the JSON equivalent of an XML message is typically fewer bytes.

The trade-off for enabling server-side compression is a slight increase in CPU usage on the server conducting the compression. The application server will have to perform additional computation to compress the response before it can be delivered. Typically, the CPU overhead is minimal.

Appendix A: Configuration Settings Provided by AIR:

Tomcat Configuration Options

```
<Connector
        protocol="org.apache.coyote.http11.Http11NioProtocol"
        port="18443" maxThreads="4000"
        acceptcount="0"
        acceptorThreadCount="2"
        connectionTimeout="60000"
        maxHttpHeaderSize="18192"
        minSpareThreads="75"
        maxSpareThreads="200"
        maxConnections="30000"
        asyncTimeout="60000"
        maxKeepAliveRequests="-1"
        tcpNoDelay="true"
        compression="on"
        compressionMinSize="2048"
        noCompressionUserAgents="gozilla, traviata"
        compressableMimeType="text/html,text/xml,text/json,application/json,t
ext/javascript,text/css,text/plain,application/x-
javascript,application/javascript,application/octet-stream"
        scheme="https" secure="true" SSLEnabled="true"
        keystoreFile="/path/to/your/tomcat.keystore" keystorePass="your-
keystore-password"
        keyAlias="*your_key_alias" clientAuth="false" sslProtocol="TLS"
/>
```

MySQL Database Configuration Options

```
[client]
port = 3306
socket = /*/*/mysqld/mysqld.sock
[mysqld_safe]
socket = /*/*/mysqld/mysqld.sock
nice = 0
log_error=/*/*/log/mysql/mysql_error.log
malloc-lib = /*/*/lib/libjemalloc.so.1
[mysqld]
skip-external-locking
explicit_defaults_for_timestamp = 1
user = mysql
pid-file = /*/*/mysqld/mysqld.pid
socket = /*/*/mysqld/mysqld.sock
port = 3306
basedir = /usr
datadir = /SDB1/mysql
tmpdir = /SDK1:/SDL1
transaction-isolation = READ-COMMITTED
myisam-recover = BACKUP
max_connections = 4500
```

```
lc-messages-dir = /*/*/mysql
skip-external-locking
slow_query_log=1
slow_query_log_file=/*/*/log/mysql/log-slow-queries.log
log_output = 'TABLE,FILE'
log_error=/**/log/mysql/mysql_error.log
query_cache_type = 1
query_cache_limit = 1M
query_cache_size = 20M
innodb_buffer_pool_size=5G
tmp_table_size=32M
max_heap_table_size=32M
open_files_limit = 65535
key_buffer_size = 16M
max_allowed_packet = 514M
back_log = 2000
connect-timeout=60
join_buffer_size = 1M
read_buffer_size = 1M
sort_buffer_size = 256K
myisam_sort_buffer_size = 8M
read_rnd_buffer_size = 524288
bulk_insert_buffer_size = 8M
query_prealloc_size = 65536
query_alloc_block_size = 131072
wait_timeout = 300
interactive_timeout = 300
innodb-log-file-size=512M
thread_stack = 192K
thread_cache_size = 10000
performance_schema_max_cond_instances=45516
performance_schema_max_file_instances=80993
performance_schema_max_mutex_instances=80674
performance_schema_max_rwlock_instances=31358
performance_schema_max_socket_instances=21078
performance_schema_max_table_handles=5000
performance_schema_max_thread_instances=21158
bind-address = 0.0.0.0
myisam-recover = BACKUP
expire_logs_days = 10
max_binlog_size = 100M
[mysqldump]
quick
quote-names
max_allowed_packet = 16M
[isamchk]
key_buffer_size = 16M
```

Appendix B: References

Document	Description
Smarter Balanced Hosting Requirements Guide, V2	Describes hosting requirements/recommendations for setting up and deploying a TDS environment
Storage for Amazon RDS	Documentation regarding various storage subsystem options for an Amazon RDS instance
AWS I/O Characteristics	Describes volume configurations available for AWS instances
AWS - Configure Sticky Sessions for Your Load Balancer	Provides guidelines and details on how to configure an HTTP/HTTPS load balancer in an AWS environment
machine_configs.txt	Output from an Excel spreadsheet used to provision a TDS environment
machines_and_types.xlsx	A spreadsheet used to dictate the AWS server instances used to provision a new TDS instance
MySQL Documentation - InnoDB Buffer Pool	Provides guidelines and details for working with the MySQL InnoDB buffer pool
MySQL Documentation - Query Cache	Provides guidelines and details for working with the MySQL query cache
MySQL Documentation - Cluster	An overview of MySQL Cluster server set up and configuration
ACID (Atomicity, Consistency, Isolation, Durability)	A definition of properties that make database transactions reliable
Google Guava	Java core libraries used by Google for interacting with collections, caching, primitives, concurrency, I/O, etc.
Memcached Documentation	Documentation regarding the `memcached` library
Eventual Consistency	Documentation describing the “eventual consistency” consistency model
Event Sourcing	An article defining the event sourcing model
AngularJS Documentation	Documentation for the AngularJS Javascript framework
AngularJS Documentation - Cache	AngularJS documentation describing the `cacheFactory.cache` implementation details
ntp Documentation	Describes the `ntp` (Network Time Protocol) protocol