APPDYNAMICS

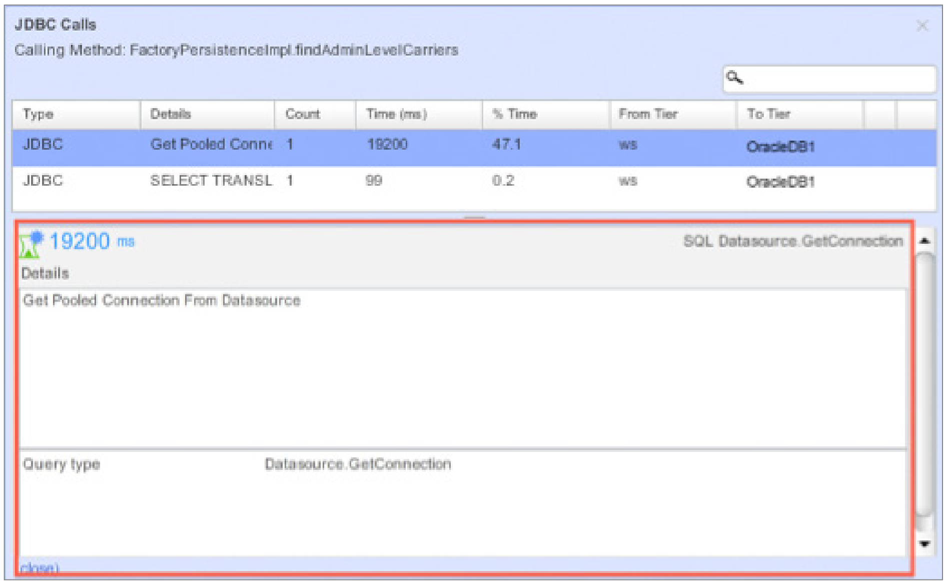
An AppDynamics Business White Paper

Top 10 reasons your eCommerce site will fail during peak periods

For eCommerce organizations, peak periods are crucial for the bottom line. In the US, there's Black Friday and Cyber Monday, however eCommerce sites globally need to plan for the extra traffic and load that happens during holidays and throughout sales. As more and more shopping is moving online, however, the success of the shopping season for the store depends more and more on the ability of its eCommerce applications to serve the high volume of demand. While many organizations prepare intensively for the season with load and performance testing, it's impossible for them to foresee every scenario that could affect application performance at a critical time. Here are a few of the most common causes of performance bottlenecks and outages to prepare for this holiday season.

1. Exhausted database connection pools

Nearly every Checkout transaction will interact with one or more databases. Database connections are therefore a precious resource, and a database connection pool that's too small can be a bottleneck when concurrency is high. Most application servers have a default connection pool size of 10-20 connections. For eCommerce sites that process 100,000 transactions per minute during peak load, this will be woefully inadequate. Many organizations will never realize this, however, because they focus their load testing efforts on the web servers and neglect to test backend services for high concurrency. The below screenshot illustrates what can happen when your database connection pool is sized too small for your application:



JDBC Calls

Calling Method: FactoryPersistenceImpl.findAdminLevelCarriers

Type	Details	Count	Time (ms)	% Time	From Tier	To Tier
JDBC	Get Pooled Conne	1	19200	47.1	ws	OracleDB1
JDBC	SELECT TRANSL	1	99	0.2	ws	OracleDB1

19200 ms SQL Datasource.GetConnection

Details

Get Pooled Connection From Datasource

Query type Datasource.GetConnection

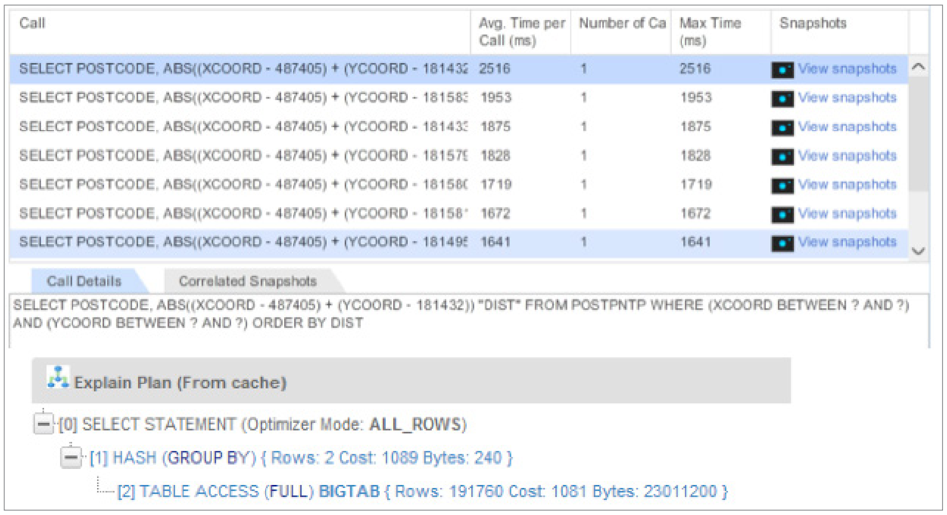
close)

In this screenshot you see that even though the database call itself only took 99 milliseconds to execute, the end user was waiting for more than 20 seconds because it took 19 seconds for the thread to get a database connection from the pool.

Even if you no longer use the default settings for your database connection pool, you should be sure to look at your configurations this November to make sure your connection pool configuration won't be a bottleneck that will affect your end users.

2. Missing database indexes

Slow-running SQL statements hold on to a database connection for longer than they should, which means the connection pool isn't recycled often enough and new threads are forced to wait for connections. The most common root cause for slow SQL statements is missing indexes on the database tables, which is often caused by miscommunication between the database administrators (who are responsible for setting up and maintaining database schemas) and the developers that write the SQL. The "full table scan" query execution, in which the database operation must scan through all the data in the table before data is returned, can be very time-consuming when the table contains millions of rows. By adding an index – a copy of one or more columns that allows the database operation to quickly find the rows it needs – you can make common database operations much faster, freeing up database connections to serve more users.



Call	Avg. Time per Call (ms)	Number of Ca	Max Time (ms)	Snapshots
SELECT POSTCODE, ABS((XCOORD - 487405) + (YCOORD - 18143	2516	1	2516	View snapshots
SELECT POSTCODE, ABS((XCOORD - 487405) + (YCOORD - 18158	1953	1	1953	View snapshots
SELECT POSTCODE, ABS((XCOORD - 487405) + (YCOORD - 18143	1875	1	1875	View snapshots
SELECT POSTCODE, ABS((XCOORD - 487405) + (YCOORD - 18157	1828	1	1828	View snapshots
SELECT POSTCODE, ABS((XCOORD - 487405) + (YCOORD - 18158	1719	1	1719	View snapshots
SELECT POSTCODE, ABS((XCOORD - 487405) + (YCOORD - 18158	1672	1	1672	View snapshots
SELECT POSTCODE, ABS((XCOORD - 487405) + (YCOORD - 18149	1641	1	1641	View snapshots

```
SELECT POSTCODE, ABS((XCOORD - 487405) + (YCOORD - 181432)) "DIST" FROM POSTPNTP WHERE (XCOORD BETWEEN ? AND ?)
AND (YCOORD BETWEEN ? AND ?) ORDER BY DIST
```

Explain Plan (From cache)

```
[0] SELECT STATEMENT (Optimizer Mode: ALL_ROWS)
  [1] HASH (GROUP BY) { Rows: 2 Cost: 1089 Bytes: 240 }
    [2] TABLE ACCESS (FULL) BIGTAB { Rows: 191760 Cost: 1081 Bytes: 23011200 }
```

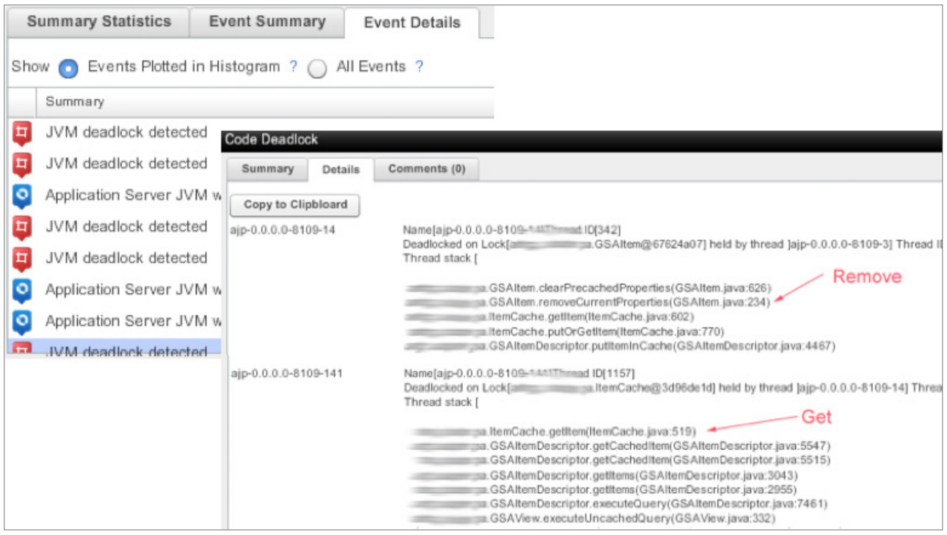
Here is an example of a transaction with several similar database calls, each of which takes 1-2 seconds to complete. If you look at the explain plan, you can see that the database operation accessed the entire table, scanning over 191,000 rows at a cost of 1 second.

While this isn't very slow by itself, because the transaction called the database a dozen times the cumulative effect on response time was quite large.

Before the holidays, do an audit of your most common database operations to ensure they have indexes, if applicable. You don't want to discover that your database is missing a crucial index after your end users are already being affected.

3. Code deadlock

High concurrency often means that application server threads contend for resources and objects more than usual. Most eCommerce applications have some form of atomicity built into their transactions to preserve data integrity for order and stock volumes. This is especially important for eCommerce applications because they're dealing with physical merchandise – if your database says there is only one item left, you must make sure only one person is able to purchase that item before the database is updated and the website reflects that the item is out of stock.



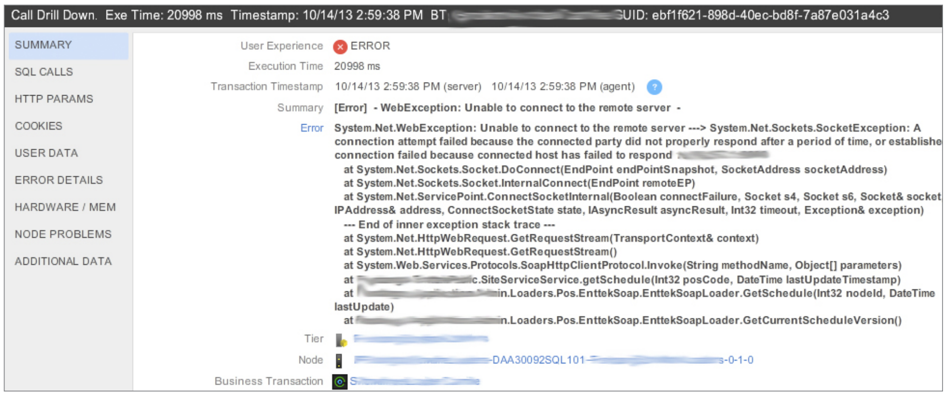
```
ajp-0.0.0.0-8109-14     Name[ajp-0.0.0.0-8109-[redacted] ID[342]
                        Deadlocked on Lock[[redacted].GSAItem@67624a07] held by thread ]ajp-0.0.0.0-8109-3] Thread I
                        Thread stack [

                        [redacted].GSAItem.clearPrecachedProperties(GSAItem.java:626)
                        [redacted].GSAItem.removeCurrentProperties(GSAItem.java:234)
                        [redacted].ItemCache.getItem(ItemCache.java:602)
                        [redacted].ItemCache.putOrGetItem(ItemCache.java:770)
                        [redacted].GSAItemDescriptor.putItemInCache(GSAItemDescriptor.java:4467)

ajp-0.0.0.0-8109-141    Name[ajp-0.0.0.0-8109-[redacted] ID[1157]
                        Deadlocked on Lock[[redacted].ItemCache@3d96de1d] held by thread ]ajp-0.0.0.0-8109-14] Threa
                        Thread stack [

                        [redacted].ItemCache.getItem(ItemCache.java:519)
                        [redacted].GSAItemDescriptor.getCachedItem(GSAItemDescriptor.java:5547)
                        [redacted].GSAItemDescriptor.getCachedItem(GSAItemDescriptor.java:5515)
                        [redacted].GSAItemDescriptor.getItems(GSAItemDescriptor.java:3043)
                        [redacted].GSAItemDescriptor.getItems(GSAItemDescriptor.java:2955)
                        [redacted].GSAItemDescriptor.executeQuery(GSAItemDescriptor.java:7461)
                        [redacted].GSAView.executeUncachedQuery(GSAView.java:332)
```

One way that eCommerce applications achieve this is by putting locks on certain resources, in this case the product. This is effective at preserving data integrity, but it can be bad for performance, especially if it causes a code deadlock. Code deadlock happens when two or more threads are both contending for the same resource, and often can be disastrous for the application server affected. In the screenshot below, you see an example of code deadlock that affected an eCommerce application. Three threads tried to perform a get, set and remove on the same cache at the same time, causing code deadlock to occur. The resulting deadlock caused over 2,500 checkout transactions to hang.

Code deadlock is usually the result of an application design that doesn't account for concurrency. Because these problems only appear during high concurrency, it can be difficult to catch them in development or test. Be sure to do load and performance tests before peak periods to surface issues like these. In addition, be careful where and how you use locks or synchronous code.

4. CPU-Intensive transaction

Server connectivity is an obvious root cause for performance issues. If you check your logs using a product like Sumo Logic or Splunk then you'll probably see hundreds of errors indicating that a transaction could not connect to a remote server. Some of these will be the result of network problems; some won't even be your services, but remote HTTP calls to third party services like shipping, billing or fraud detection. During peak periods you can expect to see many more of these, not just because your site is experiencing high demand, but because these other remote services and even the entire network are saturated. The problem for your business is that if server connectivity issues take too long (typically 30-45 seconds) they can cause important transactions to time out. Here's an example of a transaction that timed out after it was unable to connect to a server:

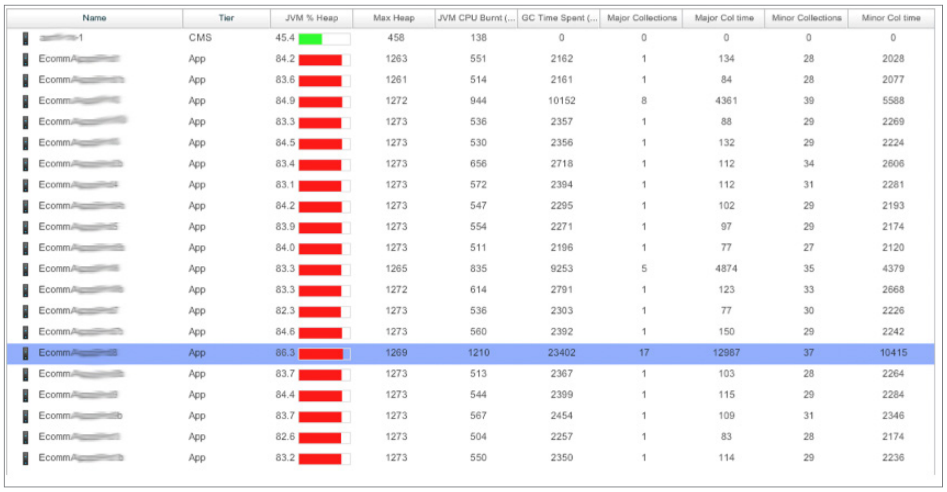


```
Call Drill Down.  Exe Time: 20998 ms  Timestamp: 10/14/13 2:59:38 PM  BT [redacted]GUID: ebf1f621-898d-40ec-bd8f-7a87e031a4c3

        User Experience  ERROR
         Execution Time  20998 ms
  Transaction Timestamp  10/14/13 2:59:38 PM (server)   10/14/13 2:59:38 PM (agent)
                Summary  [Error]  - WebException: Unable to connect to the remote server  -
                  Error  System.Net.WebException: Unable to connect to the remote server ---> System.Net.Sockets.SocketException: A
connection attempt failed because the connected party did not properly respond after a period of time, or establishe
connection failed because connected host has failed to respond [redacted]
   at System.Net.Sockets.Socket.DoConnect(EndPoint endPointSnapshot, SocketAddress socketAddress)
   at System.Net.Sockets.Socket.InternalConnect(EndPoint remoteEP)
   at System.Net.ServicePoint.ConnectSocketInternal(Boolean connectFailure, Socket s4, Socket s6, Socket& socket
IPAddress& address, ConnectSocketState state, IAsyncResult asyncResult, Int32 timeout, Exception& exception)
   --- End of inner exception stack trace ---
   at System.Net.HttpWebRequest.GetRequestStream(TransportContext& context)
   at System.Net.HttpWebRequest.GetRequestStream()
   at System.Web.Services.Protocols.SoapHttpClientProtocol.Invoke(String methodName, Object[] parameters)
   at [redacted]c.SiteServiceService.getSchedule(Int32 posCode, DateTime lastUpdateTimestamp)
   at [redacted]n.Loaders.Pos.EnttekSoap.EnttekSoapLoader.GetSchedule(Int32 nodeId, DateTime
lastUpdate)
   at [redacted]n.Loaders.Pos.EnttekSoap.EnttekSoapLoader.GetCurrentScheduleVersion()
                   Tier  [redacted]
                   Node  [redacted]-DAA30092SQL101-[redacted]s-0-1-0
   Business Transaction  S[redacted]
```

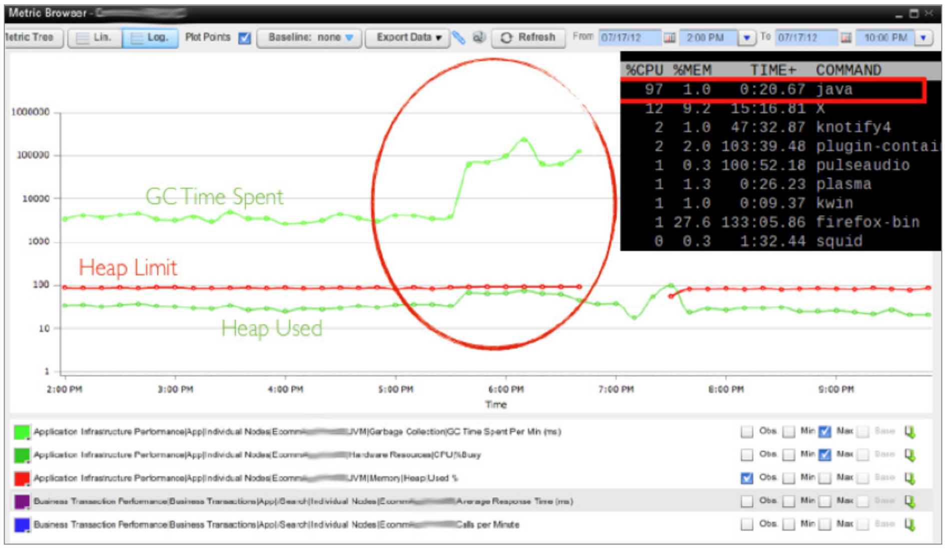
If you see a lot of these errors occurring in your system, you should investigate the issue to determine where the problem is occurring and troubleshoot the problem. In addition, you should ensure that your application uses short timeouts with retry logic to make your app more resilient to network issues.

5. Garbage collection

Caches are an easy way to speed up application performance. The closer the data is to the application logic (in memory) the faster it will execute. It is therefore no surprise that as memory has gotten bigger and cheaper, most companies have adopted some form of in-memory caching to cut down on database access for frequently used results. This means that average heap size is much larger than before; 64GB and 128GB heaps are not uncommon. As a result, garbage collection affects end users more than before. In order to reduce the impact and frequency of garbage collection cycles, you must be efficient and careful in maintaining cache data and in creating or persisting user objects. Just because you have GBs of memory to play with doesn't mean you can be lazy in how you create, maintain and destroy objects. Here are a few screenshots demonstrating how garbage collection can kill your eCommerce application:



Name	Tier	JVM % Heap	Max Heap	JVM CPU Burnt (...	GC Time Spent (...	Major Collections	Major Col time	Minor Collections	Minor Col time
[redacted]-1	CMS	45.4	458	138	0	0	0	0	0
Ecomm[redacted]	App	84.2	1263	551	2162	1	134	28	2028
Ecomm[redacted]	App	83.6	1261	514	2161	1	84	28	2077
Ecomm[redacted]	App	84.9	1272	944	10152	8	4361	39	5588
Ecomm[redacted]	App	83.3	1273	536	2357	1	88	29	2269
Ecomm[redacted]	App	84.5	1273	530	2356	1	132	29	2224
Ecomm[redacted]	App	83.4	1273	656	2718	1	112	34	2606
Ecomm[redacted]	App	83.1	1273	572	2394	1	112	31	2281
Ecomm[redacted]	App	84.2	1273	547	2295	1	102	29	2193
Ecomm[redacted]	App	83.9	1273	554	2271	1	97	29	2174
Ecomm[redacted]	App	84.0	1273	511	2196	1	77	27	2120
Ecomm[redacted]	App	83.3	1265	835	9253	5	4874	35	4379
Ecomm[redacted]	App	83.3	1272	614	2791	1	123	33	2668
Ecomm[redacted]	App	82.3	1273	536	2303	1	77	30	2226
Ecomm[redacted]	App	84.6	1273	560	2392	1	150	29	2242
Ecomm[redacted]	App	86.3	1269	1210	23402	17	12987	37	10415
Ecomm[redacted]	App	83.7	1273	513	2367	1	103	28	2264
Ecomm[redacted]	App	84.4	1273	544	2399	1	115	29	2284
Ecomm[redacted]	App	83.7	1273	567	2454	1	109	31	2346
Ecomm[redacted]	App	82.6	1273	504	2257	1	83	28	2174
Ecomm[redacted]	App	83.2	1273	550	2350	1	114	29	2236



```
%CPU %MEM    TIME+  COMMAND
  97  1.0   0:20.67 java
  12  9.2  15:16.81 X
   2  1.0  47:32.87 knotify4
   2  2.0 103:39.48 plugin-contai
   1  0.3 100:52.18 pulseaudio
   1  1.3   0:26.23 plasma
   1  1.0   0:09.37 kwin
   1 27.6 133:05.86 firefox-bin
   0  0.3   1:32.44 squid
```

🟩 Application Infrastructure Performance|App|Individual Nodes|Ecomm[redacted]JVM|Garbage Collection|GC Time Spent Per Min (ms) — ☐ Obs. ☐ Min ☑ Max ☐ Base

🟩 Application Infrastructure Performance|App|Individual Nodes|Ecomm[redacted]Hardware Resources|CPU|%Busy — ☐ Obs. ☐ Min ☑ Max ☐ Base

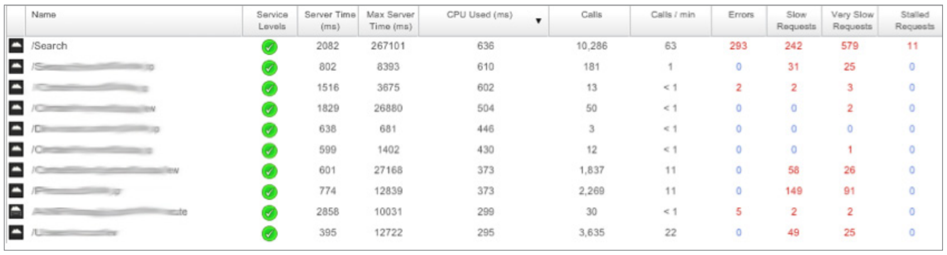
🟥 Application Infrastructure Performance|App|Individual Nodes|Ecomm[redacted]JVM|Memory|Heap|Used % — ☑ Obs. ☐ Min ☐ Max ☐ Base

🟪 Business Transaction Performance|Business Transactions|App|/Search|Individual Nodes|Ecomm[redacted]Average Response Time (ms) — ☐ Obs. ☐ Min ☐ Max ☐ Base

🟦 Business Transaction Performance|Business Transactions|App|/Search|Individual Nodes|Ecomm[redacted]Calls per Minute — ☐ Obs. ☐ Min ☐ Max ☐ Base

6. Transactions with high CPU burn

It’s no secret that inefficient application logic will require more CPU cycles than efficient logic. In practice, however, it’s much easier to speed up your application by buying more servers than by going back and optimizing your code. This practice, while it works well in the short term, is not a long-term solution to the problem. Adding capacity masks inefficient code temporarily, but if you have transactions that hog or burn CPU they will cause problems for you again as your application grows. It’s better (and cheaper) to address these problems as they arise rather than to throw money at additional infrastructure your application doesn’t need.

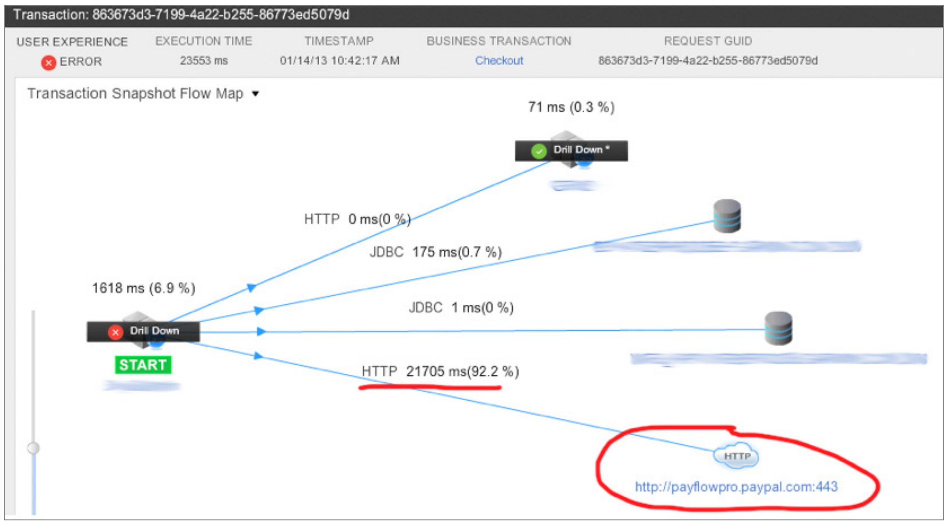


Name	Service Levels	Server Time (ms)	Max Server Time (ms)	CPU Used (ms) ▼	Calls	Calls / min	Errors	Slow Requests	Very Slow Requests	Stalled Requests
/Search	✓	2082	267101	636	10,286	63	293	242	579	11
[redacted]	✓	802	8393	610	181	1	0	31	25	0
[redacted]	✓	1516	3675	602	13	< 1	2	2	3	0
[redacted]	✓	1829	26880	504	50	< 1	0	0	2	0
[redacted]	✓	638	681	446	3	< 1	0	0	0	0
[redacted]	✓	599	1402	430	12	< 1	0	0	1	0
[redacted]	✓	601	27168	373	1,837	11	0	58	26	0
[redacted]	✓	774	12839	373	2,269	11	0	149	91	0
[redacted]	✓	2858	10031	299	30	< 1	5	2	2	0
[redacted]	✓	395	12722	295	3,635	22	0	49	25	0

The above screenshot shows the transactions in an eCommerce site sorted by the CPU time used. Monitoring the CPU usage of your business transactions is a good way to determine whether or not your application really needs new infrastructure, or if you simply need to optimize some transactions that burn a lot of CPU time.

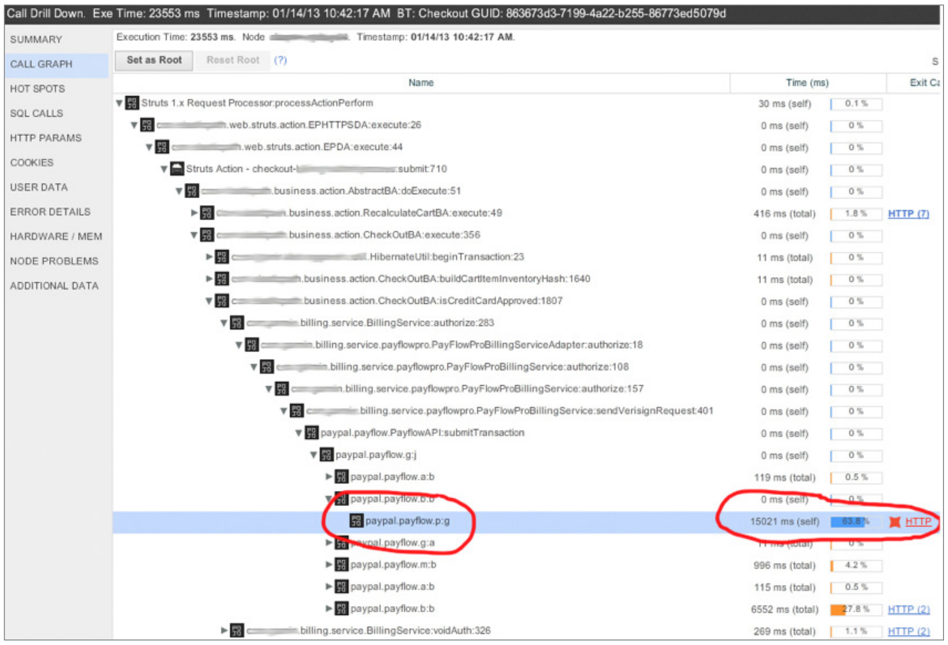
7. Slow (or unavailable) 3rd party web services

If your eCommerce application is built around a distributed service oriented architecture then your application has multiple points of failure. This can make it difficult to identify and troubleshoot a problem, especially if some of the services that your application relies on are owned and operated by third parties. For example, most payment and credit card authorization services are provided by third party vendors like PayPal, Stripe or Braintree. If these services are slow or unavailable then it's impossible for checkout transactions to complete. You need to monitor these services religiously so that when problems occur you can rapidly identify them and work with the service provider to troubleshoot them.



Transaction: 863673d3-7199-4a22-b255-86773ed5079d

USER EXPERIENCE	EXECUTION TIME	TIMESTAMP	BUSINESS TRANSACTION	REQUEST GUID
ERROR	23553 ms	01/14/13 10:42:17 AM	Checkout	863673d3-7199-4a22-b255-86773ed5079d

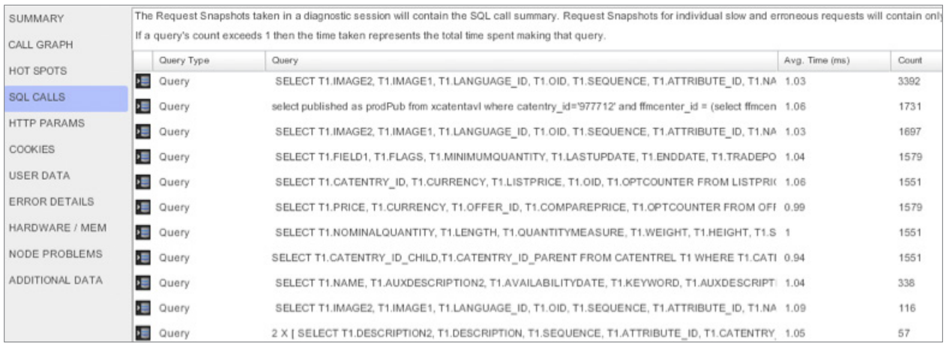


Call Drill Down. Exe Time: 23553 ms Timestamp: 01/14/13 10:42:17 AM BT: Checkout GUID: 863673d3-7199-4a22-b255-86773ed5079d

Execution Time: **23553 ms.** Node [redacted]. Timestamp: **01/14/13 10:42:17 AM.**

Name	Time (ms)		Exit Ca
Struts 1.x Request Processor:processActionPerform	30 ms (self)	0.1 %	
[redacted].web.struts.action.EPHTTPSDA:execute:26	0 ms (self)	0 %	
[redacted].web.struts.action.EPDA:execute:44	0 ms (self)	0 %	
Struts Action - checkout-[redacted]:submit:710	0 ms (self)	0 %	
[redacted].business.action.AbstractBA:doExecute:51	0 ms (self)	0 %	
[redacted].business.action.RecalculateCartBA:execute:49	416 ms (total)	1.8 %	HTTP (7)
[redacted].business.action.CheckOutBA:execute:356	0 ms (self)	0 %	
[redacted].HibernateUtil:beginTransaction:23	11 ms (total)	0 %	
[redacted].business.action.CheckOutBA:buildCartItemInventoryHash:1640	11 ms (total)	0 %	
[redacted].business.action.CheckOutBA:isCreditCardApproved:1807	0 ms (self)	0 %	
[redacted].billing.service.BillingService:authorize:283	0 ms (self)	0 %	
[redacted].billing.service.payflowpro.PayFlowProBillingServiceAdapter:authorize:18	0 ms (self)	0 %	
[redacted].billing.service.payflowpro.PayFlowProBillingService:authorize:108	0 ms (self)	0 %	
[redacted].billing.service.payflowpro.PayFlowProBillingService:authorize:157	0 ms (self)	0 %	
[redacted].billing.service.payflowpro.PayFlowProBillingService:sendVerisignRequest:401	0 ms (self)	0 %	
paypal.payflow.PayflowAPI:submitTransaction	0 ms (self)	0 %	
paypal.payflow.g:j	0 ms (self)	0 %	
paypal.payflow.a:b	119 ms (total)	0.5 %	
paypal.payflow.b:b	0 ms (self)	0 %	
paypal.payflow.p:g	15021 ms (self)	63.8 %	HTTP
paypal.payflow.g:a	11 ms (total)	0 %	
paypal.payflow.m:b	996 ms (total)	4.2 %	
paypal.payflow.a:b	115 ms (total)	0.5 %	
paypal.payflow.b:b	6552 ms (total)	27.8 %	HTTP (2)
[redacted].billing.service.BillingService:voidAuth:326	269 ms (total)	1.1 %	HTTP (2)

8. Recursive code (excessive method invocations)
Many eCommerce applications request data from multiple sources (caches, databases, web services) at the same time. Each round trip call is expensive and may involve network time along the way, so it's important to minimize the number of times a transaction makes calls to these external resources. One common mistake that causes performance issues is when a transaction calls the database multiple times (sometimes in a loop) when it could use a single query. In the screenshot below you see an example of a search transaction that made 13,000 database calls.



The Request Snapshots taken in a diagnostic session will contain the SQL call summary. Request Snapshots for individual slow and erroneous requests will contain onl
If a query's count exceeds 1 then the time taken represents the total time spent making that query.

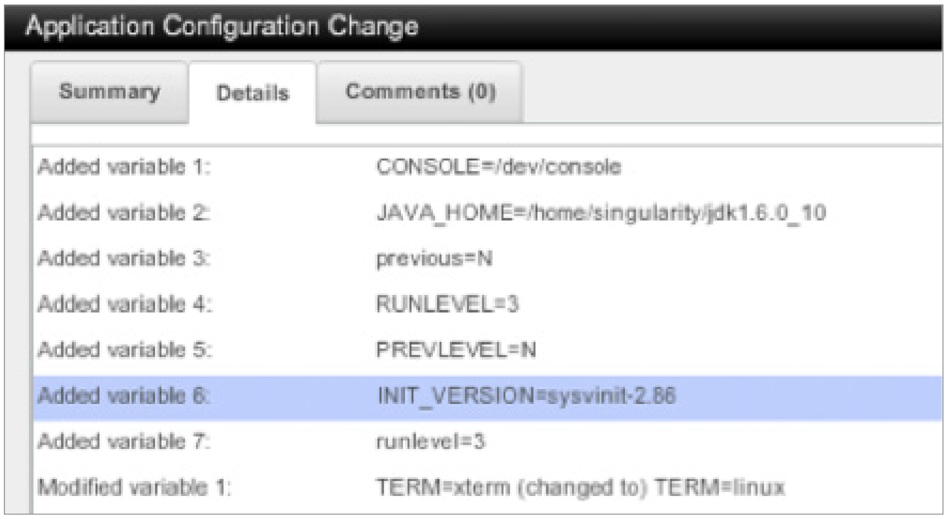
Query Type	Query	Avg. Time (ms)	Count
Query	SELECT T1.IMAGE2, T1.IMAGE1, T1.LANGUAGE_ID, T1.OID, T1.SEQUENCE, T1.ATTRIBUTE_ID, T1.NA	1.03	3392
Query	select published as prodPub from xcatentavl where catentry_id='977712' and ffmcenter_id = (select ffmcen	1.06	1731
Query	SELECT T1.IMAGE2, T1.IMAGE1, T1.LANGUAGE_ID, T1.OID, T1.SEQUENCE, T1.ATTRIBUTE_ID, T1.NA	1.03	1697
Query	SELECT T1.FIELD1, T1.FLAGS, T1.MINIMUMQUANTITY, T1.LASTUPDATE, T1.ENDDATE, T1.TRADEPO	1.04	1579
Query	SELECT T1.CATENTRY_ID, T1.CURRENCY, T1.LISTPRICE, T1.OID, T1.OPTCOUNTER FROM LISTPRI(	1.06	1551
Query	SELECT T1.PRICE, T1.CURRENCY, T1.OFFER_ID, T1.COMPAREPRICE, T1.OPTCOUNTER FROM OFI	0.99	1579
Query	SELECT T1.NOMINALQUANTITY, T1.LENGTH, T1.QUANTITYMEASURE, T1.WEIGHT, T1.HEIGHT, T1.S	1	1551
Query	SELECT T1.CATENTRY_ID_CHILD,T1.CATENTRY_ID_PARENT FROM CATENTREL T1 WHERE T1.CATI	0.94	1551
Query	SELECT T1.NAME, T1.AUXDESCRIPTION2, T1.AVAILABILITYDATE, T1.KEYWORD, T1.AUXDESCRIPT	1.04	338
Query	SELECT T1.IMAGE2, T1.IMAGE1, T1.LANGUAGE_ID, T1.OID, T1.SEQUENCE, T1.ATTRIBUTE_ID, T1.NA	1.09	116
Query	2 X [SELECT T1.DESCRIPTION2, T1.DESCRIPTION, T1.SEQUENCE, T1.ATTRIBUTE_ID, T1.CATENTRY	1.05	57

Even though each database call was relatively fast (a matter of milliseconds) the cumulative effect on response time was enormous: the transaction took almost fifteen seconds to execute, which is a long time for an impatient consumer to wait. Consolidating database calls, using eager fetching, and being very careful about how you use loops can help to prevent this problem from occurring this holiday season.

9. Configuration changes

As much as we'd like to think that production environments are locked down by the change control process, they aren't. Accidents happen, humans make mistakes and hot fixes are occasionally applied at two in the morning. Application server configuration can be very sensitive, so being able to audit, report on and compare configuration changes across your application is critical to troubleshooting configuration-related issues in production.

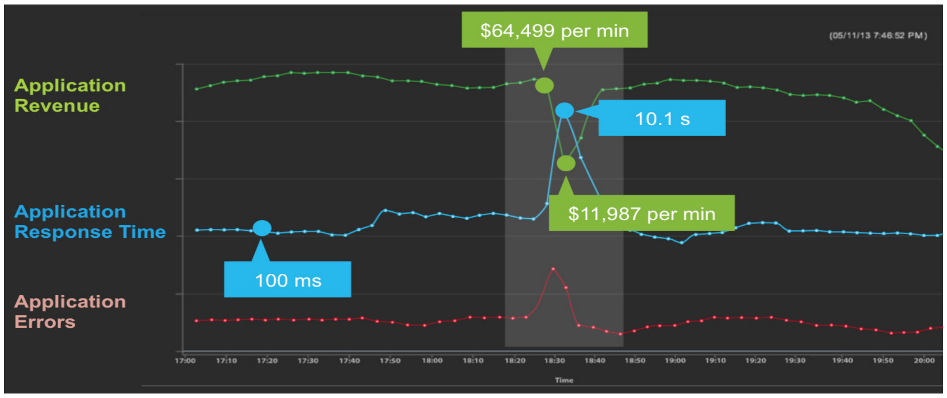
In the screenshot below you can see an example of a configuration change made in a production environment. By collecting the details around each configuration change, you can quickly discover what changes were made and how they affected performance.



Added variable 1:	CONSOLE=/dev/console
Added variable 2:	JAVA_HOME=/home/singularity/jdk1.6.0_10
Added variable 3:	previous=N
Added variable 4:	RUNLEVEL=3
Added variable 5:	PREVLEVEL=N
Added variable 6:	INIT_VERSION=sysvinit-2.86
Added variable 7:	runlevel=3
Modified variable 1:	TERM=xterm (changed to) TERM=linux

10. Out of stock exception

Sometimes what appears to be a performance issue in your application is actually a business problem. Running out of merchandise on Black Friday, for example, would anger end users and ultimately cost the business some money. Application performance management can be used to monitor business metrics like revenue and the items left in stock, however, to help you make better business decisions and prepare for events like Black Friday and Cyber Monday. Here is an example of a dashboard used by an eCommerce organization to monitor revenue through their web application and its relationship with performance.



Conclusion

Every eCommerce application is different, but the problems they experience during the holiday shopping season are usually pretty similar. The surge in traffic often surfaces performance issues and bottlenecks that have existed in the application for some time, and the solution is not to simply restart the server or throw more infrastructure at it. In order to find real, long-term solutions to some of these performance issues, you must take a good hard look at your application and work through some of the problems long before your sales roll around. With a little preparation and some powerful monitoring tools you can ensure that both the application and the business will have a successful holiday season.

About AppDynamics

AppDynamics is the next-generation application performance management solution that simplifies the management of complex, business-critical apps. No one can stand slow applications—not IT Ops and Dev teams, not the CIO, and definitely not end users. With AppDynamics, no one has to tolerate slow performing apps ever again. Visit us at appdynamics.com.

Try it FREE at
appdynamics.com