Diagnosing Production Java Applications

Madhav Sathe

Oracle India

Agenda

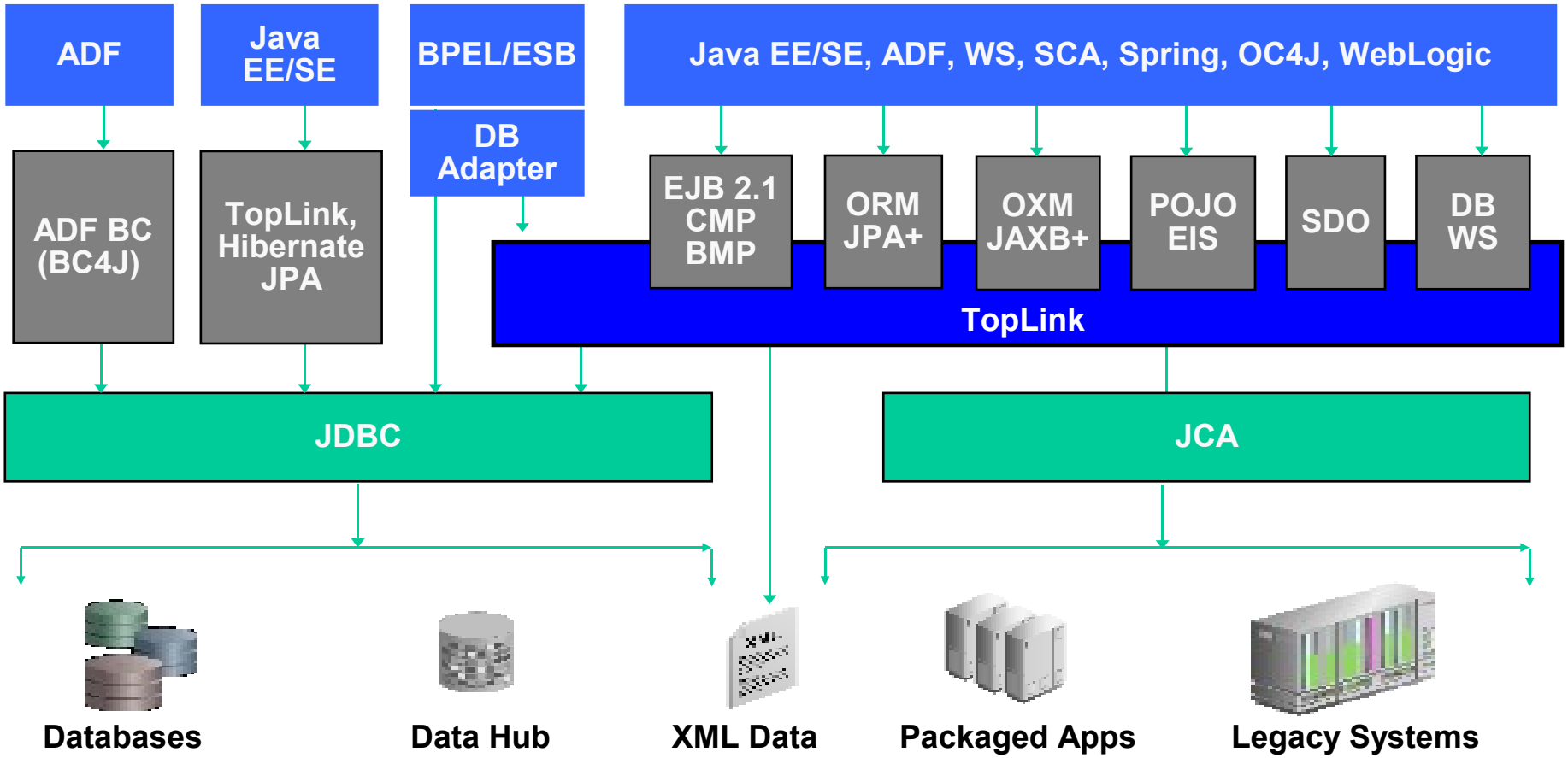
- Types of applications and issues
- Challenges
- Approaches to diagnostics
- Runtime JVM diagnostics
- Best practices
- Demo
- Q & A

Folientitel

- Font: Microsoft Sans Serif, 32 pt
  - Font: Microsoft Sans Serif, 28 pt
    - Font: Microsoft Sans Serif, 24 pt
      - Font: Microsoft Sans Serif, 20 pt

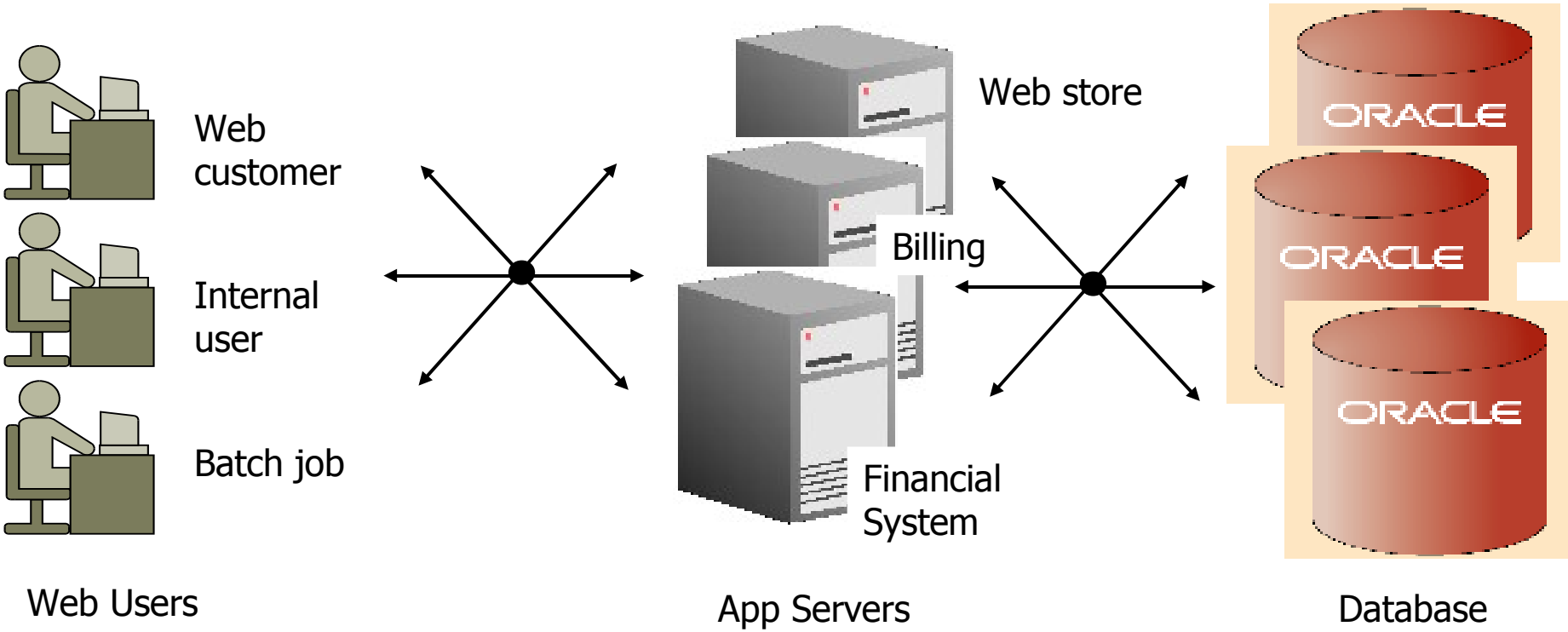
- Lorem ipsum dolor sit amet, consectetur adipscing elit, sed diam nonnumy eiusmod tempor incidunt ut labore et dolore magna aliquam erat volupat.

Applications using Java Technologies



Application Performance Issues

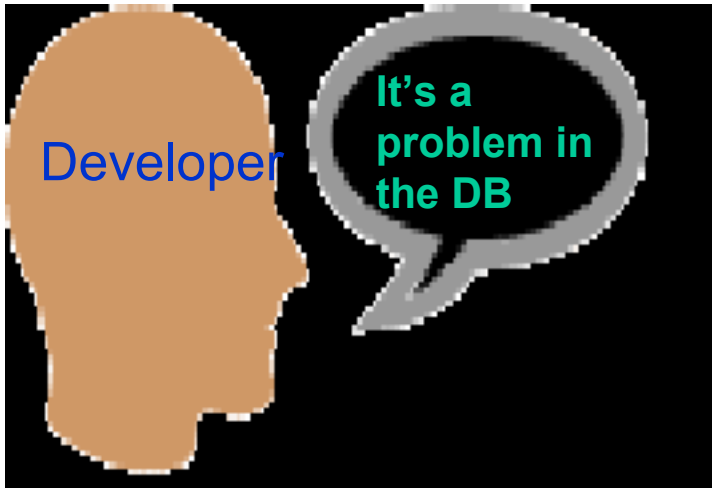
- Slow and slower response time
- Slower under heavy load
- Sporadic hangs and aberrant errors
- System locks up
- Sudden chaos / Unexpected Errors

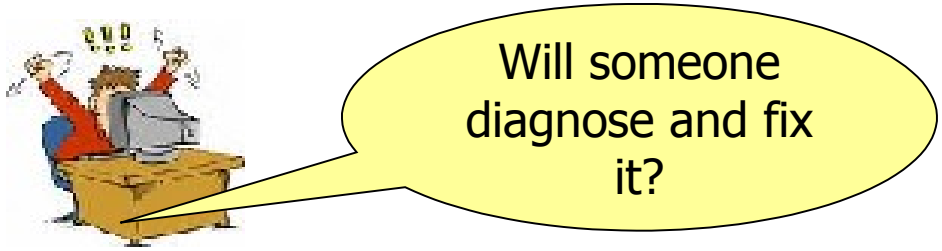


SLAs Affected

Service levels going down

Who will own the issue?

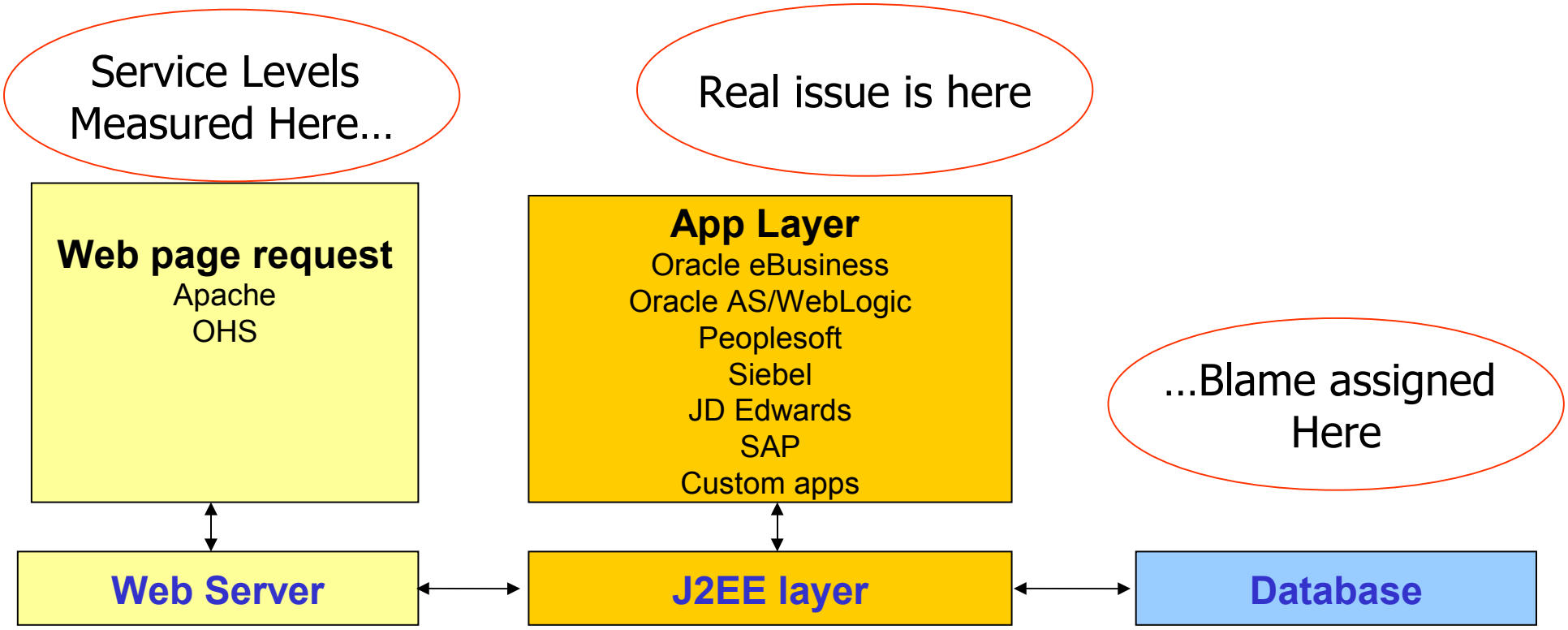




Administrator

Diagnostic Challenges: Lack of Visibility is Source of Problem

Typical Multi-Tiered J2EE System Leaves the Administrators Blind



Challenges In Diagnosing

- Highly distributed systems
- Cross tier – J2EE, DB, MOM, SOA
  - How does DB configuration affect application performance?
  - How to correlate SQLs to java sessions?
  - What about Toplink, Hibernate generated SQLs?
- Was AS tuned properly?
- No visibility into runtime environment
  - What error customer saw on browser?
  - Which part of the code is taking time?
  - Which application server instance is causing problem?
  - Was the problem in J2EE layer or DB layer?

The risk is very high as the time, effort and cost of diagnosing goes high

Application Diseases

- Java memory leak issues
  - Liner
  - Exponential
- Bad Coding
  - Infinite loops
  - Exception handling
  - Unnecessary synchronization
- Resource Leak
- Thread deadlock
- Incorrect Application Server Configuration
  - Pools, caches
  - Memory
- External Issues
  - JDBC / DB Issues
  - Messaging Provider

Approaches to Application Diagnostics

- Monitoring Metrics
  - Application, J2EE Servers, DB and Machine resources
- Find top SQL from DB monitoring and find out application code responsible
- Rerun Use case in test environment
- Use JVM diagnostic tools in test environment
- Use logs

Challenges In these approaches

- Logs don't have sufficient data
- Correlating logs across multiple application server instances and other tiers is very painful; impossible in some cases
- Test environment can't reproduce the exact scenario - load, resources contention, etc
- The actual execution context is lost forever

Challenges In these approaches

- Challenge in finding offending SQL Code in modern app using JPA or O-R Framework (TopLink, Hibernate)
- No cross tier correlation of application code with DB tier
- How to correlate metrics from various sources

Monitor in production and diagnose in test/development environment

Application Problems and Possible Causes

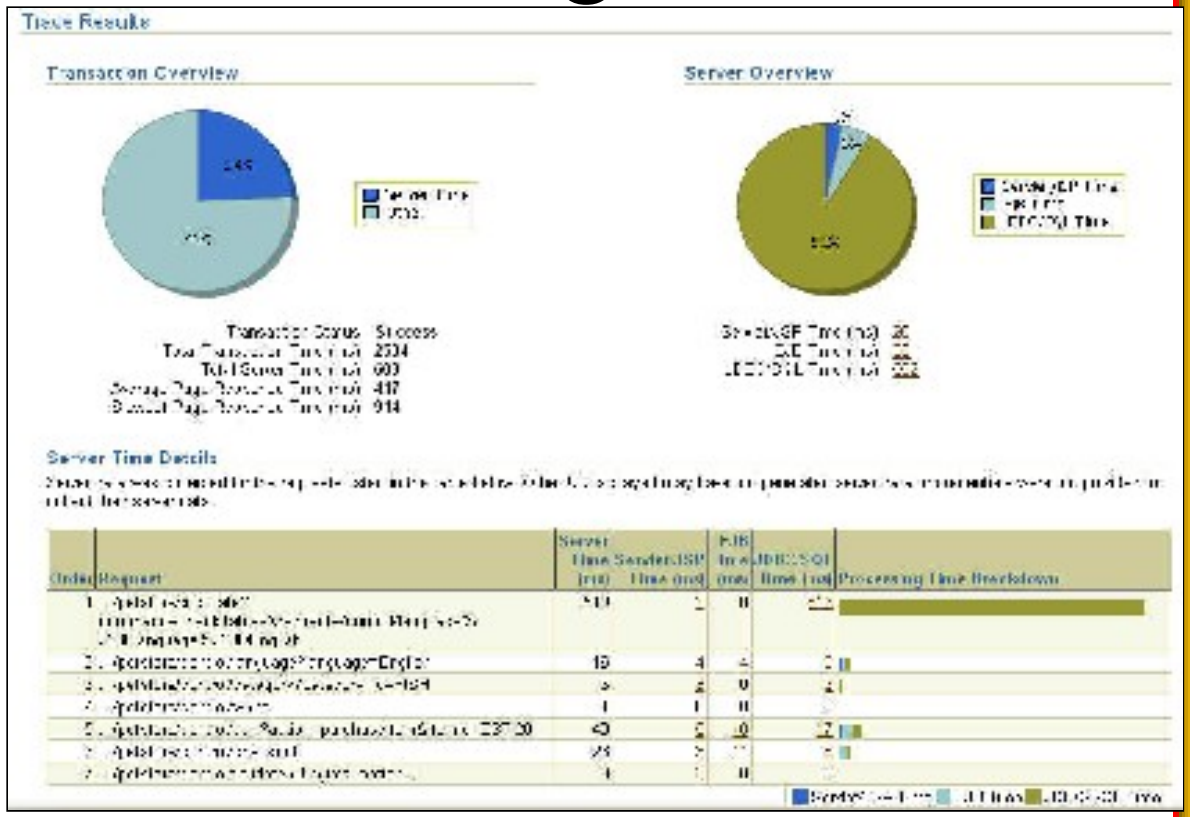
Slower over time or under load	Memory leak
Application hangs in intermittently or times out	Application code or improper app server configurion
Slower response over time, intermittent hangs	Resource Leak
Slower consistently and under load	JDBC or DB problem
Sudden chaos	Threading problem
Hangs and Sudden chaos	Application Server or Back-end DB problem

Diagnose Application Server Issues

- Monitor Application Server resources and search for bottlenecks
  - Thread pools / Work managers
  - Resource usage (JMS, Data Sources)
  - Applications (EJB Pools, bad JSPs/Servlets)
- Look for possible errors in Logs
  - Time outs
  - Exceptions
  - Memory issues
- Use tools to proactively monitor using alert/event notifications

Application Code: Use monitoring tools to find offending tier

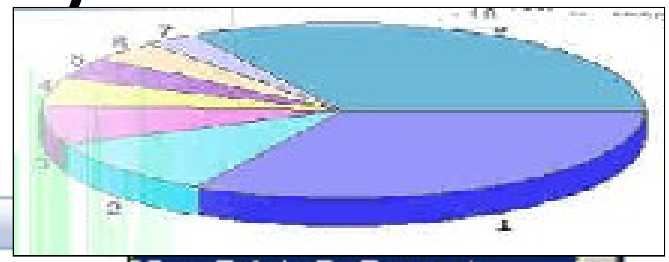
- View the time spent in server and network
- View response time breakdowns by Servlet/JSP, EJB, JDBC, and SQL



Then drill down to the offending tier

Perform Transaction Response Time Analysis

- Trace active threads in real-time
- View the bottleneck resources



Stack Trace

0.35% java.lang.String->
0.35% java.lang.Object->clone
99.30% jamagent.jamrur->main
99.30% java.lang.reflect.Method->invoke
99.30% sun.reflect.DelegatingMethodAccessorImpl->invoke
99.30% sun.reflect.NativeMethodAccessorImpl->invoke
99.30% sun.reflect.NativeMethodAccessorImp->invoke0
99.30% org.apache.catalina.startup.Bootstrap->main
81.12% org.apache.catalina.startup.Bootstrap->start
81.12% java.lang.reflect.Method->invoke
81.12% sun.reflect.DelegatingMethodAccessor
01.12% sun.reflect.NativeMethodAccessorImpl->invoke
81.12% sun.reflect.NativeMethodAccessorImpl->invoke0
81.12% org.apache.catalina.startup.Catalina->start
81.12% org.apache.catalina.core.StandardServer->start
72.38% org.apache.catalina.core.StandardService->start
3.74% org.apache.catalina.util.LifecycleSupport->fireLifecycleEvent
18.18% org.apache.catalina.startup.Bootstrap->load

View Totals By Percent
View Totals By Abs Numbers
View Lock By Percent
View Lock By Abs Numbers
View CPU By Percent
View CPU By Abs Numbers
View DB By Percent
View DB By Abs Numbers
View IO By Percent
View IO By Abs Numbers
View Net By Percent

Improve Performance by Tuning

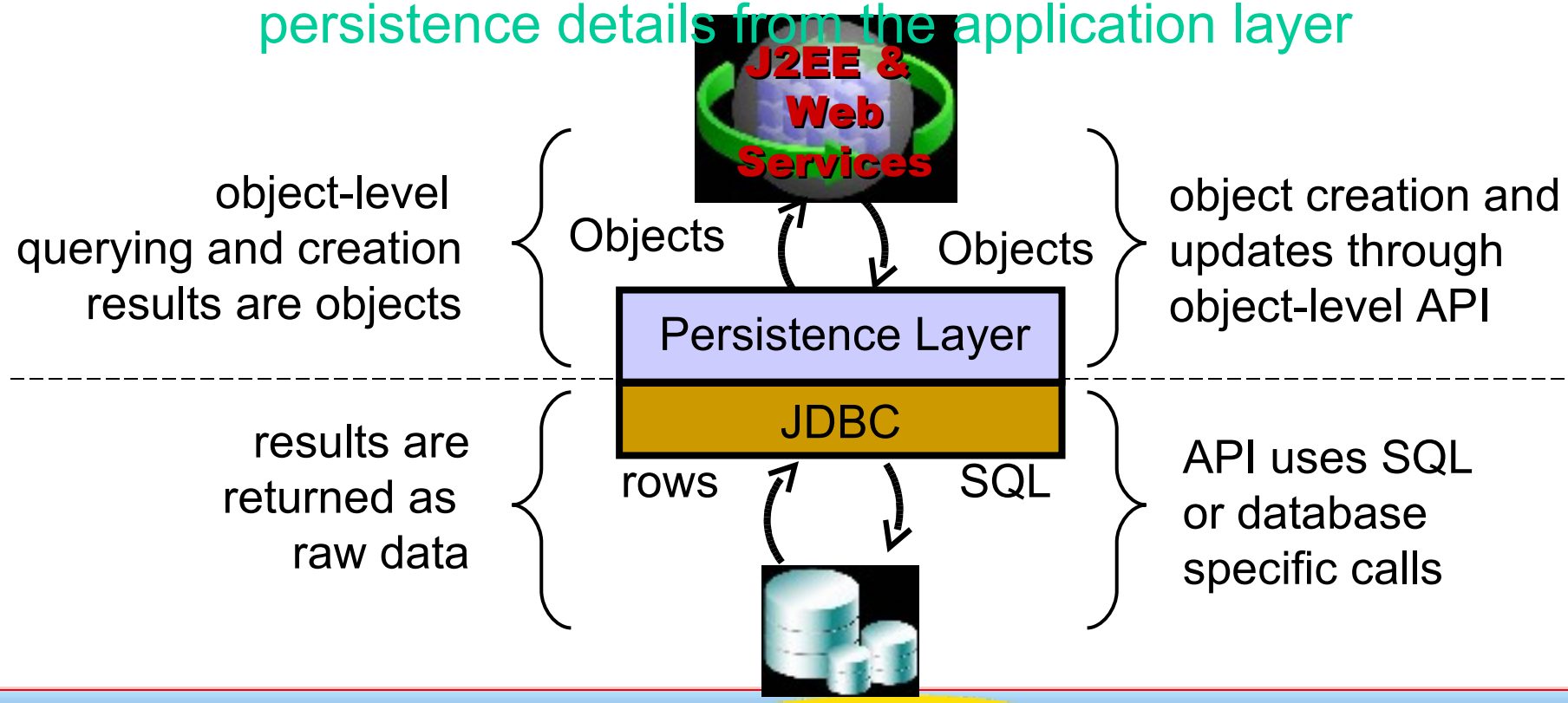
- Tune AS
  - Thread pools/work managers
  - Web Server concurrency
- JDBC / Data Source
  - Tune connection pool
  - Use SQL statement caching at DataSource level
  - Reduce round trips with pre-fetch size
  - Use lowers isolations levels when appropriate

Improve Performance by Tuning

- Improve Application DB Access
  - Enforce primary key constraint at database level
    - Avoid extra SQL for constraint checking by container/JPA provider
  - Exploit DELETE CASCADE at database level
    - Container generate multiple SQL statements
  - Batch update operations to end of transaction
- Improve code
  - Use sync code only when necessary
  - Nullify references, invalidate sessions
  - Declare variables, arguments final when possible

O-R Frameworks and its impact on DB

Technologies such as CMP Entity Beans / Java Persistence API / TopLink/Hibernate abstracts persistence details from the application layer



Tuning Tips for CMP, JPA O-R Frameworks

- Find the SQL Generated by apps
  - Log file
  - Tracing and Monitoring tools
- Make sure that SQL Generated by your finder /select method uses indexes
- Exploit statement caching by the container
- Avoid findAll() methods on large tables
- Use read-only beans when appropriate
- Reduce number of activations and passivations

Application Code Issues: Using Diagnostics Tools

- JMX based
- Byte Code Instrumentation / AOP
- JMVTI

JMX Based Monitoring

- Monitoring tools JMX data exposed by the application server and/or applications
- Get some high level metrics such as Response Time, Load, Open Connections, EJB Count, etc
- Alerts on threshold can give some indicators
- Limitations
  - No insight into code
  - Can not correlate user requests with mid tier metrics

Byte Code Instrumentation

- Find out code traces
- Can correlate user requests to code stack
- Find out code bottlenecks
- Limitations
  - Need server restart or application redeployment
  - Either instrument every thing or know what to instrument
  - Over instrumentation will be have overheads
  - Functional instrumentation is difficult and is impossible for admins

BCI Tools

- Aggregate of traces
- Segregation by layers in J2EE
- Some provide remoting support
- Capture each trace
- Some products
  - ClearApp
  - Dynatrace
  - Wily
  - Quest

JVMPI/JMVTI

- Use JVM hooks
- The agent is a shared library listening for JVM events
- Limitations
  - High overheads
  - Need JVM restart with –agentlib option
  - No cross tier visibility

JVM Native Diagnostics

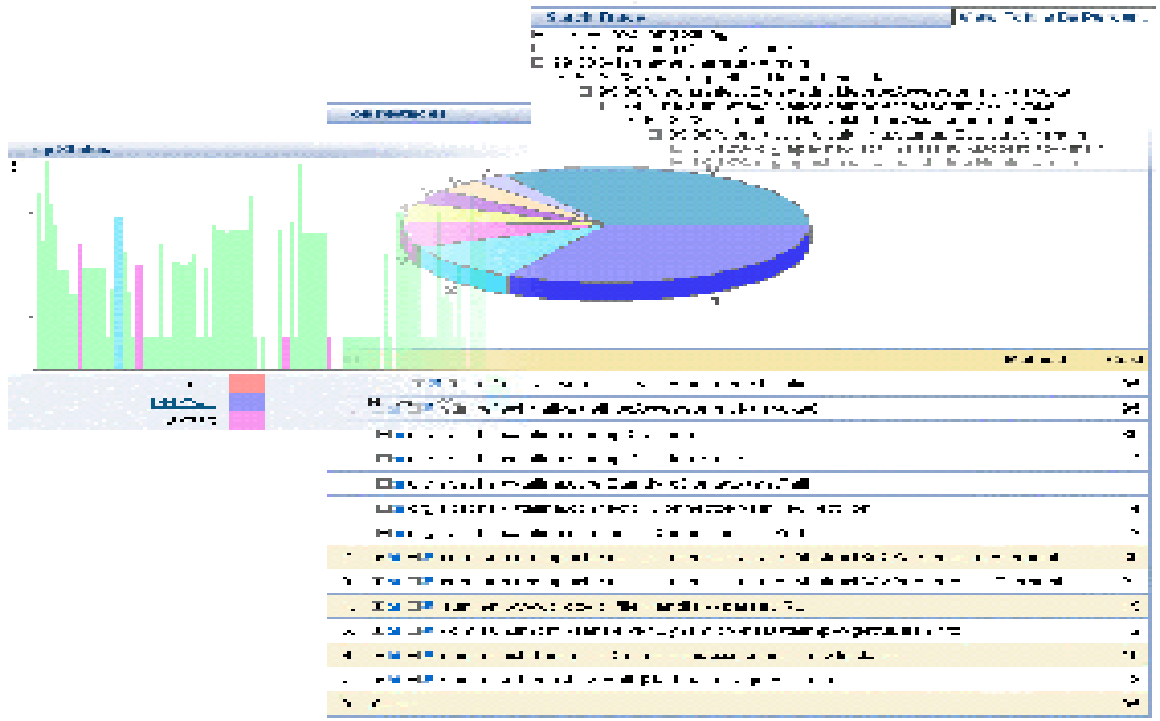
- JVM specific agent running within JVM
- Take snapshots of JVM threads and heap
- DB agent makes crenellation with J2EE stack possible
- Advantages
  - Extremely low overheads (< 1%)
  - No byte code changes
  - No server restarts or application redeployments
  - Perfect for production scenarios

A new approach: JVM Native Diagnostics

- Capabilities
  - Stack traces for all the threads
  - Correlate call stacks with end user requests
  - Find out object locks, network waits, etc
  - Memory heap analysis
  - DB agents allows to correlate DB locks to user requests and threads
  - Use repository to do historical analysis

Application Diagnostics for Java

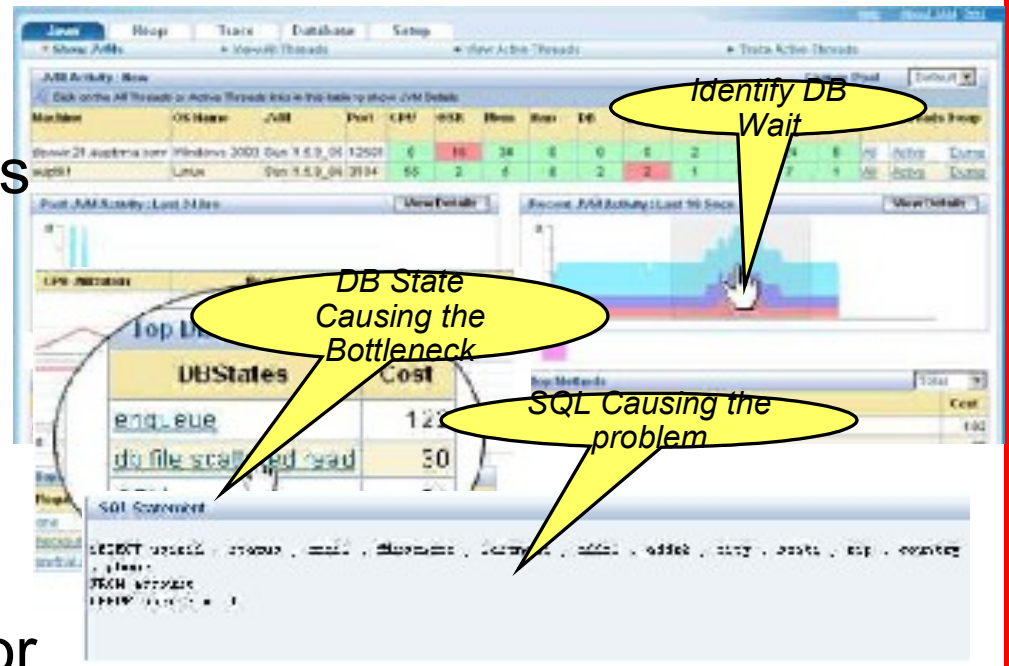
- No performance overhead
- No instrumentation
  - No server restarts
  - Need not redeploy app
- Thread state analysis.
  - Locks, n/w waits, DB waits
  - Call stack visibility
- Memory leak analysis



Only solution for diagnosing application performance issues in production

Cross-tier trace with DB

- Trace Java thread to DB session
  - Identify in-flight Java threads waiting for DB resource
  - Drill to SQL
- Trace DB session to Java thread
  - View DB sessions waiting for or holding locks
  - Identify Java thread holding DB session



Best Practices

- Use single console to monitor all system components
- Choose right JVM diagnostic tool
  - Should not need restarts/redeployments
  - Monitor threads, heap real time
  - Near zero % overheads
  - Cross tier
- Capture the actual HTTP transactions seen by clients
- Capture traces in real time and segregate performance by various J2EE layers
- Correlate transactions across JVMs

Do all this without any/negligible overhead on runtime system

Some Products

Feature	Products
Infrastructure/Services management and monitoring	Oracle Grid Control, BMC, IBM Tivoli, HP Openview
Application tracing	Oracle ClearApp, Wily, Dynatrace, Quest
Real user experience	Oracle Real User Experience Insight (Moniforce), Quest Foglight, HP Real User Monitor
Production JVM and cross-tier diagnostics	**Oracle Application Diagnostics for Java (AD4J)**

AD4J Demo

Questions?