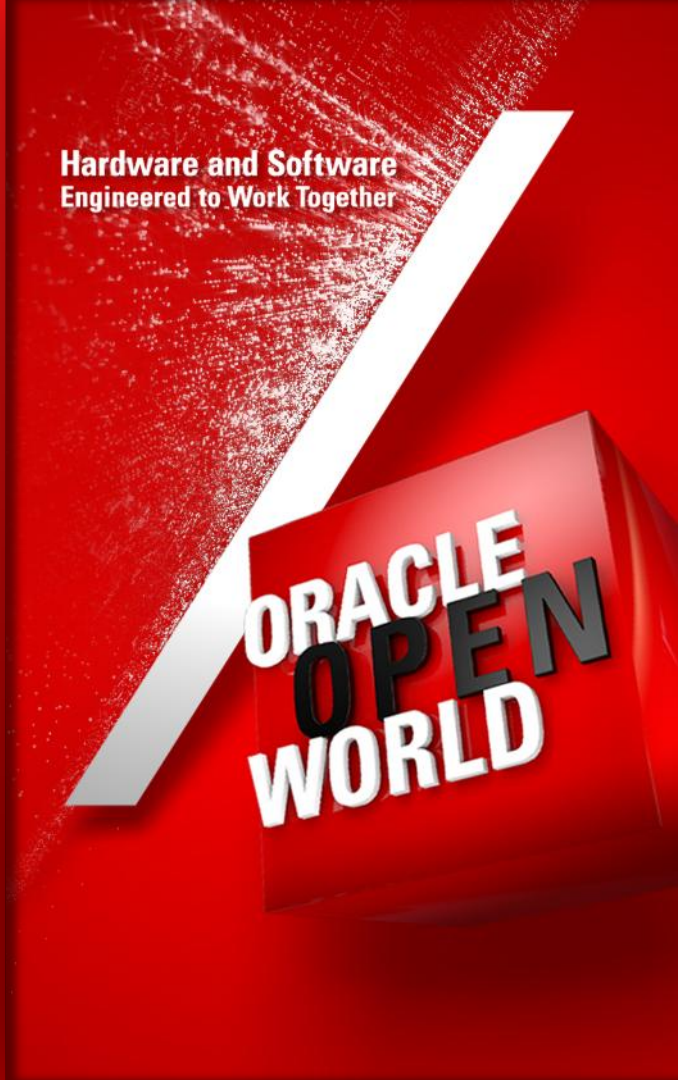

ORACLE®

ORACLE

Advanced Database Performance Analysis Techniques Using Metric Extensions and SPA

Mughees A. Minhas
VP of Product Management
Oracle

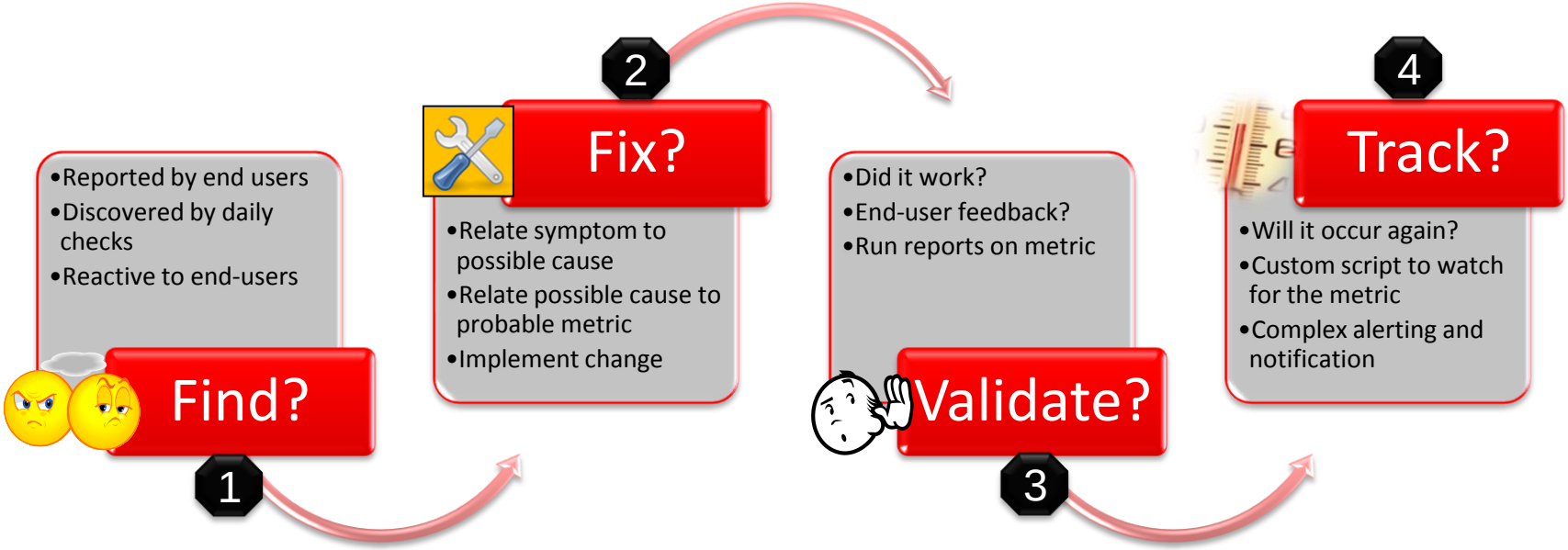


Program Agenda

- Database Performance Analysis
  - Challenges
- Advanced Use Cases
  1. Detecting Run-away Queries Using Metric Extensions
  2. Identifying High Risk SQL in Growing Data Volume Environment

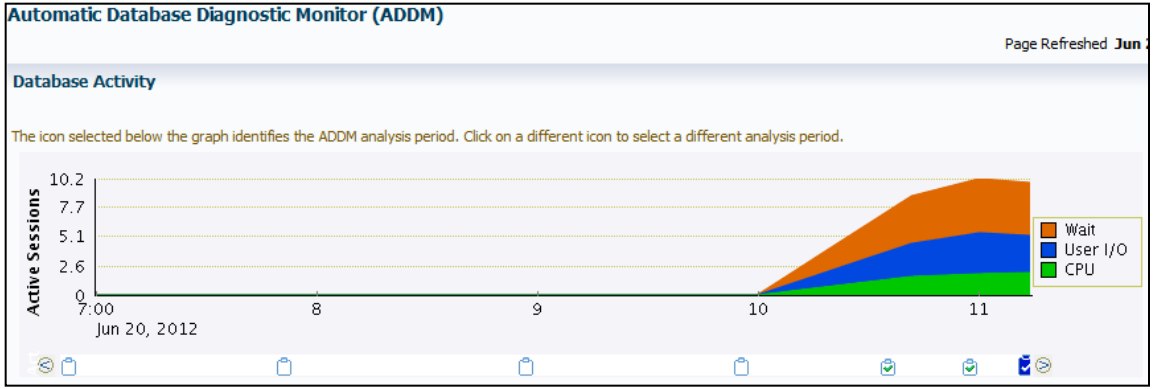
Database Performance Analysis

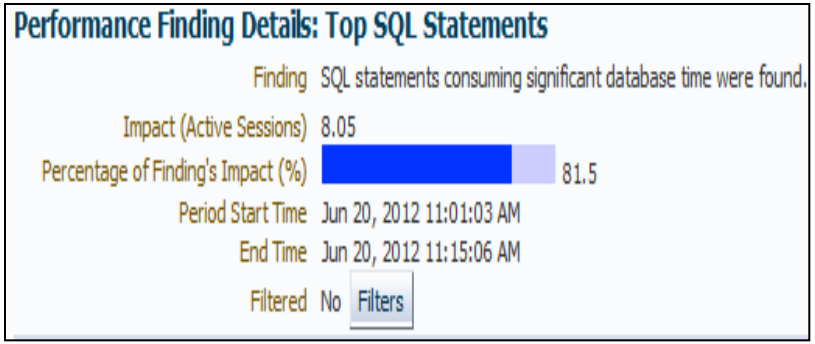
Challenges



1 – Find: Diagnostics

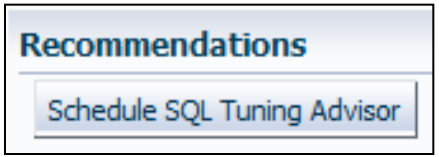
- For problem diagnosis use ADDM, EM Performance
- ADDM reveals significant problems including problematic SQL details and recommendations improve to performance
- ADDM family – Compare Period, Real-Time ADDM

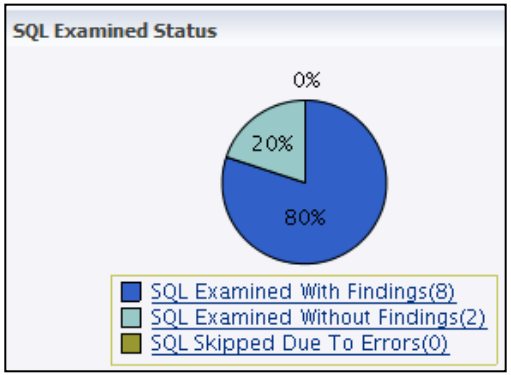


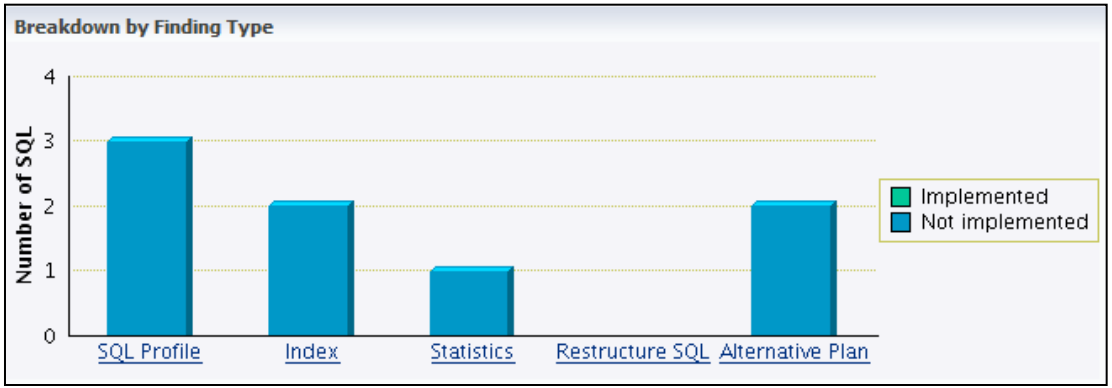


2 – Fix: Tuning

- Seamless integration between Find and Fix
- In-depth analysis and recommendation of the fix
- Gather statistics for this example…
- How would we validate this changes?

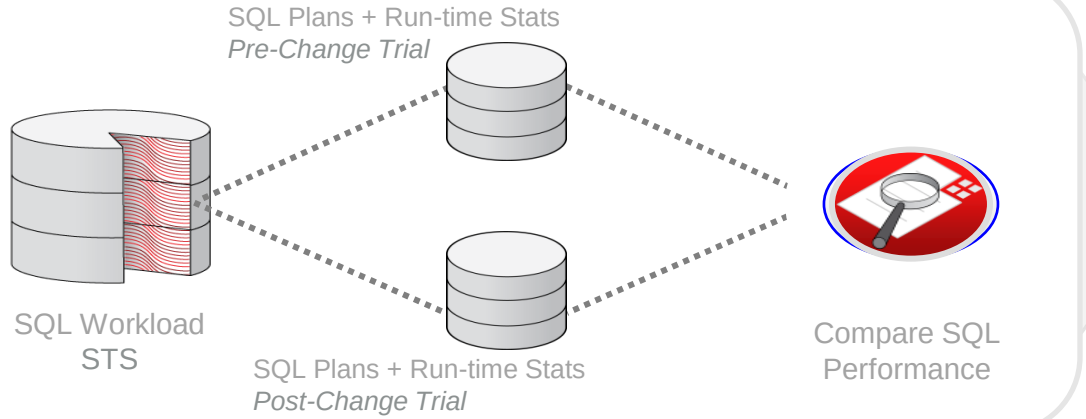


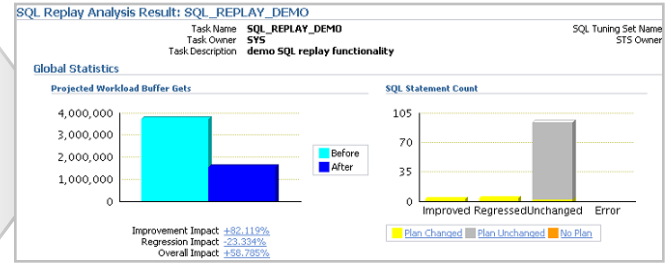




3 – Validate: Real Application Testing

SQL Performance Analyzer (SPA)



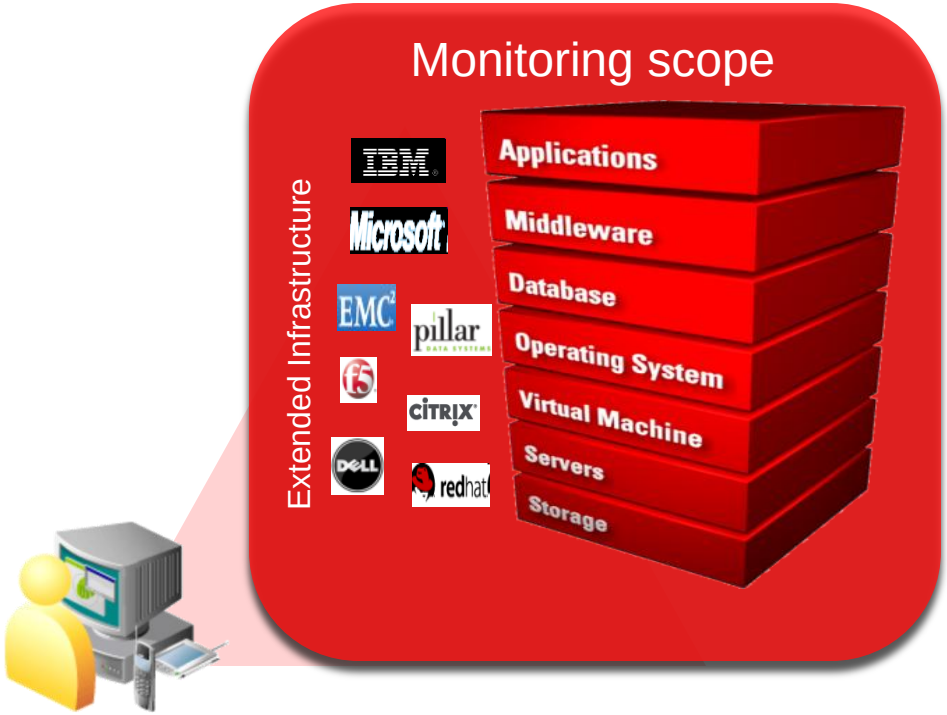


Analysis Report

- Test and predict impact of system changes on SQL query performance
- Analyze performance changes for improvements and regressions
- Comprehensive performance analysis and reporting
- Re-execute SQL queries in the given environment
- End-to-end solution: STS, SQL Plan Baselines, and SQL Tuning Advisor

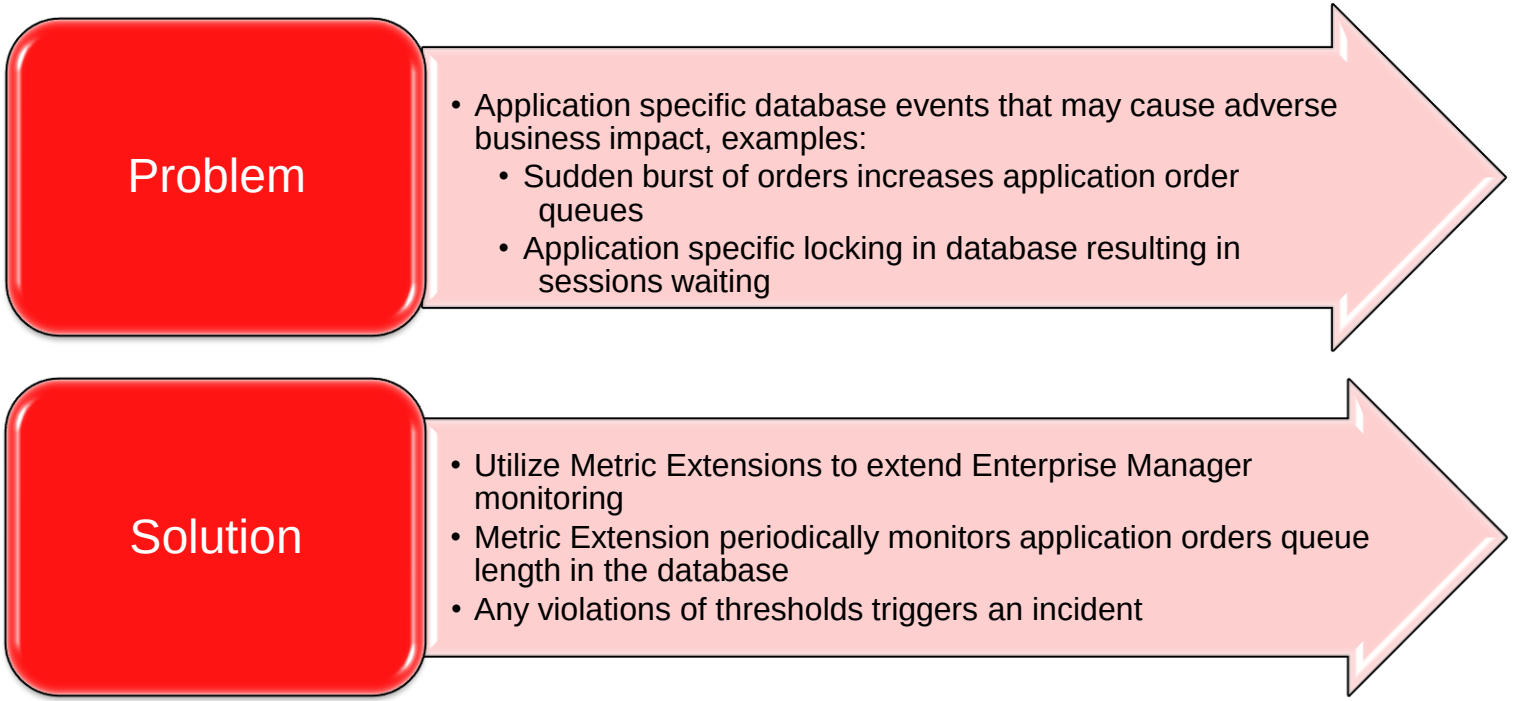
4 – Track: Monitoring in Enterprise Manager (EM)

- Lights-out data center monitoring
  - Manage by exception
  - Notifications
  - Integrated with My Oracle Support
- Complete and integrated across stack
  - Entire Oracle stack
  - Heterogeneous infrastructure monitoring via plug-ins
  - Extensible
- Integrates with third-party systems
  - Helpdesks and other management systems
- Metric Extensions (12c)
  - Next generation User Defined Metrics (UDM)
  - Can be defined for any target type, *including applications*



4 – Track: Metric Extensions

EM Metrics for Application Monitoring



4 – Track: Orders Queue Length Metric Extension

- Create a Metric Extension that monitors ORDERS_QUEUE table
- Set Warning and Critical limits for New and Processing status
- Set Incident Rules – notify when warning or critical limits are violated

Basic Properties

SQL Query

```
select decode(status,1,'NEW',2,'PROCESSING',3,'COMPLETE') as STATUS,count(status) as
ORDER_STATUS
from oe.order_queue
group by status
having STATUS in (1,2)
```

Sql or PL/SQL statement. Either SQL Query or SQL Query File should be provided

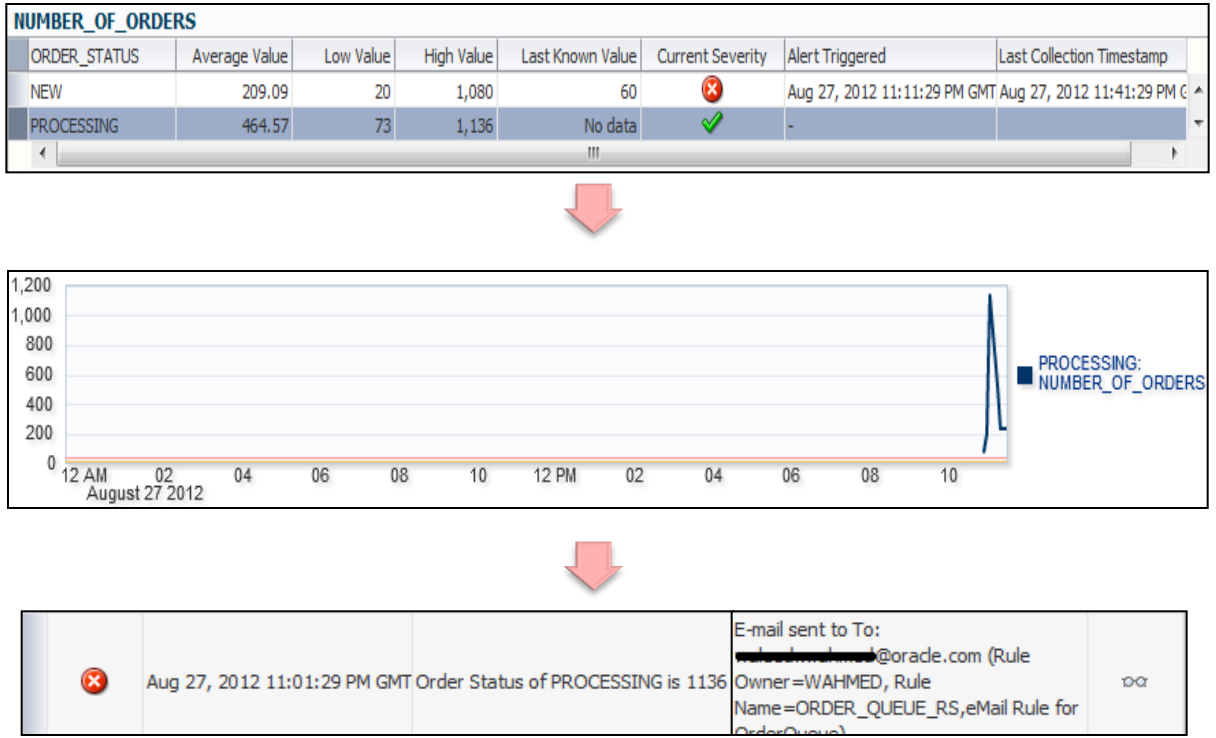
SQL Query File

Absolute path to file containing SQL script. Either SQL Query or SQL Query File should be provided

Alert Threshold		
Comparison Operator	Warning	Critical
>	20	40

4 – Track: Monitoring

- Your custom metric integrated natively into OOB metrics
- Historical metric view in one pane shows past trend of orders queues, stored in EM repository
- Alert history summary with actions taken
- Email notification sent when violations occur



Program Agenda

- Database Performance Analysis
  - Challenges
- Advanced Use Cases
  1. Detecting Run-away Queries Using Metric Extensions
  2. Identifying High Risk SQL in Growing Data Volume Environment

EM DEV SUCCESS STORY

Case Study 1: Detecting Run-Away Queries Using Metric Extensions

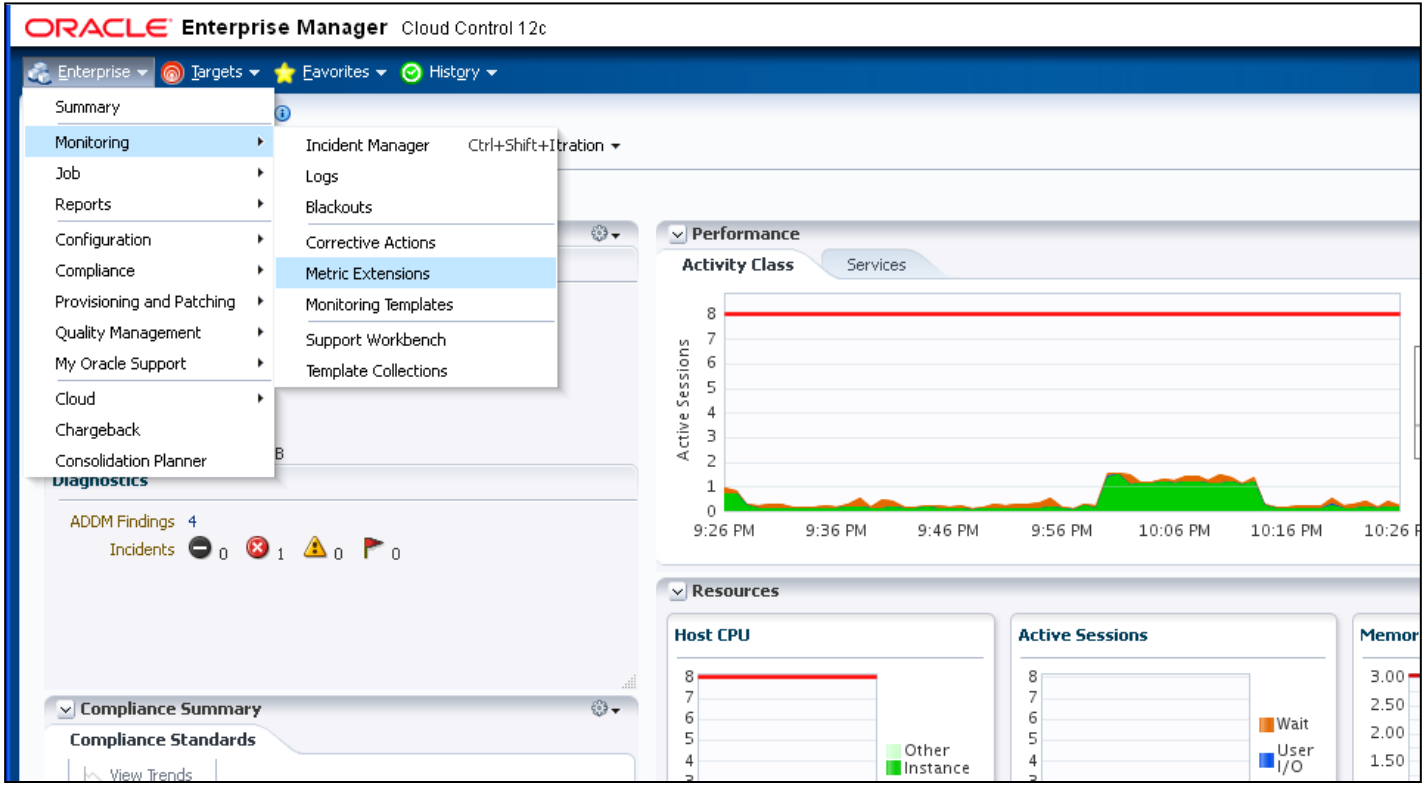
Author: John Beresniewicz, Oracle

ORACLE

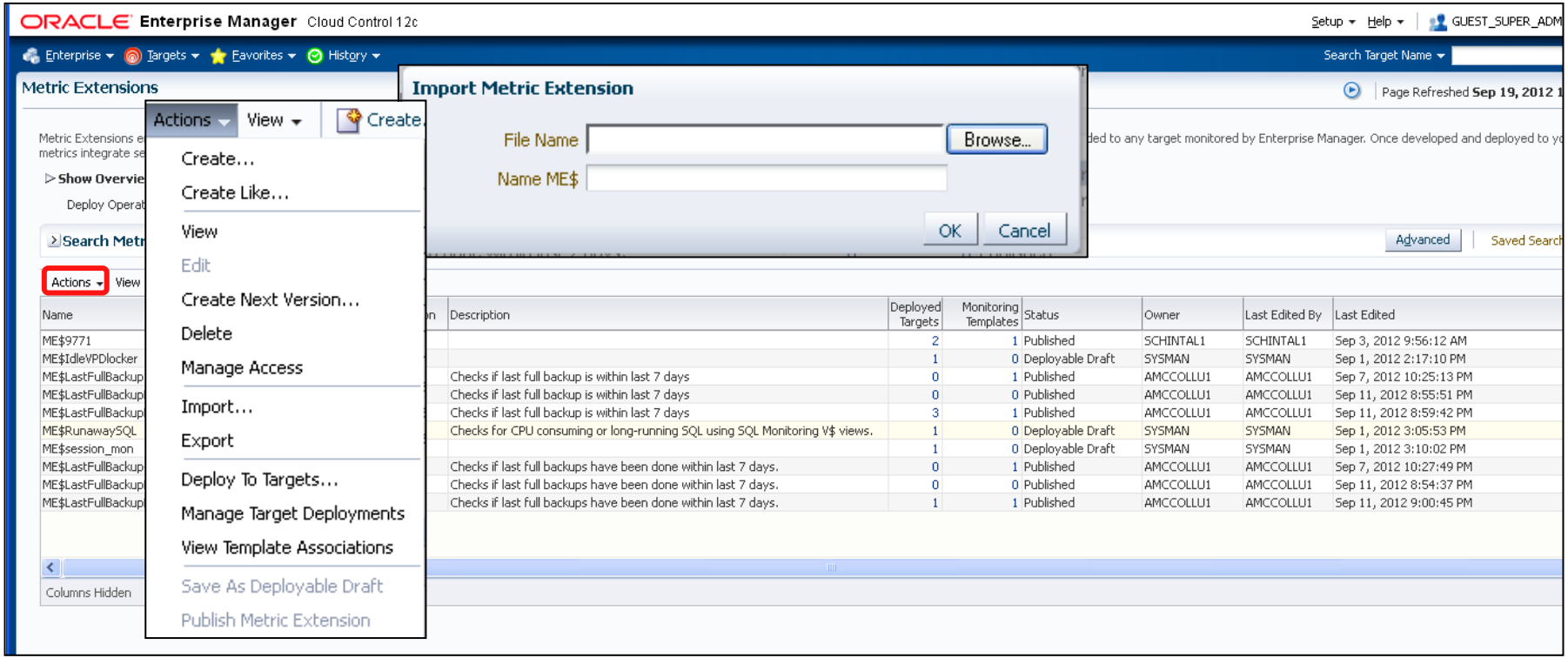
Metric Extension (ME) Lifecycle

- Develop and unit test privately
- Deploy and system test on actual targets
- Publish for general use in EM
- Export/import to propagate to other EM environments

Find Metric Extensions



Manage Metric Extensions



“Runaway Query”

- SQL that consumes too much CPU or executes too long
  - BUGS: Need to find during Development and Testing contexts
- Real-time SQL Monitoring introduced in DB 11g
  - Monitors long-running and parallel query (PQ) SQL executions
  - Rich interactive user interface and Active Report
- GV$SQL_MONITOR or V$SQL_MONITOR

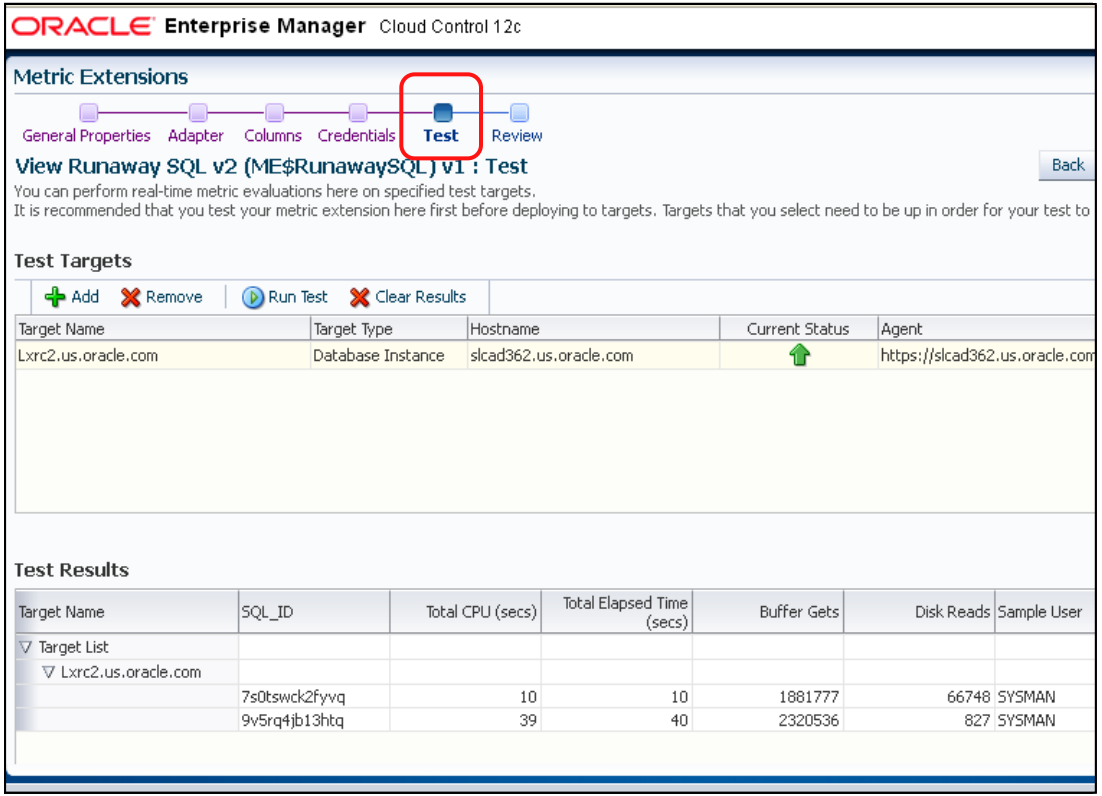
Design Metric Extension

- GV$SQL_MONITOR query
  - SQL_ID: key column
  - TotalCPUSecs
  - TotalElapsedSecs
- Execute every 15-minutes (to not miss anything)
- Set alert thresholds on CPU and Elapsed times

```
WITH SQLM as
(select sql_id,sql_exec_start,sql_exec_id
      ,MAX(M.user#)                                                    as UserNum
      ,MAX(M.username)                                                 as UserName
      ,MAX(NVL(PX_QCInst_ID,M.inst_id))                                as ExecInst
      ,MAX(NVL(PX_QCsid,M.sid))                                        as ExecSid
      ,MAX(NVL(PX_QCsid,M.session_serial#))                            as ExecSerial
      ,MAX(CASE PX_QCsid WHEN null THEN null ELSE M.status END)        as Status
      ,DECODE(count(distinct px_server#),0,'SERIAL','PARALLEL')        as PQ_SERIAL
      ,COUNT(DISTINCT M.inst_id)                                       as instances
      ,MAX(NVL(PX_MAXDOP,1))                                           as MaxDOP
      ,MAX(last_refresh_time)                                          as last_refresh_time
      ,SUM(elapsed_time/1000000)                                       as DBtimeSecs
      ,ROUND((MAX(last_refresh_time)-
              MIN(sql_exec_start))*24*60*60,1)                         as ExecElapsedSecs
      ,ROUND((MAX(last_refresh_time)-
              MIN(sql_exec_start))*24*60*60 +
              SUM(NVL(queuing_time,0)/1000000),1)                      as TotalElapsedSecs
      ,ROUND(SUM(NVL(queuing_time,0)/1000000),1)                       as QueuingSecs
      ,SUM(cpu_time/1000000)                                           as CPUsecs
      ,SUM(user_io_wait_time/1000000)                                  as IOsecs
      ,SUM((application_wait_time+concurrency_wait_time+cluster_wait_time)/1000000) as WaitSecs
      ,SUM((plsql_exec_time+java_exec_time)/1000000)                   as JavaPLSQLsecs
      ,SUM(buffer_gets)                        as BuffGets
      ,SUM(disk_reads)                         as DiskReads
      ,SUM(direct_writes)                      as DirectWrites
 from
      gv$sql_monitor M
group by sql_exec_id,sql_exec_start,sql_id
)
select sql_id                 as SQL_ID
      ,ROUND(SUM(CPUsecs))    as TotalCPUSecs
      ,SUM(ExecElapsedSecs)   as TotalElapsedSecs
--      ,IOSecs
--      ,WaitSecs
      ,SUM(BuffGets)  as TotalBuffGets
      ,SUM(DiskReads) as TotalDiskReads
      ,MAX(UserName)  as SampleUser
      ,COUNT(*)       as NumExecs
from
      SQLM
where
    ( status like 'EXECUTING%'                                                  -- currently executing or
     OR (status like 'DONE%' AND last_refresh_time > SYSDATE - 15/(24*60))      -- finished last 15 minutes?
```

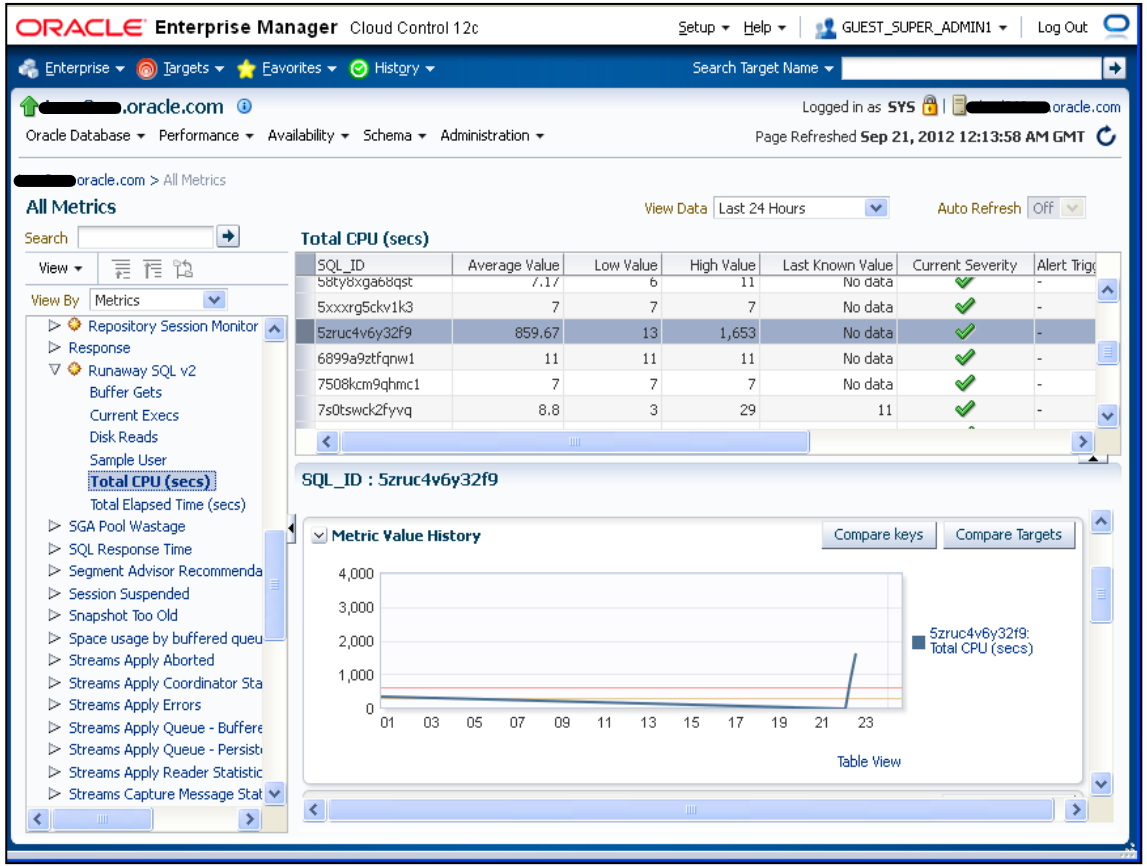
Create Metric Extension

- Guided wizard makes it easy
  - Well organized
  - Excellent explanations
- Develop and test on any database before deploying to production



Monitoring

- Email notification with custom subject line
- Informative email with context-sensitive hyperlinks into EM
- Your metric is a first-class EM metric



Metric Extensions are Cool!

- First-class metrics of your own design
- Easy-to-use user interface makes it a snap
- Lifecycle support and export/import important value-adds
- Monitor targets for application-specific issues
  - OOB metrics are for the general case
- RunawaySQL ME has found many bugs
  - Using EM to improve EM

Program Agenda

- Database Performance Analysis
  - Challenges
- Advanced Use Cases
  1. Detecting Run-away Queries Using Metric Extensions
  2. Identifying High Risk SQL in Growing Data Volume Environment

Case Study 2: Identifying High Risk SQL in Growing Data Volume Environment

Identify High Risk SQL with Growing Data Volumes

Problem

- Data stores and sources are expanding at a rate greater than ever
- The fast increase of data volume commonly results in performance slowdown
- Performance slow down can lead to unplanned downtime

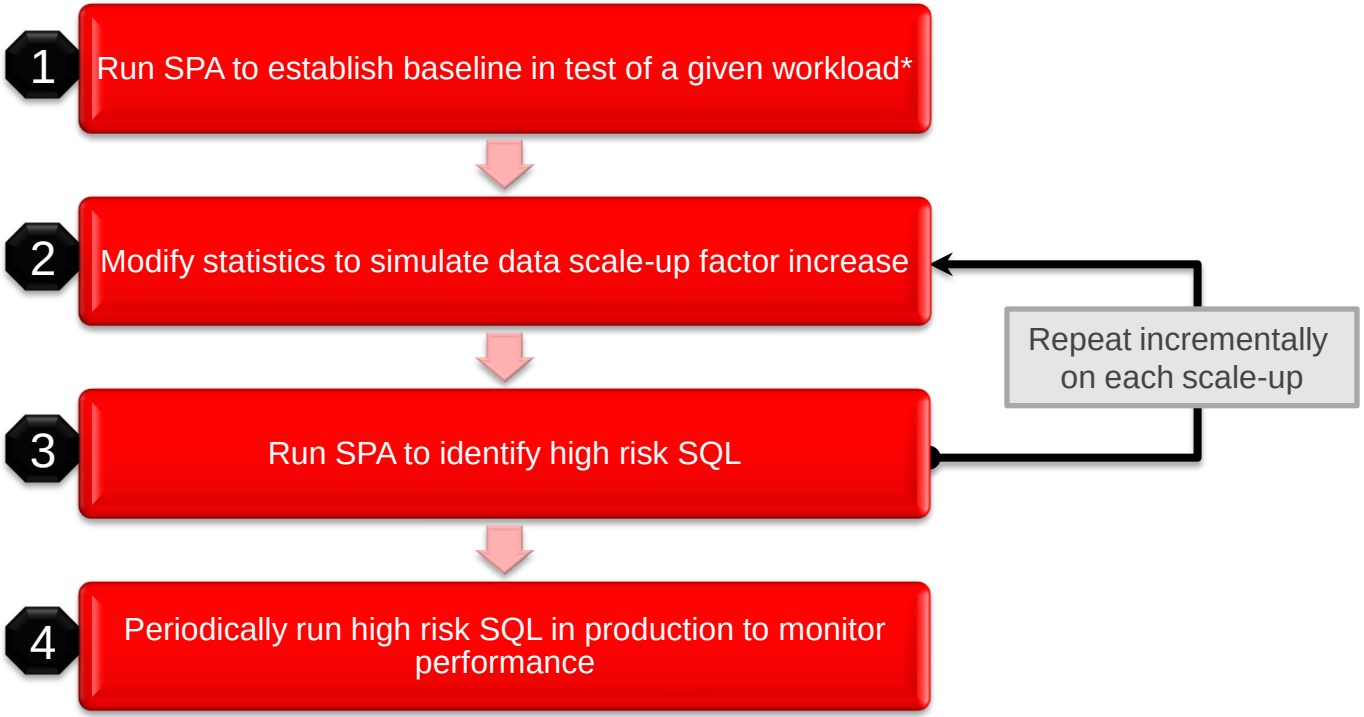
Reactive Fix

- Adjust system or db configuration
- Tune storage system or SQL reactively
- Provision more hardware
- Reduce or restrict user access

Proactive Optimization

- Identify SQL with potential volatile plans in increasing data volume DBs
- Preempt SQL performance reliably
- Assess and plan for data growth

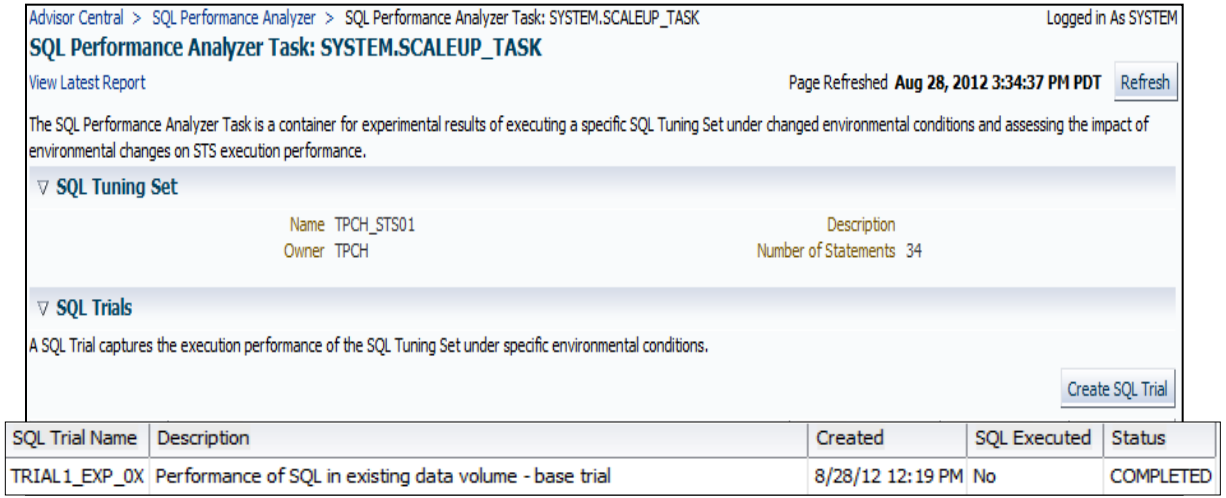
Identify High Risk SQL with Growing Data Volumes



** Workload captured into SQL Tuning Sets (STS)*

1 – SPA Baseline Trial in Test

- Setup test environment either full or subset of data
- Ensure table and index statistics are same as that of production
- Run SPA trials in explain plan mode for the workload



2 – Simulate Data Growth using Statistics

Backup existing table and index statistics

- Create table to hold original table and index statistics

```
dbms_stats.create_stat_table()
```

- Export current table and index statistics to this table

```
dbms_stats.export_table_stats()
dbms_stats.export_index_stats()
```

Simulate scale up of data volume

- Assuming column data is uniformly distributed: …For all tables increase number of rows and blocks by a factor of 1.1 (10%) using

```
dbms_stats.set_table_stats()
```

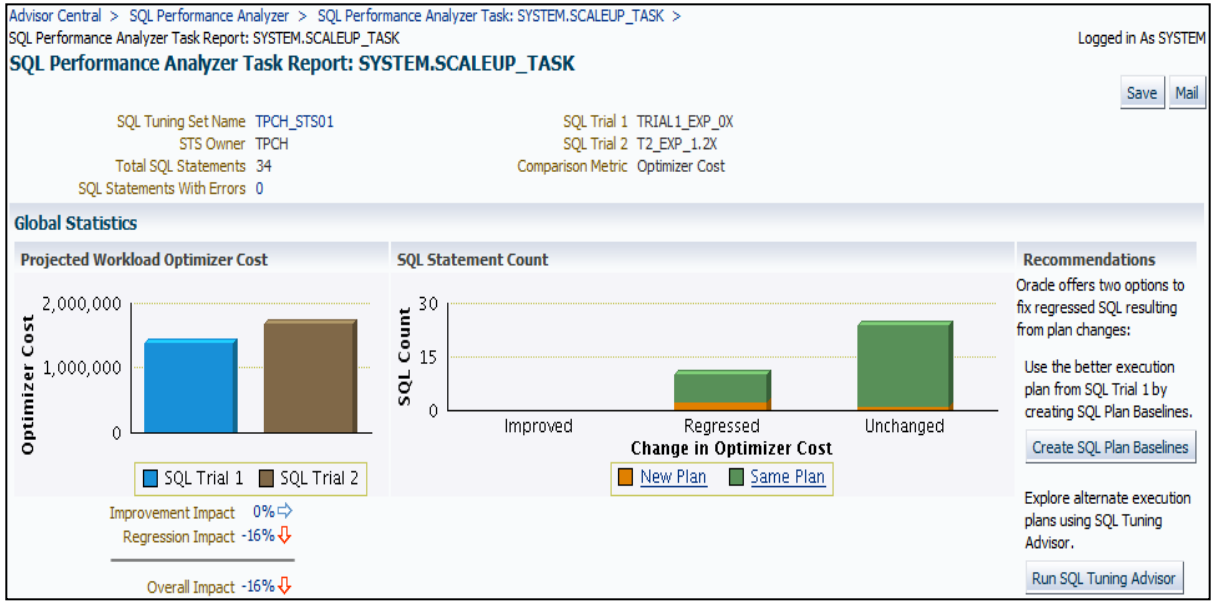
- For all indexes increase number of rows, number of leaf blocks, number of distinct keys and clustering factor by a factor of 1.1 (10%), using

```
dbms_stats.set_index_stats()
```


Refer to slide notes section for example commands

3 – Run SPA Trial Analysis

- Run SPA Trial with the scaled-up optimizer statistics to assess the impact of data volume growth
- Identify High Risk SQLs (SQL with plan changes) due to data volume growth



4 – Periodically Test High Risk SQL in Production

Subset high risk SQL (SQL with volatile execution plans) into a separate SQL Tuning Set

- Create a new empty SQL Tuning Set:

```
dbms_sqltune.CREATE_SQLSET()
```

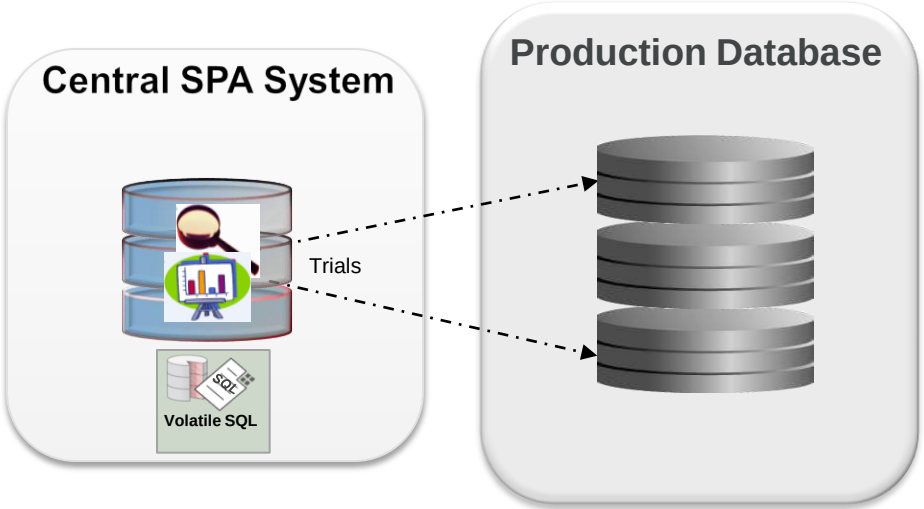
- Load high risk SQL into the new SQL Tuning set from the SPA task

```
dbms_sqltune.load_sqlset()
```

Using this STS, run SPA trials periodically with per execution time limits in production to monitor and tune volatile SQL Plans

4 – Periodically Test High Risk SQL in Production

- Baseline STS on production once with SPA
- On a routine basis run SPA Analysis and compare to baseline to identify SQL regressions
- Preempt performance degradation and fix proactively in rapidly growing data volume environments



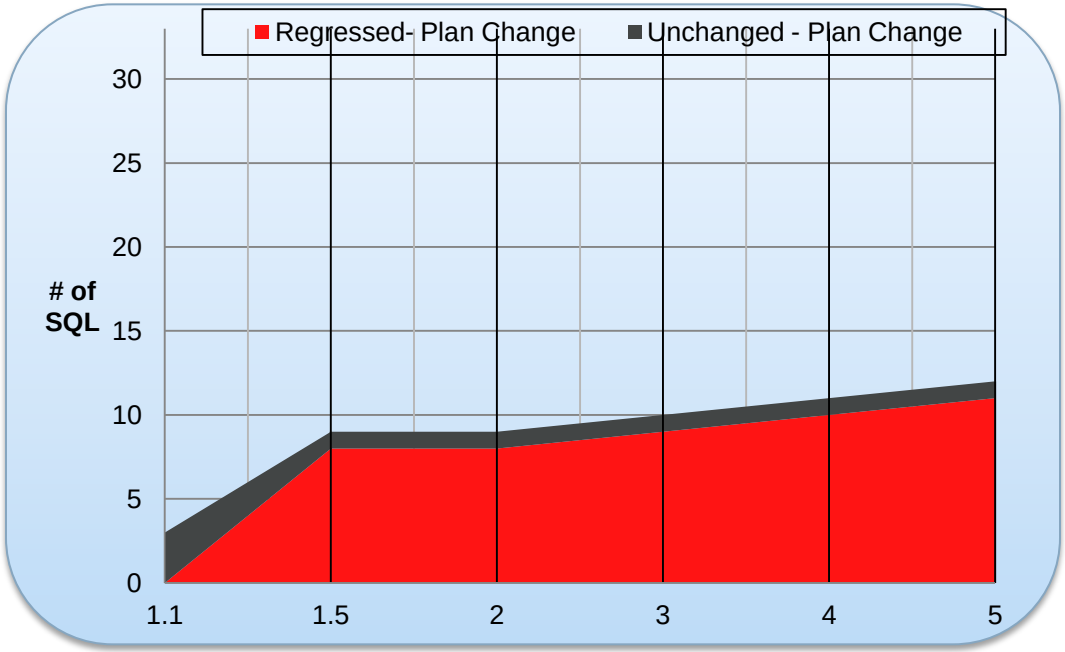
Case Study: Predict Data Growth Effect on SQL Performance

Using Optimizer Statistics and SPA, test incrementally, SQL Plan volatility with data volume growth

- TPCH workload containing 33 queries
- Run SPA tests at increments of 1.1, 1.5, 2, 3, 4 and 5 times the original data size using dbms_stats to simulate this data volume increase
- Tests were done assuming uniform column data distribution
- Optimizer Cost was measured for the workload
- SPA Trials were run in explain plan mode

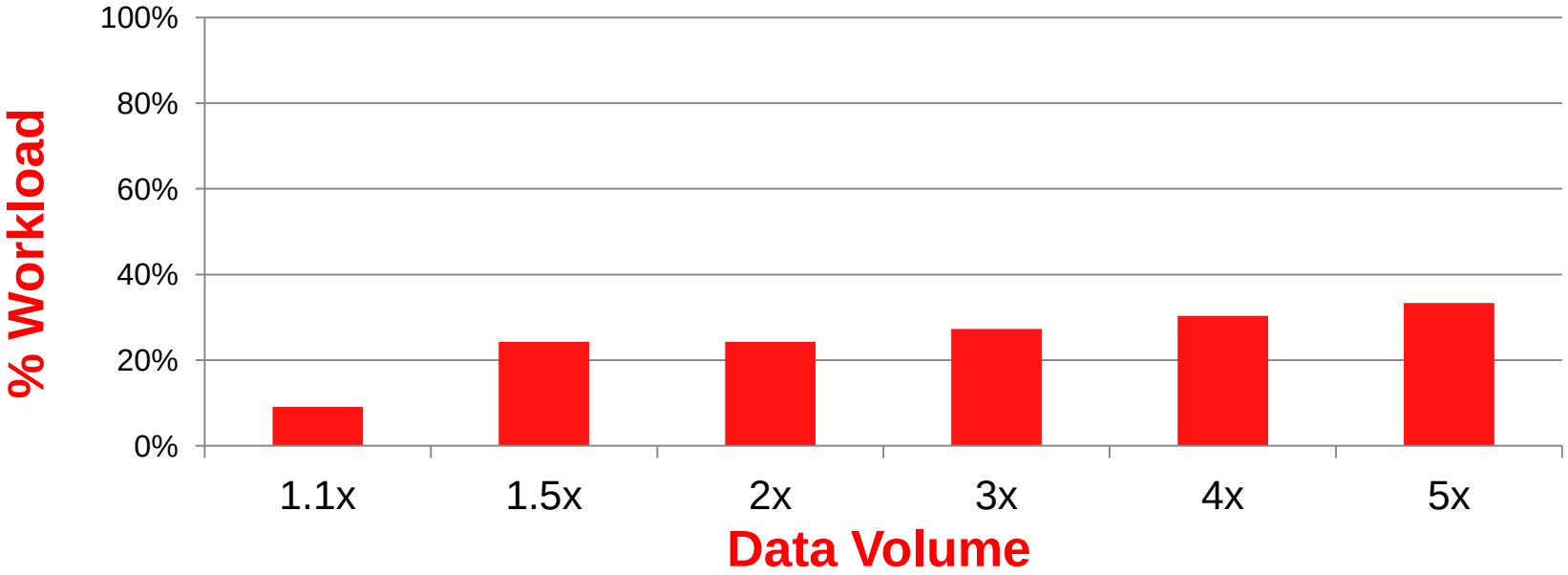
Predict Data Growth Effect on SQL Performance

- Using TPCH, captured ***33*** SQL and ran SPA Analysis on increasing data volumes
- Focused on queries with plan changes
- Unchanged performance with new plans also requires testing to minimize risk
- Regressed SQL represents only 11% of total workload (worst case scenario at 1.5X)
- Significant Cost changes past 1.5X, investigate partitioning or additional access structures at this point (SQL Access Advisor)



Data Growth (x)	1.1X	1.5X	2X	3X	4X	5X
Cost Regression %	-3	-59	-146	-436	-1480	-3679

Plan Sensitivity Distribution



SPA with Growing Data Volume

Summary

SPA can be used to identify high risk SQL with minimal effort in environments with

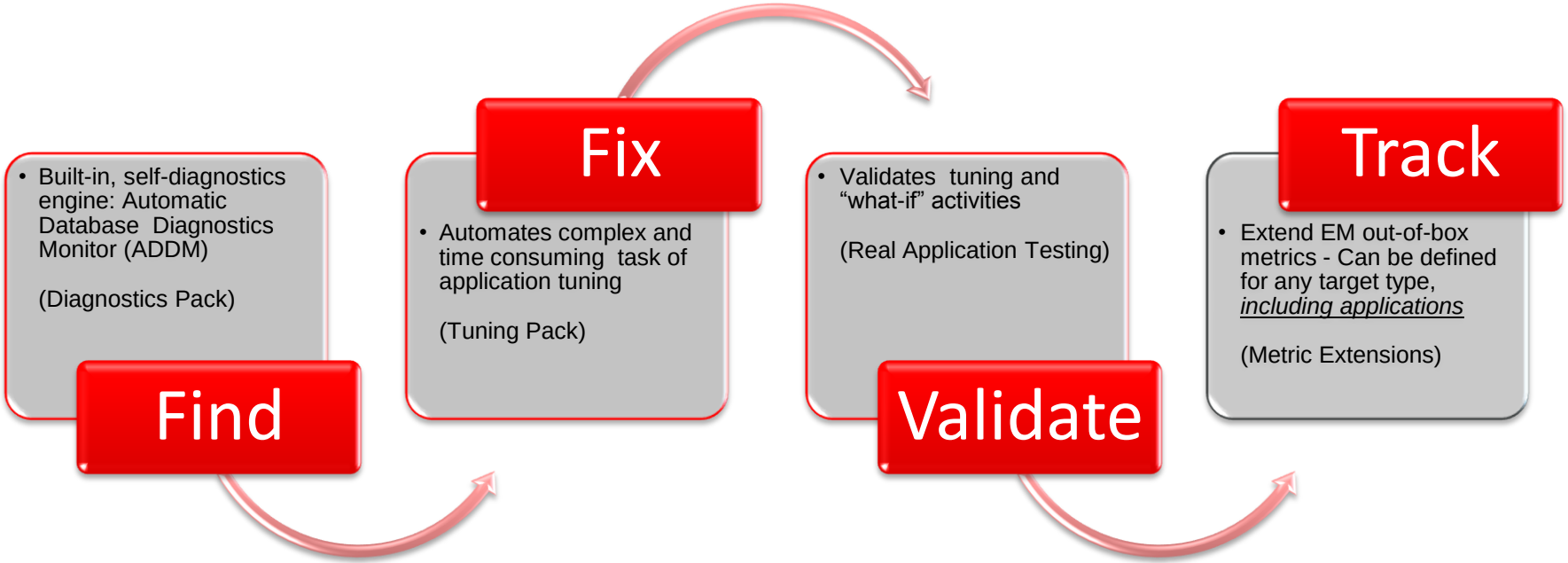
- Large workloads and increasing data volumes

Prevent unexpected performance degradation proactively

- Test high risk SQL in production and tune

Advanced Databases Performance Analysis

Solution



Safe Harbor Statements

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.

Hardware and Software
Engineered to Work Together

ORACLE


ORACLE®

Plan Sensitivity to Data Volume Growth

- *Plan Sensitivity Index (PSI) = Optimizer Cost Change % / data volume growth factor
- Linear optimizer cost increase observed with data volume scale increase up until 2X
- When data volume is double the original size, significant regression can be predicted
- For OLTP workload, increasing PSI index signals regressing application performance
- Use SQL Tuning Advisor and/or Plan Baseline to remediate regression(s)