An increasing volume of data often leads to performance degradation in most of the enterprise applications. The technology infrastructure not only has to meet the current performance requirements but also continue to scale as the business demand grows.

Caching is one of the most effective mechanisms to improve the performance of the applications. By definition, cache is a local memory area to provide faster data access as compared to external storage devices viz. Database, File servers etc. It can be used to store both static data (metadata) as well as dynamic data (transactional data). In spite of the benefits it provides, caching carries the challenge of source data synch up to minimize the probability of showing the stale data.

In order to realize the full benefits of caching, equal or more importance should be given to appropriate invalidation strategies. Conventional cache invalidation strategies like polling and time based (described below) are database independent, making them high probable candidates for showing the stale data. The newly introduced push notifications makes invalidation mechanism database dependent i.e. the cache will be invalidated / updated only when the database is changed thereby reduces the probability of showing the stale data.

This article demonstrates usage of DBMS_CHANGE_NOTIFICATION package, a push based notification mechanism provided by Oracle database. It can be used in any Java application using Oracle 10g or above as a data source.

There can be many use cases where DBMS_CHANGE_NOTIFICATION package can be leveraged but, this article describes its usage for server side cache invalidation.
**Target Audience**
This article will help IT architects and design teams working on JEE applications using Oracle database (10g or above) to come up with near real time cache invalidation mechanism at the server side. Other databases like SQL server also provide similar push notification feature.

**Cache Invalidation**
Cache Invalidation is the process of removing stale data from the cache. There are primarily three ways of invalidating the cache entries namely Poll based, Push based and Time based.

- **Poll based invalidation**
  In this approach a separate application thread polls the data source on a periodic basis to find out any update in the data. In case of any data change, the cache is updated with the latest data.

- **Push based invalidation**
  In this approach the data source itself sends notifications to the registered clients about the data change. Henceforth, the clients can either invalidate the cache or update it with the latest data.

- **Time based invalidation**
  This approach is useful when the cache entries cannot be invalidated through any other mechanism or they should be refreshed after a stipulated period of time. This mechanism works well for caching static contents like images, headers and footers of the web pages.

The table below compares the two most frequently used cache invalidation techniques i.e. (Poll vs. Push based)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Push based</th>
<th>Poll Based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Network Performance</strong></td>
<td>The server notifies the client only when data changes, hence network bandwidth is efficiently used.</td>
<td>The client keeps polling the data source leading to high network usage.</td>
</tr>
<tr>
<td><strong>Data Synchronization</strong></td>
<td>Near real time as the Database server notifies the registered client listeners instantly.</td>
<td>Can be delayed as it depends on the frequency of the client polling.</td>
</tr>
</tbody>
</table>

In this article we have demonstrated how to use Oracle Change Notification to implement Push based cache invalidation. The steps described below holds good for any Java application that uses Oracle 10g or above as the data source and uses mid tier data caching mechanism.
Technical Details

The next few sections describe the software pre-requisites, Oracle Change Notification workflow and implementation steps.

Pre-requisites

Listed below are the software pre-requisites to implement Oracle Change Notification.

<table>
<thead>
<tr>
<th>#</th>
<th>Requirement</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Database</td>
<td>Oracle 10g or above</td>
</tr>
<tr>
<td>2</td>
<td>Database User Privilege</td>
<td>CHANGE_NOTIFICATION system privilege</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SELECT privilege on tables that are to be monitored</td>
</tr>
<tr>
<td>3</td>
<td>JDBC driver jars</td>
<td>ojdbc5.jar or above should be present in the application server lib</td>
</tr>
<tr>
<td>4</td>
<td>Cache framework</td>
<td>Any mid tier caching framework like EhCache, DynaCache Coherence etc.</td>
</tr>
</tbody>
</table>

Oracle Change Notification Workflow

The first step in implementing Oracle Change Notification is to register the client listeners.

There are 2 ways to register the listeners depending on the context. (Java vs PL/SQL)

1. JDBC style of registration

In this style, client listeners are registered using JDBC driver. A new thread is spawned by the JDBC driver that listens to the notifications sent from the database server and notifies all the registered client listeners.

2. PL/SQL style of registration

In this style, listeners are actually the PL/SQL procedures which receive the notifications. They can be considered for conventional database triggers which are widely used in the applications.
Figure 1 illustrates the steps involved in Oracle Change Notification workflow.

Following steps are involved in the entire workflow:

1. Create and register listener through JDBC registration style.
2. Associate a query with the registration, which is done through select queries, i.e. if the select query is `select * from employee`, employee table is registered for change notification.
   
   **Note**: Oracle 11g provides row level granular control e.g. if the registration query is `select * from employee where employee_id=10` then notifications will be sent only when the row corresponding to employee id=10 changes.

3. The Oracle database updates data dictionary with the registration information. Following views can be queried to find relevant information.
4. DML operations (INSERT/UPDATE/DELETE) performed on registered tables as part of transaction. Notifications are sent only for committed transactions.

5. Oracle database adds a message to its internal queue with the description of the change.

6. Oracle's internal JOBQ process is notified about the changed row.

7. The JOBQ process then notifies the JDBC driver about the changed row's ROW ID, which in turn notifies all the registered listener handlers.

8. Based on the ROW ID received, handler queries the database to fetch the updated information.

9. After fetching the updated data, handler invalidates the cache and updates the cached data.

Implementing Oracle Change Notification

The below sections describes the key design considerations which can be followed while implementing Oracle change notification using JDBC style. These considerations are generic in nature and can be adapted with little or no modifications.

Design Considerations

Following considerations can be adopted during the design phase.

<table>
<thead>
<tr>
<th>#</th>
<th>Consideration</th>
<th>Benefits Realized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XML based listener configuration</td>
<td>Provides configurable approach for adding listeners, increases flexibility &amp; extensibility</td>
</tr>
<tr>
<td>2</td>
<td>Unregistering listeners during server startup</td>
<td>Prevents duplicate registration of the listeners.</td>
</tr>
</tbody>
</table>
Implementation Details

Following steps are involved in implementing Oracle Change Notification using JDBC style.

1. Listener Configuration
2. Deregister Listener
3. Register Listener
4. Associating query with Registration
5. Handling change notification

Steps 1 and 2 can be implemented in different form and order depending on each application's need.

**Step 1:**

**Listener Configuration**

This step involves configuration of listeners in the form of an XML file and the below table summarizes the corresponding properties. Listener’s registration must be a onetime activity and should be done during server startup in a servletcontextlistener class.

<table>
<thead>
<tr>
<th>Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCListeners</td>
<td>Root Element which holds all listeners info</td>
</tr>
<tr>
<td>Listener</td>
<td>Each listener object</td>
</tr>
<tr>
<td>TableName</td>
<td>Table for which notification is needed</td>
</tr>
<tr>
<td>Select Query</td>
<td>Query which is added as part of registration</td>
</tr>
<tr>
<td>ListenerClass</td>
<td>Listener class which is notified</td>
</tr>
</tbody>
</table>

Sample listener configuration file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<DCListeners>
  <Listener>
    <TableName>
      EMPLOYEE
    </TableName>
    <SelectQuery>
      select * from EMPLOYEE where rownum=1
    </SelectQuery>
    <ListenerClass>
      com.infy.dbmsChangeNotificationListener.DCNEmployeeListener
    </ListenerClass>
  </Listener>
</DCListeners>
```

Note: For receiving notification from multiple tables we need to register multiple listeners
Since listeners are registered during server startup (either through patch or planned outage) there is a possibility of them getting registered multiple times for the same server in the change notification table which will eventually lead to:

- Multiple notifications to the same listener which is incorrect.
- Maximum number of allowed registrations error.

To forbid the above, it is better to deregister the listeners or validate if the listener is already registered for server + table combination and if not, then only register the listener.

Note: In a clustered environment with multiple servers we need to query for the appropriate REGID by providing the IP address of the registered server. Failing to do so will result in de-registration of all the listeners on that table, even those which were registered through other servers.

Sample illustrative code for deregistration

```java
StringBuilder unregister = new StringBuilder();
unregister.append("SELECT REGID, CALLBACK FROM USER_CHANGE_NOTIFICATION_REGS " + "
WHERE TABLE_NAME IN (";
unregister.append(");
Statement unregisterStmt = conn.createStatement();
ResultSet unregisterRs = unregisterStmt.executeQuery(unregister.toString());
while (unregisterRs.next()) {
    long regid = unregisterRs.getLong(1);
    String callback = unregisterRs.getString(2);
    try {
        ((OracleConnection)conn).unregisterDatabaseChangeNotification(
            regid, callback);
    } catch (SQLException e)
    }
}
```
Registration of Listeners is mainly a onetime activity and should be done during the server startup and as mentioned above servletcontextlistener class is the ideal place for it. It is mainly a two step process.

1. **Setting registration properties** - OracleConnection object is used to set the type of notifications which the client is interested in. Some of the commonly used options are

<table>
<thead>
<tr>
<th>#</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DCN_NOTIFY_ROWIDS</td>
<td>Database change events will include row-level details, such as operation type and ROWID.</td>
</tr>
<tr>
<td>2</td>
<td>DCN_NOTIFY_CHANGE_LAG</td>
<td>Specifies the number of transactions by which the client is willing to lag behind.</td>
</tr>
<tr>
<td>3</td>
<td>DCN_IGNORE_DELETE_OP</td>
<td>If set to true, DELETE operations will not generate any database change event.</td>
</tr>
<tr>
<td>4</td>
<td>DCN_IGNORE_INSERT_OP</td>
<td>If set to true, INSERT operations will not generate any database change event.</td>
</tr>
<tr>
<td>5</td>
<td>DCN_IGNORE_UPDATE_OP</td>
<td>If set to true, UPDATE operations will not generate any database change event.</td>
</tr>
</tbody>
</table>

These registration properties are then passed to registerDatabaseChangeNotification method.

2. **Attaching the Listener** - After setting the notification properties the corresponding listener classes are attached to the DatabaseChangeRegistration class.

Sample illustrative code of the entire registration process

```java
/////Registering the type of notification to be received /////////////////
Properties prop = new Properties ();
prop.setProperty (OracleConnection.DCN_NOTIFY_ROWIDS, "true");
DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotification (prop);
conn.registerDatabaseChangeNotification (prop);
/////Adding the listener classes //////////////////////////////////////////////////////////////////
try {
   // add the listener:
   Class listenerClass = Class.forName("DCNEmployeeListener");
   DatabaseChangeListener
   listener = (DatabaseChangeListener)
   listenerClass.getConstructor(DCNEmployeeListener.class).newInstance(this);
   dcr.addListener(listener);
} catch (SQLException ex) {
   if(conn != null)
   conn.unregisterDatabaseChangeNotification(dcr);
   throw ex;
}
```

Note: The connection object has to be type casted to OracleConnection Object. Also, in case of exception during registration we need to close the registration in order to interrupt the thread and deregister the listener otherwise it will hanging around.- change Active to passive
Step 4:

Associating a query with registration

The next important step is to define a mapping between the registered listener class and the underlying database table on which the notifications are required. Table registration is done via SELECT clause only. The registration process extracts the table name from the query and associates it with the listener class.

To associate a query with registration setDatabaseChangeRegistration method is defined in the OracleStatement class. This method takes a DatabaseChangeRegistration object as parameter from previous step. The below code snippet illustrates how to associate a query with a registration.

```java
try {
    Statement stmt = conn.createStatement();
    ((OracleStatement) stmt).setDatabaseChangeRegistration(dcr);
    ResultSet rs = stmt.executeQuery("select * from EMPLOYEE where rownum=1");
    while (rs.next()) {}
} catch (SQLException ex) {
}
```

Step 5:

Handling change notification & cache invalidation on the server side

Whenever a database change event occurs on the registered tables, Oracle server notifies the JDBC driver taking into account the notification type specified during registration. For e.g. If DCN_NOTIFY_ROWID option was specified then listener is notified with granular row level details. The listener code can then use the ROW ID to fetch the data from database and invalidate the cache. The listener class must implement DatabaseChangeListener interface and override onDatabaseChangeNotification method.

Sample illustrative code of listener class

```java
public class DCNEmployeeListener extends DatabaseChangeListener {

public void onDatabaseChangeNotification(DatabaseChangeEvent e) {
    try{
        int len = e.getTableChangeDescription()[0].getRowChangeDescription().length;
        for (int i = 0;i<len; i++) {
            String query = "select * from EMPLOYEE WHERE rowid=?";
            pStmt = conn.prepareStatement(query);
            pStmt.setString(1,e.getTableChangeDescription()[0].getRowChangeDescription()[i].getRowid().toString());
            rs=pStmt.executeQuery(query);
            while(rs.next())
            {// Invalidate cache with the updated data
                catch(Exception ex)
                {
                }
            }
        }
    }
}
```
Caching Topology
This section describes the commonly used caching topologies in various deployments in accordance with the IT policies of the company.

Standalone Cache
This topology is used when cache replication service is not /cannot be enabled at the cluster level. In this topology, each of the cluster members holds their private copy of the data which is not shared among other cluster members. Listeners are registered for each of the clustered instances and Oracle server sends notifications to all the listeners which then invalidate or update their local cache accordingly. This caching methodology is not clustered fault tolerant.

Figure 2 illustrates standalone caching topology where each app server holds its own local copy of cache which is affected by the notifications send by the Oracle server.

Fig 2. Standalone Caching topology
**Replicated Cache**

A replicated cache is a clustered fault tolerant cache, where data is fully replicated to every member in the cluster. In this topology when Oracle server sends push notifications to one cache node, the data is replicated across all the nodes in the cluster. The caching frameworks like EhCache, Coherence, Dynacache etc, handles data replication service out of the box which need to be properly configured.

**Figure 3** illustrates replicated caching in which Oracle server sends push notifications to the registered listener of application server 1 and data is replicated across all the clustered members.

![Fig 3. Replicated Caching topology](image-url)
Distributed Cache

A distributed cache is a clustered, fault-tolerant cache. In this topology data is distributed across all the nodes in such a way that no two node is responsible for the same piece of cached data. One node is assigned to be the primary node and another is assigned as a backup node, this is done for failover purpose. The caching frameworks like EhCache, Coherence etc. handles the data distribution service out of the box that needs to be configured accordingly.

Figure 3 illustrates distributed caching. In this scenario, Oracle DB sends push notification to the registered listener on App Server 1, accordingly caching framework distributes the data across cache node 1 and cache node 3, assigning node 1 as primary and node 3 as the backup node.

![Distributed Caching topology](image-url)
Benefits realized by push based notifications
Some of the quantitative and qualitative benefits which can be realized by using push based notifications are summarized below.

Quantitative benefits realized
1. **Eliminate** periodic database hits to find the latest updates.
2. **No** thread spawning to check the database status.
3. **Reduced** connection pool usage for database polling.

Qualitative benefits realized
1. **Increased** code *configurability* to register for push notifications.
2. **Near real time data synch** up between data source & the cached data.
3. **Implicit** asynchronous *thread* management by JDBC driver.
Conclusion
In this article we have demonstrated one of the use cases of Oracle change notification framework for cache invalidation. We compared push based and poll based cache invalidation strategies and described how Oracle change notification can be leveraged for push based cache approach. We covered some of the design considerations for implementing Oracle change notification based on our experience and how the application design teams can leverage some or all of them in addition to their own based on the application needs. We also covered some of the quantitative and qualitative benefits realized by using push notifications.
References
1. http://docs.oracle.com/cd/E14072_01/java.112/e10589/dbchgnf.htm

About the Author
Ravi Shankar Anupindi
is a Senior Technical Architect with Manufacturing unit, Infosys. He has around 12 years of IT experience. He has been the active member of many CoEs and his main areas of interest include performance engineering and Cloud computing. He has been involved in data migration activities in the recent past. He has been actively participating in Infosys internal forums as well as external forums like IBM Developer works based on the live experience.

Debasis Mitra
is a Technology Architect with Manufacturing unit, Infosys. He has around 8 years of IT experience. He has architected multiple J2EE applications using latest frameworks with Oracle as data source. He has been active member of multiple CoEs and his areas of interest include Cloud Computing and Enterprise Portals.