Eine Alternative zu Database Change Notification bei Massendaten-Änderungen

Michael Griesser und Thies Rubarth, Freiberufler, sowie Nis Nagel und Andriy Terletskyy, Berenberg Bank

Dieser Artikel zeigt eine Alternative zu Oracles „Database Change Notification" beziehungsweise „Continuous Query Notification" im Bereich der Massendaten-Änderung, wobei die multilingualen User-Clients mit unterschiedlichen territorialen Einstellungen in einem festgelegten Intervall über DML-Ereignisse aus der Datenbank informiert werden.

Bei Berenberg gibt es für viele Anwendungen die Anforderung, Datensatz-Änderungen (DML-Operationen) automatisch in der GUI aller aktiven Anwender anzuzeigen. Ein ständiges Neu-Abfragen der Daten durch die GUI würde jedoch die gesamte Applikation unnötig belasten und eine viel zu große Datenmenge mehrfach bearbeiten. Stattdessen soll die Datenbank proaktiv in den Aktualisierungsprozess für die GUI einbezogen und Daten-Spam aus der Quelle vermieden werden. Bisher gab es dafür eine Trigger-basierte Lösung, die mit zunehmendem Datenvolumen und bei hochfrequenten Änderungen an ihre Grenzen stieß. Ihre wesentlichen Nachteile lagen darin, dass jede einzelne DML-Änderung an die Clients gesendet wurde und dass die Ausführung der Trigger die Performance der Businessprozesse in der Datenbank negativ beeinflusste.

Anforderungen, Analyse und Entscheidung

Heutzutage ist es besonders beim Wertpapierhandel üblich, dass ein Order-Datensatz innerhalb eines kurzen Zeitabschnitts mehrfach vom System geändert wird. Auch ist der Durchsatz an Order-Einstellungen in einem Handelssystem sehr hoch. Grund dafür ist eine hohe Anzahl maschinell gehandelter Orders an den Börsen (algorithmisches Trading).

Die neue Lösung soll deshalb in einem zeitlich festgelegten Intervall einen Snapshot der geänderten Datensätze an den Client senden und in unabhängigen, von der Datenverarbeitung getrennten Prozessen laufen (siehe Abbildung 1). Eine weitere Anforderung an das System ist die unterschiedliche Aufbereitung der Nachrichten, abhängig von den Sprach- und Ländereinstellungen der angemeldeten Benutzer. Mit diesen Anforderungen und der Tatsache, dass Oracle in ihrer Spezifikation angibt, dass das Datenbank-Feature „Database Change Notification" nur bei geringen Änderungen effizient ist und ansonsten die gesamte Query aktualisiert wird, fiel die Entscheidung zugunsten einer Eigen-Entwicklung.

Die Architektur

Abbildung 2 zeigt die System-Architektur der Client Notification für Datensatz-Änderungen. Das System kann in fünf Komponenten unterteilt werden:

- Tabellen mit Row-Dependencies
- Consumer-Verwaltung mit Queue-Überwachung
- SCN-Capture-Jobs
- Message Enqueue
- Nachrichten-Empfang auf GUI-Ebene

Es muss zunächst definiert werden, für welche Tabellen die Client Notification umzusetzen ist. Die Tabelle ist dann mit „ROWDEPENDENCIES" anzulegen oder mit „DBMS_REDEFINITION" zu ändern. Damit speichert die Oracle-Datenbank für jeden Datensatz eine eigene „ROW_SCN" (System Change Number). Diese ist eine Pseudo-Column des Datensatzes/Blocks

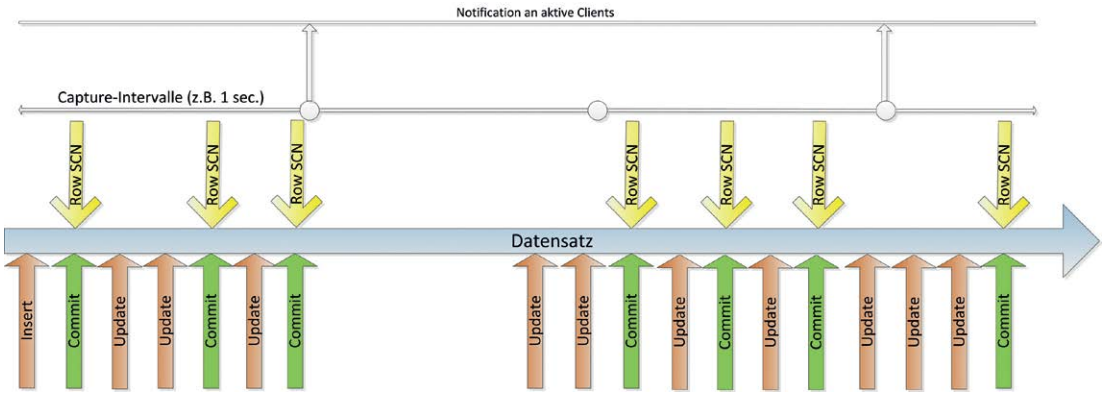

Abbildung 1: Row-SCN-Snapshots bei Massendaten-Änderungen

und beschreibt eine Art fortlaufenden „Transaktionszähler“.

Der GUI-Client (Consumer) meldet sich an das Notification-System über ein selbstgeschriebenes PL/SQL-API in der Datenbank an. Zusätzlich registriert sich jede Masken-Instanz im System als sogenannter „Client-Listener“. Die Consumer-Verwaltung stellt ein Mapping zwischen den Masken-Definitionen und den entsprechenden Business-Tabellen/Objekten her und definiert damit, welche Datenbank-Tabellen aktiv von den SCN-Capture-Jobs erfasst werden müssen. Ein zusätzlicher Job in der Consumer-Verwaltung übernimmt Monitoring-Aufgaben. Er erkennt inaktive Consumer (ausgesetzte Heartbeats) und meldet diese ab. Ein weiterer Job überwacht die Queues und entfernt Expired Messages, die in die Exception-Queue verschoben wurden.

Über Scheduler-Jobs werden die in das System konfigurierten Tabellen mithilfe der Pseudo-Spalte „ORA_ROWSCN“ auf DML-Ereignisse überwacht. Sobald sich Client-Listener für Änderungen interessieren, wird in dem jeweilig eingestellten Zeitintervall (abhängig vom Business-Umfeld) die Ergebnismenge der jüngsten DML-Änderung („ORA_ROWSCN BETWEEN <Letzter SCN> AND DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER()“) gesammelt (siehe Listing 1).

Aus der Liste wird je eine Text-Message erzeugt, gruppiert nach den jeweiligen Sprach- und sonstigen territorialen Einstellungen der User, und an die entsprechende Recipient-Liste gesendet (Siehe Listing 2).

Als Schnittstelle zwischen Datenbank und GUI dient eine Multi-Subscriber-Queue mit einem JMS-Payload-Type (siehe Listing 3). Je nach Konfiguration können diese Messages „buffered“ oder „persistent“ verschickt werden, wobei hier auch im laufenden Betrieb eingegriffen und dieses ad hoc umkonfiguriert werden kann. Die Message wird im JSON-Format erzeugt (siehe Listing 4).

Um bei „buffered Messages“ das Problem einer zu hohen Enqueue-Rate zu umgehen, muss auf die Exception „ORA-25307 Enqueue-Rate zu hoch, Fluss-Steuerung aktiviert“ reagiert werden. Hier schaltet das System im laufen-

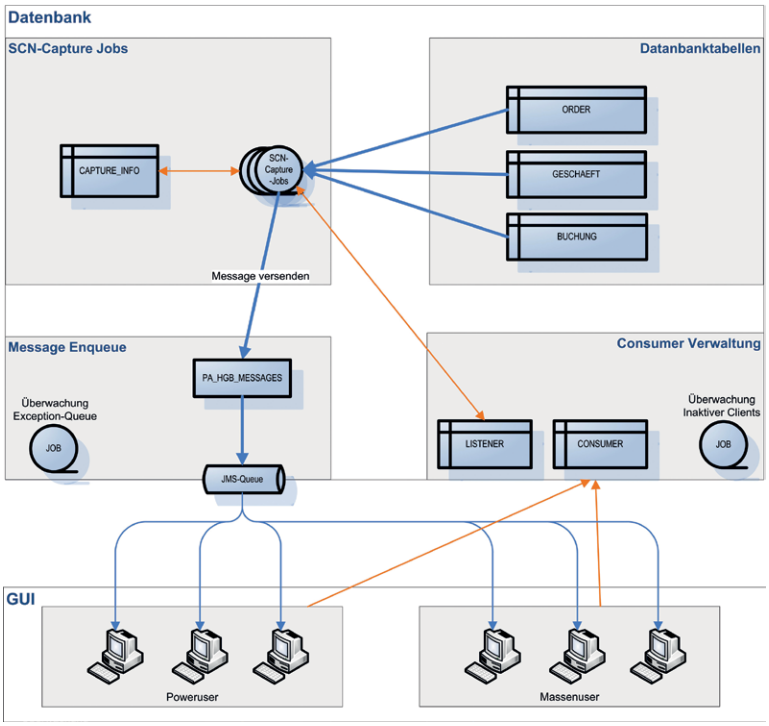

Abbildung 2: Architektur der Client Notification

```
CREATE TYPE TABLE_NUMBER AS TABLE OF NUMBER;

CREATE OR REPLACE PROCEDURE PR_CAPTURE
IS
  scn_start      NUMBER;
  scn_end        NUMBER;
  tab_order      TABLE_NUMBER := TABLE_NUMBER();
BEGIN

  -- Letzte SCN auslesen
  SELECT i.scn_last
  INTO scn_start
  FROM CAPTURE_INFO i
  WHERE i.TABLE_NAME = 'ORDER';

  scn_end   := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER();

  -- prüfen ob es angemeldete listener gibt
  IF PA_HGB_MESSAGES.LISTENER_EXISTS(IN_TABLE_NAME => 'ORDER') THEN
    -- capturen der tabelle
    SELECT o.ORDER_ID
       BULK COLLECT INTO tab_order
      FROM ORDER o
      WHERE o.ora_rowscn BETWEEN scn_start+1 AND scn_end;

    IF tab_order.COUNT() > 0 THEN
      PA_HGB_MESSAGES.VERSENDEN(
                IN_TABLE_IDS => tab_order
, IN_CLIENT_TABLE_NAME => 'ORDER');
    END IF;

  END;

  -- Letze SCN merken
  UPDATE CAPTURE_INFO i
  SET i.scn_last = scn_last
  WHERE i.TABLE_NAME = 'ORDER';
  COMMIT;

END;
```

Listing 1

den Betrieb automatisch auf „persistent Messages“ um und wechselt anschließend wieder auf „buffered Messages“.

Implementierung in Java

Um aus Java heraus auf Oracle Advanced Queuing zugreifen zu können, bietet Oracle zwei unterschiedliche APIs an. Zum einen ein proprietäres API, das direkt mit den Queuing-Funktionen arbeitet, und zum andern ein JMS-API, über das zum größten Teil mit Standard-Java-Methoden auf die Queues zugegriffen werden kann.

Da die Oracle-Datenbank keinen JNDI-Context anbietet, muss das JMS-Topic über Oracle-Methoden erstellt werden. Zusätzlich wird das Oracle-spezifische Konstrukt „TopicReceiver“ verwendet, das benötigt wird, um auf die „Multi-Subscriber-Queues“ zuzugreifen, da JMS ein solches Konzept nicht vorsieht.

In Abbildung 3 sind die in der Java-Anwendung notwendigen Schritte zum Empfangen von JMS-Nachrichten dargestellt. Die Schritte 1 plus 2 und 5 können über einen Anwendungsserver laufen, sodass die DB-Verbindung in den Clients auf das Empfangen von JMS-Nachrichten beschränkt werden kann.

Die Klasse aus Listing 5 setzt voraus, dass die Anmeldung bereits erfolgt ist und ein Listener für eine Tabelle angemeldet wurde. Zunächst muss eine Topic-Connection zur Datenbank aufgebaut werden. Mit dem Namen des Consumers kann dann auf das Topic zugegriffen werden.

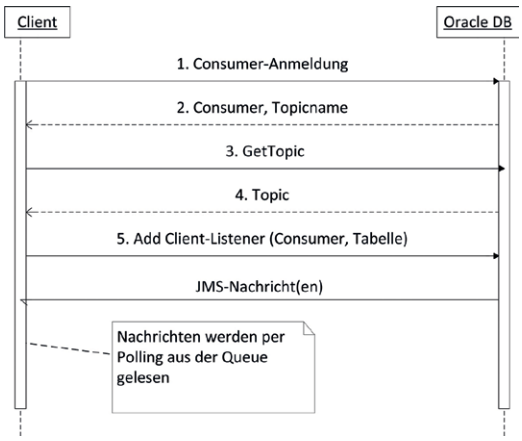

Abbildung 3: Sequenz-Diagramm für das Einrichten des Nachrichten-Empfangs

```
  t_recipients  SYS.DBMS_AQ.AQ$_RECIPIENT_LIST_T;
  ...
  SELECT sys.AQ$_AGENT(c.NAME, NULL, NULL)
  BULK COLLECT INTO t_recipients
  FROM CONSUMER c
WHERE ...;
```

Listing 2

```
BEGIN
  -- QueueTable anlegen
  SYS.DBMS_AQADM.CREATE_QUEUE_TABLE(
      QUEUE_TABLE        => 'AQ_JMS_HGB_TAB'
     ,QUEUE_PAYLOAD_TYPE => 'SYS.AQ$_JMS_TEXT_MESSAGE'
     ,COMMENT            => 'QueueTable für HGBs'
     ,MULTIPLE_CONSUMERS => TRUE
     ,SORT_LIST          => 'priority,enq_time'
                                );
  -- Queue anlegen
  SYS.DBMS_AQADM.CREATE_QUEUE(QUEUE_NAME  => 'AQ_JMS_HGB_WPS'
                             ,COMMENT     => 'Queue für HGBs'
                             ,QUEUE_TABLE => 'AQ_JMS_HGB_TAB'
                             ,QUEUE_TYPE  => SYS.DBMS_AQADM.NORMAL_QUEUE
                             ,MAX_RETRIES  => 2
                             );
  -- Queue starten
  SYS.DBMS_AQADM.START_QUEUE(QUEUE_NAME => 'AQ_JMS_HGB_WPS');
END;
```

Listing 3

```
...   -- Bevorzugtes Messagingverfahren auswerten
    IF IN_BUFFERED_PERSISTANT = ,BUFFERED'
    THEN
       -- Prüfung der Länge
       -- Wenn > max VARCHAR, dann kann die Message nicht als
       -- Buffered verschickt werden
       IF IN_OBJECT.text_len <= 32767
       THEN
         enq_opt.VISIBILITY    := DBMS_AQ.IMMEDIATE;
         enq_opt.DELIVERY_MODE := DBMS_AQ.BUFFERED;
       END IF;

    END IF;
    -- Nachrichten für explizite Recipients?
    IF IN_RECIPIENTS.COUNT() <> 0
    THEN
      msg_prop.recipient_list := IN_RECIPIENTS;
    END IF;

    DBMS_AQ.Enqueue(QUEUE_NAME         => IN_QUEUE_NAME
                   ,ENQUEUE_OPTIONS    => enq_opt
                   ,MESSAGE_PROPERTIES => msg_prop
                   ,PAYLOAD            => IN_OBJECT
                   ,MSGID              => msg_handle
                   );
...
```

Listing 4

Am einfachsten können die Nachrichten empfangen werden, indem der Topic-Receiver einen Message-Listener zugewiesen bekommt, der dann für jede Nachricht aufgerufen wird. Listing 6 zeigt ein Beispiel für einen Message-Listener, der die eingehenden Nachrichten auf der Konsole ausgibt.

Intern wird der Nachrichtenempfang durch das Oracle-JMS-API mittels Polling realisiert. Um die Dequeue-Frequenz einzustellen, können in der Java-VM die Parameter „oracle.jms.minSleepTime“ und „oracle.jms.maxSleepTime“ gesetzt werden. So kann beispielsweise für bestimmte Power-User eine höhere Dequeue-Frequenz als für die große Menge an Benutzern („Massen-User“) eingestellt werden.

Leider bietet das Oracle-JMS-API diese Funktionalität nur für persistente Nachrichten. Wenn Buffered Messages verwendet werden sollen, muss das Verfahren selbst umgesetzt werden. In Listing 7 ist ein einfacher Polling-Mechanismus implementiert.

Um Buffered Messages empfangen zu können, muss die Methode „AQjmsConumer.bufferReceive“ verwendet werden, die sowohl persistente als auch nicht-persistente Nachrichten aus dem Topic lesen kann. Die Methode gibt es in unterschiedlichen Varianten, die sich im Wesentlichen dadurch unterscheiden, dass sie die Nachrichten „blockierend“ oder „nicht-blockierend“ aus der Queue lesen. Welche der Methoden zum Tragen kommt, sollte davon abhängig gemacht werden, wie viele Clients an einem Topic lauschen. Wenn „nicht-blockierend“ gelesen wird, kommt es zu Latenzzeiten, da zwischen den Lesezugriffen Pausen eingehalten werden müssen, um den Client und die Datenbank nicht zu überlasten. Das blockierende Lesen erlaubt einen schnelleren Empfang der Nachrichten, bindet jedoch aktive Sessions in der Datenbank. Um die Zahl der Sessions in der Datenbank zu minimieren, kann ein JMS-Server zwischen der Datenbank und den Clients geschaltet werden, der als Message Broker fungiert.

```
import java.io.*;
import javax.jms.*;
import oracle.jms.*;

public class TestAqTopics {

  public static void main(String[] args) throws Exception {
    String jdbcUrl = …;
    String userName = …;
    String password = …;
    String topicOwnerName = …;
    String topicName = …;
    String consumerName = …;
    MessageListener msgListener = new TestAqMessageListener();
    TestAqThread thread = null;
    boolean persistentMessagesOnly = true;

    // Topic Connection aufbauen
    TopicConnectionFactory tcf = AQjmsFactory
        .getTopicConnectionFactory(jdbcUrl, null);
    TopicConnection con = tcf.createTopicConnection(userName,
        password);
    con.start();

    // Topic abonieren
    AQjmsSession session = (AQjmsSession) con
        .createTopicSession(false, Session.CLIENT_ACKNOWLEDGE);
    Topic topic = session.getTopic(topicOwnerName, topicName);
    TopicReceiver receiver = session.createTopicReceiver(
        topic, consumerName, “”);
    if (persistentMessagesOnly) {
      // Wenn mit persistenen Nachrichten gearbeitet wird,
      // können wir den MessageListener direkt setzen
      receiver.setMessageListener(msgListener);
    } else {
      // Für nicht-persistente Nachrichten, muss das Polling
      // selbst implementiert werden
      AQjmsConsumer consumer = (AQjmsConsumer) receiver;
      thread = new TestAqThread(consumer, msgListener);
      thread.start();
    }
    String cmd = “”;
    BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));

    while (!”QUIT”.equals(cmd)) {
      cmd = in.readLine();
    }
    if (thread != null) {
      thread.interrupt();
      thread.join();
    }
    con.close();
  }
}
```

Listing 5

Fazit

Im Bereich von hochfrequenter Massendaten-Änderung sollte in Bezug auf die Business-Anforderungen überlegt werden, ob alle DML-Änderungen visualisiert werden sollen oder ob es ausreicht, in bestimmten zeitlichen Intervallen einen Snapshot an die GUI zu senden. Buffered Messages haben einen klaren Performance-Vorteil und sind ideal für Systeme, bei denen die Ausfallsicherheit nicht im Vordergrund steht. Bei deren Einsatz sind die jeweils aktuellen Restriktionen des Datenbank-Release zu bewerten. Zurzeit werden folgende Oracle-Streams-Advanced-Queuing-Features für Buffered Messages nicht unterstützt:

- Message retention
- Message delay
- Transaction grouping
- Array enqueue
- Array dequeue
- Message export and import
- Posting for subscriber notification
- Messaging Gateway

Als Nachrichtenformat haben sich die Autoren für JSON entschieden, da die-

```
import javax.jms.*;

public class TestAqMessageListener implements MessageListener {
  @Override
  public void onMessage(Message message) {
    try {
      TextMessage tm = (TextMessage) message;
      String text = tm.getText();
      System.out.println("Message received: " + text);
    } catch (JMSException e) {
      e.printStackTrace();
    }
  }
}
```

Listing 6

```
import javax.jms.*;
import oracle.jms.*;

public class TestAqThread extends Thread{

  private static long WAIT_TIME = 500;
  private AQjmsConsumer receiver;
  private MessageListener listener;

  public TestAqThread(AQjmsConsumer receiver, MessageListener listener){
    this.receiver = receiver;
    this.listener = listener;
  }

  public void run(){
    while(!isInterrupted()){
      try {
        Message msg = receiver.bufferReceive(1);
        if (msg != null) {
          listener.onMessage(msg);
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
      try {
        sleep(WAIT_TIME);
      } catch (InterruptedException e) {
        break;
      }
    }
  }
}
```

Listing 7

ses Format gegenüber XML beim Erstellen und Parsen schneller und die Nachrichtengröße deutlich kompakter ist. Als Payload-Type bietet JMS Vorteile bei der Performance und der Flexibilität bezüglich Nachrichtengröße und Zusatzattributen im Vergleich zu XML und RAW.

Ein Nachteil der hier vorgestellten Implementierung ist noch, dass bei sehr großen Tabellen und der Tatsache, dass eine Indizierung der „ORA_ROWSCN“ in der aktuellen Datenbank-Version nicht möglich ist, der Snapshot etwas länger dauert. Ein Enhancement-Request wurde jedoch schon von Oracle angenommen.

Das JMS-API von Oracle wirkt an einigen Stellen noch nicht ausgereift. Es fehlt die Möglichkeit, Buffered Messages mit einem Message-Listener zu empfangen und beim nicht-blockierenden Lesen von Buffered Messages kommt es dazu, dass Nachrichten verloren gehen. Als Workaround kann blockierend gelesen werden, mit einem Timeout von einer Millisekunde.

Andriy Terletskyy
andriy.terletskyy@berenberg.de

Nis Nagel
nis.nagel@berenberg.de

Michael Griesser
michael.griesser.ext@berenberg.de

Thies Rubarth
thies.rubarth.ext@berenberg.de