**package** com.rbc.newton.mask.service;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Types;

**import** java.util.ArrayList;

**import** java.util.Date;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.concurrent.ArrayBlockingQueue;

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.Semaphore;

**import** java.util.concurrent.ThreadPoolExecutor;

**import** java.util.concurrent.TimeUnit;

**import** java.util.concurrent.atomic.AtomicInteger;

**import** javax.inject.Inject;

**import** org.apache.commons.lang.StringUtils;

**import** org.apache.log4j.Logger;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** org.springframework.jdbc.core.BatchPreparedStatementSetter;

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.jdbc.core.RowMapper;

**import** org.springframework.stereotype.Service;

**import** org.springframework.transaction.annotation.Transactional;

**import** com.rbc.newton.mask.dao.DmRequestDataDao;

**import** com.rbc.newton.mask.dao.MappingsDao;

**import** com.rbc.newton.mask.domain.Constants;

**import** com.rbc.newton.mask.domain.DmRequestData;

**import** com.rbc.newton.mask.domain.Document;

**import** com.rbc.newton.mask.domain.IdElemValue;

**import** com.rbc.newton.mask.domain.Mapping;

**import** com.rbc.newton.mask.domain.PathValue;

**import** com.rbc.newton.mask.domain.Revision;

**import** com.rbc.newton.util.SimpleGNUCommandLine;

**import** com.rbc.newton.util.Utils;

@Service

**public** **class** MaskingService {

**private** Logger logger = Logger.getLogger(MaskingService.**class**);

@Value("${core-pool-size}")

**private** **int** core\_pool\_size;

@Value("${maximum-pool-size}")

**private** **int** maximum\_pool\_size;

@Value("${keep-alive-time-milliseconds}")

**private** **int** keep\_alive\_time\_milliseconds;

@Value("${queue-size}")

**private** **int** queue\_size;

@Inject

**private** JdbcTemplate jdbcTemplate;

@Inject

**private** DmRequestDataDao dmRequestDataDao;

@Inject

**private** MappingsDao mappingsDao;

**private** **static** **final** String REINDEX\_SQL\_STMT = "insert into DM\_INDEX\_JOURNAL values ( -1,'MODIFIED',? )";

**private** **static** **final** String AUDIT = "insert into AUDIT\_MASKING values ( ?, ? ,?) ";

**private** **static** **final** String UPDATE\_DM\_REQUEST\_DATA="update DM\_REQUEST\_DATA set ELEMENT\_VALUE=? where RULE\_EXECUTION\_ID=? and ELEMENT\_PATH=?";

**private** **static** **final** String UPDATE\_DM\_RD\_DICT="update DM\_RD\_DICT set ELEMENT\_VALUE=? where DM\_RD\_DICT\_ID=? and ELEMENT\_PATH=?";

**private** **int** update(PathValue pv, List<Long> ids) {

StringBuilder sb=**new** StringBuilder();

**for**(Long id:ids){

sb.append(sb.length()==0?" (":",").append(id);

}

sb.append(") ");

String in\_str=sb.toString();

sb.setLength(0);

sb.append("update ").append(pv.getSource()==1?"DM\_REQUEST\_DATA":"DM\_RD\_DICT");

sb.append("\nset ELEMENT\_VALUE=").append(pv.getValue()==**null**?"null":"'"+pv.getValue()+"'");

sb.append("\nwhere ").append(pv.getSource()==1?"RULE\_EXECUTION\_ID in ":"DM\_RD\_DICT\_ID in ").append(in\_str);

sb.append("\nand ELEMENT\_PATH='").append(pv.getPath()).append("'");

//System.out.println(sb.toString());

**return** jdbcTemplate.update(sb.toString());

}

// private void addValue(DmRequestData rd, String value,Map<PathValue,List<Long>> map){

// if ( StringUtils.equals(rd.getElementValue(), value)){

// return;

// }

// PathValue pv=new PathValue(rd.getElementPath(),value,rd.getSource());

// List<Long> list=map.get(pv);

// if ( list==null ){

// list=new ArrayList<Long>();

// map.put(pv, list);

// }

// list.add(rd.getSource()==1?rd.getRuleExecutionId():rd.getRequestDataId());

// }

// private void processPath(Revision rev, List<Mapping> listMappings,Map<PathValue,List<Long>> map) {

// boolean found=false;

// for (Mapping mapping : listMappings) {

// String prefix = mapping.getPrefix() + " ";

// int ix = mapping.getPath().indexOf("[n]");

// if (ix == -1) {

// if ( ! found ){

// DmRequestData rd = rev.getElements().get(mapping.getPath());

// if (rd != null) {

// if ("null".equals(mapping.getValue())) {

// this.addValue(rd, null, map);

// found=true;

// } else {

// DmRequestData rdval = rev.getElements().get(mapping.getValue());

// if (rdval != null) {

// this.addValue(rd, prefix + rdval.getElementValue(), map);

// found=true;

// }

// }

// }

// }

// } else {

// for (int i = 0; i < 10; i++) {

// String path = mapping.getPath().replaceFirst("\\[n\\]", "[" + i + "]");

// DmRequestData rd = rev.getElements().get(path);

// if (rd == null) {

// continue;

// }

// if ("null".equals(mapping.getValue())) {

// this.addValue(rd, null, map);

// } else {

// DmRequestData rdval = rev.getElements()

// .get(mapping.getValue().replaceFirst("\\[n\\]", "[" + i + "]"));

// if (rdval != null) {

// this.addValue(rd, prefix + rdval.getElementValue(), map);

// }

// }

// }

// }

// }

// }

//

// private int processElements(Document doc, Map<Long, Revision> revisions, Map<String, List<Mapping>> mappings) {

// int rows\_updated = 0;

// Map<PathValue,List<Long>> map=new HashMap<PathValue,List<Long>>();

// for (Revision rev : revisions.values()) {

// for (String path : mappings.keySet()) {

// this.processPath(rev, mappings.get(path), map);

// }

// }

// for(PathValue pv:map.keySet()){

// List<Long> list=map.get(pv);

// if ( list==null || list.size() < 1 ){

// continue;

// }

// rows\_updated+=update(pv,list);

// }

// return rows\_updated;

// }

**private** **void** addValue(DmRequestData rd, String value, List<IdElemValue> dm\_list, List<IdElemValue> dict\_list) {

**if** ( StringUtils.equals(rd.getElementValue(), value)){

**return**;

}

**if** ( rd.getSource()==1 ){

dm\_list.add(**new** IdElemValue(rd.getRuleExecutionId(),rd.getElementPath(),value));

} **else** {

dict\_list.add(**new** IdElemValue(rd.getRequestDataId(),rd.getElementPath(),value));

}

}

**private** **void** processPath(Revision rev, List<Mapping> listMappings,List<IdElemValue> dm\_list, List<IdElemValue> dict\_list) {

**boolean** found=**false**;

**for** (Mapping mapping : listMappings) {

String prefix = mapping.getPrefix() + " ";

**int** ix = mapping.getPath().indexOf("[n]");

**if** (ix == -1) {

**if** ( ! found ){

DmRequestData rd = rev.getElements().get(mapping.getPath());

**if** (rd != **null**) {

**if** ("null".equals(mapping.getValue())) {

**this**.addValue(rd, **null**, dm\_list, dict\_list);

found=**true**;

} **else** {

DmRequestData rdval = rev.getElements().get(mapping.getValue());

**if** (rdval != **null**) {

**this**.addValue(rd, prefix + rdval.getElementValue(), dm\_list, dict\_list);

found=**true**;

}

}

}

}

} **else** {

**for** (**int** i = 0; i < 10; i++) {

String path = mapping.getPath().replaceFirst("\\[n\\]", "[" + i + "]");

DmRequestData rd = rev.getElements().get(path);

**if** (rd == **null**) {

**continue**;

}

**if** ("null".equals(mapping.getValue())) {

**this**.addValue(rd, **null**, dm\_list, dict\_list);

} **else** {

DmRequestData rdval = rev.getElements()

.get(mapping.getValue().replaceFirst("\\[n\\]", "[" + i + "]"));

**if** (rdval != **null**) {

**this**.addValue(rd, prefix + rdval.getElementValue(), dm\_list, dict\_list);

}

}

}

}

}

}

**private** **int** batchUpdate(**int** source, **final** List<IdElemValue> list) **throws** Exception {

**int**[] rows=jdbcTemplate.batchUpdate(source==1?UPDATE\_DM\_REQUEST\_DATA:UPDATE\_DM\_RD\_DICT, **new** BatchPreparedStatementSetter(){

@Override

**public** **int** getBatchSize() {

**return** list.size();

}

@Override

**public** **void** setValues(PreparedStatement pstmt, **int** i) **throws** SQLException {

IdElemValue iev=list.get(i);

**if** ( iev.getValue()==**null** ){

pstmt.setNull(1, Types.VARCHAR);

} **else** {

pstmt.setString(1, iev.getValue());

}

pstmt.setLong(2, iev.getId());

pstmt.setString(3, iev.getPath());

}});

**int** nbr=0;

**for**(**int** i:rows){

**if** ( i == -2 ) {

nbr ++;

}

}

**return** nbr;

}

**private** **synchronized** **int** processDictBatch(List<IdElemValue> dict\_list) **throws** Exception {

**return** **this**.batchUpdate(2, dict\_list);

}

**private** **int** processElements(Document doc, Map<Long, Revision> revisions, Map<String, List<Mapping>> mappings) **throws** Exception {

**int** rows\_updated = 0;

List<IdElemValue> dm\_list=**new** ArrayList<IdElemValue>();

List<IdElemValue> dict\_list=**new** ArrayList<IdElemValue>();

**for** (Revision rev : revisions.values()) {

**for** (String path : mappings.keySet()) {

**this**.processPath(rev, mappings.get(path), dm\_list, dict\_list);

}

}

**if**( dm\_list.size() > 0 ){

rows\_updated += **this**.batchUpdate(1, dm\_list);

}

**if**( dict\_list.size() > 0 ){

rows\_updated += **this**.processDictBatch(dict\_list);

}

**return** rows\_updated;

}

//@Transactional

**private** **int** processDocument(Document doc, Map<Long, Revision> revisions, **boolean** onlyCurrentRevision) **throws** Exception {

logger.info("Processing " + doc.getEntity() + " " + doc.getId());

**int** rows\_updated = 0;

**if** (Constants.CLIENT.equals(doc.getEntity())) {

rows\_updated += jdbcTemplate.update("update RBC\_P\_CLIENT set CLIENT\_NM=?, LEGAL\_NAME=? where ID=?",

Constants.NAME\_PREFIX + doc.getId(), Constants.LEGAL\_PREFIX + doc.getId(), doc.getId());

} **else** **if** (Constants.STMT.equals(doc.getEntity())) {

rows\_updated += jdbcTemplate.update("update FS\_STATEMENT set BUSINESS\_PARTNER\_NAME=? where STATEMENT\_ID=?",

Constants.NAME\_PREFIX + doc.getId(), doc.getId());

}

**if** (revisions.size() == 0) {

logger.info("No revisions for the document " + doc.getId());

} **else** {

**switch** (doc.getEntity()) {

**case** Constants.CLIENT:

rows\_updated += **this**.processElements(doc, revisions, mappingsDao.getClientMappings());

**break**;

**case** Constants.RATING:

rows\_updated += **this**.processElements(doc, revisions, mappingsDao.getAssessmentMappings());

**break**;

**case** Constants.STMT:

rows\_updated += **this**.processElements(doc, revisions, mappingsDao.getFinancialMappings());

**break**;

}

**if** ( onlyCurrentRevision ){

jdbcTemplate.update(REINDEX\_SQL\_STMT, doc.getId());

}

}

jdbcTemplate.update(AUDIT, doc.getId(), **new** Date(), rows\_updated);

logger.info("Updated " + rows\_updated + " row(s) in " + doc.getEntity() + " " + doc.getId());

**return** rows\_updated;

}

**private** String getSql(String entity, **boolean** onlyCurrentRevision) {

String whereClause = **null**;

**switch** (entity) {

**case** Constants.CLIENT:

whereClause = mappingsDao.getClientWhereClause();

**break**;

**case** Constants.STMT:

whereClause = mappingsDao.getStmtWhereClause();

**break**;

**case** Constants.RATING:

whereClause = mappingsDao.getRatingWhereClause();

**break**;

}

StringBuilder sb = **new** StringBuilder();

sb.append(

"select rev.REVISION\_ID, 1 as SRC,rd.REQUEST\_DATA\_ID as ID, rd.RULE\_EXECUTION\_ID, rd.ELEMENT\_NAME, rd.ELEMENT\_PATH, rd.ELEMENT\_VALUE");

sb.append("\nfrom DM\_REQUEST\_DATA rd");

sb.append("\ninner join DM\_RULE\_EXECUTION re on re.RULE\_EXECUTION\_ID=rd.RULE\_EXECUTION\_ID");

sb.append("\ninner join DM\_REVISION\_REQUEST rr on rr.REQUEST\_ID=re.REQUEST\_ID");

sb.append("\ninner join DM\_REVISION rev on rev.REVISION\_ID=rr.REVISION\_ID");

**if** ( onlyCurrentRevision ){

sb.append("\ninner join DM\_HEADER h ON h.LATEST\_REVISION\_ID=rev.REVISION\_ID and h.HEADER\_ID=?");

} **else** {

sb.append("\ninner join DM\_HEADER h ON h.HEADER\_ID=rev.HEADER\_ID and h.LATEST\_REVISION\_ID != rev.REVISION\_ID and h.HEADER\_ID=?");

}

sb.append(whereClause);

sb.append("\nUNION ALL");

sb.append(

"\nselect rev.REVISION\_ID, 2 as SRC, rd.DM\_RD\_DICT\_ID as ID, rel.DM\_RD\_RULE\_EXECUTION\_ID, rd.ELEMENT\_NAME, rd.ELEMENT\_PATH, rd.ELEMENT\_VALUE");

sb.append("\nfrom DM\_RD\_DICT rd");

sb.append("\ninner join DM\_RD\_DICT\_REL rel on rel.DM\_RD\_DICT\_ID=rd.DM\_RD\_DICT\_ID");

sb.append("\ninner join DM\_RULE\_EXECUTION re on re.RULE\_EXECUTION\_ID=rel.DM\_RD\_RULE\_EXECUTION\_ID");

sb.append("\ninner join DM\_REVISION\_REQUEST rr on rr.REQUEST\_ID=re.REQUEST\_ID");

sb.append("\ninner join DM\_REVISION rev on rev.REVISION\_ID=rr.REVISION\_ID");

**if** ( onlyCurrentRevision ){

sb.append("\ninner join DM\_HEADER h ON h.LATEST\_REVISION\_ID=rev.REVISION\_ID and h.HEADER\_ID=?");

} **else** {

sb.append("\ninner join DM\_HEADER h ON h.HEADER\_ID=rev.HEADER\_ID and h.LATEST\_REVISION\_ID != rev.REVISION\_ID and h.HEADER\_ID=?");

}

sb.append(whereClause);

**return** sb.toString();

}

**private** **void** process(**final** String entity, String idstr, Integer limit, **final** **boolean** onlyCurrentRevision) **throws** Exception {

mappingsDao.load();

String clientSql = **this**.getSql(Constants.CLIENT, onlyCurrentRevision);

String stmtSql = **this**.getSql(Constants.STMT, onlyCurrentRevision);

String ratingSql = **this**.getSql(Constants.RATING, onlyCurrentRevision);

List<Document> docs = **new** ArrayList<Document>();

**if** ("all".equals(idstr)) {

**if** (Constants.CLIENT.equals(entity) || "all".equals(entity)) {

logger.info("Identify client ids");

List<Long> ids = jdbcTemplate

.query("select distinct c.ID from RBC\_P\_CLIENT c join DM\_HEADER h on h.HEADER\_ID=c.ID left join AUDIT\_MASKING m on c.ID=m.DOC\_ID where m.DOC\_ID is null"

+ (limit != **null** && limit > 0 ? " and rownum <= " + limit : ""), **new** RowMapper<Long>() {

**public** Long mapRow(ResultSet rs, **int** arg1) **throws** SQLException {

**return** rs.getLong(1);

}

});

logger.info("Number of entries: " + ids.size());

**for** (Long id : ids) {

docs.add(**new** Document(Constants.CLIENT, id));

}

}

**if** (Constants.STMT.equals(entity) || "all".equals(entity)) {

logger.info("Identify statement ids");

List<Long> ids = jdbcTemplate

.query("select distinct s.DOCUMENT\_ID from FS\_STATEMENT s join DM\_HEADER h on h.HEADER\_ID=s.STATEMENT\_ID left join AUDIT\_MASKING m on s.DOCUMENT\_ID=m.DOC\_ID where m.DOC\_ID is null"

+ (limit != **null** && limit > 0 ? " and rownum <= " + limit : ""), **new** RowMapper<Long>() {

**public** Long mapRow(ResultSet rs, **int** arg1) **throws** SQLException {

**return** rs.getLong(1);

}

});

logger.info("Number of entries: " + ids.size());

**for** (Long id : ids) {

docs.add(**new** Document(Constants.STMT, id));

}

}

**if** (Constants.RATING.equals(entity) || "all".equals(entity)) {

logger.info("Identify risk assessment ids");

List<Long> ids = jdbcTemplate

.query("select distinct r.ID from RBC\_P\_RATING r join DM\_HEADER h on h.HEADER\_ID=r.ID left join AUDIT\_MASKING m on r.ID=m.DOC\_ID where m.DOC\_ID is null"

+ (limit != **null** && limit > 0 ? " and rownum <= " + limit : ""), **new** RowMapper<Long>() {

**public** Long mapRow(ResultSet rs, **int** arg1) **throws** SQLException {

**return** rs.getLong(1);

}

});

logger.info("Number of entries: " + ids.size());

**for** (Long id : ids) {

docs.add(**new** Document(Constants.RATING, id));

}

}

} **else** {

String[] ss = idstr.split(",");

**switch** (entity) {

**case** Constants.CLIENT:

**for** (String s : ss) {

docs.add(**new** Document(Constants.CLIENT, Long.valueOf(s.trim())));

}

**break**;

**case** Constants.RATING:

**for** (String s : ss) {

docs.add(**new** Document(Constants.RATING, Long.valueOf(s.trim())));

}

**break**;

**case** Constants.STMT:

**for** (String s : ss) {

docs.add(**new** Document(Constants.STMT, Long.valueOf(s.trim())));

}

**break**;

}

}

**final** AtomicInteger rows\_updated = **new** AtomicInteger(0);

**final** AtomicInteger docs\_updated = **new** AtomicInteger(0);

BlockingQueue<Runnable> queue = **new** ArrayBlockingQueue<Runnable>(queue\_size);

ThreadPoolExecutor exec = **new** ThreadPoolExecutor(core\_pool\_size, maximum\_pool\_size,

keep\_alive\_time\_milliseconds, TimeUnit.MILLISECONDS, queue, **new** ThreadPoolExecutor.CallerRunsPolicy());

**final** Semaphore semaphore = **new** Semaphore(maximum\_pool\_size);

**for** (**final** Document doc : docs) {

**final** String sql = Constants.CLIENT.equals(doc.getEntity()) ? clientSql

: (Constants.STMT.equals(doc.getEntity()) ? stmtSql : ratingSql);

exec.execute(**new** Runnable() {

@Override

**public** **void** run() {

**try** {

semaphore.acquire();

Map<Long, Revision> revisions = dmRequestDataDao.getRevisions(doc.getId(), sql);

**int** nbr\_rows = processDocument(doc, revisions,onlyCurrentRevision);

rows\_updated.addAndGet(nbr\_rows);

revisions.clear();

revisions = **null**;

**int** nbr = docs\_updated.incrementAndGet();

**if** (nbr % 100 == 0) {

logger.info("Processed " + nbr + " documents and " + rows\_updated.get() + " rows");

}

} **catch** (Exception ex) {

logger.error("Error processing document "+doc.getId(),ex);

} **finally** {

semaphore.release();

}

}

});

}

exec.shutdown();

exec.awaitTermination(4, TimeUnit.HOURS);

logger.info("Processing completed. Total Documents: " + docs\_updated + ", Rows: " + rows\_updated);

}

**public** **static** **void** main(String[] args) **throws** Exception {

System.setProperty("app.name", "masking-service");

Utils.setEnvironment();

Logger logger = Logger.getLogger(MaskingService.**class**);

SimpleGNUCommandLine cmd = **new** SimpleGNUCommandLine(args);

cmd.addOption("type", "client, rating, stmt or all", **true**, **true**);

cmd.addOption("id", "all or comma separated list of doc ids", **true**, **true**);

cmd.addOption("limit", "Limit", **true**, **false**);

ClassPathXmlApplicationContext appln\_context = **null**;

**try** {

appln\_context = **new** ClassPathXmlApplicationContext("spring/Context.xml");

MaskingService processor = appln\_context.getBean(MaskingService.**class**);

//logger.info("Process current revisions");

//processor.process(cmd.getString("type"), cmd.getString("id"), cmd.getInteger("limit"),true);

logger.info("Process other revisions");

processor.process(cmd.getString("type"), cmd.getString("id"), cmd.getInteger("limit"),**false**);

System.exit(1);

} **catch** (Exception ex) {

logger.error("Data masking failed ", ex);

System.exit(0);

} **finally** {

**if** (appln\_context != **null**) {

appln\_context.close();

}

}

}

}