# GoldenGate

Разного рода статьи и полезности, в части настроек и эксплуатации Оракл ГолденГейт

- Параметры ОГГ
  - Параметр INSERTMISSINGUPDATES в Oracle GoldenGate: как работает и где может пригодиться
- Разное
  - Компрессия и голденгейт
  - Партиционируем и жмем огромную таблицу, с работающей репликацией
- Настройка репликации
  - Инитлоад через промежуточный сервер:
- Veridata
  - Oracle GoldenGate Veridata 26c: Эволюция от утилиты к платформе автоматизации
- Debezium
  - Debezium для CDC-репликации Oracle → Oracle

# Параметры ОГГ

# Параметр INSERTMISSINGUPDATES в Oracle GoldenGate: как работает и где может пригодиться

Параметр **`INSERTMISSINGUPDATES`** **(** **`NOINSERTMISSINGUPDATES`** **)** управляет поведением процесса **Replicat** в ситуации, когда для операции обновления (UPDATE) не найдена целевая запись. Если параметр включён, **Replicat** вставит новую строку на основе данных из источника; если выключен (значение по умолчанию) – запись отсутствует, возникает ошибка, и дальнейшая судьба транзакции зависит от настройки `REPERROR`.

## Как это работает по документации

**`INSERTMISSINGUPDATES`** следует использовать только тогда, когда исходная база данных гарантированно логирует **все** столбцы строки (даже те, что не изменились). Это необходимо, потому что для вставки новой записи требуются значения всех столбцов, а не только изменённых.

- Если источник передаёт только изменённые столбцы **(сжатый формат обновлений),** параметр всё же может работать при условии, что целевая таблица допускает `NULL` в отсутствующих колонках.
- Если же база данных по умолчанию логирует все столбцы (например, включено дополненное логирование), то для корректной работы **`INSERTMISSINGUPDATES`** необходимо также использовать параметры **`NOCOMPRESSUPDATES`** и **`NOCOMPRESSDELETES`** (если они поддерживаются). В противном случае потребуется **`FETCHOPTIONS MISSINGCOLS`**, чтобы добрать недостающие колонки из источника.

Важно помнить: параметр является таблично-специфичным и действует для всех последующих операторов `MAP`, пока не встретится противоположная директива.

# Дополнительные требования к источнику

Из описания ясно, что для надёжной работы **INSERTMISSINGUPDATES** на таблицах источника должно быть включено дополненное логирование всех столбцов:

```
ALTER TABLE SCHEME.CUSTOMERS ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

Без этого при обновлении в трейл попадут только изменившиеся колонки, и **Replicat** не сможет сформировать полноценную вставку.

# Неочевидный сценарий использования: заполнение пробелов

Параметр может пригодиться не только для обработки редких ситуаций рассинхронизации, но и для целенаправленного «долива» пропущенных данных. Предположим, в таблице SCHEME.CUSTOMERS по какой-то причине образовались пробелы (часть строк отсутствует на приёмнике), мы знаем, что DELETE-операций в этом наборе нет. Мы знаем временной интервал, в котором могли быть пропуски.

Тогда можно выполнить на источнике фиктивный UPDATE, который затронет все потенциально пропущенные строки:

```
UPDATE SCHEME.CUSTOMERS SET ID = ID
WHERE DATE_CHANGE BETWEEN TO_DATE('...') AND TO_DATE('...');
```

Поскольку условие ID = ID не меняет данные, на источнике не происходит реальных изменений, но в трейл всё равно попадут образы строк (**NOCOMPRESSUPDATES** за это отвечает). На приёмнике, если строка уже существует, обновление пройдёт без эффекта; если же строки нет, **INSERTMISSINGUPDATES** вставит её. Так можно аккуратно засинкать таблицы, без полной перезаливки или использования Веридаты.

# Грабли: порядок операций

Несмотря на заверения оракл, что **Replicat** применяет изменения в том же порядке, что и на источнике, на практике возможны спецэффекты. Например, если после всех обновлений строки она была удалена, но из-за особенностей транспорта или параллельной работы DELETE прилетит раньше финального UPDATE, то на приёмнике картина может исказиться:

| Источник | Приёмник (ожидание) | Реальность при нарушении порядка |
|---|---|---|
| INSERT | INSERT | INSERT |
| UPDATE | UPDATE | UPDATE |
| UPDATE | UPDATE | UPDATE |
| UPDATE | UPDATE | DELETE (пришёл раньше последнего UPDATE) |
| DELETE | DELETE | UPDATE (вставляет пропущенную строку!) |

В результате удалённая на источнике строка снова появится на приёмнике из-за **`INSERTMISSINGUPDATES`**, который вставит её при обработке запоздавшего UPDATE. Поэтому включать параметр'чтоб было' - не стоит.

# Разное

Разное

# Компрессия и голденгейт

Представим ситуацию. У нас есть табличка, которая содержит информацию о заказах. Когда-то заказов в табличке были сотни, потом тысячи, десятки тысяч, и в какой-то момент оказалось, что заказов у нас миллиарды, а бизнес не хочет удалять исторические данные. При этом место у нас не резиновое и стоит дорого.

Тогда мы вспоминаем, что Oracle предоставляет неплохой инструмент для сжатия данных в таблицах — от классических Advanced/OLTP/Basic до специфических компрессий, которые есть на Exadata: FOR QUERY HIGH/LOW и FOR ARCHIVE HIGH/LOW. И это хорошо.

Но в таблицу идёт активная вставка данных (ведь у нас очень много покупателей и заказов), плюс на таблице выполняются операции обновлений, потому что заказы могут меняться, отменяться и т.д. Поэтому, если мы просто сожмём таблицу на нашей замечательной Exadata, мы, конечно, получим выигрыш в месте (возможно, даже 20-кратный), но при этом полностью «убьём» приложения, которые изменяют таблицу.

Остановимся на том, что наша таблица заполняется на неком хранилище репликатом GoldenGate. С таблицей работает координированный репликат, потому что если не распараллеливать вставки и обновления, то у нас начинает накапливаться отставание.

Как только мы сожмём таблицу, наш координированный репликат начнёт выдавать ошибку ORA-00060 (deadlock detected) и падать на взаимоблокировках: одновременно несколько потоков пытаются изменить один и тот же блок.

Но выход есть. Надо таблицу партиционировать. Мы предполагаем, что заказы старше определённой даты уже не могут быть отменены или изменены. Мы партиционируем таблицу по интервалу в один месяц, сжимаем «старые» партиции, которые не будут меняться, той компрессией, которую мы выбрали, а новые партиции оставляем как есть — несжатыми.

По мере заполнения партиций мы можем автоматически сжимать устаревающие партиции. Для этого нам надо предусмотреть, чтобы индексы (для работы репликата нужен уникальный индекс в любом случае) были локальными.

# Пример реализации

Создадим таблицу заказов, партиционированную по диапазону дат с интервалом в один месяц:

```
CREATE TABLE orders (
    order_id      NUMBER PRIMARY KEY,
    order_date    DATE,
    customer_id   NUMBER,
    order_status  VARCHAR2(20),
    ...
)
PARTITION BY RANGE (order_date) INTERVAL (INTERVAL '1' MONTH)
(
    PARTITION p_historical VALUES LESS THAN (TO_DATE('2020-01-01','YYYY-MM-DD'))
);
```

Затем создадим локальный уникальный индекс на поле `order_id`, необходимый для работы репликации:

```
CREATE UNIQUE INDEX orders_pk ON orders (order_id, order_date) LOCAL;
```

Теперь можно сжать партицию, данные в которой уже не меняются. Например, для партиции за январь 2020 года:

```
ALTER TABLE orders
    MOVE PARTITION p_2020_jan
    PARALLEL 16
    COMPRESS FOR ARCHIVE HIGH;


SELECT 'ALTER INDEX ' || INDEX_NAME || ' REBUILD PARTITION ' || PARTITION_NAME || ' PARALLEL 16;'
FROM DBA_IND_PARTITIONS IP
WHERE IP.STATUS = 'UNUSABLE';
```

Это сжатие не затронет активные партиции, и репликат продолжит работать без конфликтов за блоки.

Таким образом, партиционирование в сочетании с выборочным сжатием позволяет эффективно экономить место, не жертвуя производительностью операций DML и не создавая проблем для параллельной репликации.

Разное

# Партиционируем и жмем огромную таблицу, с работающей репликацией

Итак, мы решили, что будем партиционировать таблицу ORDERS, про которую шла речь в предыдущем примере. В принципе, мы можем просто создать рядом таблицу с атрибутом компрессии на уровне таблицы и сделать инсерт из нашей большой таблицы. Но так не получится. Инсерт будет работать очень долго, параллелиться он не будет, так как партиции, куда льются данные, сжаты, и в результате сначала отработают воркеры, которые делают селект из исходной таблицы, а потом мы будем ждать, когда данные в один поток зальются в приемник. Мы так делать не будем. Мы проверим, какие партиции у нас в принципе могут быть созданы с интервалом в месяц:

```
SELECT /*+ PARALLEL(t 32)  INDEX(t ORDERS_DATE_IDX) */
      COUNT(1),
      TO_CHAR(t.order_date, 'rrrr.mm')
FROM   orders t
GROUP BY TO_CHAR(t.order_date, 'rrrr.mm');
```

(у нас есть индекс по полю, которое будет впоследствии ключом партиционирования, и это хорошо)

Получим что-то вроде такого результата:

| COUNT(1) | ORDER_DATE |
|---:|---:|
| 14281 | 2013.12 |
| 16223 | 2014.01 |
| 15018 | 2014.02 |
| 14926 | 2014.03 |
| 22395 | 2014.04 |
| 18620 | 2014.05 |

| | |
|---|---|
| 18111 | 2014.06 |
| 27140 | 2014.07 |
| 234983 | 2014.08 |
| 492579 | 2014.09 |
| ... | ... |
| 436314181 | 2025.12 |
| 570153932 | 2026.01 |
| 346233850 | 2026.02 |

Теперь создадим таблицу, в которой есть все партиции, которые нам нужны. Атрибут компрессии мы выставлять не будем:

```
CREATE TABLE orders_parts (
    order_id      NUMBER PRIMARY KEY,
    order_date    DATE,
    customer_id   NUMBER,
    order_status  VARCHAR2(20),
    ...
)
PARTITION BY RANGE (order_date) INTERVAL (INTERVAL '1' MONTH)
(
  PARTITION P_BEFORE_2013 VALUES LESS THAN (DATE '2013-12-01'),
  PARTITION P_201312 VALUES LESS THAN (DATE '2014-01-01') TABLESPACE CUST_DATA,
  PARTITION P_201401 VALUES LESS THAN (DATE '2014-02-01') TABLESPACE CUST_DATA,
  PARTITION P_201402 VALUES LESS THAN (DATE '2014-03-01') TABLESPACE CUST_DATA,
  ........
  PARTITION P_202511 VALUES LESS THAN (DATE '2025-12-01') TABLESPACE CUST_DATA,
  PARTITION P_202512 VALUES LESS THAN (DATE '2026-01-01') TABLESPACE CUST_DATA,
  PARTITION P_202601 VALUES LESS THAN (DATE '2026-02-01') TABLESPACE CUST_DATA,
  PARTITION P_202602 VALUES LESS THAN (DATE '2026-03-01') TABLESPACE CUST_DATA
);
```

После чего нам нужно будет залить данные до уровня, когда мы уверены, что эти данные останутся неизменными.

Для этого используем DBMS_PARALLEL_EXECUTE...

```
SET SERVEROUTPUT ON SIZE UNLIMITED
SET LINESIZE 200
```

```
SET PAGESIZE 0
SET FEEDBACK OFF
SET VERIFY OFF

ALTER SESSION ENABLE PARALLEL DML;

-- 1. Создаём задачу
BEGIN
  DBMS_PARALLEL_EXECUTE.CREATE_TASK('LOAD_ORDERS_PARTS');
END;
/

-- 2. Создаём чанки (диапазоны дат) для всех партиций, кроме двух последних
BEGIN
  DBMS_PARALLEL_EXECUTE.CREATE_CHUNKS_BY_SQL(
    task_name    => 'LOAD_ORDERS_PARTS',
    sql_stmt     =>
      'WITH part_info AS (
         SELECT partition_name,
                CASE
                  WHEN partition_name = ''P_BEFORE_2013'' THEN DATE ''1900-01-01''
                  ELSE TO_DATE(SUBSTR(partition_name, 3, 6) || ''01'', ''YYYYMMDD'')
                END AS start_date,
                CASE
                  WHEN partition_name = ''P_BEFORE_2013'' THEN DATE ''2013-12-01''
                  ELSE ADD_MONTHS(TO_DATE(SUBSTR(partition_name, 3, 6) || ''01'', ''YYYYMMDD''), 1)
                END AS end_date,
                partition_position
         FROM user_tab_partitions
         WHERE table_name = ''ORDERS_PARTS''
       )
       SELECT start_date, end_date
       FROM part_info
       WHERE partition_position <= (SELECT MAX(partition_position)-2 FROM part_info)
       ORDER BY partition_position',
    by_rowid => FALSE
  );
END;
/
```

```
-- 3. Запускаем задачу с параллельностью 16
BEGIN
  DBMS_PARALLEL_EXECUTE.RUN_TASK(
    task_name      => 'LOAD_ORDERS_PARTS',
    sql_stmt       => 'INSERT /*+ APPEND PARALLEL(16) */ INTO orders_parts
    SELECT /*+ PARALLEL(orders,16) */ * FROM orders
    WHERE order_date >= :start_id AND order_date < :end_id',
    language_flag  => DBMS_SQL.NATIVE,
    parallel_level => 16
  );
END;
/

-- 4. Зачистка задач
exec DBMS_PARALLEL_EXECUTE.DROP_TASK('LOAD_ORDERS_PARTS');

-- 5. Посмотреть, что там с обработкой чанков
SELECT
       task_name,
       status
FROM
       user_parallel_execute_tasks;

SELECT
        chunk_id,
        status,
        start_rowid,
        end_rowid
FROM
        user_parallel_execute_chunks
WHERE
        task_name = 'LOAD_ORDERS_PARTS'
ORDER BY
        chunk_id;
```

Или, если не хочется по каким-то причинам использовать DBMS_PARALLEL_EXECUTE, всё то же самое сделаем вручную:

```
-- Партиция P_BEFORE_2013: все заказы до 2013-12-01
INSERT /*+ APPEND PARALLEL(16) */ INTO orders_parts
```

```
SELECT /*+ PARALLEL(orders,16) */ * FROM orders
WHERE order_date < DATE '2013-12-01';
COMMIT;

-- Партиция P_201401: декабрь 2013 (2013-12-01 <= order_date < 2014-01-01)
INSERT /*+ APPEND PARALLEL(16) */ INTO orders_parts
SELECT /*+ PARALLEL(orders,16) */ * FROM orders
WHERE order_date >= DATE '2013-12-01' AND order_date < DATE '2014-01-01';
COMMIT;

-- Партиция P_201402: январь 2014
INSERT /*+ APPEND PARALLEL(16) */ INTO orders_parts
SELECT /*+ PARALLEL(orders,16) */ * FROM orders
WHERE order_date >= DATE '2014-01-01' AND order_date < DATE '2014-02-01';
COMMIT;

-- Партиция P_201403: февраль 2014
INSERT /*+ APPEND PARALLEL(16) */ INTO orders_parts
SELECT /*+ PARALLEL(orders,16) */ * FROM orders
WHERE order_date >= DATE '2014-02-01' AND order_date < DATE '2014-03-01';
COMMIT;

-- ... аналогично для всех промежуточных месяцев до 2025 ...

-- Партиция P_202511: ноябрь 2025
INSERT /*+ APPEND PARALLEL(16) */ INTO orders_parts
SELECT /*+ PARALLEL(orders,16) */ * FROM orders
WHERE order_date >= DATE '2025-11-01' AND order_date < DATE '2025-12-01';
COMMIT;

-- Партиция P_202512: декабрь 2025
INSERT /*+ APPEND PARALLEL(16) */ INTO orders_parts
SELECT /*+ PARALLEL(orders,16) */ * FROM orders
WHERE order_date >= DATE '2025-12-01' AND order_date < DATE '2026-01-01';
COMMIT;

-- Две последнии партиции не трогаем
```

Вариант с ручным переносом может быть полезен в случае, если места у нас осталось впритык и как только мы перенесли одну партицию, мы её сразу же компрессируем.

Теперь собственно мы берём наши партиции и добавляем в них компрессию:

```
ALTER TABLE ORDERS MOVE PARTITION P_BEFORE_2013 COMPRESS FOR QUERY HIGH PARALLEL 16;
ALTER TABLE ORDERS MOVE PARTITION P_201312 COMPRESS FOR QUERY HIGH PARALLEL 16;
ALTER TABLE ORDERS MOVE PARTITION P_201401 COMPRESS FOR QUERY HIGH PARALLEL 16;
ALTER TABLE ORDERS MOVE PARTITION P_201402 COMPRESS FOR QUERY HIGH PARALLEL 16;
.... и так далее
```

После того как всё сжато, нам осталось долить данные в партиции, которые будут активно изменяться в текущем периоде. И тут у нас есть два подхода. Если мы можем позволить себе небольшой даунтайм (отключение записи в таблицу) на время, пока мы переливаем данные в последние партиции и строим нужные индексы, то, в принципе, следующий этап совсем прост. Нам нужно остановить репликат, по аналогии с тем, как мы заливали предыдущие данные, долить оставшиеся, построить локальный индекс, сделать переименование таблицы (возможно, придётся откомпилировать зависимые объекты) и снова запустить репликат.

В случае, если даунтайм, должен быть минимальный (минуты) мы сделаем по другому:

1 У нас уже есть репликат, который пишет данные в исходную таблицу, и нам нужно засечь SCN, начиная с которого мы будем заполнять нашу новую таблицу. Для этого мы создадим новый репликат с маппингом, аналогичным исходному, например так:

```
--Наш исходный репликат, потребляет трэйл ./dirdat/ad
ADD REPLICAT RORDPART, EXTTRAIL ./dirdat/ad checkpointtable GGATE.CHECKPOINTS

--На этом же этапе, подготовим репликаты, для синхронной остановки через  eventactions( stop )
--Создадим в базах - источнике и приемнике, таблицу:
create table GGATE.STOPACTION
(
  id        number GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  stopdate  date
);
--Добавим логирвоане на эту таблицу:
ALTER TABLE GGATE.STOPACTION ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;

-- В экстратор, который поставляет трэйл - файлы для репликатов, пропишем:
TABLE GGATE.STOPACTION, tokens( commit_scn=@GETENV('TRANSACTION', 'CSN') );

--Перезапустим экстратор

--Добавим конструкцию в оба репликата:
```

```
MAP GGATE.STOPACTION, target GGATE.STOPACTION, colmap(usedefaults, stop_scn=@TOKEN('commit_scn')),
eventactions( stop );
```

2 Пропишем конфиг, аналогичный исходному репликату, в наш новый репликат, оставив в новом репликате маппинг только на одну таблицу ORDER_PARTS.

3 Остановим исходный репликат, и засечем SCN

```
SELECT CURRENT_SCN FROM V$DATABASE;
```

4 Переставим наш новый репликат на тот RBA, на котором у нас остановлен основной репликат:

```
--Посмотрим где у нас остановлен основной репликат
GGSCI (ggate.local.net) 2> info RORDERS

Replicat   RORDERS   Last Started 2026-02-24 21:03   Status STOPPED
Checkpoint Lag       00:02:05 (updated 00:00:02 ago)
Process ID           2394620
Log Read Checkpoint  File ./dirdat/ad002328787
                     2026-02-25 12:22:12.000000  RBA 407552118

--Переставим новый репликат на нужный RBA
GGSCI (ggate.local.net) 3> ALTER REPLICAT RORDPART EXTSEQNO 2328787 EXTRBA 407552118

--Запустим основной репликат
GGSCI (ggate.local.net) 4> START RORDERS
```

Теперь новый репликат, когда будет запущен, начнёт писать изменения ровно с того места, где был остановлен основной репликат и для которого мы засекли SCN. Это первое время простоя репликации, и мы, скорее всего, уложились в пару минут.

5 Перенесём данные из исходной таблицы, который у нас остались:

```
-- Партиция P_202601: январь 2026
INSERT /*+ APPEND PARALLEL(16) */ INTO orders_parts
SELECT /*+ PARALLEL(orders,16) */ * FROM orders AS OF SCN (<значени SCN, которое мы засекли>)
WHERE order_date >= DATE '2026-01-01' AND order_date < DATE '2026-02-01';
COMMIT;
-- Партиция P_202602: февраль 2026
INSERT /*+ APPEND PARALLEL(16) */ INTO orders_parts
SELECT /*+ PARALLEL(orders,16) */ * FROM orders AS OF SCN (<значени SCN, которое мы засекли>)
```

```
WHERE order_date >= DATE '2026-02-01' AND order_date < DATE '2026-03-01';
COMMIT;
```

6 Проиндексируем таблицу:

```
CREATE UNIQUE INDEX orders_parts_idx on orders_parts(order_id, order_date) local parallel 16;
alter index orders_parts_idx noparallel;
```

7 Запустим репликат, который мы создали:

```
GGSCI (gghubs.local.net) 3> START RORDPART
```

8 После того, как репликат догонит таблицу по данным, мы остановим оба репликата синхронно. Для этого у нас есть несколько вариантов:

- остановка пампа\экстратора, который формирует трэйл. Это самое простое, но, возможно, что экстратор, после рестарта, будет долго выполнят рекавер
- остановка, через событие  eventactions( stop ) самый подходящий для нас вариант

9 Остановка событием. Для этого, просто, делаем инсерт в таблицу, которую мы создавали на источнике:

```
insert into GGATE.STOPACTION (STOPDATE) values (SYSDATE);
COMMIT;
```

Оба репликата, синхронно остановтся

10 Переименовываем таблицы:

```
--Преименуем таблицы
ALTER TABLE ORDERS RENAME TO ORDERS_BACK;
ALTER TABLE ORDERS_PARTS RENAME TO ORDERS;

--Проверим, что у нас развалились зависимые пакеты и сделаем рекмпиляцию пакетов, которые развалились
ALTER PACKAGE ... COMPILE
```

11 Мы можем запустить 'старый' репликат и удалить 'новый'

```
GGSCI (gghubs.local.net) 3> START RORDERS

GGSCI (gghubs.local.net) 3> STOP RORDPART
```

```
--Для того, чтобы в базе не осталось ошметков в таблице чекпоинтов, перед удалением, логинимсяв базу
GGSCI (gghubs.local.net) 3> DBLOGIN USERIDALIAS <алиас>
GGSCI (gghubs.local.net) 3> DELETE RORDPART
```

Итого, у нас теперь есть партиционированная таблица, с пожатыми историческими партициями и  общее время простоя репликации - несколько минут.

# Настройка репликации

# Инитлоад через промежуточный сервер:

Возможна ситуация, когда по каким-то причинам выгрузить и перебросить дамп-файл с источника на приёмник не представляется возможным. Тогда мы можем сделать первоначальную загрузку средствами самого GoldenGate:

На источнике создадим экстракт:

```
GGSCI> DBLOGIN USERIDALIAS source_alias
GGSCI> ADD EXTRACT E_LOAD, SOURCEISTABLE
GGSCI> EDIT PARAMS E_LOAD
```

```
EXTRACT E_LOAD
USERIDALIAS source_alias
RMTHOST intermediate.local.net, MGRPORT 7809
RMTTASK extract, GROUP p_load
RMTTRAIL ./dirdat/ip
TABLE ACCOUNTS.PROT_ID;
TABLE ACCOUNTS.COMPANY;
TABLE ACCOUNTS.ACCOUNTS;
TABLE ACCOUNTS.ACCOUNT_HISTORY;
```

Экстракт прочитает таблицы напрямую (SOURCEISTABLE) и отправит данные в удалённый трейл на промежуточном сервере.

На промежуточном сервере создадим обычный памп (extract в роли дата-пампа), который будет читать только что записанный трейл и пересылать его дальше, на конечный приёмник:

```
GGSCI> ADD EXTRACT p_load, EXTTRAILSOURCE ./dirdat/ip
GGSCI> EDIT PARAMS p_load
```

```
EXTRACT p_load
RMTHOST dest.local.net, MGRPORT 7809
```

```
RMTTRAIL ./dirdat/ip
TCPBUFSIZE 1000000
TCPFLUSHBYTES 1000000
COMPRESS
TABLE ACCOUNTS.PROT_ID;
TABLE ACCOUNTS.COMPANY;
TABLE ACCOUNTS.ACCOUNTS;
TABLE ACCOUNTS.ACCOUNT_HISTORY;
```

На приёмнике создадим репликат, который будет выполнять загрузку (параметр SPECIALRUN – репликат отработает один раз и завершится):

```
GGSCI> DBLOGIN USERIDALIAS dest_alias
GGSCI> ADD REPLICAT R_LOAD, SPECIALRUN
GGSCI> EDIT PARAMS R_LOAD
```

```
REPLICAT R_LOAD
EXTTRAIL ./dirdat/ip
USERIDALIAS dest_alias
EOFDELAY 30
SETENV ( NLS_LANG="AMERICAN_AMERICA.CL8MSWIN1251" )
ASSUMETARGETDEFS
MAP ACCOUNTS.PROT_ID         TARGET ACCOUNTS.PROT_ID;
MAP ACCOUNTS.COMPANY         TARGET ACCOUNTS.COMPANY;
MAP ACCOUNTS.ACCOUNTS        TARGET ACCOUNTS.ACCOUNTS;
MAP ACCOUNTS.ACCOUNT_HISTORY TARGET ACCOUNTS.ACCOUNT_HISTORY;
```

После настройки запускаем процессы в правильном порядке: сначала репликат на приёмнике (он будет ждать данные), затем экстракт на источнике. Репликат завершится сам после применения всех записей, а экстракт закончит работу, когда вычитает все строки из таблиц. Готово – первоначальная загрузка выполнена без создания физического дампа.

Следить за состоянием проуесса, обычными командами SEND <Имя процесса> STATUS, или можо посомтреть статистику, командой STATS <Имя процесса>

# Veridata

Случилось то, чего все ждали. Оракл наконец то начал развивать веридату.

Veridata

# Oracle GoldenGate Veridata 26c: Эволюция от утилиты к платформе автоматизации

Что принципиально нового появилось в 26-м релизе Oracle GoldenGate Veridata? Если говорить кратко — он наконец-то стал полноценным гражданином мира DevOps.

Я помню времена Veridata 12c. Мощный инструмент для сравнения данных, спору нет. Но попытки встроить его в конвейеры автоматизации наталкивались на суровую реальность: либо веб-интерфейс, либо `vericom.sh`. Скрипты, шелл, костыли — и вот ты уже гордый владелец сложной системы из вызовов коммандника и парсинга вывода. Но, да, это работало и довольно неплохо.

В 23-м релизе что-то начало меняться, но именно 26-й версии можно смело сказать: **наконец-то**. То, что мы раньше делали с помощью `vericom.sh` и баш-скриптов, теперь можно делать через полноценный REST API. И это не просто "еще один способ", это фундаментальный сдвиг в том, как мы строим процессы вокруг верификации данных.

## API как фундамент автоматизации

Открываем документацию по REST endpoints и видим не просто пару методов "для галочки", а полноценное покрытие жизненного цикла сравнения:

- **Управление конфигурацией**: создание соединений (`/connections`), групп сравнения (`/groups`), джобов (`/jobs`) и правил сравнения (`/cps`). Больше не нужно накликивать это в GUI или тащить за собой XML-ки через утилиты импорта/экспорта, если вы строите динамическую инфраструктуру.
- **Запуск и мониторинг**: запуск джобы (`/execution/jobs/{id}`), остановка, мониторинг статуса (`/monitoring/jobs`). Это именно то, чего так не хватало для интеграции с Jenkins, GitLab CI или вашей внутренней системой оркестрации.

- **Работа с расхождениями**: получение данных о "раз sync" записях (`/oos/{runId}`) и, что важнее, запуск ремонта (`/repair/jobs/{runId}`). Мы теперь можем строить умные пайплайны: обнаружил расхождение — запусти автоматический ремонт или создай тикет в ServiceNow.
- **Администрирование**: управление пользователями (`/admin/users`), настройка уровней логирования (`/logs/server/configuration`). Все это теперь можно централизованно конфигурировать через код.

# Встроенный планировщик: прощай, Cron

Второй ключевой момент, который идет рука об руку с API — нативная поддержка планировщика. Раньше, чтобы запускать сравнение по ночам, мы использовали `cron`, `vericom.sh` . Это было внешнее по отношению к Veridata решение.

Теперь планировщик — это часть продукта. Судя по блогу Oracle, он доступен и через UI, и через API (`/configuration/schedule/jobs`). Это означает, что мы можем создавать периодические задания на сравнение данных программно. Хотите сравнивать критичную витрину каждые 15 минут? Пожалуйста, просто POST на нужный эндпоинт с расписанием в формате cron. Все это ловится, логируется и управляется из единого центра.

# Что это меняет на практике?

1. **Инфраструктура как код (IaC)**: теперь конфигурацию Veridata можно хранить в Git. При развертывании нового окружения мы не вспоминаем, как настроить соединения к тестовой и продовой БД. Мы просто применяем манифесты через API.
2. **Интеграция с пайплайнами**: этап верификации данных после миграции или репликации становится "first-class citizen" в CI/CD. Джоба запускается автоматически, статус проверки влияет на прохождение пайплайна. Зеленая сборка означает не только "код скомпилировался", но и "данные сошлись".
3. **Автоматическое устранение расхождений**: мы можем строить реактивные системы. Обнаружили расхождение в некритичных справочниках? Запустили `POST /repair/jobs`. Обнаружили в финансовом журнале? Отправили алерт в Telegram и создали задачу администратору, даже не заходя в веб-морду.

Конечно, приятно, что в 26-й версии появилась поддержка MongoDB и Azure Synapse, а сам продукт теперь доступен в виде контейнеров (что тоже упрощает автоматизацию развертывания). Но для меня, как для человека, который годами писал обертки над `vericom.sh`, главная новость именно в этом: **Veridata превратился из инструмента с ручным управлением в платформу для автоматизации доверия к данным**. API и встроенный планировщик — вот то, чего мы ждали долгие годы.

# Debezium

дистрибутив, деплой и сравнение с Oracle GoldenGate

# Debezium для CDC-репликации Oracle → Oracle

Материал собран по официальным источникам Debezium (blog, GitHub, документация), официальной документации Oracle GoldenGate и обсуждениям в issue-трекере/форуме проекта.

## 1. Где брать дистрибутив

- **Maven-артефакт:** `io.debezium:debezium-connector-oracle` — доступен на Maven Central / Sonatype (central.sonatype.com, mvnrepository.com). Актуальная на середину 2026 линейка — 3.6.x.
- **Docker-образы:** с релиза **Debezium 3.0.0.Final** новые образы (2.7.x и 3.x+) публикуются **только на quay.io/debezium**; на Docker Hub (docker.io) остаются исторические образы 1.x/2.x вплоть до 3.0.0.Final включительно, новых версий там больше не появляется — официально анонсировано в блоге Debezium (debezium.io/blog/2024/09/18/quay-io-reminder).
- **GitHub:** архивы плагина коннектора (tar.gz с jar-файлами для Kafka Connect plugin path) публикуются в релизах репозитория `debezium/debezium`.
- **Актуальная версия:** финальный релиз **Debezium 3.6** вышел 1 июля 2026 (debezium.io/blog/2026/07/01/debezium-3-6-final-release), предыдущая стабильная серия — 3.5 (последний патч 3.5.2.Final, 2 июня 2026, debezium.io/blog/2026/06/02/debezium-3-5-2-final-released). Номера версий стоит перепроверить на debezium.io/releases непосредственно перед установкой — релизы выходят часто.

## 2. Краткая инструкция по развёртыванию Oracle → Oracle

Debezium — это **только источник CDC** (capture + публикация в Kafka), а не готовый инструмент репликации. Доставки «из коробки» в целевую Oracle-БД у Debezium нет — нужен отдельный sink-коннектор.

## 2.1. Подготовка источника Oracle

- **ARCHIVELOG mode обязателен** — без него LogMiner не сможет работать.
- **Supplemental logging** — нужен минимум database-level supplemental logging; для полноценной фиксации значений колонок в redo-логах дополнительно нужен table-level supplemental logging уровня `ALL COLUMNS`. Начиная с **Debezium 3.4** коннектор распознаёт этот табличный режим как эквивалент минимального, поэтому нужен один из двух вариантов, а не оба одновременно (тикет DBZ-7341).
- **Права пользователя коннектора** — отдельный технический пользователь с грантами вида `CREATE TABLE`, `CREATE SEQUENCE`, `LOGMINING`, `SELECT ANY TRANSACTION`, `SELECT ANY DICTIONARY` и доступом к представлениям LogMiner.
- **Multitenant (CDB/PDB)** — если Oracle работает в архитектуре CDB/PDB, обязательно нужно задать `database.pdb.name`; без этого параметра коннектор майнит только корневую CDB и пропускает изменения в PDB.

## 2.2. Компоненты инфраструктуры

1. Кластер Kafka + Kafka Connect (distributed mode) — стандартная точка развёртывания коннекторов Debezium.
2. Плагин `debezium-connector-oracle` — устанавливается в plugin path воркеров Kafka Connect (из образа quay.io/debezium или архива с GitHub Releases).
3. **Sink для доставки в целевую Oracle** — на выбор:
   - официальный **`debezium-connector-jdbc`** (часть монорепозитория Debezium, `debezium.io/documentation/reference/stable/connectors/jdbc.html`) — «понимает» нативную структуру событий Debezium без дополнительных трансформаций;
   - сторонний (не Debezium) **Confluent Kafka Connect JDBC Sink Connector** — тоже официально поддерживает Oracle как target через MERGE-upsert, но с задокументированным ограничением: upsert ломается на первичном ключе типа `CHAR`, работает на `VARCHAR2` (docs.confluent.io/kafka-connectors/jdbc).

## 2.3. Ключевые параметры конфигурации source-коннектора

- `database.connection.adapter` — режим захвата:
  - `logminer` (по умолчанию) — LogMiner в режиме некоммиченных изменений, Debezium сам буферизует транзакции;
  - `logminer_unbuffered` — LogMiner отдаёт только закоммиченные изменения, экономит память коннектора, но перекладывает нагрузку на PGA источника (проблема для больших транзакций);
  - `olr` — через сторонний open-source OpenLogReplicator;
  - `xstream` — через Oracle XStream API, **требует лицензии на продукт Oracle GoldenGate** для production-использования, даже если сам GoldenGate физически не установлен (README debezium-connector-oracle).
- `log.mining.strategy`:
  - `redo_log_catalog` (по умолчанию) — самый надёжный вариант отслеживания DDL вперемешку с DML (LogMiner интерполирует между дамп-снимками словаря

данных), но самый дорогой — форсирует переключение redo-логов и генерирует больше архивных логов;
    - `online_catalog` — заметно быстрее, но не отслеживает DDL-изменения; подходит, если схема таблиц стабильна.
- Данные о `snapshot.mode` и точный список привилегий для XStream в собранном пуле не были независимо подтверждены до высокой степени детализации — перед продакшен-настройкой сверяйтесь напрямую с `debezium.io/documentation/reference/stable/connectors/oracle.html`.

# 3. Сравнение с Oracle GoldenGate

| | Oracle GoldenGate | Debezium (Oracle connector) |
|---|---|---|
| Архитектура | Единый коммерческий продукт: **Extract** читает redo/архивные логи и пишет в **Trail**-файлы; **Replicat** читает Trail и применяет изменения на target — либо через OCI (non-integrated), либо через LCR и inbound server (integrated mode) (docs.oracle.com — GoldenGate processes) | Только capture: коннектор публикует события в Kafka-топик. Доставка на target — отдельный, независимо настраиваемый sink-коннектор |
| Доставка на target | Встроена (Replicat) | Не встроена — нужен `debezium-connector-jdbc` или сторонний JDBC sink |
| Лицензирование | Платный продукт Oracle | Debezium — open source (Apache 2.0), но режим **XStream** внутри Oracle-коннектора Debezium требует лицензии GoldenGate |
| DDL | Integrated Replicat применяет DDL напрямую на target | Debezium парсит DDL из redo-потока, обновляет схему в памяти и пишет её в отдельный Kafka-топик schema history; при рестарте схема восстанавливается разбором этого топика с начала до точки рестарта |
| LOB/XMLType | Официально документировано Oracle как часть поддерживаемых типов GoldenGate (docs.oracle.com — data types) — детального сравнения в собранных источниках нет | CLOB/NCLOB/BLOB: неизменившееся значение не попадает в событие (плейсхолдер вместо значения). XMLType добавлен в 2.4, требует `lob.enabled=true`; были баги парсинга для не-binary storage моделей XMLType (CLOB/object-relational), исправлены позже |
| Производительность/мониторинг | открытый вопрос | JMX-метрики (`OffsetScn` и др.); trade-off `redo_log_catalog` vs `online_catalog` описан выше |

# 4. Трудности при переходе с GoldenGate на Debezium

1. **Нет доставки на target «из коробки».** Придётся отдельно разворачивать и эксплуатировать JDBC sink-коннектор — это отдельный компонент со своим жизненным циклом, ограничениями (например, CHAR-PK у Confluent JDBC Sink) и мониторингом.
2. **XStream не избавляет от лицензии GoldenGate.** Если ради производительности выбрать адаптер `xstream` вместо LogMiner, лицензия на GoldenGate всё равно нужна — миграция «с GoldenGate на Debezium ради экономии на лицензии» работает только при использовании LogMiner-адаптера.
3. **Trade-off LogMiner-стратегий.** `redo_log_catalog` (надёжное отслеживание DDL) против `online_catalog` (быстрее, но без DDL) — нужно явно решить, что важнее для конкретной репликации.
4. **Долгие транзакции.** В буферизованном режиме (`logminer`) долгие транзакции накапливаются в памяти коннектора; `logminer_unbuffered` снимает эту нагрузку с Debezium, но перекладывает её на PGA источника.
5. **Multitenant-архитектура (CDB/PDB)** добавляет обязательный параметр `database.pdb.name` и усложняет настройку прав и supplemental logging — GoldenGate решает это иначе, и при миграции это нужно перепроверять отдельно.
6. **LOB/XMLType — источник специфичных багов.** Известны нетривиальные проблемы с представлением XMLType в зависимости от storage-модели (CLOB / object-relational / binary XML), а также сочетание XMLType с generated columns — стоит тестировать на репрезентативной схеме перед переносом продакшена.
7. **Механика рестарта по SCN и зависимость от архивных логов**
   Debezium хранит в offset **два SCN**: *low watermark* (`scn`) — безопасная точка возобновления, как правило указывающая на начало самой старой ещё не закоммиченной транзакции на момент остановки, и *high watermark* (`commit_scn`) — позиция последней эмитированной транзакции (Debezium for Oracle — Part 3). При старте коннектор сравнивает low watermark с самым старым доступным на источнике архивным логом. Если этот лог начинается **позже** сохранённого SCN, коннектор падает с ошибкой `None of the log files contains offset SCN` и требует полного ре-снапшота (troubleshooting-разбор Debezium, июль 2025).
   Отсюда прямо следует наблюдаемый на практике эффект: если на момент остановки капчуринга существовала долгая незакоммиченная транзакция (или коннектор просто долго простаивал), low watermark может оказаться далеко в прошлом — и тогда для успешного рестарта на источнике нужны архивные логи, покрывающие этот старый SCN, а не только логи «с последней обработанной точки». Политика retention архивных логов на источнике должна учитывать этот сценарий — иначе логи будут вычищены раньше, чем понадобятся, и единственным выходом останется полный ре-снапшот данных.
   В Oracle RAC добавляется требование иметь логи по **всем redo-thread'ам** для восстановления глобального порядка транзакций; с версии Debezium 2.7 есть проверка консистентности логов (Redo Thread Consistency check), детектирующая разрывы между threads (единственный источник на это — блог Debezium, независимое подтверждение не найдено, поэтому уровень доверия — средний).
   Отдельно задокументирован конкретный **регрессионный баг в версиях 3.4.0/3.4.1**: коннектор мог «залипать» на SCN, даже если нужный лог физически присутствовал на диске в архиве — подтверждено мейнтейнером Debezium (Chris

Cranford) в официальной ветке форума, исправлено в **3.4.3.Final** (groups.google.com/g/debezium). Это отдельный повод при тестировании поведения рестарта фиксировать версию коннектора и не путать баг конкретного релиза с общей архитектурной особенностью.