Что нового в Firebird 5.0. Предложение SKIP LOCKED

Симонов Денис

Version 1.0 от 03.12.2023

Этот материал был создан при поддержке и спонсорстве компании iBase.ru, которая разрабатывает инструменты Firebird SQL для предприятий и предоставляет сервис технической поддержки для Firebird SQL.

Предисловие

Недавно вышел Release Candidate СУБД Firebird 5.0, а это обозначает, что пришло время ознакомиться с новыми возможностями предстоящего релиза. Это восьмой основной выпуск СУБД Firebird, разработка которого началась в мае 2021 года.

В Firebird 5.0 команда разработчиков сосредоточила свои усилия на повышение производительности СУБД в различных аспектах, таких как:

- параллельное выполнение для распространённых задач: backup, restore, sweep, создание и перестроение индекса;
- улучшение масштабирования в многопользовательской среде;
- ускорение повторной подготовки запросов (кеш компилированных запросов);
- улучшение оптимизатора;
- улучшение алгоритма сжатия записей;
- поиск узких мест с помощью плагина профилирования.

Поскольку объём материала довольно большой, то я разделю описание новых функций на несколько частей:

- улучшение в оптимизаторе запросов;
- новые возможности в языке SQL Firebird 5.0;
- другие функции появившиеся в Firebird 5.0;
- поиск узких места с помощью плагина PSQL профилирования.

В прошлый раз я рассказал о том, что нового появилось в языке SQL Firebird 5.0. Теперь подробней поговорим о предложении `SKIP LOCKED`.

Использованием конструкции SKIP LOCKED в Firebird 5.0

При разработке бизнес логики часто возникает задача по организации очередей обработки некоторых заданий. В этом случае один или несколько постановщиков ставят задания в очередь, а исполнители берут свободное невыполненное задание из очереди и выполняют его, после чего обновляют статус задания. Если исполнитель всего один, то проблем не возникает. С увеличением количества исполнителей возникает конкуренция за задачу и конфликты между исполнителями.

Подготовка базы данных

Попробуем реализовать очередь обработки заданий. Для этого создадим тестовую базу данных и создадим в ней таблицу QUEUE_TASK. В эту таблицу постановщики будут добавлять задачи, а исполнители брать свободные задачи и выполнять их. Скрипт создания базы данных с комментариями приведён ниже:

```
CREATE DATABASE 'inet://localhost:3055/c:\fbdata\5.0\queue.fdb'
USER SYSDBA password 'masterkey'
DEFAULT CHARACTER SET UTF8;

CREATE DOMAIN D_QUEUE_TASK_STATUS
AS SMALLINT CHECK(VALUE IN (0, 1));

COMMENT ON DOMAIN D_QUEUE_TASK_STATUS IS 'Статус завершения задачи';

CREATE TABLE QUEUE_TASK (
  ID BIGINT GENERATED BY DEFAULT AS IDENTITY NOT NULL,
  NAME VARCHAR(50) NOT NULL,
  STARTED BOOLEAN DEFAULT FALSE NOT NULL,
  WORKER_ID BIGINT,
  START_TIME TIMESTAMP,
  FINISH_TIME TIMESTAMP,
  FINISH_STATUS D_QUEUE_TASK_STATUS,
  STATUS_TEXT VARCHAR(100),
  CONSTRAINT PK_QUEUE_TASK PRIMARY KEY(ID)
);

COMMENT ON TABLE QUEUE_TASK IS 'Очередь задач';
COMMENT ON COLUMN QUEUE_TASK.ID IS 'Идентификатор задачи';
COMMENT ON COLUMN QUEUE_TASK.NAME IS 'Имя задачи';
COMMENT ON COLUMN QUEUE_TASK.STARTED IS 'Признак того что задача взята в обработку';
COMMENT ON COLUMN QUEUE_TASK.WORKER_ID IS 'Идентификатор исполнителя задачи';
COMMENT ON COLUMN QUEUE_TASK.START_TIME IS 'Время начала выполнения задачи';
COMMENT ON COLUMN QUEUE_TASK.FINISH_TIME IS 'Время завершения выполнения задачи';
COMMENT ON COLUMN QUEUE_TASK.FINISH_STATUS IS 'Статус с которым завершилось выполнение задачи 0 - успешно, 1 - с ошибкой
';
COMMENT ON COLUMN QUEUE_TASK.STATUS_TEXT IS 'Текст статуса. Если задача выполнена без ошибок, то "OK", иначе текст
ошибки';
```

Для добавления новой задачи достаточно выполнить оператор

```
INSERT INTO QUEUE_TASK(NAME) VALUES (?)
```

В данном случае мы передаём только имя задачи, на практике параметров может быть больше.

Каждый исполнитель должен выбрать одну свободную задачу и установить её признак "Взята в обработку".

Получить свободную задачу можно с помощью следующего запроса:

```
SELECT ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY ID
FETCH FIRST ROW ONLY
```

Далее исполнитель помечает задачу как "Взята в обработку", устанавливает время старта задачи и идентификатор исполнителя. Это делается запросом:

```
UPDATE QUEUE_TASK
SET
    STARTED = TRUE,
    WORKER_ID = ?,
    START_TIME = CURRENT_TIMESTAMP
WHERE ID = ?
```

После того как задача взята в обработку, начинается собственно само выполнение задачи. Когда задача выполнена необходимо установить время завершения задачи и её статус. Выполнение задачи может завершиться с ошибкой, в этом случае устанавливается соответствующий статус и сохраняется текст ошибки.

```
UPDATE QUEUE_TASK
SET
    FINISH_STATUS = ?,
    STATUS_TEXT = ?,
    FINISH_TIME = CURRENT_TIMESTAMP
WHERE ID = ?
```

Скрипт моделирующий очередь заданий

Попробуем проверить нашу идею. Для этого напишем простой скрипт на языке Python.

Для написания скрипта нам потребуется установить две библиотеки:

```
pip install firebird-driver
pip install prettytable
```

Теперь можно приступить к написанию скрипта. Скрипт написан для запуска под Windows, однако его можно запускать и под Linux изменив некоторые константы и путь к библиотеке `fbclient`. Сохраним написанный скрипт его в файл `queue_exec.py`:

```
#!/usr/bin/python3

import concurrent.futures as pool
import logging
import random
import time

from firebird.driver import connect, DatabaseError
from firebird.driver import driver_config
from firebird.driver import tpb, Isolation, TraAccessMode
from firebird.driver.core import TransactionManager
from prettytable import PrettyTable

driver_config.fb_client_library.value = "c:\\firebird\\5.0\\fbclient.dll"

DB_URI = 'inet://localhost:3055/d:\\fbdata\\5.0\\queue.fdb'
DB_USER = 'SYSDBA'
DB_PASSWORD = 'masterkey'
DB_CHARSET = 'UTF8'

WORKERS_COUNT = 4  # Количество исполнителей
WORKS_COUNT = 40   # Количество задач

# set up logging to console
stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.INFO)

logging.basicConfig(level=logging.DEBUG,
                    handlers=[stream_handler])

class Worker:
    """Класс Worker представляет собой исполнителя задачи"""

    def __init__(self, worker_id: int):
        self.worker_id = worker_id

    @staticmethod
    def __next_task(tnx: TransactionManager):
        """Извлекает следующую задачу из очереди.

        Arguments:
            tnx: Транзакция в которой выполняется запрос
        """
        cur = tnx.cursor()
```

```
        cur.execute("""
            SELECT ID, NAME
            FROM QUEUE_TASK
            WHERE STARTED IS FALSE
            ORDER BY ID
            FETCH FIRST ROW ONLY
        """)

        row = cur.fetchone()
        cur.close()
        return row

    def __on_start_task(self, tnx: TransactionManager, task_id: int) -> None:
        """Срабатывает при старте выполнения задачи.

        Устанавливает задаче признак того, что она запущена и время старта.

        Arguments:
            tnx: Транзакция в которой выполняется запрос
            task_id: Идентификатор задачи
        """
        cur = tnx.cursor()
        cur.execute(
            """
            UPDATE QUEUE_TASK
            SET
                STARTED = TRUE,
                WORKER_ID = ?,
                START_TIME = CURRENT_TIMESTAMP
            WHERE ID = ?
            """,
            (self.worker_id, task_id,)
        )

    @staticmethod
    def __on_finish_task(tnx: TransactionManager, task_id: int, status: int, status_text: str) -> None:
        """Срабатывает при завершении выполнения задачи.

        Устанавливает задаче время завершения и статус с которым завершилась задача.

        Arguments:
            tnx: Транзакция в которой выполняется запрос
            task_id: Идентификатор задачи
            status: Код статуса завершения. 0 - успешно, 1 - завершено с ошибкой
            status_text: Текст статуса завершения. При успешном завершении записываем "OK",
                в противном случае текст ошибки.
        """
        cur = tnx.cursor()
        cur.execute(
            """
                UPDATE QUEUE_TASK
                SET
                    FINISH_STATUS = ?,
                    STATUS_TEXT = ?,
                    FINISH_TIME = CURRENT_TIMESTAMP
                WHERE ID = ?
            """,
            (status, status_text, task_id,)
        )

    def on_task_execute(self, task_id: int, name: str) -> None:
        """Этот метод приведён как пример функции выполнения некоторой задачи.
```

```
        В реальных задачах он будет другим и с другим набором параметров.

        Arguments:
            task_id: Идентификатор задачи
            name: Имя задачи
        """
        # выбор случайной задержки
        t = random.randint(1, 4)
        time.sleep(t * 0.01)
        # для демонстрации того, что задача может выполняться с ошибками,
        # генерируем исключение для двух из случайных чисел.
        if t == 3:
            raise Exception("Some error")

    def run(self) -> int:
        """Выполнение задачи"""
        conflict_counter = 0
        # Для параллельного выполнения каждый поток должен иметь своё соединение с БД.
        with connect(DB_URI, user=DB_USER, password=DB_PASSWORD, charset=DB_CHARSET) as con:
            tnx = con.transaction_manager(tpb(Isolation.SNAPSHOT, lock_timeout=0, access_mode=TraAccessMode.WRITE))
            while True:
                # Извлекаем очередную задачу и ставим ей признак того что она выполняется.
                # Поскольку задача может выполниться с ошибкой, то признак старта задачи
                # выставляем в отдельной транзакции.
                tnx.begin()
                try:
                    task_row = self.__next_task(tnx)
                    # Если задачи закончились завершаем поток
                    if task_row is None:
                        tnx.commit()
                        break
                    (task_id, name,) = task_row
                    self.__on_start_task(tnx, task_id)
                    tnx.commit()
                except DatabaseError as err:
                    if err.sqlstate == "40001":
                        conflict_counter = conflict_counter + 1
                        logging.error(f"Worker: {self.worker_id}, Task: {self.worker_id}, Error: {err}")
                    else:
                        logging.exception('')
                    tnx.rollback()
                    continue

                # Выполняем задачу
                status = 0
                status_text = "OK"
                try:
                    self.on_task_execute(task_id, name)
                except Exception as err:
                    # Если при выполнении возникла ошибка,
                    # то ставим соответствующий код статуса и сохраняем текст ошибки.
                    status = 1
                    status_text = f"{err}"
                    # logging.error(status_text)

                # Сохраняем время завершения задачи и записываем статус её завершения.
                tnx.begin()
                try:
                    self.__on_finish_task(tnx, task_id, status, status_text)
                    tnx.commit()
                except DatabaseError:
                    if err.sqlstate == "40001":
                        conflict_counter = conflict_counter + 1
                        logging.error(f"Worker: {self.worker_id}, Task: {self.worker_id}, Error: {err}")
```

```
                    else:
                        logging.exception('')
                    tnx.rollback()

        return conflict_counter

def main():
    print(f"Start execute script. Works: {WORKS_COUNT}, workers: {WORKERS_COUNT}\n")

    with connect(DB_URI, user=DB_USER, password=DB_PASSWORD, charset=DB_CHARSET) as con:
        # Чистим предыдущие задачи
        con.begin()
        with con.cursor() as cur:
            cur.execute("DELETE FROM QUEUE_TASK")
        con.commit()
        # Постановщик ставит 40 задач
        con.begin()
        with con.cursor() as cur:
            cur.execute(
                """
                EXECUTE BLOCK (CNT INTEGER = ?)
                AS
                DECLARE I INTEGER;
                BEGIN
                  I = 0;
                  WHILE (I < CNT) DO
                  BEGIN
                    I = I + 1;
                    INSERT INTO QUEUE_TASK(NAME)
                    VALUES ('Task ' || :I);
                  END
                END
                """,
                (WORKS_COUNT,)
            )
        con.commit()

    # создаём исполнителей
    workers = map(lambda worker_id: Worker(worker_id), range(WORKERS_COUNT))
    with pool.ProcessPoolExecutor(max_workers=WORKERS_COUNT) as executer:
        features = map(lambda worker: executer.submit(worker.run), workers)
        conflicts = map(lambda feature: feature.result(), pool.as_completed(features))
        conflict_count = sum(conflicts)

    # считаем статистику
    with connect(DB_URI, user=DB_USER, password=DB_PASSWORD, charset=DB_CHARSET) as con:
        cur = con.cursor()
        cur.execute("""
            SELECT
              COUNT(*) AS CNT_TASK,
              COUNT(*) FILTER(WHERE STARTED IS TRUE AND FINISH_TIME IS NULL) AS CNT_ACTIVE_TASK,
              COUNT(*) FILTER(WHERE FINISH_TIME IS NOT NULL) AS CNT_FINISHED_TASK,
              COUNT(*) FILTER(WHERE FINISH_STATUS = 0) AS CNT_SUCCESS,
              COUNT(*) FILTER(WHERE FINISH_STATUS = 1) AS CNT_ERROR,
              AVG(DATEDIFF(MILLISECOND FROM START_TIME TO FINISH_TIME)) AS AVG_ELAPSED_TIME,
              DATEDIFF(MILLISECOND FROM MIN(START_TIME) TO MAX(FINISH_TIME)) AS SUM_ELAPSED_TIME,
              CAST(? AS BIGINT) AS CONFLICTS
            FROM QUEUE_TASK
        """, (conflict_count,))
        row = cur.fetchone()
        cur.close()

        stat_columns = ["TASKS", "ACTIVE_TASKS", "FINISHED_TASKS", "SUCCESS", "ERROR", "AVG_ELAPSED_TIME",
                        "SUM_ELAPSED_TIME", "CONFLICTS"]
```

```
        stat_table = PrettyTable(stat_columns)
        stat_table.add_row(row)
        print("\nStatistics:")
        print(stat_table)

        cur = con.cursor()
        cur.execute("""
            SELECT
              ID,
              NAME,
              STARTED,
              WORKER_ID,
              START_TIME,
              FINISH_TIME,
              FINISH_STATUS,
              STATUS_TEXT
            FROM QUEUE_TASK
        """)
        rows = cur.fetchall()
        cur.close()

        columns = ["ID", "NAME", "STARTED", "WORKER", "START_TIME", "FINISH_TIME",
                   "STATUS", "STATUS_TEXT"]

        table = PrettyTable(columns)
        table.add_rows(rows)
        print("\nTasks:")
        print(table)

if __name__ == "__main__":
    main()
```

В этом скрипте постановщик ставит 40 задач, которые должны выполнить 4 исполнителя. Каждый исполнитель работает в собственном потоке. По результатам работы скрипта выводится статистика выполнения задач, а также количество конфликтов и сами задачи.

Попробуем запустит наш скрипт:

```
python ./queue_exec.py
```

```
Start execute script. Works: 40, workers: 4

ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95695
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95697
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95703
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95706
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95713
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95722
ERROR:root:Worker: 3, Task: 3, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95722
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95722
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95728
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95734
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95736
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95741
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95744
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95749

Statistics:
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+
| TASKS | ACTIVE_TASKS | FINISHED_TASKS | SUCCESS | ERROR | AVG_ELAPSED_TIME | SUM_ELAPSED_TIME | CONFLICTS |
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+
|   40  |      0       |       40       |    28   |   12  |       43.1       |       1353       |     14    |
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+

Tasks:
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
|  ID  |   NAME  | STARTED | WORKER |         START_TIME      |        FINISH_TIME      | STATUS | STATUS_TEXT |
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
| 1341 |  Task 1 |   True  |   0    | 2023-07-06 15:35:29.9800 | 2023-07-06 15:35:30.0320 |    1   |  Some error |
| 1342 |  Task 2 |   True  |   0    | 2023-07-06 15:35:30.0420 | 2023-07-06 15:35:30.0800 |    1   |  Some error |
| 1343 |  Task 3 |   True  |   0    | 2023-07-06 15:35:30.0900 | 2023-07-06 15:35:30.1130 |    0   |      OK     |
| 1344 |  Task 4 |   True  |   0    | 2023-07-06 15:35:30.1220 | 2023-07-06 15:35:30.1450 |    0   |      OK     |
...
```

Из результатов выполнения скрипта видно, что 4 исполнителя постоянно конфликтуют за задачу. Чем быстрее выполняется задача и чем больше будет исполнителей, тем выше будет вероятность конфликтов.

Фраза SKIP LOCKED

Как же изменить наше решение, чтобы оно работало эффективно и без ошибок? Тут нам на помощь приходит новая конструкция `SKIP LOCKED` из Firebird 5.0.

Фраза `SKIP LOCKED` позволяет пропускать уже заблокированные записи, позволяя тем самым работать без конфликтов. Она может применяться в запросах, в которых есть вероятность возникновения конфликта обновления, то есть в запросах `SELECT ⋯ WITH LOCK`, `UPDATE` и `DELETE`. Посмотрим на её синтаксис:

```
SELECT
  [FIRST ...]
  [SKIP ...]
  FROM <sometable>
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [{ ROWS ... } | {OFFSET ...} | {FETCH ...}]
  [FOR UPDATE [OF ...]]
  [WITH LOCK [SKIP LOCKED]]
```

```
UPDATE <sometable>
  SET ...
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]
```

```
DELETE FROM <sometable>
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]
```

Очередь заданий без конфликтов

Попробуем исправить наш скрипт, так чтобы исполнители не конфликтовали за задачи.

Для этого нам необходимо немного переписать запрос в методе `__next_task` класса `Worker`.

```
    @staticmethod
    def __next_task(tnx: TransactionManager):
        """Извлекает следующую задачу из очереди.

        Arguments:
            tnx: Транзакция в которой выполняется запрос
        """
        cur = tnx.cursor()

        cur.execute("""
            SELECT ID, NAME
            FROM QUEUE_TASK
            WHERE STARTED IS FALSE
            ORDER BY ID
            FETCH FIRST ROW ONLY
            FOR UPDATE WITH LOCK SKIP LOCKED
        """)

        row = cur.fetchone()
        cur.close()
        return row
```

Попробуем запустит исправленный скрипт:

```
python ./queue_exec.py
```

```
Start execute script. Works: 40, workers: 4

Statistics:
+-------+--------------+---------------+---------+-------+-----------------+-----------------+-----------+
| TASKS | ACTIVE_TASKS | FINISHED_TASKS | SUCCESS | ERROR | AVG_ELAPSED_TIME | SUM_ELAPSED_TIME | CONFLICTS |
+-------+--------------+---------------+---------+-------+-----------------+-----------------+-----------+
|   40  |      0       |       40      |    32   |   8   |       39.1      |       1048      |     0     |
+-------+--------------+---------------+---------+-------+-----------------+-----------------+-----------+

Tasks:
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
|  ID  |   NAME  | STARTED | WORKER |         START_TIME      |        FINISH_TIME      | STATUS | STATUS_TEXT |
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
| 1381 |  Task 1 |   True  |   0    | 2023-07-06 15:57:22.0360 | 2023-07-06 15:57:22.0740 |   0    |      OK     |
| 1382 |  Task 2 |   True  |   0    | 2023-07-06 15:57:22.0840 | 2023-07-06 15:57:22.1130 |   0    |      OK     |
| 1383 |  Task 3 |   True  |   0    | 2023-07-06 15:57:22.1220 | 2023-07-06 15:57:22.1630 |   0    |      OK     |
| 1384 |  Task 4 |   True  |   0    | 2023-07-06 15:57:22.1720 | 2023-07-06 15:57:22.1910 |   0    |      OK     |
| 1385 |  Task 5 |   True  |   0    | 2023-07-06 15:57:22.2020 | 2023-07-06 15:57:22.2540 |   0    |      OK     |
| 1386 |  Task 6 |   True  |   0    | 2023-07-06 15:57:22.2620 | 2023-07-06 15:57:22.3220 |   0    |      OK     |
| 1387 |  Task 7 |   True  |   0    | 2023-07-06 15:57:22.3300 | 2023-07-06 15:57:22.3790 |   1    |  Some error |
...
```

На этот раз никаких конфликтов нет. Таким образом, в Firebird 5.0 вы можете использовать фразу `SKIP LOCKED` для того, чтобы избежать ненужных конфликтов обновлений.

Заключение

Нашу очередь заданий можно ещё немного улучшить. Давайте посмотрим на план выполнения запроса

```
SELECT
  ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY ID
FETCH FIRST ROW ONLY
FOR UPDATE WITH LOCK SKIP LOCKED
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "PK_QUEUE_TASK" Full Scan
```

Этот план выполнения не очень хороший. Запись из таблицы QUEUE_TASK извлекается с помощью навигации по индексу, однако сканирование индекса полное. Если таблицу QUEUE_TASK не очищать как мы это делали в нашем скрипте, то со временем выборка необработанных задач будет становиться всё медленней и медленней.

Можно создать индекс для поля STARTED. Если постановщик постоянно добавляет новые задачи, а исполнители выполняют их, то количество не начатых задач всегда меньше, количества завершённых, таким образом, этот индекс будет эффективно фильтровать задачи. Проверим это утверждение:

```
CREATE INDEX IDX_QUEUE_TASK_INACTIVE ON QUEUE_TASK(STARTED);

SELECT
  ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY ID
FETCH FIRST ROW ONLY
FOR UPDATE WITH LOCK SKIP LOCKED;
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "PK_QUEUE_TASK" Full Scan
                        -> Bitmap
                            -> Index "IDX_QUEUE_TASK_INACTIVE" Range Scan (full match)
```

Это действительно так, но теперь используется два индекса, один для фильтрации, а второй для навигации.

Можно пойти дальше и создать композитный индекс:

```
DROP INDEX IDX_QUEUE_TASK_INACTIVE;

CREATE INDEX IDX_QUEUE_TASK_INACTIVE ON QUEUE_TASK(STARTED, ID);
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "IDX_QUEUE_TASK_INACTIVE" Range Scan (partial match: 1/2)
```

Это будет эффективнее поскольку используется только один индекс для навигации, и он сканируется частично. Однако у такого индекса есть существенный недостаток, он не будет компактным.

Для решения этой проблемы можно задействовать ещё одну новую возможность из Firebird 5.0, так называемые частичные индексы.

> Частичный индекс — это индекс, который строится по подмножеству строк таблицы, определяемому условным выражением (оно называется предикатом частичного индекса). Такой индекс содержит записи только для строк, удовлетворяющих предикату.

Давайте попробуем построить такой индекс:

```
DROP INDEX IDX_QUEUE_TASK_INACTIVE;

CREATE INDEX IDX_QUEUE_TASK_INACTIVE ON QUEUE_TASK (STARTED, ID) WHERE (STARTED IS FALSE);

SELECT
  ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY STARTED, ID
FETCH FIRST ROW ONLY
FOR UPDATE WITH LOCK SKIP LOCKED
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "IDX_QUEUE_TASK_INACTIVE" Full Scan
```

Запись из таблицы `QUEUE_TASK` извлекается с помощью навигации по индексу `IDX_QUEUE_TASK_INACTIVE`. Несмотря на то, что сканирование индекса полное, сам по себе индекс очень компактный, поскольку содержит только ключи для которых выполняется условие `STARTED IS FALSE`. Таких записей в нормальной очереди задач всегда сильно меньше, чем записей с выполненными задачами.

В этой статье мы показали как применять новую функциональность `SKIP LOCKED`, которая появилась в Firebird 5.0. Кроме того, немного рассказано о "частичных индексах", которые тоже появилась в Firebird 5.0. Частичные индексы могут также использоваться для сложных "ограничений" уникальности. Но об этом в следующий раз.

DDL скрипт для создания базы данных, а также Python скрипт с эмуляцией очереди задач можно скачать по следующим ссылкам:

- ddl.sql
- queue_exec.py