Параллельное чтение данных в СУБД Firebird

Симонов Денис, Влад Хорсун

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

В Firebird 5.0 появилась возможность использовать параллелизм при создании резервной копии с помощью утилиты `gbak`. Надо отметить, что первоначально эта функция появилась в HQbird 2.5, и уже после обкатке её реальными заказчиками была портирована в Firebird 5.0.

В этой статье мы познакомимся с некоторыми механизмами, которые используется при параллельном создании резервных копий внутри утилиты `gbak`, а также покажем как их можно использовать в ваших приложениях для параллельного чтения данных.

Важно отметить, что здесь идёт речь не о параллельном сканировании таблиц внутри движка Firebird при выполнении SQL запросов, а чтении данных внутри вашего приложения параллельными потоками.

Для демонстрации параллельного чтения данных из СУБД Firebird я написал простенькую утилиту, которые экспортирует данные из одной или нескольких таблиц в формат CSV. Её описание и сходный код можно найти по адресу https://github.com/IBSurgeon/FBCSVExport.git

Организация параллельного чтения

И так, давайте подумаем как читать данные из нескольких таблиц параллельно. Для начала надо отметить, что Firebird, позволяет выполнять запросы параллельно только если каждый из запросов будет выполняться в отдельном соединении.

Для этой цели необходимо сделать пул рабочих потоков. Основной поток приложения также является рабочим потоком, поэтому количество дополнительных рабочих потоков должно быть равно N - 1, где N - количество параллельных обработчиков. Каждый рабочий поток будет работать с собственным соединением и транзакцией.

И тут возникает первая проблема, а как обеспечить согласованность прочитанных данных?

Согласованное чтение данных

Поскольку каждый рабочий поток использует собственное соединение и собственную транзакцию, то возникает проблема несогласованного чтения - если таблица параллельно меняется, то прочитанные данные могут быть несогласованными. В однопоточном режиме `gbak` использовал транзакцию с режимом изолированности SNAPSHOT, что позволяло читать согласованную информацию, на начало старта SNAPSHOT транзакции. Но здесь у нас несколько транзакций и необходимо, чтобы они видели один и тот же "снимок", для того чтобы они читали одни и те же неизменные данные.

Механизм создания общего снимка для разных транзакций с режимом изолированности SNAPSHOT появился в Firebird 4.0 (первоначально HQBird 2.5, но в Firebird 4.0/HQBird 4.0 он более простая и эффективная). Для создания общего снимка существует два способа:

1. Через SQL.
   - получаем номер моментального снимка из главной транзакции, то есть транзакции с которой работает основной рабочий поток.

     ```
     SELECT RDB$GET_CONTEXT('SYSTEM', 'SNAPSHOT_NUMBER') FROM RDB$DATABASE
     ```

   - стартуем другие транзакции запросом:

     ```
     SET TRANSACTION SNAPSHOT AT NUMBER snapshot_number
     ```

     где `snapshot_number` номер моментального снимка.
2. Через API.
   - получить номер моментального снимка из главной транзакции, то есть транзакции с которой работает основной рабочий поток, используя функцию `isc_transaction_info` или `ITransaction.getInfo` с тегом `fb_info_tra_snapshot_number`;
   - стартуем другие транзакции, передавая туда номер снимка с помощью тега `isc_tpb_at_snapshot_number`.

CSVExport так же как и gbak использует второй подход. Эти подходы можно комбинировать, например получив номер моментального снимка запросом, но передавая этот номер для старта других транзакций с помощью API или наоборот.

В CSVExport получение номера моментального снимка реализовано следующим образом:

```
ISC_INT64 getSnapshotNumber(Firebird::ThrowStatusWrapper* status, Firebird::ITransaction* tra)
{
    ISC_INT64 ret = 0;
    unsigned char in_buf[] = { fb_info_tra_snapshot_number, isc_info_end };
    unsigned char out_buf[16] = { 0 };

    tra->getInfo(status, sizeof(in_buf), in_buf, sizeof(out_buf), out_buf);

    unsigned char* p = out_buf, * e = out_buf + sizeof(out_buf);
    while (p < e)
    {
        short len = 0;
        switch (*p++)
        {
        case isc_info_error:
        case isc_info_end:
            p = e;
            break;

        case fb_info_tra_snapshot_number:
            len = static_cast<short>(isc_vax_integer(reinterpret_cast<char*>(p), 2));
            p += 2;
            ret = isc_portable_integer(p, len);
            p += len;
            break;
        }
    }
    return ret;
}
```

А старт транзакции с полученным номером моментального снимка так:

```
Firebird::AutoDispose<Firebird::IXpbBuilder> tpbWorkerBuilder(fbUtil->getXpbBuilder(&status, Firebird::IXpbBuilder::TPB,
nullptr, 0));
tpbWorkerBuilder->insertTag(&status, isc_tpb_concurrency);
tpbWorkerBuilder->insertBigInt(&status, isc_tpb_at_snapshot_number, snapshotNumber);

Firebird::AutoRelease<Firebird::ITransaction> workerTra(
    workerAtt->startTransaction(
        &status,
        tpbWorkerBuilder->getBufferLength(&status),
        tpbWorkerBuilder->getBuffer(&status)
    )
);
```

Теперь данные прочитанные из разных соединений будут согласованными, можно распределять нагрузку по рабочим потокам.

Как именно распределить нагрузку между рабочими потоками? В случае полного экспорта таблиц или резервной копии самым простым вариантом является - один рабочий поток на одну таблицу. Но при таком подходе есть одна проблема: а что если у нас много маленьких и одна большая таблица или вообще таблица всего одна и она огромная? В этом случае какому-то потоку достанется большая таблица, а остальные потоки будут простаивать. Для того чтобы этого не происходило необходимо обрабатывать большую таблицу по частям.

NOTE	Дальнейшие рассуждения касаются только полного чтения таблиц, если вы хотите организовать параллельное чтение из некоторого запроса (представления), то вам необходимо продумать как разбивать этот запрос на части самостоятельно.

Разбиение большой таблицы на части

Допустим у нас всего одна большая таблица, которую хотим прочитать целиком и как можно быстрее. Предлагается разбить её на несколько частей и каждую часть читать из своего потока независимо. Каждый поток должен иметь свой коннект с БД.

В этом случае возникают следующие вопросы:

- на сколько частей разбить таблицу?
- как это лучше сделать?

Ответим на эти вопросы по порядку.

На сколько частей разбить таблицу

Для начала предположим идеальный вариант — сервер и клиент больше ничем не заняты, то есть все CPU полностью в нашем распоряжении. Тогда я бы рекомендовал:

а) взять за максимальное количество частей двойное количество ядер на сервере. Обычно рекомендуется разбивать подобные параллельные задачи согласно количеству ядер, но мы точно знаем, что у нас будут задержки связанные с IO, поэтому можем себе позволить некоторое превышение. Более точно определить может только практика.

б) учитывать количество ядер на клиенте: если на сервере их сильно больше (обычная ситуация), то возможно будет иметь смысл сильнее ограничить количество частей разбиения, чтобы не перегружать клиента (он всё равно больше не сможет обработать, а расходы на переключение потоков никуда не денутся). Точнее можно будет решить, наблюдая за загрузкой CPU клиента и сервера — если на клиенте 100%, а на сервере заметно меньше, то имеет смысл уменьшить количество частей.

в) если клиент и сервер — один и тот же хост, то см. (а)

Если клиент и/или сервер заняты чем-то ещё, то возможно придётся ещё уменьшить количество частей. Так же на это может повлиять способность дисков на сервере обрабатывать множество IO запросов одновременно (наблюдать за размером очереди и временем отклика).

Как лучше разбить таблицу на части

Для любой параллельной обработки важно обеспечить равномерное распределение заданий по обработчикам и свести к минимуму их взаимную синхронизацию. Причём нужно помнить, что синхронизация обработчиков может происходить как на стороне сервера, так и на стороне клиента. Например — не стоит нескольким обработчикам использовать один и тот же коннект к БД. Менее очевидный пример: плохо, если разные обработчики будут читать записи с одних и тех же страниц БД. Например, когда два обработчика читают чётные и нечётные записи соответственно — совсем не эффективно. Синхронизация на клиенте может возникнуть при раздаче заданий, при обработке полученных данных (при выделение памяти под результаты) и так далее.

Для "честного" разбиения одна из проблем в том, что клиенту не известно как именно распределены записи по страницам (и по ключам индексов), сколько вообще есть записей (для больших таблиц дорого считать заранее), да и сколько есть страниц — тоже дорого посчитать. Серверу это обычно тоже не известно.

Ниже описано как это делает `gbak`.

В `gbak` единицей работы является набор записей со страниц данных (DP), принадлежащих одной и той же странице указателей (pointer page, или PP). Это, с одной стороны, достаточно большое количество записей, чтобы обеспечить обработчик работой без необходимости часто просить новый кусок данных (синхронизация). С другой стороны, даже если такие наборы записей будут иметь не очень одинаковый размер, их (наборов) количество позволит относительно равномерно загрузить работой все обработчики. То есть вполне возможны случаи, когда один обработчик прочитает N записей с одной PP, а другой — M записей, и M будет достаточно отличаться от N. Но это не проблема. Такой подход не идеален, но он весьма прост в реализации и обычно достаточно эффективен, по крайней мере на больших таблицах — с десятками или сотнями (и больше) PP.

Теперь необходимо получить количество PP (Pointer Pages) для заданной таблицы. Это довольно легко, а главное быстро, можно вычислить из таблицы `RDB$PAGES`:

```
SELECT RDB$PAGE_SEQUENCE
FROM RDB$PAGES
WHERE RDB$RELATION_ID = ? AND RDB$PAGE_TYPE = 4
ORDER BY RDB$PAGE_SEQUENCE DESC ROWS 1
```

Далее можно было бы просто поделить количество PP на количество обработчиков, и выдать каждому свой кусок. Но, как я писал выше, нет никакой гарантии, что такие "большие" куски будут означать одинаковый объём работы. Нам же не интересно наблюдать как 15 обработчиков закончили свою работу и простаивают, а 16-ый долго читает свои 100500 записей.

Поэтому в `gbak` это сделано иначе. Там есть координатор работы, который выдаёт каждому обработчику по 1-ой PP за раз. Координатор знает сколько PP есть всего и сколько уже выдано в работу. Когда обработчик прочитает свои записи, он обращается к координатору за новым номером PP. Это продолжается до тех пор, пока не закончатся PP и пока есть

работающие обработчики. Конечно, такое взаимодействие обработчиков с координатором требует синхронизации. Опыт показывает, что объём работы, заданный одной PP, позволяет не синхронизироваться слишком часто. Такой подход позволяет достаточно равномерно загрузить работой все обработчики (а значит и ядра CPU) независимо от реального количества записей, принадлежащих каждой PP.

Как же обработчик читает записи со своей PP? Для этого начиная с Firebird 4.0 (впервые появилось в HQBird 2.5) есть встроенная функция `MAKE_DBKEY()`. С её помощью можно получить `RDB$DB_KEY` (физический номер записи) для первой записи на указанной PP. И с помощью таких `RDB$DB_KEY` и осуществляется отбор нужных записей:

```
SELECT *
FROM relation
WHERE RDB$DB_KEY >= MAKE_DBKEY(:rel_id, 0, 0, :loPP)
    AND RDB$DB_KEY < MAKE_DBKEY(:rel_id, 0, 0, :hiPP)
```

Например, если задать loPP = 0 и hiPP = 1, то будут прочитаны все записи с PP = 0, и только из неё.

Теперь, когда есть представление о том как работает `gbak` можно перейти к описанию реализации утилиты `CSVExport`.

Реализация утилиты CSVExport

Утилита CSVExport предназначена для экспорта данных из таблиц БД Firebird в формат CSV.

Каждая таблица экспортируется в файл с именем <tablename>.csv. В обычном (однопоточном режиме) данные из таблиц экспортируется последовательно в алфавитном порядке имени таблиц.

В параллельном режиме, таблицы экспортируются параллельно, каждая таблица в отдельном потоке. Если таблица очень большая, то она разбивается на части, и каждая часть экспортируется в отдельном потоке. Для каждой части большой таблицы создаётся отдельный файл с именем <tablename>.csv.partN, где N - номер части. Когда все части большой таблицы экспортированы, файлы частей сливаются в общий файл с именем <tablename>.csv.

Для того, чтобы указать какие именно таблицы будут экспортированы используется регулярное выражение. Возможен экспорт только обычных таблиц (системные таблицы, GTT, представления, внешние таблицы не поддерживаются). Регулярные выражения должны быть в SQL синтаксисе, то есть такие, которые используются в предикате SIMILAR TO.

Для отбора списка экспортируемых таблиц, а также списка их PP в многопоточном режиме я использую следующий запрос:

```
SELECT
    R.RDB$RELATION_ID AS RELATION_ID,
    TRIM(R.RDB$RELATION_NAME) AS RELATION_NAME,
    P.RDB$PAGE_SEQUENCE AS PAGE_SEQUENCE,
    COUNT(P.RDB$PAGE_SEQUENCE) OVER(PARTITION BY R.RDB$RELATION_NAME) AS PP_CNT
FROM RDB$RELATIONS R
JOIN RDB$PAGES P ON P.RDB$RELATION_ID = R.RDB$RELATION_ID
WHERE R.RDB$SYSTEM_FLAG = 0 AND
      R.RDB$RELATION_TYPE = 0 AND
      P.RDB$PAGE_TYPE = 4 AND
      TRIM(R.RDB$RELATION_NAME) SIMILAR TO CAST(? AS VARCHAR(8191))
ORDER BY R.RDB$RELATION_NAME, P.RDB$PAGE_SEQUENCE
```

В однопоточном режиме этот запрос можно упростить до

```
SELECT
    R.RDB$RELATION_ID AS RELATION_ID,
    TRIM(R.RDB$RELATION_NAME) AS RELATION_NAME,
    0 AS PAGE_SEQUENCE,
    1 AS PP_CNT
FROM RDB$RELATIONS R
WHERE R.RDB$SYSTEM_FLAG = 0 AND
      R.RDB$RELATION_TYPE = 0 AND
      TRIM(R.RDB$RELATION_NAME) SIMILAR TO CAST(? AS VARCHAR(8191))
ORDER BY R.RDB$RELATION_NAME
```

В однопоточном режиме значения полей PAGE_SEQUENCE и PP_CNT не используются, они добавлены в запрос с целью унификации выходных сообщений.

Результат этого запроса складывается в вектор структур:

```
struct TableDesc
{
    TableDesc() = default;
    TableDesc(const OutputRecord& rec)
        : releation_id(rec->releation_id)
        , relation_name(rec->relation_name.str, rec->relation_name.length)
        , page_sequence(rec->page_sequence)
        , pp_cnt(rec->pp_cnt)
    {}

    short releation_id;
    std::string relation_name;
    int32_t page_sequence;
    int64_t pp_cnt;
};
```

Этот вектор заполняется при помощи функции объявленной как:

```
std::vector<TableDesc> getTablesDesc(
    Firebird::ThrowStatusWrapper* status,
    Firebird::IAttachment* att,
    Firebird::ITransaction* tra,
    unsigned int sqlDialect,
    const std::string& tableIncludeFilter,
    bool singleWorker = true);
```

Последний параметр `singleWorker` переключает режим заполнения `std::vector<TableDesc>`, если `singleWorker = true`, то используется запрос для однопоточного режима, если `singleWorker = false`, то используется более дорогой и сложный запрос для многопоточного режима. Саму реализацию я не буду приводить, она довольно проста, и вы можете посмотреть её в исходном коде проекта.

Для экспорта таблицы в формат CSV разработан класс CSVExportTable, который содержит следующие методы:

```
    void prepare(Firebird::ThrowStatusWrapper* status, const std::string& tableName,
                 unsigned int sqlDialect, bool withDbkeyFilter = false);

    void printHeader(Firebird::ThrowStatusWrapper* status, csv::CSVFile& csv);

    void printData(Firebird::ThrowStatusWrapper* status, csv::CSVFile& csv, int64_t ppNum = 0);
```

Метод prepare предназначен для построения и подготовки запроса, который используется для экспорта таблицы в формат CSV. Внутренний запрос строится по разному в зависимости от параметра withDbkeyFilter. Если withDbkeyFilter = true, то запрос строится с фильтрацией по диапазону RDB$DB_KEY:

```
SELECT *
FROM tableName
WHERE RDB$DB_KEY >= MAKE_DBKEY('tableName', 0, 0, ?)
  AND RDB$DB_KEY < MAKE_DBKEY('tableName', 0, 0, ?)
```

в противном случае используется упрощённый запрос:

```
SELECT *
FROM tableName
```

Значение параметра withDbkeyFilter устанавливается в true, если используется многопоточный режим, и таблица является большой. Считаем таблицу большой, если pp_cnt > 1.

Метод printHeader предназначен для печати заголовка CSV файла (имён столбцов таблицы).

Метод printData печатает данные таблицы в CSV файл с PP страницы с номером ppNum, если запрос был подготовлен с использованием фильтра по диапазону RDB$DB_KEY, и всех данных таблицы в противном случае.

Теперь посмотрим фрагмент кода для работы в однопоточном режиме

```
...

// Открываем главное соединение
Firebird::AutoRelease<Firebird::IAttachment> att(
    provider->attachDatabase(
        &status,
        m_database.c_str(),
        dbpLength,
        dpb
    )
);

// Стартуем главную транзакцию в режиме изолированности SNAPSHOT
Firebird::AutoDispose<Firebird::IXpbBuilder> tpbBuilder(fbUtil->getXpbBuilder(&status, Firebird::IXpbBuilder::TPB,
nullptr, 0));
tpbBuilder->insertTag(&status, isc_tpb_concurrency);

Firebird::AutoRelease<Firebird::ITransaction> tra(
    att->startTransaction(
        &status,
        tpbBuilder->getBufferLength(&status),
        tpbBuilder->getBuffer(&status)
    )
);
// Получаем список таблиц по регулярному выражению в m_filter.
// m_parallel задаёт количество параллельных потоков, когда она равна 1,
// то используется упрощённый запрос для получения списка таблиц,
// в противном случае, для каждой таблицы формируется список PP и их количество.
auto tables = getTablesDesc(&status, att, tra, m_sqlDialect, m_filter, m_parallel == 1);

if (m_parallel == 1) {
    FBExport::CSVExportTable csvExport(att, tra, fb_master);
    for (const auto& tableDesc : tables) {
        // здесь нет смысла использовать фильтр по диапазону RDB$DB_KEY
        csvExport.prepare(&status, tableDesc.relation_name, m_sqlDialect, false);
        const std::string fileName = tableDesc.relation_name + ".csv";
        csv::CSVFile csv(m_outputDir / fileName);
        if (m_printHeader) {
            csvExport.printHeader(&status, csv);
        }
        csvExport.printData(&status, csv);
    }
}
```

Здесь всё довольно просто и не требует дополнительных пояснений, поэтому перейдем к многопоточной части.

Для того, чтобы экспорт происходил в многопоточном режиме, необходимо создать дополнительные `m_parallel - 1` рабочих потоков. Почему количество дополнительных потоков на 1 меньше? Да потому что основной поток, тоже будет заниматься экспортом данных и он является равноправным с дополнительными потоками. Вынесем общую часть основного и дополнительного потока в отдельную функцию:

```
void ExportApp::exportByTableDesc(Firebird::ThrowStatusWrapper* status, FBExport::CSVExportTable& csvExport, const
TableDesc& tableDesc)
{
    // Если в tableDesc pp_cnt > 1, то она описывает только часть таблицы, и необходимо построить
    // запрос с использованием фильтра по диапазону RDB$DB_KEY.
    bool withDbKeyFilter = tableDesc.pp_cnt > 1;
    csvExport.prepare(status, tableDesc.relation_name, m_sqlDialect, withDbKeyFilter);
    std::string fileName = tableDesc.relation_name + ".csv";
    // Если это не первая часть таблицы, то записываем эту часть в файл <tableName>.csv.part<N>, где
    // N - номер PP. Позднее части таблицы будут соединены в единый файл <tableName>.csv
    if (tableDesc.page_sequence > 0) {
        fileName += ".part_" + std::to_string(tableDesc.page_sequence);
    }
    csv::CSVFile csv(m_outputDir / fileName);
    // Заголовок CSV файла нужно печатать только в первую часть таблицы.
    if (tableDesc.page_sequence == 0 && m_printHeader) {
        csvExport.printHeader(status, csv);
    }
    csvExport.printData(status, csv, tableDesc.page_sequence);
}
```

Описание таблиц или её частей расположено в общем векторе со структурами TableDesc. Из этого вектора каждый рабочий поток берёт таблицу или очередную часть Для предотвращения data races необходимо синхронизация доступа к общему ресурсу. Но сам std::vector<TableDesc> не меняется, поэтому можно синхронизировать только общую переменную, которая является индексом в этом векторе. Это легко сделать используя в качестве такой переменной std::atomic<size_t>.

```
if (m_parallel == 1) {
    ...
}
else {
    // Определяем количество дополнительных рабочих потоков
    const auto workerCount = m_parallel - 1;

    // Получаем номер моментального снимка из основной транзакции
    auto snapshotNumber = getSnapshotNumber(&status, tra);
    // переменная для сохранения исключения внутри потока
    std::exception_ptr exceptionPointer = nullptr;
    std::mutex m;
    // атомарный счётчик
    // является индексом очередной таблицы или её части
    std::atomic<size_t> counter = 0;
    // пул рабочих потоков
    std::vector<std::thread> thread_pool;
    thread_pool.reserve(workerCount);
    for (int i = 0; i < workerCount; i++) {
        // для каждого потока создаём своё соединение
        Firebird::AutoRelease<Firebird::IAttachment> workerAtt(
            provider->attachDatabase(
                &status,
                m_database.c_str(),
                dbpLength,
                dpb
            )
        );
        // и свою транзакцию в которую передаём номер моментального снимка
        // для создания общего снимка
        Firebird::AutoDispose<Firebird::IXpbBuilder> tpbWorkerBuilder(fbUtil->getXpbBuilder(&status, Firebird
::IXpbBuilder::TPB, nullptr, 0));
        tpbWorkerBuilder->insertTag(&status, isc_tpb_concurrency);
```

```
        tpbWorkerBuilder->insertBigInt(&status, isc_tpb_at_snapshot_number, snapshotNumber);

        Firebird::AutoRelease<Firebird::ITransaction> workerTra(
            workerAtt->startTransaction(
                &status,
                tpbWorkerBuilder->getBufferLength(&status),
                tpbWorkerBuilder->getBuffer(&status)
            )
        );
        // создаём поток
        std::thread t([att = std::move(workerAtt), tra = std::move(workerTra), this,
                       &m, &tables, &counter, &exceptionPointer]() mutable {

            Firebird::ThrowStatusWrapper status(fb_master->getStatus());
            try {
                FBExport::CSVExportTable csvExport(att, tra, fb_master);
                while (true) {
                    // увеличиваем атомарный счётчик
                    size_t localCounter = counter++;
                    // если таблицы или их части закончились выходим
                    // из бесконечного цикла и завершаем поток
                    if (localCounter >= tables.size())
                        break;
                    // получаем описание таблицы или её части
                    const auto& tableDesc = tables[localCounter];
                    // и делаем экспорт
                    exportByTableDesc(&status, csvExport, tableDesc);
                }
                if (tra) {
                    tra->commit(&status);
                    tra.release();
                }

                if (att) {
                    att->detach(&status);
                    att.release();
                }
            }
            catch (...) {
                // если возникло исключение, то сохраняем его для
                // последующего выброса в основном потоке
                std::unique_lock<std::mutex> lock(m);
                exceptionPointer = std::current_exception();
            }
        });
        thread_pool.push_back(std::move(t));
    }

    // экспорт в основном потоке
    FBExport::CSVExportTable csvExport(att, tra, fb_master);
    while (true) {
        // увеличиваем атомарный счётчик
        size_t localCounter = counter++;
        if (localCounter >= tables.size())
            break;
        // если таблицы или их части закончились выходим
        // из бесконечного цикла
        const auto& tableDesc = tables[localCounter];
        exportByTableDesc(&status, csvExport, tableDesc);
    }
    // ждём завершения рабочих потоков
    for (auto& th : thread_pool) {
        th.join();
    }
    // если в рабочих потоках было исключение выбрасываем его повторно
    if (exceptionPointer) {
        std::rethrow_exception(exceptionPointer);
```

```
    }
    ...
```

Осталось соединить файлы, которые были созданы для частей таблиц в единой файл для каждой из этих таблиц.

```
for (size_t i = 0; i < tables.size(); i++) {
    const auto& tableDesc = tables[i];
    // если количество PP больше 1,
    // то таблица большая и для неё было несколько частей
    if (tableDesc.pp_cnt > 1) {
        // основной файл для таблицы
        std::string fileName = tableDesc.relation_name + ".csv";
        std::ofstream ofile(m_outputDir / fileName, std::ios::out | std::ios::app);
        i++;
        for (int64_t j = 1; j < tableDesc.pp_cnt; j++, i++) {
            // файлы частей таблицы
            std::string partFileName = fileName + ".part_" + std::to_string(j);
            auto partFilePath = m_outputDir / partFileName;
            std::ifstream ifile(partFilePath, std::ios::in);
            ofile << ifile.rdbuf();
            ifile.close();
            fs::remove(partFilePath);
        }
        ofile.close();
    }
}
```

Ну вот и всё пришло время померить производительность нашей утилиты в однопоточном и многопоточном режиме.

Бенчмарк утилиты CSVExport

Для начала посмотрим на результаты сравнения многопоточного и однопоточного режима экспорта на моём домашнем не самом современном компьютере.

Windows

- Операционная система: Windows 10 x64.
- Процессор: Intel Core i3 8100, 4 ядра, 4 потока.
- Память: 16 Гб
- Дисковая подсистема: NVME SSD (база данных), SATA SSD (папка для размещения CSV файлов).
- Firebird 4.0.4 x64

Результаты:

```
CSVExport.exe -H --table-filter="COLOR|BREED|HORSE|COVER|MEASURE|LAB_LINE|SEX" --parallel=1 \
  -d inet://localhost:3054/horses -u SYSDBA -p masterkey --charset=WIN1251 -o ./single

Elapsed time in milliseconds parallel_part: 35894 ms
Elapsed time in milliseconds: 36317 ms

CSVExport.exe -H --table-filter="COLOR|BREED|HORSE|COVER|MEASURE|LAB_LINE|SEX" --parallel=4 \
  -d inet://localhost:3054/horses -u SYSDBA -p masterkey --charset=WIN1251 -o ./multi

Elapsed time in milliseconds parallel_part: 19259 ms
Elapsed time in milliseconds: 20760 ms

CSVExport.exe -H --table-filter="COLOR|BREED|HORSE|COVER|MEASURE|LAB_LINE|SEX" --parallel=4 \
  -d inet://localhost:3054/horses -u SYSDBA -p masterkey --charset=WIN1251 -o ./multi

Elapsed time in milliseconds parallel_part: 19600 ms
Elapsed time in milliseconds: 21137 ms
```

Из результата тестирования видно, что при использовании двух потоков, ускорении составило 1.8 раза, что является хорошим результатом. Но параллельное выполнение экспорта в 4 потоках, тоже дало ускорение в 1.8 раза. Почему не в 3-4? Дело в том, что сервер Firebird и утилита экспорта запущены на одном и том же компьютере, у которого всего 4 ядра. Таким образом сам сервер Firebird, использует 4 потока для чтения таблицы и утилита CSVExport, тоже использует 4 потока. Очевидно, что в таком случае довольно затруднительно получить ускорение более чем в 2 раза. Поэтому попробуем на другом железе, где количество ядер существенно больше.

Linux

- Операционная система: CentOS 8.

- Процессор: 2 процессора Intel Xeon E5-2603 v4, всего 12 ядер, 12 потоков.
- Память: 32 Гб
- Дисковая подсистема: SAS HDD (RAID 10)
- Firebird 4.0.4 x64

Результаты:

```
[denis@copyserver build]$ ./CSVExport -H --table-filter="COLOR|BREED|HORSE|COVER|MEASURE|LAB_LINE|SEX" --parallel=1 \
  -d inet://localhost/horses -u SYSDBA -p masterkey --charset=UTF8 -o ./single

Elapsed time in milliseconds parallel_part: 57547 ms
Elapsed time in milliseconds: 57595 ms

[denis@copyserver build]$ ./CSVExport -H --table-filter="COLOR|BREED|HORSE|COVER|MEASURE|LAB_LINE|SEX" --parallel=4 \
  -d inet://localhost/horses -u SYSDBA -p masterkey --charset=UTF8 -o ./multi

Elapsed time in milliseconds parallel_part: 17755 ms
Elapsed time in milliseconds: 18148 ms

[denis@copyserver build]$ ./CSVExport -H --table-filter="COLOR|BREED|HORSE|COVER|MEASURE|LAB_LINE|SEX" --parallel=6 \
  -d inet://localhost/horses -u SYSDBA -p masterkey --charset=UTF8 -o ./multi

Elapsed time in milliseconds parallel_part: 13243 ms
Elapsed time in milliseconds: 13624 ms

[denis@copyserver build]$ ./CSVExport -H --table-filter="COLOR|BREED|HORSE|COVER|MEASURE|LAB_LINE|SEX" --parallel=12 \
  -d inet://localhost/horses -u SYSDBA -p masterkey --charset=UTF8 -o ./multi

Elapsed time in milliseconds parallel_part: 12712 ms
Elapsed time in milliseconds: 13140 ms
```

В данном случае оптимальном числом потоков для экспорта является 6 (6 потоков для Firebird и 6 потоков для утилиты CSVExport). При этом удалось получить ускорение в 5 раз, что говорит о достаточно хорошей масштабируемости. Хотелось бы отметить, что для проверки на Linux и Windows использовались идентичные базы данных почти одинакового размера. В одном потоке, на Windows экспорт прошёл почти в 2 раза быстрее, из-за более быстрой дисковой подсистемы. Всё таки NVME диски намного быстрее SAS дисков объединённых в RAID.

Заключение

В этой статье мы научились эффективно использовать читать данные из таблиц СУБД Firebird используя параллелизм, и что самое согласовано читать их. Кроме того, был показан пример как можно использовать некоторые возможности СУБД Firebird для организации такого тения в своих утилитах.

Огромное спасибо Владиславу Хорсуну, автору многопоточного `gbak` в Firebird 5.0 и HQbird, за подробное объяснение принципов работы параллельного резервного копирования. Без него этой статьи не было бы.