Передача BLOB по сети в Firebird

Симонов Денис

Версия 1.0 от 16.03.2025

Содержание

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

У разработчиков приложений, а также администраторов, использующий СУБД Firebird часто возникает вопрос, а что если разместить СУБД Firebird в облаке и работать с ней через интернет канал? Однако, после тестирования такого режима работы многие остаются разочарованными, поскольку скорость обмена данными по сети с высокой латентностью (интернет канал) оставляет желать лучшего. В большинстве случаев скорость выборки данных из курсоров, порождённых SQL запросами устраивает, но как только в таких запросах встречаются BLOB поля (бинарные или тестовые данные) скорость передачи данных падает катастрофически.

В данной статье мы поговорим о том как передаются BLOB по сети, какие проблемы возникают у пользователей в случае использования Firebird в сетях с высокой латентностью (работа через интернет канал), расскажем о способах решения этих проблем, а также об улучшениях в передачи BLOB в последних версиях Firebird (5.0.2 и 5.0.3).

Глава 1. Приложение и база данных для тестирования

Для демонстрации различных способов работы с BLOB полями, а также замеров производительности было написано небольшое тестовое приложение исходные коды которого доступны по адресу https://github.com/IBSurgeon/fb-blob-test. На этой же странице вы можете скачать готовую сборку под Windows x64 и тестовую базу данных.

В данное приложении производится тестирование производительность передачи только текстовых BLOB полей, но те же механизмы могут быть применены и к бинарным BLOB.

Для демонстрации передачи BLOB по сети нам потребуется база данных содержащая таблицы с BLOB полями, причём желательно чтобы размер этих BLOB полей был разный от совсем небольших до средних. Для этой цели можно использовать исходные коды какого нибудь Open Source проекта, например UDR библиотеки lucene_udr.

Содержимое файлов будет хранится в таблице со следующей структурой:

```
CREATE TABLE BLOB_SAMPLE (
    ID         BIGINT GENERATED BY DEFAULT AS IDENTITY,
    FILE_NAME  VARCHAR(255) CHARACTER SET UTF8 NOT NULL,
    CONTENT    BLOB SUB_TYPE TEXT CHARACTER SET UTF8
);

ALTER TABLE BLOB_SAMPLE ADD PRIMARY KEY (ID);
ALTER TABLE BLOB_SAMPLE ADD UNIQUE (FILE_NAME);
```

Поскольку проект не большой, то количество файлов с исходными текстами в нём не так много как хотелось бы. Чтобы результаты тестирования были более наглядны в цифрах, доведём количество записей с BLOB до 10000. Для этого создадим отдельную таблицу BLOB_TEST со следующей структурой:

```
RECREATE TABLE BLOB_TEST (
    ID             BIGINT GENERATED BY DEFAULT AS IDENTITY,
    SHORT_CONTENT  VARCHAR(8191) CHARACTER SET UTF8,
    CONTENT        BLOB SUB_TYPE TEXT CHARACTER SET UTF8,
    SHORT_BLOB     BOOLEAN DEFAULT FALSE NOT NULL,
    CONSTRAINT PK_BLOB_TEST PRIMARY KEY (ID)
);
```

Здесь мы убрали поле для хранения имя файла FILE_NAME, но зато добавили поле SHORT_CONTENT. Будем заполнять это поле, если содержимое BLOB поля CONTENT, может быть целиком сохранено в поле типа VARCHAR(8191) CHARACTER SET UTF8. Также добавим поле SHORT_BLOB, которое является признаком того, что BLOB "короткий" (помещается в VARCHAR). Данные поля потребуются нам при выполнении различных сравнительных тестов.

Итак нам требуется заполнить таблицу BLOB_TEST из таблицы BLOB_SAMPLE, так чтобы в

целевой таблице было 10000 записей. Для этого воспользуемся следующим скриптом:

```
SET TERM ^;

EXECUTE BLOCK
AS
DECLARE I INTEGER = 0;
DECLARE IS_SHORT BOOLEAN;
BEGIN
  WHILE (TRUE) DO
  BEGIN
    FOR
      SELECT
        ID,
        CONTENT,
        CHAR_LENGTH(CONTENT) AS CH_L
      FROM BLOB_SAMPLE
      ORDER BY FILE_NAME
      AS CURSOR C
    DO
    BEGIN
      I = I + 1;
      -- BLOB помещается в строковую переменную 8191
      IS_SHORT = (C.CH_L < 8191);

      INSERT INTO BLOB_TEST (
        SHORT_CONTENT,
        CONTENT,
        SHORT_BLOB
      )
      VALUES (
        IIF(:IS_SHORT, :C.CONTENT, NULL), -- если BLOB короткий пишем в VARCHAR поле
        :C.CONTENT,
        :IS_SHORT
      );
      -- выходим когда вставлено 10000 записей
      IF (I = 10000) THEN EXIT;
    END
  END
END^

SET TERM ;^

COMMIT;
```

База данных с BLOB полями различной длины готова для тестирования.

Чтобы сравнение различных вариантов передачи BLOB полей было честным необходимо "прогреть" страничный кеш, то есть сделать так чтобы в него попали все страницы данных таблицы `BLOB_TEST`, а также blob страницы. Если этого не сделать, то первый запрос может выполняться существенно медленнее остальных. В приложении для тестирования производительности передачи BLOB по сети автоматически выполняет SQL запрос для чтобы "прогреть" страничный кеш.

Для тестирования я использую версию Firebird 5.0.3 в архитектуре SuperServer. Значение параметра `DefaultDbCachePages = 32K`, что достаточно для того, чтобы все наши запросы не производили физических чтений, после заполнения страничного кеша.

Глава 2. BLOB vs VARCHAR

Попробуем выяснить почему работать по сети с высокой латентностью (интернет канал) становится некомфортно, если запросы выбираются данные содержащие BLOB столбцы. Для этого проведём сравнительный тест передачи одних и тех же данных, когда эти данные расположены в полях типа VARCHAR и BLOB. Тестирование будет производится с использованием fbclient версии 5.0.1 (более ранние версии ведут себя аналогично).

Напомню в Firebird столбец типа VARCHAR может вмещать в себя 32765 байт, если в нём текст в кодировки UTF8, то VARCHAR может вместить до 8191 символа (4 байта на символ). Именно поэтому в таблице `BLOB_TEST` столбец `SHORT_CONTENT` определён как

```
SHORT_CONTENT  VARCHAR(8191) CHARACTER SET UTF8
```

Сначала посмотрим на статистику выполнения запроса передающие данные с использованием столбца типа BLOB, длина которого не превышает 8191 символ:

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
WHERE SHORT_BLOB IS TRUE
FETCH FIRST 1000 ROWS ONLY
```

Статистика

```
Elapsed time: 36544ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
```

А теперь сравним со статистикой выполнения запроса, использующего VARCHAR столбец:

```
SELECT
  ID,
  SHORT_CONTENT
FROM BLOB_TEST
WHERE SHORT_BLOB IS TRUE
FETCH FIRST 1000 ROWS ONLY
```

Статистика

```
Elapsed time: 574ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
```

Ого, передача данных с использованием столбца типа VARCHAR быстрее 64 раза!

Теперь попробуем измерить передачу не только коротких, но и средних BLOB полей:

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

Статистика

```
Elapsed time: 38256ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
```

Это ужасно медленно. Но начиная с Firebird 3.0 мы можем использовать сжатие трафика, и может быть в этом случае результаты будут лучше?

Глава 3. BLOB vs VARCHAR + сжатие сетевого трафика

Ну что же попробуем включить сжатие сетевого трафика. Этом можно сделать указав при подключении к базе данных параметр `WireCompression=True`.

Тест передачи коротких BLOB:

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
WHERE SHORT_BLOB IS TRUE
FETCH FIRST 1000 ROWS ONLY
```

```
Elapsed time: 36396ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
```

Тест передачи данных в типе `VARCHAR(8191)`:

```
SELECT
  ID,
  SHORT_CONTENT
FROM BLOB_TEST
WHERE SHORT_BLOB IS TRUE
FETCH FIRST 1000 ROWS ONLY
```

```
Elapsed time: 489ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
```

Тест передачи коротких и средних BLOB:

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

```
Elapsed time: 38107ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
```

Ситуация почти не изменилась. Давайте попробуем разобраться в причинах.

Глава 4. Как передаются данные типа BLOB по сети

Чтобы понять почему так получается необходимо погрузится во внутреннюю кухню сетевого протокола сервера Firebird. Прежде всего необходимо понять две базовые вещи. Сетевой протокол и API разработаны для получения больших двоичных объектов или длинных строк (BLOB):

- небольшими частями (не более 64 Кбайт);
- в отложенном режиме.

Если первое реализовано почти во всех SQL серверах одинаково, то второе может стать неожиданностью для тех кто не работал с BLOB на уровне API (только через высокоуровневые компоненты доступа).

Давайте рассмотрим типичный код для получения и обработки записей курсора:

```
// открываем курсор
Firebird::IResultSet* rs = stmt->openCursor(status, tra, inMetadata, nullptr, outMetadata, 0);
// получение очередной записи из курсора
while (rs->fetchNext(status, outBuffer) == Firebird::IStatus::RESULT_OK) {
    // обработка очередной записи
    recordProcess(outBuffer);
}
// закрытие курсора
rs->close(status);
```

Здесь упрощённо происходит следующее. При открытии курсора на сторону сервера посылается соответствующий сетевой пакет `op_execute2`. Вызов `fetchNext` посылает на сервер сетевой пакет `op_fetch`, после чего сервер возвращает в качестве ответа столько записей, сколько помещается в размер сетевого буфера, и последующие вызовы `fetchNext` не будут отправлять на сервер сетевые пакеты, а будут читать очередную запись из буфера до тех пор пока записи в буфера не исчерпаются. Когда буфер будет пуст вызов `fetchNext` вновь отправит на сервер сетевой пакет `fetchNext`. Такая схема позволяет значительно уменьшить количество roundtrips. Под roundtrip понимается отправка сетевого пакета на сторону сервера и отправка ответного сетевого пакета со стороны сервера на клиент. Чем меньше таких roundtrip, тем выше скорость работы сетевого протокола.

Буфер в который помещается запись после выполнения `fetchNext` называется выходным сообщением. Выходное сообщение описывается с помощью метаданных выходного сообщения, которые либо возвращаются при подготовке SQL запроса, либо подготавливаются в приложении. Давайте посмотрим как можно отобразить выходные сообщения на структуры в зависимости от столбцов запроса.

Для SQL запроса

```
SELECT
  ID,
  SHORT_CONTENT
FROM BLOB_TEST
WHERE SHORT_BLOB IS TRUE
FETCH FIRST 1000 ROWS ONLY
```

выходное сообщение можно отобразить на следующую структуру

```
struct message {
    int64_t id;                 // значение поля ID
    short idNull;               // NULL индикатор поля ID
    struct {
        unsigned short length;  // актуальная длина поля типа VARCHAR в байтах
        char[8191 * 4] str;     // буфер для данных строки типа VARCHAR
    } short_content;            // значение поля SHORT_CONTENT
    short short_contentNull;    // NULL индикатор поля SHORT_CONTENT
}
```

Таким образом при выполнении `fetchNext` значение поля типа `VARCHAR` становится доступно сразу. Сервер использует так называемый prefetch записей для более эффективной передачи по сети.

Теперь посмотрим на структуру выходного сообщения для SQL запроса:

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

выходное сообщение можно отобразить на следующую структуру

```
struct message {
    int64_t id;                 // значение поля ID
    short idNull;               // NULL индикатор поля ID
    ISC_QUAD content;           // идентификатор для BLOB поля CONTENT
    short contentNull;          // NULL индикатор поля CONTENT
}
```

Здесь `ISC_QUAD` структура определённая следующим образом:

```
struct GDS_QUAD_t {
    ISC_LONG gds_quad_high;
    ISC_ULONG gds_quad_low;
};

typedef struct GDS_QUAD_t ISC_QUAD;
```

Эта структура описывает только идентификатор BLOB, в котором нет содержимого. Содержимое BLOB поля надо извлекать отдельными API функциями.

Собственно, если мы будем получать только идентификаторы BLOB без их содержимого, то наш тест будет показывать отличные результаты, но это не то что нам требуется.

```
Elapsed time: 38ms
Max id: 1000
Record count: 1000
```

Таким образом, последний запрос получается только идентификатор BLOB и теперь требуется получить их содержимое. Для строковых BLOB это можно сделать с помощью следующих функций:

```
std::string readBlob(Firebird::ThrowStatusWrapper* status, Firebird::IAttachment* att,
    Firebird::Transaction* tra, ISC_QUAD* blobId)
{
    // открываем BLOB по заданному идентификатору
    Firebird::IBlob* blob = att->openBlob(status, tra, blobId, 0, nullptr);

    // Получаем информацию о BLOB (размер)
    FbBlobInfo blobInfo;
    std::memset(&blobInfo, 0, sizeof(blobInfo));
    getBlobStat(status, blob, blobInfo);

    std::string s;
    s.reserve(blobInfo.blob_total_length);
    bool eof = false;
    std::vector<char> vBuffer(MAX_SEGMENT_SIZE);
    auto buffer = vBuffer.data();
    while (!eof) {
        unsigned int l = 0;
        // чтение очередной порции из BLOB или его сегмента
        switch (blob->getSegment(status, MAX_SEGMENT_SIZE, buffer, &l))
        {
        case Firebird::IStatus::RESULT_OK:
        case Firebird::IStatus::RESULT_SEGMENT:
            s.append(buffer, l);
            break;
        default:
            eof = true;
            break;
        }
    }
    blob->close(status);
    return s;
}


void getBlobStat(Firebird::ThrowStatusWrapper* status, Firebird::IBlob* blob, FbBlobInfo& stat)
{
    ISC_UCHAR buffer[1024];
    const unsigned char info_options[] = {
```

```
        isc_info_blob_num_segments, isc_info_blob_max_segment,
        isc_info_blob_total_length, isc_info_blob_type,
        isc_info_end };
    // получение информации о BLOB
    blob->getInfo(status, sizeof(info_options), info_options, sizeof(buffer), buffer);
    for (ISC_UCHAR* p = buffer; *p != isc_info_end; ) {
        const unsigned char item = *p++;
        const ISC_SHORT length = static_cast<ISC_SHORT>(portable_integer(p, 2));
        p += 2;
        switch (item) {
        case isc_info_blob_num_segments:
            stat.blob_num_segments = portable_integer(p, length);
            break;
        case isc_info_blob_max_segment:
            stat.blob_max_segment = portable_integer(p, length);
            break;
        case isc_info_blob_total_length:
            stat.blob_total_length = portable_integer(p, length);
            break;
        case isc_info_blob_type:
            stat.blob_type = static_cast<short>(portable_integer(p, length));
            break;
        default:
            break;
        }
        p += length;
    };
}
```

Примерно так выглядит та работа которая проделывается на уровне API при вызове `BlobField.AsString` в высокоуровневых компонентах доступа для получения содержимого BLOB поля как строки.

Теперь рассмотрим какие дополнительные сетевые обращения делаются в этом коде. Функция `IAttachment::openBlob` открывает BLOB по заданному идентификатору посылая сетевой пакет `op_open_blob2`. Далее мы запрашиваем информацию о BLOB с помощью `IBlob::getInfo`, которая посылает ещё один сетевой пакет `op_info_blob` и ждёт возврата информации о BLOB. После чего мы начинаем читать BLOB порциями с помощью функции `IBlob::getSegment`, которая посылает ещё один сетевой пакет `op_get_segment`. Отмечу, что `IBlob::getSegment` оптимизирована таким образом, чтобы читать BLOB как можно большими порциями за одно сетевое обращение, то есть если вы вызовете `getSegment` с размером 10 байт, то во внутренний буфер будет прочтено гораздо большая порция, по аналогии с тем как это делает `IResultSet::fetchNext`. Когда весь BLOB прочитан будет вызван метод `IBlob::close`, которая отправит ещё один сетевой пакет `op_close_blob`.

Из описанного выше видно, что даже самый короткий BLOB требует 4 дополнительных сетевых пакета: `op_open_blob2`, `op_info_blob`, `op_get_segment`, `op_close_blob`. Вы можете отказаться от использования `op_info_blob` для предварительного резервирования буфера под выходную строку, что сэкономит один roundtrip. Однако большинство высокоуровневых компонентов доступа при работе с BLOB делают именно так как я описал.

Теперь становится понятно почему при использовании выборок содержащих BLOB столбцы

ваши приложения тормозят в сетях с высокой латентностью (интернет канал). Можно ли как-то улучшить ситуацию?

Глава 5. Совместное использование BLOB и VARCHAR для оптимизации передачи по сети

Как было показано выше основные накладные расходы приходятся на передачу именно коротких BLOB. Более большие BLOB требуют дополнительных пакетов `op_get_segment`, в то время как остальные сетевые пакеты связанные с BLOB посылаются максимум один раз. Это неизбежное зло, поскольку большие BLOB невозможно передать за один сетевой пакет.

Но что если, мы будем передавать содержимое BLOB как VARCHAR, если оно может поместиться в этом типе данных, а остальные BLOB передавать стандартным способом? Давайте попробуем это.

Перепишем наш запрос следующим образом:

```
SELECT
  BLOB_TEST.ID,
  CASE
    WHEN CHAR_LENGTH(BLOB_TEST.CONTENT) <= 8191
    THEN CAST(BLOB_TEST.CONTENT AS VARCHAR(8191))
  END AS SHORT_CONTENT,
  CASE
    WHEN CHAR_LENGTH(BLOB_TEST.CONTENT) > 8191
    THEN CONTENT
  END AS CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

Теперь нам следует переписать код нашего приложения, чтобы оно могло выбирать откуда читать данные:

```
Firebird::IResultSet* rs = stmt->openCursor(status, tra, inMetadata, nullptr, outMetadata, 0);

// описание структуры выходного сообщения
FB_MESSAGE(OutMessage, Firebird::ThrowStatusWrapper,
    (FB_BIGINT, id)
    (FB_VARCHAR(8191 * 4), short_content)
    (FB_BLOB, content)
) out(status, master);


size_t blb_size = 0;
while (rs->fetchNext(status, out.getData()) == Firebird::IStatus::RESULT_OK) {
    std::string s;
    if (out->short_contentNull && !out->contentNull) {
        // Если поле SHORT_CONTENT IS NULL и CONTENT IS NOT NULL читаем из BLOB
        Firebird::IBlob* blob = att->openBlob(status, tra, &out->content, 0, nullptr);
        s = readBlob(status, blob);
        blob->close(status);
    }
    else {
        // в противном случае читаем из VARCHAR
        s = std::string(out->short_content.str, out->short_content.length);
    }
    blb_size += s.size();
}
rs->close(status);
```

Посмотрим производительность этого решения:

Статистика (`WireCompression=False`):

```
Elapsed time: 20212ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
```

Теперь измерим производительность со включенным сжатие сетевого трафика (`WireCompression=True`):

Статистика (`WireCompression=True`):

```
Elapsed time: 15927ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
```

Намного лучше. Напомню результаты чтения только BLOB полей были 38256ms и 38107ms.

Можно ли ещё улучшить наш результат? Да, поскольку если в нашей таблице уже хранятся короткие BLOB как VARCHAR. В этом случае SQL запрос выглядит следующим образом:

```
SELECT
  BLOB_TEST.ID,
  CASE
    WHEN BLOB_TEST.SHORT_BLOB IS TRUE
    THEN BLOB_TEST.SHORT_CONTENT
  END AS SHORT_CONTENT,
  CASE
    WHEN BLOB_TEST.SHORT_BLOB IS FALSE
    THEN BLOB_TEST.CONTENT
  END AS CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`*):*

```
Elapsed time: 19288ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
```

Статистика (`WireCompression=True`*):*

```
Elapsed time: 15752ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
```

Глава 6. Улучшения при передачи BLOB с fbclient версии 5.0.2

В Firebird 5.0.2 была сделана небольшая оптимизация передачи BLOB по сети. На самом деле изменения коснулись только клиентской части Firebird, то есть fbclient. Вы можете почувствовать при передаче BLOB с любым Firebird старше 2.1 при использовании fbclient версии 5.0.2 и выше. Прежде чем объяснить, что именно было улучшено приведём результаты тестирования.

Тест передачи `VARCHAR(8191)` (`WireCompression=False`):

```
SELECT
  ID,
  SHORT_CONTENT
FROM BLOB_TEST
WHERE SHORT_BLOB IS TRUE
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`):

```
Elapsed time: 569ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
Wire logical statistics:
  send packets = 34
  recv packets = 1034
  send bytes = 712
  recv bytes = 3396028
Wire physical statistics:
  send packets = 33
  recv packets = 2179
  send bytes = 712
  recv bytes = 3396028
  roundtrips = 33
```

Здесь помимо статистики выполнения приведена статистика сетевого трафика. Статистика сетевого трафика это новая функция доступная на клиентской стороне с fbclient версии 5.0.2 и выше.

Статистика (`WireCompression=True`*):*

```
Elapsed time: 478ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
Wire logical statistics:
  send packets = 34
  recv packets = 1034
  send bytes = 712
  recv bytes = 3396028
Wire physical statistics:
  send packets = 33
  recv packets = 457
  send bytes = 297
  recv bytes = 648654
  roundtrips = 33
```

Поля типа VARCHAR передаются без изменений, изменения в статистике выполнения в пределах погрешности.

Тест передачи коротких BLOB:

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
WHERE SHORT_BLOB IS TRUE
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`*):*

```
Elapsed time: 12739ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
Wire logical statistics:
  send packets = 4002
  recv packets = 5002
  send bytes = 72084
  recv bytes = 3557424
Wire physical statistics:
  send packets = 1002
  recv packets = 4106
  send bytes = 72084
  recv bytes = 3557424
  roundtrips = 1001
```

Статистика (`WireCompression=True`*):*

```
Elapsed time: 12693ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
Wire logical statistics:
  send packets = 4002
  recv packets = 5002
  send bytes = 72084
  recv bytes = 3557424
Wire physical statistics:
  send packets = 1002
  recv packets = 2563
  send bytes = 12337
  recv bytes = 731253
  roundtrips = 1001
```

Здесь изменения более чем заметны. Напомню для клиента версии 5.0.1 время выполнения тестов было: 36544ms и 36396ms. Таким образом короткие BLOB передаются до 3-х раз быстрее, но всё равно значительно хуже чем VARCHAR.

Посмотрим на статистику передачи коротких и средних BLOB:

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`*):*

```
Elapsed time: 17907ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
Wire logical statistics:
  send packets = 4325
  recv packets = 5325
  send bytes = 77252
  recv bytes = 12810832
Wire physical statistics:
  send packets = 1325
  recv packets = 10578
  send bytes = 77252
  recv bytes = 12810832
  roundtrips = 1324
```

Статистика (`WireCompression=True`*):*

```
Elapsed time: 17044ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
Wire logical statistics:
  send packets = 4325
  recv packets = 5325
  send bytes = 77252
  recv bytes = 12810832
Wire physical statistics:
  send packets = 1325
  recv packets = 3468
  send bytes = 14883
  recv bytes = 2261821
  roundtrips = 1324
```

Здесь улучшения тоже заметны. Для клиента версии 5.0.1 время выполнения тестов было: 38256ms и 38107ms.

Посмотрим улучшает ли производительность наш метод с совместным использованием BLOB + VARCHAR.

```
SELECT
  BLOB_TEST.ID,
  CASE
    WHEN BLOB_TEST.SHORT_BLOB IS TRUE
    THEN BLOB_TEST.SHORT_CONTENT
  END AS SHORT_CONTENT,
  CASE
    WHEN BLOB_TEST.SHORT_BLOB IS FALSE
    THEN BLOB_TEST.CONTENT
  END AS CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`*):*

```
Elapsed time: 10843ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
Wire logical statistics:
  send packets = 2000
  recv packets = 3000
  send bytes = 35472
  recv bytes = 12715904
Wire physical statistics:
  send packets = 767
  recv packets = 9732
  send bytes = 35472
  recv bytes = 12715904
  roundtrips = 735
```

Статистика (`WireCompression=True`*):*

```
Elapsed time: 9476ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
Wire logical statistics:
  send packets = 2000
  recv packets = 3000
  send bytes = 35472
  recv bytes = 12715904
Wire physical statistics:
  send packets = 767
  recv packets = 2385
  send bytes = 7878
  recv bytes = 2234602
  roundtrips = 735
```

Совместное использование столбца BLOB для блинных строк и `VARCHAR(8191)` для коротких всё равно лучше, хотя отрыв уже не такой большой как было с клиентской библиотекой версии 5.0.1.

Так в чём же суть изменений клиента fbclient версии 5.0.2 и почему она намного быстрее работает с BLOB без изменения сетевого протокола и даже со старыми версиями сервера?

Как было описано выше при чтении BLOB клиент версии 5.0.1 посылает следующие пакеты:

- `op_open_blob2` - открытие BLOB;
- `op_info_blob` - получение информации о BLOB (необязательно);
- `op_get_segment` - чтение очередной порции данных или сегмента BLOB (1 и более раз, в зависимости от размера BLOB);
- `op_close_blob` - закрытие BLOB.

Клиент Firebird 5.0.2 группирует следующие пакеты `op_open_blob2`, `op_info_blob` и `op_get_segment` в один логический пакет и посылает их при открытии BLOB (вызов `IAttachment::openBlob`). В ответ он получает в одном логическом пакете информацию о BLOB и первую порцию данных (до 64 Кбайт), то есть выполняется так называемый prefetch информации и первой порции данных. Группировка физических пакетов в логические доступна начиная с Firebird 2.1, но она не выполнялась для API функции `IAttachment::openBlob` на уровне клиента до версии 5.0.2.

Таким образом коротких BLOB вместо отправки 3-4 сетевых пакетов отправляется 2 сетевых пакета, что приводит к значительному увеличению производительности передачи BLOB по сети.

Глава 7. Улучшения передачи BLOB по сети в Firebird 5.0.3

В Firebird 5.0.3 была сделана ещё одна оптимизация передачи BLOB по сети. На этот раз изменения коснулись сетевого протокола. От клиентской части и сервера требуется поддержка сетевого протокола версии 19. Поэтому для того чтобы задействовать эту оптимизацию необходимо обновить Firebird сервер и fbclient до версии 5.0.3.

Посмотрим на результаты наших тестов с новыми версиями клиента и сервера.

Тест передачи `VARCHAR(8191)` (`WireCompression=False`):

```
SELECT
  ID,
  SHORT_CONTENT
FROM BLOB_TEST
WHERE SHORT_BLOB IS TRUE
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`):

```
Elapsed time: 554ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
Wire logical statistics:
  send packets = 34
  recv packets = 1034
  send bytes = 716
  recv bytes = 3396028
Wire physical statistics:
  send packets = 33
  recv packets = 2394
  send bytes = 716
  recv bytes = 3396028
  roundtrips = 33
```

Статистика (`WireCompression=True`):

```
Elapsed time: 482ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
Wire logical statistics:
  send packets = 34
  recv packets = 1034
  send bytes = 716
  recv bytes = 3396028
Wire physical statistics:
  send packets = 33
  recv packets = 472
  send bytes = 277
  recv bytes = 648656
  roundtrips = 33
```

Тут всё ожидаемо, передача полей типа VARCHAR не изменялась.

Тест передачи коротких BLOB:

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
WHERE SHORT_BLOB IS TRUE
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`):

```
MaxInlineBlobSize = 65535
Elapsed time: 1110ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
Wire logical statistics:
  send packets = 27
  recv packets = 2027
  send bytes = 576
  recv bytes = 3453744
Wire physical statistics:
  send packets = 26
  recv packets = 2458
  send bytes = 576
  recv bytes = 3453744
  roundtrips = 26
```

Статистика (`WireCompression=True`):

```
MaxInlineBlobSize = 65535
Elapsed time: 157ms
Max id: 1700
Record count: 1000
Content size: 3366000 bytes
Wire logical statistics:
  send packets = 6
  recv packets = 2006
  send bytes = 156
  recv bytes = 3453492
Wire physical statistics:
  send packets = 5
  recv packets = 454
  send bytes = 58
  recv bytes = 672345
  roundtrips = 5
```

Вот это да! Скорость передачи коротких BLOB без использования сжатия сетевого трафика выросла в 11 раз по сравнению с версией 5.0.2 (было 12739ms) и в 33 раза по сравнению с версией 5.0.1 (было 36544ms).

При использовании сжатия сетевого трафика скорость передачи выросла в 81 раз по сравнению с 5.0.2 (было 12693ms) и в 232 раза по сравнению с 5.0.1 (было 36396ms). Но самое удивительное, короткие BLOB стали передаваться даже быстрее чем `VARCHAR(8191)` 482ms vs 157ms. Отличный результат!

Попробуем посмотреть на статистику передачи коротких и средних BLOB:

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`*):*

```
MaxInlineBlobSize = 65535
Elapsed time: 3254ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
Wire logical statistics:
  send packets = 249
  recv packets = 2220
  send bytes = 4552
  recv bytes = 12701676
Wire physical statistics:
  send packets = 161
  recv packets = 8872
  send bytes = 4552
  recv bytes = 12701676
  roundtrips = 161
```

Статистика (`WireCompression=True`*):*

```
MaxInlineBlobSize = 65535
Elapsed time: 1365ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
Wire logical statistics:
  send packets = 184
  recv packets = 2155
  send bytes = 3264
  recv bytes = 12700876
Wire physical statistics:
  send packets = 97
  recv packets = 1470
  send bytes = 951
  recv bytes = 2187089
  roundtrips = 88
```

Отличный результат. Результаты предыдущих тестов:

- 5.0.1 (`WireCompression=False`) 38256ms
- 5.0.1 (`WireCompression=True`) 38107ms
- 5.0.2 (`WireCompression=False`) 17907ms
- 5.0.2 (`WireCompression=True`) 17044ms

Теперь посмотрим, а имеет ли смысл использовать наш велосипед, когда короткие BLOB передаются как VARCHAR, а длинные как BLOB.

```
SELECT
  BLOB_TEST.ID,
  CASE
    WHEN BLOB_TEST.SHORT_BLOB IS TRUE
    THEN BLOB_TEST.SHORT_CONTENT
  END AS SHORT_CONTENT,
  CASE
    WHEN BLOB_TEST.SHORT_BLOB IS FALSE
    THEN BLOB_TEST.CONTENT
  END AS CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`*):*

```
MaxInlineBlobSize = 65535
Elapsed time: 3678ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
Wire logical statistics:
  send packets = 249
  recv packets = 1631
  send bytes = 4560
  recv bytes = 12667632
Wire physical statistics:
  send packets = 161
  recv packets = 8958
  send bytes = 4560
  recv bytes = 12667632
  roundtrips = 161
```

Статистика (`WireCompression=True`*):*

```
MaxInlineBlobSize = 65535
Elapsed time: 1576ms
Max id: 1000
Record count: 1000
Content size: 12607388 bytes
Wire logical statistics:
  send packets = 207
  recv packets = 1589
  send bytes = 3732
  recv bytes = 12667108
Wire physical statistics:
  send packets = 120
  recv packets = 1527
  send bytes = 1086
  recv bytes = 2187418
  roundtrips = 110
```

Нет, данный способ передачи данных медленней, чем непосредственная передача данных

как BLOB.

В целом получены отличные результаты, теперь можно смело использовать в выборках столбцы типа BLOB при размещении сервера Firebird в сетях с высокой латентностью (интернет канал).

7.1. Как это работает?

Если размер BLOB объекта меньше значения параметра `MaxInlineBlobSize` (по умолчанию 64 Кбайт - 1), то содержимое BLOB отправляется в том же потоке данных, что и основной набор результатов (ResultSet).

Метаданные (размер, количество сегментов, тип) и данные BLOB объектов отправляются с использованием нового типа пакета `op_inline_blob` и новой структуры `P_INLINE_BLOB`.

Пакет `op_inline_blob` отправляется перед соответствующим `op_sql_response` (в случае ответа на `op_execute2` или `op_exec_immediate2`) или `op_fetch_response` (ответ на `op_fetch`).

Количество пакетов `op_inline_blob` может соответствовать количеству полей BLOB-объектов в выходном формате. Если BLOB-объект имеет значение `NULL` или слишком большой, то BLOB-объекты не отправляются.

BLOB-объект отправляется целиком, то есть текущая реализация не поддерживает отправку части BLOB-объекта. Причины — более простой код и тот факт, что поиск не реализован для сегментированных BLOB.

Отправленные inline BLOB-объекты кешируются на стороне клиента на уровне соединения (`IAttachment`). На стороне клиента существует структура для быстрого поиска содержимого BLOB и его метаданных по BLOB идентификатору. Когда приложение открывает BLOB с помощью `IAttachment::openBlob` его метаданные и содержимое извлекаются из кеша BLOB. Вызовы `IAttachment::openBlob`, `IBlob::getSegment` и `IBlob::close` не передают никаких дополнительных сетевых пакетов. Вызов `IBlob::close` удаляет BLOB из кеша. Таким образом повторное открытие и использование BLOB будет приводить к дополнительным сетевым пакетам.

Размер кеша под inline BLOB-объекты ограничен параметром `MaxBlobCacheSize` (по умолчанию 10 Мбайт). Если в кеше для inline BLOB-объекта нет места, то такой объект отбрасывается. Значение параметра `MaxBlobCacheSize` может быть установлено с помощью `isc_dpb_max_blob_cache_size` при соединении с базой данных и изменено позже с помощью метода `IAttachment::setMaxBlobCacheSize`. Изменение предела не применяется немедленно, то есть если новый предел меньше текущего используемого размера, то ничего не происходит.

Максимальный размер inline BLOB-объекта регулируется парамером `MaxInlineBlobSize`, который по умолчанию равен 64 Кбайт - 1 (63535 байт). Это значение устанавливается для каждого подготовленного запроса перед началом его выполнения с помощью метода `IStatement::setMaxInlineBlobSize`. Если `MaxInlineBlobSize` установлен равным 0, то передача inline BLOB будет отключена. На уровне соединения можно изменить значение по умолчанию для вновь подготавливаемых запросов с помощью метода `IAttachment::setMaxInlineBlobSize`. Так же значение по умолчанию для параметра

`MaxInlineBlobSize` можно установить с помощью `isc_dpb_max_inline_blob_size`.

7.2. Всегда ли подходят параметры по умолчанию?

Для ответа на этот вопрос попробуем запустить тест, который читает только идентификаторы BLOB без их содержимого и метаданных.

```
SELECT
  ID,
  CONTENT
FROM BLOB_TEST
FETCH FIRST 1000 ROWS ONLY
```

Статистика (`WireCompression=False`):

```
MaxInlineBlobSize = 65535
Elapsed time: 2049ms
Max id: 1000
Record count: 1000
Wire logical statistics:
  send packets = 75
  recv packets = 2046
  send bytes = 1536
  recv bytes = 10438516
Wire physical statistics:
  send packets = 74
  recv packets = 7170
  send bytes = 1536
  recv bytes = 10438516
  roundtrips = 74
```

Статистика (`WireCompression=True`):

```
MaxInlineBlobSize = 65535
Elapsed time: 280ms
Max id: 1000
Record count: 1000
Wire logical statistics:
  send packets = 11
  recv packets = 1982
  send bytes = 256
  recv bytes = 10437748
Wire physical statistics:
  send packets = 10
  recv packets = 1171
  send bytes = 86
  recv bytes = 1618835
  roundtrips = 10
```

Сравним эти результаты с клиентом версии 5.0.2:

- `WireCompression=False` - 26ms
- `WireCompression=True` - 28ms

Мы видим, что время выполнения этого теста увеличилось. Что же произошло?

Для всех BLOB-объектов длина, которых меньше чем значение параметра `MaxInlineBlobSize` сервер посылал дополнительный сетевой пакет `op_inline_blob`, но мы не использовали данные которые пересылались этим пакетом.

Но для чего нужен такой режим - спросите вы? На самом деле такой режим часто используется в приложениях с сетками данных, в которых содержимое BLOB не отображается непосредственно, а отображается в отдельном контроле при изменении позиции курсора в сетке. Например вы выбираете в сетке некоторую запись, а в отдельном контроле отображается картинка, хранящая с BLOB.

В некоторых компонентах доступа Delphi на основе DataSet BLOB объекты могут извлекаться сразу и кешироваться на уровне набора данных (читаться по мере фетча данных из курсора) или откладываться до тех пор пока пользователь не начнёт читать данные из поля типа BLOB. Например, в компонентах доступа FireDac это зависит от флага `fiBlobs` который может быть установлен в свойстве `FetchOptions.Items` набора данных.

Так как же быть в этом случае? Либо смириться с тем, что в режиме отложенного чтения BLOB ваш набор данных будет загружаться немного дольше, либо устанавливать значением параметра `MaxInlineBlobSize` в 0 с помощью `IStatement::setMaxInlineBlobSize`. Давайте попробуем сделать это для версии 5.0.3 и посмотрим на результат предыдущего теста.

Статистика (`WireCompression=False`):

```
MaxInlineBlobSize = 0
Elapsed time: 26ms
Max id: 1000
Record count: 1000
Wire logical statistics:
  send packets = 3
  recv packets = 1003
  send bytes = 96
  recv bytes = 32056
Wire physical statistics:
  send packets = 2
  recv packets = 23
  send bytes = 96
  recv bytes = 32056
  roundtrips = 2
```

Статистика (`WireCompression=True`*):*

```
MaxInlineBlobSize = 0
Elapsed time: 36ms
Max id: 1000
Record count: 1000
Wire logical statistics:
  send packets = 3
  recv packets = 1003
  send bytes = 96
  recv bytes = 32056
Wire physical statistics:
  send packets = 2
  recv packets = 2
  send bytes = 37
  recv bytes = 5796
  roundtrips = 2
```

Загрузка inline BLOB-объектов отключена, чтение только идентификаторов BLOB показывает тоже время что и в 5.0.2.

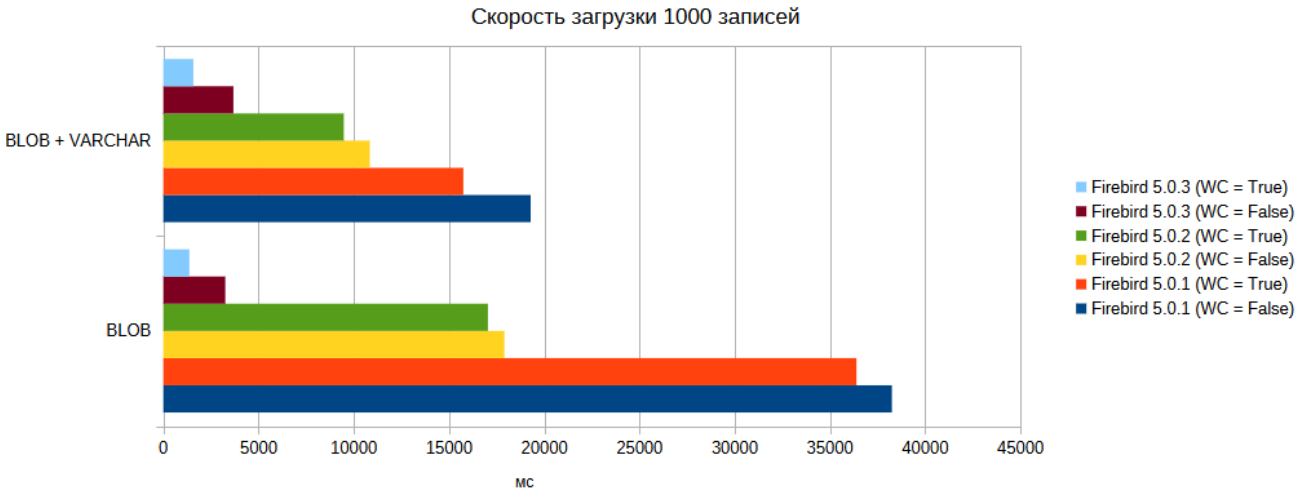
Глава 8. Сравнение скорости передачи BLOB в различных версиях Firebird

Для наглядности приведём сравнения времени загрузки 1000 записей коротких BLOB против VARCHAR(8191) в различных версиях Firebird и различными значениями параметра WireCompression (сокращённо WC).

Версия Firebird и WireCompression	Тип данных	
	VARCHAR(8191)	BLOB SUB_TYPE TEXT
Firebird 5.0.1 (WC = False)	574	36544
Firebird 5.0.1 (WC = True)	489	36396
Firebird 5.0.2 (WC = False)	569	12739
Firebird 5.0.2 (WC = True)	478	12693
Firebird 5.0.3 (WC = False)	554	1110
Firebird 5.0.3 (WC = True)	482	157

Также приведём сравнения времени загрузки 1000 записей разными способами: только BLOB или небольшие данные в VARCHAR, а большие - в BLOB.

Версия Firebird и WireCompression	Способ загрузки	
	BLOB	BLOB + VARCHAR
Firebird 5.0.1 (WC = False)	38256	19288
Firebird 5.0.1 (WC = True)	36396	15752
Firebird 5.0.2 (WC = False)	17907	10843
Firebird 5.0.2 (WC = True)	17044	9476
Firebird 5.0.3 (WC = False)	3254	3678
Firebird 5.0.3 (WC = True)	1365	1576


Скорость загрузки 1000 записей
BLOB + VARCHAR
BLOB
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
мс
Firebird 5.0.3 (WC = True)
Firebird 5.0.3 (WC = False)
Firebird 5.0.2 (WC = True)
Firebird 5.0.2 (WC = False)
Firebird 5.0.1 (WC = True)
Firebird 5.0.1 (WC = False)

Глава 9. Выводы

Если вы пытались разместить сервер Firebird в облаке и работать с ним через интернет-канал, но отказались от этой идеи из-за проблем с производительностью при передаче BLOB-объектов, то рекомендуем попробовать ещё раз!

В настоящее время эта функциональность доступна в HQbird (начиная с версии 2024 R2 Update 2 и более поздних). Ванильная версия Firebird 5.0.3 ещё не выпущена, но вы можете попробовать снапшоты Firebird 5.0.3 (https://github.com/FirebirdSQL/snapshots/releases/tag/snapshot-v5.0-release).