Transferring BLOBs over the wire in Firebird

Simonov Denis

Version 1.0 by 2025-03-16

Table of Contents

This material is sponsored and created with the sponsorship and support of IBSurgeon https://www.ib-aid.com, vendor of HQbird (advanced distribution of Firebird) and supplier of performance optimization, migration and technical support services for Firebird.

Preface

Application developers and administrators using the Firebird DBMS often wonder: what if they deploy Firebird in the cloud and access it over an internet connection? However, after testing this setup, many are left disappointed, as the data transfer speed over high-latency networks (such as the internet) leaves much to be desired. In most cases, the speed of fetching data from cursors generated by SQL queries is acceptable, but as soon as BLOB fields (binary or text data) appear in such queries, the data transfer speed drops catastrophically.

In this article, we will discuss how BLOBs are transmitted over the wire, the challenges users face when using Firebird in high-latency networks (working over the internet), and explore solutions to these issues. We will also cover the improvements in BLOB transmission in the latest versions of Firebird (5.0.2 and 5.0.3).

Chapter 1. Application and Database for Testing

To demonstrate various ways of working with BLOB fields, as well as performance measurements, a small test application was written, the source codes of which are available at https://github.com/IBSurgeon/fb-blob-test. On the same page, you can download a ready-made assembly for Windows x64 and a test database.

This application tests the performance of transferring only text BLOB fields, but the same mechanisms can be applied to binary BLOBs.

To demonstrate the transfer of BLOBs over the network, we will need a database containing tables with BLOB fields, and it is desirable that the size of these BLOB fields varies from very small to medium. For this purpose, you can use the source codes of some Open Source project, for example, the UDR library lucene_udr.

The contents of the files will be stored in a table with the following structure:

```
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);
```

Since the project is not large, the number of source code files in it is not as large as we would like. To make the testing results more visual in numbers, we will increase the number of BLOB records to 10,000. To do this, we will create a separate table `BLOB_TEST` with the following structure:

```
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)
);
```

Here we have removed the file name storage field `FILE_NAME`, but added the field `SHORT_CONTENT`. We will fill this field if the contents of the BLOB field `CONTENT` can be stored entirely in a field of type `VARCHAR(8191) CHARACTER SET UTF8`. We will also add the field `SHORT_BLOB`, which is an indication that the BLOB is "short" (fits into VARCHAR). We will need these fields when performing various comparative tests.

So, we need to fill the table `BLOB_TEST` from the table `BLOB_SAMPLE`, so that the target table has 10,000 records. To do this, we will use the following script:

```
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;
      -- The contents of the BLOB are placed into a string variable
      -- with a length of 8191 characters
      IS_SHORT = (C.CH_L < 8191);

      INSERT INTO BLOB_TEST (
        SHORT_CONTENT,
        CONTENT,
        SHORT_BLOB
      )
      VALUES (
        IIF(:IS_SHORT, :C.CONTENT, NULL), -- if BLOB is short we write it in VARCHAR field
        :C.CONTENT,
        :IS_SHORT
      );
      -- exit when 10000 records are inserted
      IF (I = 10000) THEN EXIT;
    END
  END
END^

SET TERM ;^

COMMIT;
```

The database with BLOB fields of different lengths is ready for testing.

> ❗ In order to compare different BLOB field transfer options fairly, it is necessary to "warm up" the page cache, i.e. to make sure that all the data pages of the BLOB_TEST table, as well as the BLOB pages, are included in it. If this is not done, the first query may be executed significantly slower than the others. The application for testing the performance of BLOB transfer over the network automatically executes a SQL query to "warm up" the page cache.

For testing, I use Firebird 5.0.3 in the SuperServer architecture. The value of the `DefaultDbCachePages` parameter is 32K, which is enough to ensure that all our queries do not perform physical reads after the page cache is filled.

Chapter 2. BLOB vs VARCHAR

Let’s try to find out why working over a high-latency network (Internet channel) becomes uncomfortable if queries select data containing BLOB columns. To do this, we will conduct a comparative test of transferring the same data when this data is located in VARCHAR and BLOB fields. Testing will be performed using fbclient version 5.0.1 (earlier versions behave similarly).

Let me remind you that in Firebird a VARCHAR column can hold 32765 bytes, if it contains text in UTF8 encoding, then VARCHAR can hold up to 8191 characters (UTF-8 uses variable-length encoding 1-4 bytes per character). That is why in the `BLOB_TEST` table the `SHORT_CONTENT` column is defined as

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

First, let’s look at the statistics for executing a query that transfers data using a BLOB column whose length does not exceed 8191 characters:

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

Statistics

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

Now let’s compare it with the statistics of the query execution using a VARCHAR column:

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

Statistics

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

Wow, data transfer using a VARCHAR column is 64 times faster!

Now let’s try to measure the transfer of not only short, but also medium BLOB fields:

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

Statistics

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

This is terribly slow. But starting with Firebird 3.0, we can use wire compression, and perhaps in this case, the results will be better?

Chapter 3. BLOB vs VARCHAR + wire compression

Well, let’s try enabling wire compression. This can be done by specifying the `WireCompression=True` parameter when connecting to the database.

Test of transferring short BLOBs:

```
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
```

Test of transferring data in the `VARCHAR(8191)` type:

```
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
```

Test of transferring short and medium-sized BLOBs:

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

```
Elapsed time: 38107ms
Max id: 1000
Record count: 1000
```

```
Content size: 12607388 bytes
```

The situation has hardly changed. Let’s try to understand the reasons.

Chapter 4. How BLOB data is transmitted over the wire

To understand why this happens, we need to delve into the inner workings of the Firebird server’s network protocol. First and foremost, it’s important to understand two fundamental aspects. The network protocol and API are designed to handle large binary objects or long strings (BLOBs):

- in small chunks (no larger than 64 KB);
- in a deferred (lazy) mode.

If the first point is implemented similarly in almost all SQL servers, the second might come as a surprise to those who haven’t worked with BLOBs at the API level (only through high-level access components).

Let’s take a look at a typical code snippet for fetching and processing cursor records:

```
Firebird::IResultSet* rs = stmt->openCursor(status, tra, inMetadata, nullptr, outMetadata, 0);
while (rs->fetchNext(status, outBuffer) == Firebird::IStatus::RESULT_OK) {
    recordProcess(outBuffer);
}
rs->close(status);
```

Here’s a simplified explanation of what happens. When the cursor is opened, a corresponding network packet `op_execute2` is sent to the server. The `fetchNext` call sends a network packet `op_fetch` to the server, after which the server returns as many records as can fit into the network buffer. Subsequent `fetchNext` calls will not send network packets to the server but will instead read the next record from the buffer until the buffer is exhausted. When the buffer is empty, the `fetchNext` call will again send a network packet `op_fetch` to the server. This approach significantly reduces the number of **roundtrips**. A **roundtrip** refers to sending a network packet to the server and receiving a response packet from the server back to the client. The fewer such roundtrips, the higher the efficiency of the network protocol.

The buffer into which a record is placed after executing `fetchNext` is called the **output message**. The output message is described using **output message metadata**, which is either returned when preparing the SQL query or prepared within the application. Let’s take a look at how output messages can be mapped to structures based on the columns of the query.

For the SQL query:

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

the output message can be mapped to the following structure:

```
struct message {
    int64_t id;                  // value of the ID field
    short idNull;                // NULL indicator for the ID field
    struct {
        unsigned short length;   // actual length of the VARCHAR field in bytes
        char[8191 * 4] str;      // buffer for VARCHAR string data
    } short_content;             // value of the SHORT_CONTENT field
    short contentNull;           // NULL indicator for the SHORT_CONTENT field
}
```

Thus, when fetchNext is executed, the value of the VARCHAR field is immediately available. The server uses so-called prefetch of records for more efficient transmission over the network.

Now let’s look at the structure of the output message for the SQL query:

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

the output message can be mapped to the following structure:

```
struct message {
    int64_t id;                  // value of the ID field
    short idNull;                // NULL indicator for the ID field
    ISC_QUAD content;            // identifier for the BLOB field CONTENT
    contentNull;                 // NULL indicator for the CONTENT field
}
```

Here, ISC_QUAD is a structure defined as follows:

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

typedef struct GDS_QUAD_t ISC_QUAD;
```

This structure only describes the BLOB identifier, which does not include the actual content. The content of the BLOB field must be retrieved using separate API functions.

Indeed, if we were to fetch only the BLOB identifiers without their content, our test would show excellent results, but that’s not what we need.

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

```
Record count: 1000
```

Thus, the last query retrieves only the BLOB identifier, and now we need to fetch its content. For string BLOBs, this can be done using the following functions:

```
std::string readBlob(Firebird::ThrowStatusWrapper* status, Firebird::IAttachment* att,
    Firebird::Transaction* tra, ISC_QUAD* blobId)
{
    // Open the BLOB using the specified identifier
    Firebird::IBlob* blob = att->openBlob(status, tra, blobId, 0, nullptr);

    // Retrieve BLOB information (size)
    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;
        // Read the next portion of the BLOB or its segment
        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 };
    // Retrieve BLOB information
    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;
    };
}
```

This is roughly what happens under the hood at the API level when you call `BlobField.AsString` in high-level access components to retrieve the content of a BLOB field as a string.

Now let’s look at the additional network calls made in this code. The `IAttachment::openBlob` function opens a BLOB by the given identifier by sending the `op_open_blob2` network packet. Next, we request information about the BLOB using `IBlob::getInfo`, which sends another `op_info_blob` network packet and waits for the BLOB information to be returned. After that, we start reading the BLOB in chunks using the `IBlob::getSegment` function, which sends another `op_get_segment` network packet. Note that `IBlob::getSegment` is optimized to read the BLOB in as many chunks as possible in one network call, i.e. if you call `getSegment` with a size of 10 bytes, a much larger chunk will be read into the internal buffer, similar to how `IResultSet::fetchNext` does it. When the entire BLOB has been read, the `IBlob::close` method will be called, which will send another `op_close_blob` network packet.

From the above, it is clear that even the shortest BLOB requires 4 additional network packets: `op_open_blob2`, `op_info_blob`, `op_get_segment`, `op_close_blob`. You can avoid using `op_info_blob` to reserve a buffer for the output string in advance, which will save one roundtrip. However, most high-level access components do exactly what I described when working with BLOBs.

Now it becomes clear why your applications slow down in high-latency networks (Internet channel) when using selections containing BLOB columns. Is there any way to improve the situation?

Chapter 5. Using BLOB and VARCHAR together to optimize wire transfer

As shown above, the main overhead is incurred when transferring short BLOBs. Larger BLOBs require additional `op_get_segment` packets, while other network packets associated with the BLOB are sent at most once. This is an unavoidable evil, since large BLOBs cannot be transferred in a single network packet.

But what if we transfer the BLOB contents as VARCHAR if they can fit in this data type, and transfer the rest of the BLOBs in the standard way? Let's try that.

Let's rewrite our query as follows:

```
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
```

Now we need to rewrite our application code so that it can choose where to read data from:

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

// Description of the output message structure
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) {
        // If the field SHORT_CONTENT IS NULL and CONTENT IS NOT NULL read from BLOB
        Firebird::IBlob* blob = att->openBlob(status, tra, &out->content, 0, nullptr);
        s = readBlob(status, blob);
        blob->close(status);
    }
    else {
        // otherwise read from VARCHAR
        s = std::string(out->short_content.str, out->short_content.length);
    }
    blb_size += s.size();
}
rs->close(status);
```

Let’s look at the performance of this solution:

Statistics (`WireCompression=False`*):*

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

Now let’s measure the performance with network traffic compression enabled (`WireCompression=True`):

Statistics (`WireCompression=True`*):*

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

Much better. Let me remind you that the results of reading only BLOB fields were 38256ms and 38107ms.

Can we improve our result even more? Yes, because if our table already stores short BLOBs as VARCHAR. In this case, the SQL query looks like this:

```sql
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
```

Statistics (`WireCompression=False`*):*

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

Statistics (`WireCompression=True`*):*

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

Chapter 6. Improvements in BLOB transfer with fbclient version 5.0.2

In Firebird 5.0.2, a small optimization of BLOB transfer over the network was made. In fact, the changes affected only the client part of Firebird, that is, fbclient. You can feel it when transferring BLOB with any Firebird older than 2.1 when using fbclient version 5.0.2 and higher. Before explaining what exactly was improved, we will provide the test results.

Test transmission `VARCHAR(8191)` (`WireCompression=False`):

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

Statistics (`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
```

In addition to execution statistics, wire statistics are provided here. Wire statistics is a new feature available on the client side with fbclient version 5.0.2 and higher.

Statistics (`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
```

`VARCHAR` fields are transferred unchanged, changes in execution statistics are within the margin of error.

Short BLOB transfer test:

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

Statistics (`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
```

Statistics (`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
```

Here the changes are more than noticeable. Let me remind you that for the client version 5.0.1 the test execution time was: 36544ms and 36396ms. Thus, short BLOBs are transferred up to 3 times faster, but still significantly worse than VARCHAR.

Let’s look at the statistics of short and medium BLOB transfer:

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

Statistics (`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
```

Statistics (`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
```

Here, improvements are also noticeable. For the client version 5.0.1, the test execution time was:

38256ms and 38107ms.

Let’s see if our method with combined use of BLOB + VARCHAR improves performance.

```
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
```

Statistics (`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
```

Statistics (`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
```

Using a BLOB column for long strings and `VARCHAR(8191)` for short ones is still better, although the gap is not as big as it was with the client library version 5.0.1.

So what is the essence of the changes in the fbclient version 5.0.2 and why does it work much faster with BLOBs without changing the network protocol and even with older versions of the server?

As described above, when reading a BLOB, the client version 5.0.1 sends the following packets:

- `op_open_blob2` - opening a BLOB;
- `op_info_blob` - getting information about a BLOB (optional);
- `op_get_segment` - reading the next portion of data or a BLOB segment (1 or more times, depending on the BLOB size);
- `op_close_blob` - closing a BLOB.

Firebird 5.0.2 client groups the following `op_open_blob2`, `op_info_blob` and `op_get_segment` packets into one logical packet and sends them when opening a BLOB (calling `IAttachment::openBlob`). In response, it receives information about the BLOB and the first portion of data (up to 64 KB) in one logical packet, i.e. the so-called prefetch of information and the first portion of data is performed. Grouping physical packets into logical ones is available since Firebird 2.1, but it was not performed for the `IAttachment::openBlob` API function at the client level before version 5.0.2.

Thus, for short BLOBs, instead of sending 3-4 network packets, 2 network packets are sent, which leads to a significant increase in the performance of BLOB transfer over the network.

Chapter 7. Improvements to BLOB wire transfer in Firebird 5.0.3

Firebird 5.0.3 has another optimization of BLOB transfer over the wire. This time the changes affected the network protocol. The client and server are required to support the network protocol version 19. Therefore, in order to use this optimization, it is necessary to update the Firebird server and fbclient to version 5.0.3.

Let’s look at the results of our tests with the new versions of the client and server.

Test of transferring `VARCHAR(8191)` (`WireCompression=False`):

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

Statistics (`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
```

Statistics (`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
```

Here everything is as expected, the transfer of VARCHAR type fields has not changed.

Test of transfer of short BLOBs:

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

Statistics (`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
```

Statistics (`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
```

Wow! The speed of short BLOB transfer without using network traffic compression increased 11 times compared to version 5.0.2 (was 12739ms) and 33 times compared to version 5.0.1 (was 36544ms).

When using network traffic compression, the transfer speed increased 81 times compared to 5.0.2 (was 12693ms) and 232 times compared to 5.0.1 (was 36396ms). But the most amazing thing is that short BLOBs began to be transferred even faster than `VARCHAR(8191)` 482ms vs 157ms. Excellent result!

Let’s try to look at the statistics of short and medium BLOB transfer:

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

Statistics (`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
```

Statistics (`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
```

Excellent result. Results of previous tests:

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

Now let's see if it makes sense to use our bicycle when short BLOBs are passed as VARCHAR, and long ones as BLOBs.

```
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
```

Statistics (`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
```

Statistics (`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
```

No, this method of data transfer is slower than direct data transfer as BLOB.

Overall, excellent results were obtained, now you can safely use BLOB columns in selections when placing the Firebird server in high-latency networks (Internet channel).

7.1. How does this work?

If the BLOB size is less than the `MaxInlineBlobSize` parameter (default 64 KB - 1), the BLOB contents are sent in the same data stream as the main ResultSet.

The BLOB metadata (size, number of buckets, type) and data are sent using the new `op_inline_blob` packet type and the new `P_INLINE_BLOB` structure.

The `op_inline_blob` packet is sent before the corresponding `op_sql_response` (in case of a response to `op_execute2` or `op_exec_immediate2`) or `op_fetch_response` (response to `op_fetch`).

The number of `op_inline_blob` packets can match the number of BLOB fields in the output format. If the BLOB is `NULL` or too large, the BLOBs are not sent.

The entire blob is sent, meaning the current implementation does not support sending a portion of a blob. The reasons are simpler code and the fact that search is not implemented for segmented BLOBs.

Sent inline BLOBs are cached on the client side at the connection level (`IAttachment`). There is a structure on the client side for quickly looking up the BLOB contents and its metadata by BLOB identifier. When an application opens a BLOB using `IAttachment::openBlob`, its metadata and contents are retrieved from the BLOB cache. Calls to `IAttachment::openBlob`, `IBlob::getSegment` and `IBlob::close` do not transmit any additional network packets. Calling `IBlob::close` removes the BLOB from the cache. Thus, reopening and using the BLOB will result in additional network packets.

The size of the cache for inline BLOBs is limited by the `MaxBlobCacheSize` parameter (default is 10 MB). If there is no room in the cache for an inline BLOB, then the BLOB is discarded. The `MaxBlobCacheSize` parameter can be set with `isc_dpb_max_blob_cache_size` when connecting to the database, and changed later with the `IAttachment::setMaxBlobCacheSize` method. Changing the limit is not applied immediately, i.e. if the new limit is smaller than the currently used size, nothing happens.

The maximum inline BLOB size is controlled by the `MaxInlineBlobSize` parameter, which defaults to 64 KB - 1 (63535 bytes). This value is set for each prepared statement before it is executed with the `IStatement::setMaxInlineBlobSize` method. If `MaxInlineBlobSize` is set to 0, inline BLOB transmission will be disabled. The default value for newly prepared statements can be changed at the connection level with the `IAttachment::setMaxInlineBlobSize` method. The default value for the `MaxInlineBlobSize` parameter can also be set using `isc_dpb_max_inline_blob_size`.

7.2. Are the default settings always appropriate?

To answer this question, let’s try running a test that reads only BLOB identifiers without their contents or metadata.

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

Statistics (`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
```

Statistics (`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
```

Let’s compare these results with the client version 5.0.2:

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

We see that the execution time of this test has increased. What happened?

For all BLOBs whose length is less than the value of the `MaxInlineBlobSize` parameter, the server sent an additional `op_inline_blob` network packet, but we did not use the data that was sent by this packet.

But why do we need this mode, you ask? In fact, this mode is often used in applications with data grids, in which the BLOB content is not displayed directly, but is displayed in a separate control when the cursor position in the grid changes. For example, you select a record in the grid, and a picture containing the BLOB is displayed in a separate control.

In some Delphi DataSet-based access components, BLOBs can be retrieved immediately and cached at the dataset level (read as data is fetched from the cursor) or postponed until the user starts reading data from the BLOB field. For example, in FireDac access components, it depends on the `fiBlobs` flag that can be set in the `FetchOptions.Items` property of the dataset.

So what to do in this case? Either accept that in the lazy BLOB reading mode your dataset will load a little longer, or set the `MaxInlineBlobSize` parameter to 0 using `IStatement::setMaxInlineBlobSize`. Let’s try to do this for version 5.0.3 and see the result of the previous test.

Statistics (`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
```

Statistics (`WireCompression=True`*):*

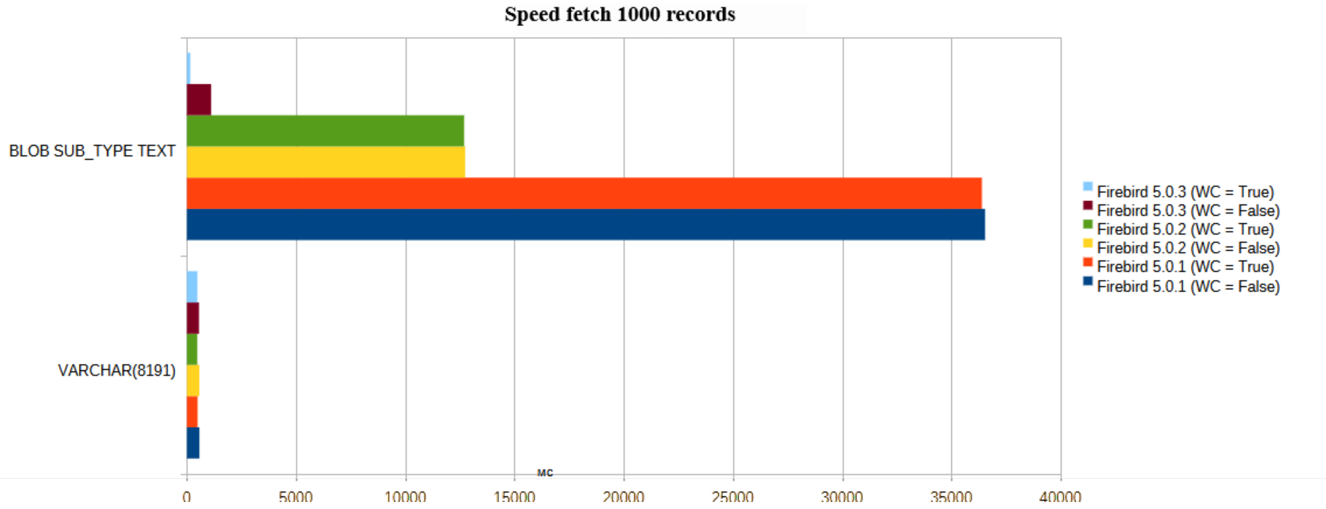
```
MaxInlineBlobSize = 0
Elapsed time: 36ms
Max id: 1000
Record count: 1000
Wire logical statistics:
  send packets = 3
  recv packets = 1003
  send bytes = 96
  recv bytes = 32056
Wire physical statistics:
  send packets = 2
  recv packets = 2
  send bytes = 37
  recv bytes = 5796
  roundtrips = 2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

Loading inline BLOBs is disabled, reading only BLOB IDs shows the same time as in 5.0.2.

Chapter 8. Comparison of BLOB transfer speed in different Firebird versions

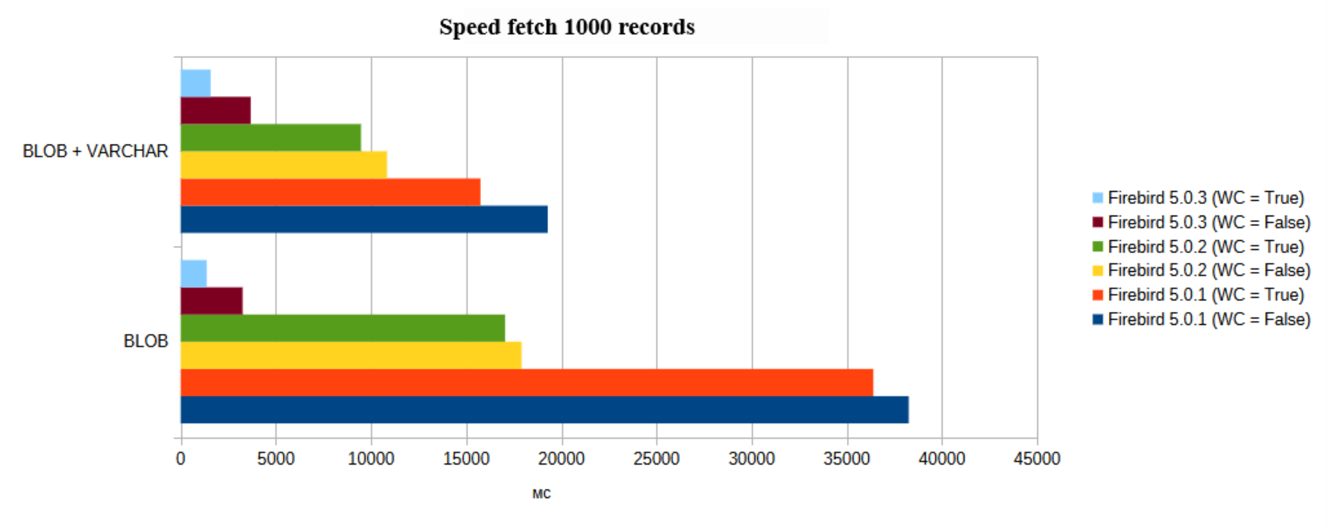
For clarity, we will compare the loading time of 1000 short BLOB records against VARCHAR(8191) in different versions of Firebird and different values of the WireCompression parameter (abbreviated WC).

Firebird version and WireCompression	Data Type	
	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



We will also provide comparisons of loading times for 1000 records in different ways: only BLOB or small data in VARCHAR and large data in BLOB.

Firebird version and WireCompression	Content loading method	
	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


Speed fetch 1000 records
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)

Chapter 9. Conclusions

If you tried to place the Firebird server in the cloud and work with it via the Internet channel, but abandoned this idea due to performance issues when transferring BLOB objects, then we recommend trying again!

At the moment, this functionality is available in HQbird (from version 2024 R2 Update 2 and more recent, version 5.0.3.1629+). Vanilla version Firebird 5.0.3 has not yet been released, but you can try snapshots of Firebird 5.0.3.