Building an “architecture agnostic” Firebird

Ann Harrison

IBPhoenix

Overview

Currently, Firebird databases are architecture specific. They use the local machine format for binary values and the native alignment. To distribute data in a Firebird database on a CD, for example, you need a different database for Intel platforms and Sun platforms. The remote interface and gbak produce “architecture agnostic” formats of data, but the database itself is tied to the machine type that created it.

I looked at that problem recently and found an interesting solution.

Goals

1) no impact on the performance of native databases. Databases that were created on the current platform or a platform with the same byte ordering will not be affected.

2) minimal code changes

3) no ODS changes

Essentially, the differences between the on disk representation of firebird databases are in two areas: alignment and “endianness” or byte ordering.

For ODS-11, alignment is the same all other currently supported platforms.

Byte ordering

However, our platforms do us different byte ordering for numbers. Some systems represent multi-byte numbers with the lowest byte first, others with the highest byte first. The PDP-ll represented 4-byte binary numbers with the second highest byte first, then the highest, then the lowest, then the second lowest. Happily, that’s not a supported platform.

From the Wikipedia:

Decimal value:	**1245391901**
Hex value:	**4A3B2C1D**

Big endian:	**4A 3B 2C 1D**
Little endian:	**1D 2C 3B 4A**
Mixed endian:	**3B 4A 1D 2C**

Detecting an "other endian" database

When Firebird opens a database, it reads the database header page, starting with the page header. If the database was created on a big-endian system and read on a little-endian system, reading the page header will produce a "bad checksum" error. The error is a holdover from the ancient days when pages were actually checksummed; now it just means that the value in the 16 bit checksum element of the page header is not "12345".

When an "endian agnostic" Firebird encounters that error, it converts the checksum to the other endian format – reversing the bytes. If that inversion produces 12345, the mixed-endian Firebird converts the whole header page and checks that the implementation belongs to an architecture with compatible alignment, and that the database is at least ODS 11. If those checks pass, Firebird sets a flag saying that this is an "other endian" database and continues to open it.

Inverting an "other endian" database

When Firebird reads a page from an "other endian" database, it converts all the structural information on the page to the "native endian" format. Before Firebird writes a page from the page cache to an "other endian" database, it copies the page to a spare buffer and converts structural information back to "other endian" format. Those two conversions - immediately after reading and immediately before writing - handle all the endian problems except those in user and system data.

User data is converted in the record buffer, after it is expanded by SQZ_decompress, using the format descriptor named in the record header. This gets a bit tricky when applying deltas, since the delta must be applied to the "other endian" record, not the local format, but those cases are handled. Blob segment lengths and the page pointers for upper level blobs are also converted. Blob data is not.

The trial implementation did not include arrays, but array data will be converted. Nor did it include external tables. There is no good way to recognize the endian affinity of an external file, but we could add information to the external table declaration.

Implementation details...

Once Firebird has established that a database is "other endian" but has a matching alignment and is ODS 11 or greater, it transforms the structure of each page into the local endian format when the page is read into cache. When the page is written, Firebird copies it to a scratch buffer and converts the structural data back to the original endian format.

Specifically... on read or write, a pointer to the page is sent to the invert code, together with a boolean that indicates whether the operation is read or write, and a pointer to the

master database block. The code that performs the inversion is the same for input and output, except where the page includes a count of the items on it. In that case, the count must be kept in the local format so iteration works.

Page conversions

The page header of every page is converted.

The conversion then cases on page type to convert the remainder of the page. For some page types, it passes information from the master database block.

Header page (HDR). The invert code switches each binary field and the timestamp field in the fixed part of the header. Data stored in clumplets in the variable portion of the header is not converted because the clumplets are designed to be architecture independent.

Page Inventory Page (PIP). The invert code fixes the pip_min field which indicates the lowest unallocated page in the bit vector that fills the rest of the page. The bit vector is unaffected.

Transaction Inventory Page (TIP). The invert code fixes the page number of the next TIP. The transaction array is unaffected.

Pointer Page (PPG). If the page is being written, the current value of ppg_count is saved then inverted, otherwise the count is inverted and saved. Then the sequence, next page number, relation id, min_space, and max_space are inverted. Then the array of page numbers is inverted, using the saved count to avoid inverting the page fill level information at the bottom of the pages.

Data Pages (DPG). As with the PPG, save the count in its local format. Then fix the sequence and relation. Then, using the count as a limit, iterate through the index changing the offset and length of each entry. Use the offset to find the header for each entry. Pass the in/out boolean to the code that converts headers so it will look at the local version of the flags word and case on the type : blob, record, or fragmented record. In the case of a blob header, if the blob is not a level 0, use the local value of blh_max_sequence as a limit and invert each page number in the array of blob pages numbers.

Index Root Pages (IRT). As with PPG, save the count in local format. Iterate through the descriptors, inverting the binary values in the descriptors. For each descriptor, also invert the values in the individual field descriptors. Since field descriptors come up from the bottom of the page, the routine that inverts index root pages takes the page size as an extra parameter.

Index pages (BTR). Invert the fixed information at the top of the page (sibling, left sibling, prefix total, relation, and length. For indexes with jump nodes - which is all

ODS-11 indexes with a page size > 1024 and large keys and all record number, also invert the description of the jump node area and the pointers in the jump nodes.

Blob pages (BLB). Invert the fixed header portion of the page (lead page, sequence, and length) then if the blob page is not level zero, iterate through it, inverting the page numbers. There's nothing interesting at the bottom of a blob page, so just stop the iteration before the end of the page.

Generator pages (GPG). Invert the fixed header portion of the page, then starting at the first generator location invert all the generators - including those allocated and those that might be allocated in the future. This routine takes an extra parameter, the number of generators that fit on a generator page.

Inverting user data

The conversion of user data is also very localized. The module vio.cpp handles the conversion of compressed data as stored in the database to expanded records that are used in the higher levels of Firebird. With the exception of blob segment lengths and array data, all inversions are called from vio.

Records on disk are stored with the format number that was current when they were stored. When a field in a table is added, dropped, or altered, the table gets a new format descriptor. Old records are stored in the old format, new records in the new format. So blindly inverting a data stream based on the current table definition works badly. However, vio needs the same information as the inversion code so it can present the record to the higher levels in the newest format.

The format is a descriptor of the record, field by field, type and length. Data is inverted from VIO, after it has been expanded when it is read, and before it is compressed on write. The inversion routine takes the record pointer and the format block and uses the format block to iterate through fields, inverting as necessary.

Subtle points.

When the transaction needs to reference a back version and the back version is a delta, the primary record version is inverted to local format when it is read, then converted back to its native format to have the differences applied. The result is then inverted again to local format. This gets slightly more complicated when garbage collecting because the record must be in local format when it goes on the going and staying lists for blob and index garbage collection.

Within a record, there’s an additional complication with blobs. A blob field is represented in a record as a blob ID. Blob ID's have two states - permanent and temporary. They're handled differently because the temporary blob id is 32 bits of null value and 32 bits to locate the temporary blob. The permanent blob id has a 16 bit relation id, eight unused bits, and a 40 bit record number which is one byte plus one native format 32 bit integer.

Blob data

Generally blob data is left to the application designer, as the database has no clue about the content of blobs. The exception in user data is that segment lengths in segmented blobs must be inverted.

However, blob data in system tables is Firebird's responsibility. Calls from met.epp and dfw.epp handle the conversion of format descriptors and the RDB$RUNTIME blob.

Outside the engine

Unlike the other utilities, gstat does its own I/O on the database file. It calls the routines that invert structural data to work on other-endian databases. Since that's a single call to the inversion code on read and gstat doesn't ever write anything, that should be easy.

Impact of the changes

The total amount of code in existing modules (excluding debugging code) is less than 100 lines. The impact on performance of local-endian databases is a few if statements. The inversion code is very simple. The result is that any Firebird using ODS-11 or greater can read, write, and change the metadata of a database created anywhere.