Database - Bug #5802

a date temp-table field gets assigned to incorrect date

11/02/2021 11:43 AM - Constantin Asofiei

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**case_num:**	
vendor_id:	GCD		

Description

History

#1 - 11/02/2021 11:45 AM - Constantin Asofiei

This test will assign tt1.f1 to 1900-11-10, only on the first iteration. I saw this when working with #3310 SOAP support, and I thought there is a problem with getting the date from the SOAP envelope all the way to remote appserver call - but it's not, the problem is that for some reason FWD sets the field to 1900-11-10, but only on the very first call:

```
def temp-table tt1 field f1 as date.
def var i as int.
def var d as date.
d = date("11/11/1900").
def var c as int.

do i = 1 to 1000:
   create tt1.
   tt1.f1 = d.
   if tt1.f1 <> d then c = c + 1. // first attempt sets tt1.f1 to '1900-11-10'
end.
message "failures:" c.
```

The problem looks like is in Record._setDate:

```
      Date datum = null;
      if (d != null && !d.isUnknown())
      {
         datum = new Date(d.dateValue(null).getTime()); // here, sometimes the java.util.Date is wrong
      }

      setDatum(offset, datum);
```

#2 - 11/02/2021 09:03 PM - Ovidiu Maxiniuc

I did a bit of investigations. Here is the stack downstream from _setDate:

```
      at com.goldencode.p2j.util.date.neutralMillisToCalendar(date.java:1280)
      at com.goldencode.p2j.util.date.neutralMillisToDate(date.java:1224)
      at com.goldencode.p2j.util.date.dateValue(date.java:2817)
      at com.goldencode.p2j.persist.Record._setDate(Record.java:469)
```

The problem is the GregorianCalendar gc used at this level. For all calls, there is a single instance of the calendar which is altered and then reused. It seems like the first setup is somewhat incorrect. Since the default timezone is unlikely to change, only the very first access is broken.

#3 - 11/03/2021 09:10 AM - Greg Shah

> It seems like the first setup is somewhat incorrect. Since the default timezone is unlikely to change, only the very first access is broken.

Can you be more specific about the problem?

#4 - 11/03/2021 08:05 PM - Ovidiu Maxiniuc

I continued investigations on this issue. The problem seems to be at the moment when the timezone of the GregorianCalendar is updated. Remember that the gc instance is reused for each usage of the same timezone (see date.neutralMillisToCalendar() which gets the gc cached instance from getZoneCalendar()).

I added some traces in the method and here is what I get:

1st iteration:

- TimeZone zone parameter is: Central European Time / raw offset: 3600000
- returned from getZoneCalendar(): 2021-11-04T00:28:25.792+0100
- tzOffset as obtained from gc.get(Calendar.ZONE_OFFSET): 3600000
- after switching to new instant (gc.setTimeInMillis(millis)), gc prints as: 1900-11-10T23:09:21.000+0009
  Note that the datetime instant is off by about 50 minutes, and the timezone was set to an invalid +0009!

2nd iteration:

- TimeZone zone parameter is the same: Central European Time / raw offset: 3600000
- returned from getZoneCalendar(): 1900-11-10T23:09:21.000+0009. The change from 1st usage was kept in cache
- tzOffset as obtained from gc.get(Calendar.ZONE_OFFSET): 561000. This is interesting. At the same time, it justifies the skew in calendar. The difference between 3600000 and 561000 represents exactly those missing 50 minues.
- the attempt to reset gc makes it switch to desired value: 1900-11-11T00:00:00.000+0009. Well, with the exception of the timezone which remains odd.

3rd and next iteration, both the tzOffset and gc remain unchanged and incorrect.

These tests were performed with system timezone set to Europe/Paris (GMT+1). Switching to Bucharest (GMT+2), the values change to:

1st iteration: 2021-11-04T01:46:40.516+0200 / 7200000 / 1900-11-10T23:44:24.000+0144
2nd iteration: 1900-11-10T23:44:24.000+0144 / 6264000 / 1900-11-11T00:00:00.000+0144
etc..

This is strange. Where does these strange time offset (+0009, +0144) come from? They are supposed to be multiple of 15 minutes if not full hours.

I tried debugging the process. I went as far as:

```
      at sun.util.calendar.ZoneInfo.getOffsets(ZoneInfo.java:259)
      at sun.util.calendar.ZoneInfo.getOffsets(ZoneInfo.java:236)
      at java.util.GregorianCalendar.computeFields(GregorianCalendar.java:2336)
      at java.util.GregorianCalendar.computeFields(GregorianCalendar.java:2308)
      at java.util.Calendar.setTimeInMillis(Calendar.java:1804)
      at com.goldencode.p2j.util.date.neutralMillisToCalendar(date.java:1269)
```

but at this level the code is obfuscated, even the IDE is not able to display the local values. However, there is an array member field of the ZoneInfo which looks like this: offsets = [7200000, 6264000, 10800000, 3600000].

I can interpret that

- the 1st element 7200000 is the Bucharest timezone (GMT+2),
- the 3rd element 10800000 is the summer time (GMT+3),
- the 4th element is the DST (1 hour).

Question of the day: what does 6264000 represent and why is it used as a valid time offset in these instances of gc?
The second question: why is the time offset of gc changed when setTimeInMillis() is invoked? We cache it assuming the TZ will not change. In fact, the original TZ is the key on which the cached calendars are mapped in date.tzcals.

LE: I forgot to mention that switching to Europe/London time-zone will cause the testcase to work correctly. I expected so happen so.

#5 - 11/04/2021 11:09 AM - Greg Shah

Do you have a simple Java recreate for this? I assume we can recreate this with simple Java code with a main() that uses the FWD classes.

Then we can test it here and on older versions of JVM and/or FWD. This code hasn't changed in a long time. I don't think this was seen in the past. I wonder if this is a JVM bug, probably a recent one.

#6 - 11/05/2021 12:13 PM - Ovidiu Maxiniuc

I used Constantin's test case. I added traces (System.out.format()) in date.neutralMillisToCalendar(long millis, TimeZone zone):

- gc.getTime().getCalendarDate() after obtaining gc at line 1259;
- tzOffset after its assign (line 1262);
- again, gc.getTime().getCalendarDate() just before the return (line 1281).

To see the internals, you need to step into java time-related code.

Here is a more isolated testcase, but it will not expose the cause:

```
   public static void main(String[] args)
   {
      date d = new date(new integer(11), new integer(11), new integer(1900));
      Date jDate1 = d.dateValue(null);
      System.out.format("ABL: %s -> Java: %s %s\n", d.toStringExport(), jDate1.toString(), jDate1.getTime());
      Date jDate2 = d.dateValue(null);
      System.out.format("ABL: %s -> Java: %s %s\n", d.toStringExport(), jDate2.toString(), jDate2.getTime());
      Date jDate3 = d.dateValue(null);
      System.out.format("ABL: %s -> Java: %s %s\n", d.toStringExport(), jDate3.toString(), jDate3.getTime());
   }
```

For my (GMT+2) it prints:

```
ABL: 11/11/1900 -> Java: Sat Nov 10 23:44:24 EET 1900 -2181866400000
ABL: 11/11/1900 -> Java: Sun Nov 11 00:00:00 EET 1900 -2181865464000
ABL: 11/11/1900 -> Java: Sun Nov 11 00:00:00 EET 1900 -2181865464000
```

I extracted a pure Java code:

```
   public static void main(String[] args)
   {
      final long JULIAN_EPOCH_OFFSET = 2440589L;
      final long MILLIS_PER_DAY = 1000 * 60 * 60 * 24;
      final int  ABL_11_NOV_1900 = 2415336;
      final long date_millis = (ABL_11_NOV_1900 - JULIAN_EPOCH_OFFSET) * MILLIS_PER_DAY;

      final SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSSZ");
      final TimeZone tz = TimeZone.getDefault();
      final GregorianCalendar gc = new GregorianCalendar(tz);

      int tzOffset1 = gc.get(Calendar.ZONE_OFFSET);
      gc.setTimeInMillis(date_millis - tzOffset1);
      System.out.format("1: Offset: %s, Date: %s, Millis: %s\n"
, tzOffset1, sdf.format(gc.getTime()), gc.getTimeInMillis());

      System.out.format("Original millis: %s\n", date_millis);

      int tzOffset2 = gc.get(Calendar.ZONE_OFFSET);
      gc.setTimeInMillis(date_millis - tzOffset2);
      System.out.format("2: Offset: %s, Date: %s, Millis: %s\n"
, tzOffset2, sdf.format(gc.getTime()), gc.getTimeInMillis());

      int tzOffset3 = gc.get(Calendar.ZONE_OFFSET);
      gc.setTimeInMillis(date_millis - tzOffset3);
      System.out.format("2: Offset: %s, Date: %s, Millis: %s\n"
, tzOffset3, sdf.format(gc.getTime()), gc.getTimeInMillis());
   }
```

This prints:

```
Original millis: -2181859200000
1: Offset: 7200000, Date: 1900-11-10 23:44:24.000+0144, Millis: -2181866400000
2: Offset: 6264000, Date: 1900-11-11 00:00:00.000+0144, Millis: -2181865464000
2: Offset: 6264000, Date: 1900-11-11 00:00:00.000+0144, Millis: -2181865464000
```

#7 - 11/05/2021 12:20 PM - Constantin Asofiei

Ovidiu, is this only with this specific 1900-11-11 date? Can you run all dates from 1900 to 2030 in a loop, a log the failures?

#8 - 11/05/2021 12:24 PM - Greg Shah

I don't see the issue on my system:

```
1: Offset: -18000000, Date: 1900-11-11 00:00:00.000-0500, Millis: -2181841200000
Original millis: -2181859200000
2: Offset: -18000000, Date: 1900-11-11 00:00:00.000-0500, Millis: -2181841200000
2: Offset: -18000000, Date: 1900-11-11 00:00:00.000-0500, Millis: -2181841200000
```

My java -version output:

```
openjdk version "1.8.0_292"
OpenJDK Runtime Environment (build 1.8.0_292-8u292-b10-0ubuntu1~20.10-b10)
OpenJDK 64-Bit Server VM (build 25.292-b10, mixed mode)
```

#9 - 11/05/2021 12:36 PM - Constantin Asofiei

If I'm using TimeZone.getTimeZone("Europe/Bucharest"), the program shows:

```
1: Offset: 7200000, Date: 1900-11-10 23:44:24.000+0144, Millis: -2181866400000
Original millis: -2181859200000
2: Offset: 6264000, Date: 1900-11-11 00:00:00.000+0144, Millis: -2181865464000
2: Offset: 6264000, Date: 1900-11-11 00:00:00.000+0144, Millis: -2181865464000
```

If I'm using PST, it shows:

```
1: Offset: -28800000, Date: 1900-11-11 09:44:24.000+0144, Millis: -2181830400000
Original millis: -2181859200000
2: Offset: -28800000, Date: 1900-11-11 09:44:24.000+0144, Millis: -2181830400000
2: Offset: -28800000, Date: 1900-11-11 09:44:24.000+0144, Millis: -2181830400000
```

for EST it shows:

```
1: Offset: -18000000, Date: 1900-11-11 06:44:24.000+0144, Millis: -2181841200000
Original millis: -2181859200000
2: Offset: -18000000, Date: 1900-11-11 06:44:24.000+0144, Millis: -2181841200000
```

```
2: Offset: -18000000, Date: 1900-11-11 06:44:24.000+0144, Millis: -2181841200000
```

This may be a JRE bug specific to Europe/Bucharest time zone, and maybe related to 1900's dates (or I'm missing some concept on how the time-zone affects a date in the JRE implementation).

But a potential problem may be: when working with pure dates, maybe FWD should not depend on/use the time part at all. Isn't a calendar date the same regardless of the time-zone you set to it (and the time portion it should always be 00:00:00)?

#10 - 11/05/2021 12:41 PM - Ovidiu Maxiniuc

Same Java version here.
Greg, please try with ABL_11_NOV_1900 = 2430137 (05/21/1941).

#11 - 11/05/2021 12:46 PM - Constantin Asofiei

Greg, what does TimeZone.getDefault(); show for you?

Ovidiu: I don't think this is specific on a user's machine (as long as TimeZone.getDefault() is not used), you can experiment with different time-zones via TimeZone.getTimeZone.

Same Java version for me, too.

#12 - 11/05/2021 12:47 PM - Ovidiu Maxiniuc

Constantin Asofiei wrote:

> Ovidiu, is this only with this specific 1900-11-11 date? Can you run all dates from 1900 to 2030 in a loop, a log the failures?

```
   final static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSSZ");

   public static void main(String[] args)
   {
      final TimeZone tz = TimeZone.getDefault();
      final GregorianCalendar gc = new GregorianCalendar(tz);

      int errs = 0;
      for (int i = 0; i < 130 * 366; i++)
      {
         errs += doTest(2415336 + i);
      }
      System.out.format("%s errors from %s dates.", errs, 130 * 366);
   }

   static int doTest(int ablDate)
   {
      final long JULIAN_EPOCH_OFFSET = 2440589L;
      final long MILLIS_PER_DAY = 1000 * 60 * 60 * 24;

      final TimeZone tz = TimeZone.getDefault();
      final GregorianCalendar gc = new GregorianCalendar(tz);
      final long date_millis = (ablDate - JULIAN_EPOCH_OFFSET) * MILLIS_PER_DAY;

      int tzOffset1 = gc.get(Calendar.ZONE_OFFSET);
```

```
      gc.setTimeInMillis(date_millis - tzOffset1);
//      System.out.format("1: Offset: %s, Date: %s, Millis: %s\n", tzOffset1, sdf.format(gc.getTime()), gc.get
TimeInMillis());

      int tzOffset2 = gc.get(Calendar.ZONE_OFFSET);
      gc.setTimeInMillis(date_millis - tzOffset2);
//      System.out.format("2: Offset: %s, Date: %s, Millis: %s\n", tzOffset2, sdf.format(gc.getTime()), gc.get
TimeInMillis());

      int tzOffset3 = gc.get(Calendar.ZONE_OFFSET);
      gc.setTimeInMillis(date_millis - tzOffset3);
//      System.out.format("2: Offset: %s, Date: %s, Millis: %s\n", tzOffset3, sdf.format(gc.getTime()), gc.get
TimeInMillis());

      if (tzOffset1 != tzOffset2 || tzOffset2 != tzOffset3)
      {
         System.out.format("%s: %s %s %s %s\n", ablDate, new
 date(ablDate).toStringExport(), tzOffset1, tzOffset2, tzOffset3);
         return 1;
      }
      return 0;
   }
```

For GMT+2 (Bucharest) I get: 11213 errors from 47580 dates. (dates between 11/11/1900 and 07/24/1931)
For US/East-Indiana I get: 36513 errors from 47580 dates. (dates between 05/21/1941 and 10/29/00)

#13 - 11/05/2021 12:48 PM - Ovidiu Maxiniuc

Constantin Asofiei wrote:

> Ovidiu: I don't think this is specific on a user's machine (as long as TimeZone.getDefault() is not used), you can experiment with different time-zones via TimeZone.getTimeZone.

I changed the time configuration of my workstation so the TimeZone.getDefault() is fine.

#14 - 11/05/2021 01:17 PM - Greg Shah

Ovidiu Maxiniuc wrote:

> Same Java version here.
> Greg, please try with ABL_11_NOV_1900 = 2430137 (05/21/1941).

```
sun.util.calendar.ZoneInfo[id="America/New_York",offset=-18000000,dstSavings=3600000,useDaylight=true,transiti
ons=235,lastRule=java.util.SimpleTimeZone[id=America/New_York,offset=-18000000,dstSavings=3600000,useDaylight=
true,startYear=0,startMode=3,startMonth=2,startDay=8,startDayOfWeek=1,startTime=7200000,startTimeMode=0,endMod
e=3,endMonth=10,endDay=1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]]
1: Offset: -18000000, Date: 1941-05-21 01:00:00.000-0400, Millis: -903034800000
Original millis: -903052800000
2: Offset: -18000000, Date: 1941-05-21 01:00:00.000-0400, Millis: -903034800000
2: Offset: -18000000, Date: 1941-05-21 01:00:00.000-0400, Millis: -903034800000
```

#15 - 11/05/2021 01:21 PM - Greg Shah

> But a potential problem may be: when working with pure dates, maybe FWD should not depend on/use the time part at all. Isn't a calendar date the same regardless of the time-zone you set to it (and the time portion it should always be 00:00:00)?

At the time I wrote this code, the use of Date with the day, month, year was already deprecated. In addition, we had a range of other features that included things like date math. Using the GregorianCalendar was the best option. Sadly, I'm not sure we have a better option now. I'm open to suggestions and hopefully I'm wrong.

#16 - 11/05/2021 01:33 PM - Constantin Asofiei

Greg Shah wrote:

> At the time I wrote this code, the use of Date with the day, month, year was already deprecated. In addition, we had a range of other features that included things like date math. Using the GregorianCalendar was the best option. Sadly, I'm not sure we have a better option now. I'm open to suggestions and hopefully I'm wrong.

The possible problem I see is that we rely on time-zone offsets (and DST) when working with pure dates (we are subtracting or adding millis related to this)- we should ensure that each date is on midnight on that timezone and DST.

#17 - 11/05/2021 01:34 PM - Ovidiu Maxiniuc

I believe it may be possible that we are doing some over-calculations with the date. From my experience with this BDT, it is not more than an integer in Progress. Indeed, we do store it as such (julian) and I think this is enough. If needed, we may use some lookup tables for non-contiguous date intervals when converting to plain Date / sql Date. The rest of computations are probably not necessary.

There is one more alternative: switching to new Java 8 Date/Time API.

#18 - 11/05/2021 10:34 PM - Ovidiu Maxiniuc

I did some small scale tests/investigations. The LocalDate might be the utility date needs. There are methods for constructing these immutable objects like this:

```
LocalDate ldNow = LocalDate.now(); // ABL's TODAY equivalent
LocalDate ld1900 = LocalDate.of(1900, 11, 11); // the date which started the thread
LocalDate ldJavaEpoch = LocalDate.ofEpochDay(0); // the Java Epoch
```

This seems to work fine with modern dates, the bad part is that dates older than the Gregorian leap (1582-10-15) are not well supported.

#19 - 11/09/2021 12:14 PM - Greg Shah

If LocalDate handled historical dates properly, then we would certainly need to consider moving to it. Without that support, I think we are "stuck" on the GregorianCalendar usage. At some point we also will need the ability to change the timezone per session, which is something we need to ensure is supported in any framework we use (for 2 threads using differnet timezones, the machine time may correspond to 2 different days).

We could consider a framework like Time4J but I hesitate to add such a dependency unless it really makes our implementation much simpler and more robust.