Timestamp interoperability using writer-agnostic adjustments

Zoltan Ivanfi, 2017-07-18

Introduction

This is an attempt at fixing the timestamp incompatibility between Impala, Hive and SparkSQL. There was an earlier attempt that required different adjustment logic depending the writer component. The Spark community opposed that change, asking for a more clear approach. This new proposal addresses these concerns by being writer-agnostic.

Background

According to the ANSI SQL standard, the plain unqualified TIMESTAMP type should have the semantics of the TIMESTAMP WITHOUT TIMEZONE type, which in turn is described as having timezone-agnostic behaviour: regardless of the local or session-local timezone, the timestamp value should be displayed the same. It should essentially behave like a severely restricted string type.

Some SQL engines of the Hadoop stack adhere to the behaviour described above, but in some others the TIMESTAMP type has TIMESTAMP WITH TIMEZONE semantics, which essentially means that the timestamp is stored normalized to UTC. Timestamps of this semantics may be displayed differently in different timezones so that displayed value in the local timezone corresponds to the same time instant as the original timestamp did.

Specifically,

- Impala follows timezone-agnostic behaviour,
- SparkSQL follows UTC-normalized behaviour,
- Hive follows timezone-agnostic behaviour except for Parquet tables, where it follows UTC-normalized behaviour.

This results in incompatibility between these components. Timestamps written by one SQL engine may result in different values when read back by other SQL engines.

Scope

Goals

- Interoperability of Parquet timestamps between major SQL engines of the Hadoop stack. Timestamps written by one SQL engine should be displayed the same in other SQL engines.
- ANSI-compliant timezone-agnostic behaviour of the TIMESTAMP SQL type.
- The properties listed above should be available for existing data as well.
- Temporary workaround for existing data (and also for new data until proper support is added for new SQL types that are explicit about their behaviour).
- No behaviour change without explicit user request.

Explicit non-goals

- UTC-normalized timestamp behaviour across components (TIMESTAMP WITH TIMEZONE type of SQL).
- Direct data access in Spark without using SQL.
- Addressing the inaccuracy of Impala's timezone database.
- Existing data that is already inconsistent due to being written by different components or in different timezones.

Definition of terms

STZ

STZ stands for stored timezone. It is a table-specific setting that is specified in the parquet.timezone-adjustment table property (exact name open for suggestions). For existing tables, STZ must be set to match the timezone that was in effect when writing the data. For new tables, STZ can be set arbitrarily, but the intended/recommended way of usage is to set it to UTC.

LTZ

LTZ stands for local timezone and corresponds to the actual timezone in effect. In components that support a session-local timezone, LTZ corresponds to the session-local timezone.

UTC

UTC is the reference timezone in terms of which all other timezones are defined. UTC does not observe daylight saving time.

Timestamps vs. time instants

According to the SQL definition, one should not think about timestamps as actual instants in time but as strings instead that are ambiguous without an explicit timezone. (This is only true for the unqualified TIMESTAMP type or the TIMESTAMP WITHOUT TIME ZONE type. The TIMESTAMP WITH TIME ZONE type does correspond to an actual instant in time.)
Let $Instant_{TZ}\ (x)$ denote the time instant that x interpreted in timezone TZ corresponds to. (If x has a timezone specifier part, that has to be ignored in order to interpret the rest in timezone TZ.) If x is not valid in timezone TZ (due to being skipped by a DST change), this should be handled gracefully by using a valid timestamp close to the invalid range, but differs only in the hour part, similar to how Java handles such cases.

$TZ_1 \rightarrow TZ_2$ conversion

$y = Convert_{TZ_1 \rightarrow TZ_2}(x)$denotes converting the timezone-agnostic timestamp x from timezone TZ_1 to timezone TZ_2. The result is another timezone-agnostic timestamp y that satisfies $Instant_{TZ_2}(y) = Instant_{TZ_1}(x)$, i.e., the result y will correspond to the same time instant when interpreted in timezone TZ_2as x corresponds to when interpreted in timezoneTZ_1. For example, $Convert_{America/Los\ Angeles \rightarrow Europe/Berlin}(\text{2017-07-20 10:00}) = \text{2017-07-20 19:00}$, because 2017-07-20 10:00 in Los Angeles corresponds to the same instant in time as 2017-07-20 19:00 does in Berlin. If no such equivalent timestamp exists due to timestamps skipped by DST changes, the original timestamp was invalid in timezone X. This should be handled gracefully by using a valid timestamp close to the invalid range, but differs only in the hour part, similar to how Java handles such cases.

One way to conceptualize UTC-normalized systems is by saying that in such systems parsing timestamps automatically and implicitly involves a $Convert_{LTZ \rightarrow UTC}$ operation and displaying timestamps automatically and implicitly involves a $Convert_{UTC \rightarrow LTZ}$ operation.

Exit criteria

Impala

- If the parquet.timezone-adjustment table property is not set, Impala behavior will not change.
- If the parquet.timezone-adjustment table property is set:

    - For Parquet files, Impala will adjust timestamp values upon reading and writing based on the value of the table property:
        - The value of the table property must be a valid timezone specifier, otherwise when a query tries to access the table either for reading or writing, Impala will indicate an error to the user.
        - When reading timestamps, values must be adjusted according to the following formula: $y = Convert_{UTC \rightarrow STZ}(x)$, where x is the on-disk timestamp to read and y is the resulting in-memory timestamp.
        - When writing timestamps, values must be adjusted in the opposite direction according to the following formula: $y = Convert_{STZ \rightarrow UTC}(x)$, where x is the in-memory timestamp to be written and y is the resulting on-disk timestamp.
        - Tables created using CREATE TABLE LIKE <other table> will copy the parquet.timezone-adjustment property of the table that is copied.
        - If the specified time zone is UTC, no noticeable performance regression should be observable.
    - For files in any other file formats than Parquet, Impala behavior will not change.

Hive

- If the parquet.timezone-adjustment table property is not set, Hive behavior will not change.
- If the parquet.timezone-adjustment table property is set:
    - For Parquet files, Hive will adjust timestamp values upon reading and writing based on the value of the table property:
        - The value of the table property must be a valid timezone specifier, otherwise when a query tries to access the table either for reading or writing, Hive will indicate an error to the user.
        - When reading timestamps, values must be adjusted according to the following formula: $y = Convert_{LTZ \rightarrow UTC}(Convert_{UTC \rightarrow STZ}(x))$, where x is the on-disk timestamp to read and y is the resulting in-memory timestamp. (Since in Hive displaying a timestamp automatically and implicitly involves a $Convert_{UTC \rightarrow LTZ}$ operation, the displayed value $z = Convert_{UTC \rightarrow LTZ}(y) = Convert_{UTC \rightarrow LTZ}(Convert_{LTZ \rightarrow UTC}(Convert_{UTC \rightarrow STZ}(x)))$ $= Convert_{UTC \rightarrow STZ}(x)$ will be identical to the one calculated for Impala above.)
        - When writing timestamps, values must be adjusted in the opposite direction according to the following formula: $y = Convert_{STZ \rightarrow UTC}(Convert_{UTC \rightarrow LTZ}(x))$, where x is the in-memory timestamp to be written and y is the resulting on-disk timestamp.

        - Tables created using CREATE TABLE LIKE <other table> will copy the parquet.timezone-adjustment property of the table that is copied.
    - For files in any other file formats than Parquet, Hive behavior will not change.

SparkSQL

- If the parquet.timezone-adjustment table property is not set, SparkSQL behavior will not change.
- If the parquet.timezone-adjustment table property is set:
    - For Parquet tables, SparkSQL will adjust timestamp values upon reading and writing based on the value of the table property:
        - The value of the table property must be a valid timezone specifier, otherwise when a query tries to access the table either for reading or writing, SparkSQL will indicate an error to the user.
        - When reading timestamps, values must be adjusted according to the following formula: $y = Convert_{LTZ \rightarrow UTC}(Convert_{UTC \rightarrow STZ}(x))$, where x is the on-disk timestamp to read and y is the resulting in-memory timestamp. (Since in Spark displaying a timestamp automatically and implicitly involves a $Convert_{UTC \rightarrow LTZ}$ operation, the displayed value $z = Convert_{UTC \rightarrow LTZ}(y)$ $= Convert_{UTC \rightarrow LTZ}(Convert_{LTZ \rightarrow UTC}(Convert_{UTC \rightarrow STZ}(x)))$ $= Convert_{UTC \rightarrow STZ}(x)$ will be identical to the one calculated for Impala above.)
        - When writing timestamps, values must be adjusted in the opposite direction according to the following formula: $y = Convert_{STZ \rightarrow UTC}(Convert_{UTC \rightarrow LTZ}(x))$, where x is the in-memory timestamp to be written and y is the resulting on-disk timestamp.
        - Tables created using CREATE TABLE LIKE <other table> will copy the parquet.timezone-adjustment property of the table that is copied.
    - For files in any other file formats than SparkSQL, Hive behavior will not change.

Extensibility

If needed, similar <file-format-name>.timezone-adjustment table properties may be added in the future for other file formats. Please note that we can not use the same table property for all file formats, because timestamps are already handled differently in some formats, therefore no uniform adjustment can fix timestamps for all file formats of the same table at the same time.

Justification

The mechanism described above approximates timezone-agnostic behavior in all components.

Impala uses a timezone-agnostic in-memory timestamp representation, thereby a UTC-normalized on-disk timestamp can easily be adjusted to its original displayed value upon reading by doing a UTC→STZ adjustment (provided that STZ is correctly set to the timezone that was in effect when writing the data). If the timestamps on disk are already timezone-agnostic, then specifying UTC as STZ will preserve the timestamps without modification upon reading. For writing new data, a reverse adjustment has to be applied.

Hive and Spark uses a UTC-normalized in-memory timestamp representation in which LTZ→UTC conversion automatically happens upon parsing timestamps and a UTC→LTZ conversion automatically happens upon displaying timestamps. The timestamps are saved to and read back from disk UTC-normalized. In order to approximate timezone-agnostic behaviour, one has to do an adjustment that results in a UTC value that will result in the desired display value after the UTC→LTZ conversion that happens automatically upon displaying timestamps.

For example: The timestamp 2:00 (date and sub-hour parts omitted for brevity) was saved in the "America/Los Angeles" timezone as 10:00 UTC. When loading this timestamp later or on a different system, the local timezone is "Europe/Berlin". The timezone offsets corresponding to the timestamp are UTC-08:00 and UTC+01:00 for STZ and LTZ respectively. One can simulate timezone-agnostic behaviour on top of this data as follows:

- Calculate that 10:00 UTC is 2:00 in "America/Los Angeles".
- Reinterpret this 2:00 as being in "Europe/Berlin" without actually modifying it.
- Calculate that 2:00 in "Europe/Berlin" is 1:00 UTC.
- This value (1:00 UTC) should be used as the UTC-based in-memory representation.

Why is this good?

- Since "America/Los Angeles" is the stored time zone, we know that the user actually wanted to save the 2:00 displayed representation, but it was saved as 10:00 UTC instead. The first conversion gives us the proper display time.
- This is not enough, since the in-memory representation is still UTC and is adjusted to local time for displaying, which would result in 3:00 being displayed instead of 2:00.
- So this 2:00 should be interpreted as a local time and converted to 1:00 UTC for in-memory representation. When this gets displayed, it is shown as 2:00 as desired. (The reason why this method is only an approximation is that timestamps skipped due to DST changes can not be represented this way.)

Please note that if STZ==LTZ then this is a no-op, because the two conversions are opposites of each other. This case should be explicitly handled in a more efficient way (by skipping the conversions).

Upon writing, the reverse calculation should happen.