Handling of DateTime Default Values in Prisma

Historically Prisma has been using two ways to define default values:

- As a direct definition `@default("1996-12-19T16:39:57-08:00")`
- . . . or as a database-generated default `@default(dbgenerated("'16:20:00'"))`.

If **using the first definition, the default value of the column is inserted via the Query Engine**, while **the second lets the database set the value**. This has been convenient for us; we by default handle the defaults in the Engine – not needing to worry if the database does support them. We always have a default for a column, making things simpler. **Both definition variants are used by the Migration Engine to define the default value to the database DDL.**

Defaults in dates and times

A problem we face with our handling of datetimes is how in almost all of the types, if the user is using some other time zone than UTC, we either write a completely wrong value without any idea of the time zone, or we write a wrong value with the time zone; allowing user to fix the broken data.

The same issues affect both: storing values from the client, and with using the default values defined in the PSL.

An example of a data model defining a datetime default format using the RFC-3339 format:

```
model A {
  id  Int      @id
  val DateTime @default("1975-08-19T23:15:30+07:00")
}
```

When the user writes a query that creates a new `A` to the database and doesn't define `val` in the insert, the Query Engine generates a value based on the definition in the PSL `@default` attribute.

The Result in Different Datetime Types

In the following examples we take a look what actually gets stored when we either use a non-UTC default value or the client uses a non-UTC value in their insert query.

MySQL

In our example we store a default value of `@default("1975-08-19T23:15:30+07:00")` to different types of datetime columns, observing what goes wrong. In all cases the Query Engine converts the value to `1975-08-19T16:15:30Z` before storing (see below where we modify the value).

@db.DateTime

Storing the value `1975-08-19 16:15:30`. In the database we have values stored from other systems respecting the database default with the value of `1975-08-19 23:15:30`. We have two different default values in the database because of Prisma at this point.

This is the default native type Prisma uses for datetimes.

@db.Time

Storing the value `16:15:30`. Other systems store the given `23:15:30` and now we have two different default values in the database with no idea which ones are correct.

@db.Date

In this example we have in the PSL the default `1975-08-19T00:05:30+01:00`. Due to the conversion to UTC, the value gets converted into `1975-08-18T23:05:30Z`. We store the value of `1975-08-18` and other systems writing to the database use the value `1975-08-19`.

@db.Timestamp

A timestamp value is always a number of (micro)seconds since `1970-01-01 00:00:00` in UTC. Converting between timezones does not change the underlying value. The `TIMESTAMP` type is the only one that is not set wrong by Prisma if the user is using a non-UTC timezone.

PostgreSQL

In our example we store a default value of `@default("1975-08-19T23:15:30+07:00")` to different types of datetime columns, observing what goes wrong. In all cases the Query Engine converts the value to `1975-08-19T16:15:30Z` before storing.

@db.Timestamp

A timestamp value is always a number of (micro)seconds since `1970-01-01 00:00:00` in UTC. Converting between timezones does not change the underlying value. The `TIMESTAMP` type is the only one that is not set wrong by Prisma if the user is using a non-UTC timezone.

This is the default native type Prisma uses for datetimes.

@db.Timestamptz

A timestamp with an additional offset given. Prisma stores the timestamp as in the example above, and stores the offset as `0` while we should be storing `7`.

@db.Date

We have in the PSL the default `@default("1975-08-19T00:05:30+01:00")`. Due to the conversion to UTC, the value gets converted into `1975-08-18T23:05:30Z`. We store the value of `1975-08-18` and other systems writing to the database use the value `1975-08-19`.

@db.Time

Storing the value `16:15:30`. Other systems store the given `23:15:30` and now we have two different default values in the database.

@db.Timetz

Storing the value `16:15:30Z`. Other systems store the given `23:15:30+07:00` and now we have two different default values in the database.

SQL Server

In our example we store a default value of `@default("1975-08-19T23:15:30+07:00")` to different types of datetime columns, observing what goes wrong. In all cases the Query Engine converts the value to `1975-08-19T16:15:30Z` before storing.

@db.Datetime2 / @db.Datetime / @db.SmallDateTime

Storing the value `1975-08-19 16:15:30`. In the database we have values stored from other systems with the default of `1975-08-19T23:15:30+07:00`, which in those systems will be stored as `1975-08-19 23:15:30`. We have two different default values in the database because of Prisma at this point.

`DateTime2` is the default native type Prisma uses for datetimes.

@db.Date

We have in the PSL the default `@default("1975-08-19T00:05:30+01:00")`. Due to the conversion to UTC, the value gets converted into `1975-08-18T23:05:30Z`. We store the value of `1975-08-18` and other systems writing to the database use the value `1975-08-19`.

@db.Time

Storing the value `16:15:30`. Other systems store the given `23:15:30` and now we have two different default values in the database.

@db.DateTimeOffset

We don't really strong a wrong value here or lose information as we do with the other types. We just store a wrong offset in for the `datetimeoffset` type. We store `1975-08-19T16:15:30Z` and any other instance that is not Prisma writes `1975-08-19T23:15:30+07:00`. This data can be corrected, but is still not correct.

Datetime in Prisma Client

Assuming this model:

```
model A {
  id  Int      @default(autoincrement()) @id
  val DateTime @default("1975-08-19T23:15:30+07:00")
  foo String
}
```

Datetime via Prisma Client query

A typical Prisma Client request to insert a datetime would start from the Client query:

```
await prisma.a.create({ data: {
  val: new Date('August 19, 1975 23:15:30 GMT+07:00'),
  foo: "bar"
}})
```

We translate this to a GraphQL query, using the `JSON.stringify` function, which converts the datetime to UTC:

```
> const d = new Date('August 19, 1975 23:15:30 GMT+07:00')
undefined
> JSON.stringify(d)
'"1975-08-19T16:15:30.000Z"'
```

The client query in GraphQL then gets the value in UTC:

```
mutation {
  createOneA(data: {
    val: "1975-08-19T16:15:30.000Z"
    foo: "bar"
  }) {
    id
    val
    foo
  }
}
```

Crossing the boundary to the Rust code base in Query Engine, the datetime value will get converted to the internal `Value` representation. In the case of a `DateTime` value we parse the string to an instance of `DateTime<FixedOffset>`, keeping the given offset as-is.

Datetime via default in Prisma Client

In the case of using a default value for the datetime, the client in this case will not send anything to this field; letting the Query Engine to take the value from the PSL. PSL is parsing the given datetime as `DateTime<FixedOffset>`, giving it to the Query Engine as-is without converting it to UTC.

```
await prisma.a.create({ data: {
  foo: "bar"
}})
```

```
mutation {
  createOneA(data: {
    val: "1975-08-19T16:15:30.000Z"
    foo: "bar"
  }) {
    id
    val
    foo
  }
}
```

Result in the database

Before writing to the database, the SQL connector in the Query Engine converts the user-provided value or the PSL default once again to `DateTime<Utc>`.

```
// SQL with Params
Query: INSERT INTO "public"."A" ("val","foo") VALUES ($1,$2) RETURNING "public"."A"."id"
Params: [1975-08-19 16:15:30 UTC,"bar"]

// Result when read back
{ id: 5, val: 1975-08-19T16:15:30.000Z, foo: 'bar' }
```

The final outcome is we have no way of using any other timezones in the Prisma Client than UTC.

Our difference in this point of time between the user-provided and the default value is how the user-provided value is always in UTC due to the JavaScript code, and the default value in the given timezone.

> Broken: DateTime value read back after creating it (either via PSL `@default` or Prisma Client query) is different timezone than defined in PSL `@default` and Prisma Client query parameter.
>
> Broken: DateTime value read back after creating it is different depending on if it was created via `@default` or via `@default(dbgenerated(...))`

> Broken: Migrated version PSL type `DateTime` does not have timezone in **PostgreSQL** but accepts datetime strings with timezone in both PSL `@default` and Prisma Client query parameter.

Default Values in Migrations

In the next experiments, we try to migrate a default value to our database. First we'll try with using the default datetime type we choose for different databases, and what happens when we try to `push` the following schema:

```
model foo {
  id Int      @id @default(autoincrement())
  a  DateTime @default("1995-05-02T16:20:00+07:00")
}
```

MySQL 8.0.18

The SQL we generate:

```sql
CREATE TABLE `foo` (
    `id` INTEGER NOT NULL AUTO_INCREMENT,
    `a` DATETIME NOT NULL DEFAULT '1995-05-02T16:20:00+07:00',

    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

The result `prisma db push` gives to us:

```
reading the prisma schema from test.prisma
Error: Invalid default value for 'a'
```

MySQL 8.0.19

For all MySQL versions we generate the same SQL.

The result `prisma db push` gives to us:

```
Schema pushed to database. (1 steps)
```

Introspecting the data model we just pushed gives us a different result compared to where we started:

```
model foo {
  id Int      @id @default(autoincrement())
  a  DateTime @default(dbgenerated("'1995-05-02 09:20:00.000'"))
}
```

See how the time is different to the data model we started from.

We can try to push this introspected data model once more. Which works in all MySQL 8.0.19 examples.

We have a value that migrates, so we can see how it works with different native types:

1. Date

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default("1995-05-02T16:20:00+07:00") @db.Date
   }
   ```

   Push works. Introspection returns:

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default(dbgenerated("'1995-05-02'")) @db.Date
   }
   ```

2. Time

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default("1995-05-02T16:20:00+07:00") @db.Time
   }
   ```

   Push returns an error:

   ```
   Error: Invalid default value for 'a'
   ```

3. Timestamp

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default("1995-05-02T16:20:00+07:00") @db.Timestamp
   }
   ```

   Push works, we introspect the following data model back:

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default(dbgenerated("'1995-05-02 09:20:00'")) @db.Timestamp(0)
   }
   ```

MySQL 5.7.32

The result `prisma db push` gives to us:

```
Error: Incorrect datetime value: '1995-05-02T16:20:00+07:00' for column 'a' at row 1
```

MariaDB 10

The result `prisma db push` gives to us:

```
Error: Invalid default value for 'a'
```

PostgreSQL 14

```sql
CREATE TABLE "foo" (
    "id" SERIAL NOT NULL,
    "a" TIMESTAMP(3) NOT NULL DEFAULT '1995-05-02 16:20:00 +07:00',

    CONSTRAINT "foo_pkey" PRIMARY KEY ("id")
);
```

The result `prisma db push` gives to us:

```
Schema pushed to database. (1 steps)
```

Introspecting gives a different data model back:

```
model foo {
  id Int      @id @default(autoincrement())
  a  DateTime @default(dbgenerated("'1995-05-02 16:20:00'::timestamp without time zone"))
}
```

Pushing the introspected datamodel back works in all PostgreSQL examples.

1. Date

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default("1995-05-02T16:20:00+07:00") @db.Date
   }
   ```

   Push works, introspection result:

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default(dbgenerated("'1995-05-02'::date")) @db.Date
   }
   ```

2. Time

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default("1995-05-02T16:20:00+07:00") @db.Time
   }
   ```

   Push works, introspection result:

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default(dbgenerated("'16:20:00'::time without time zone")) @db.Time(6)
   }
   ```

3. Timetz

   ```
   model foo {
   ```

```
  id Int      @id @default(autoincrement())
  a  DateTime @default("1995-05-02T16:20:00+07:00") @db.Timetz
}
```

Push works, introspection result

```
model foo {
  id Int      @id @default(autoincrement())
  a  DateTime @default(dbgenerated("'16:20:00+07'::time with time zone")) @db.Timetz(6)
}
```

4. Timestamptz

```
model foo {
  id Int      @id @default(autoincrement())
  a  DateTime @default("1995-05-02T16:20:00+07:00") @db.Timestamptz
}
```

Push works, introspection result:

```
model foo {
  id Int      @id @default(autoincrement())
  a  DateTime @default(dbgenerated("'1995-05-02 09:20:00+00'::timestamp with time zone"
}
```

SQL Server 2019

The generated DDL:

```sql
CREATE TABLE [dbo].[foo] (
    [id] INT NOT NULL IDENTITY(1,1),
    [a] DATETIME2 NOT NULL CONSTRAINT [foo_a_df] DEFAULT '1995-05-02 16:20:00 +07:00',
    CONSTRAINT [foo_pkey] PRIMARY KEY ([id])
);
```

Push works, introspection returns:

```
model foo {
  id Int      @id @default(autoincrement())
  a  DateTime @default(dbgenerated("1995-05-02 16:20:00 +07:00"))
}
```

When we push this again, we get the error:

```
Error: Incorrect syntax near '16'.
```

The faulty DDL:

```sql
CREATE TABLE [dbo].[foo] (
    [id] INT NOT NULL IDENTITY(1,1),
    [a] DATETIME2 NOT NULL CONSTRAINT [foo_a_df] DEFAULT 1995-05-02 16:20:00 +07:00,
```

```
    CONSTRAINT [foo_pkey] PRIMARY KEY ([id])
);
```

1. Date

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default("1995-05-02T16:20:00+07:00") @db.Date
   }
   ```

   Push works, introspected result:

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default(dbgenerated("1995-05-02 16:20:00 +07:00")) @db.Date
   }
   ```

   Funnily enough, pushing this one AGAIN gives a syntax error:

   ```
   Error: Incorrect syntax near '16'.
   ```

   The faulty SQL in this case:

   ```
   CREATE TABLE [dbo].[foo] (
       [id] INT NOT NULL IDENTITY(1,1),
       [a] DATE NOT NULL CONSTRAINT [foo_a_df] DEFAULT 1995-05-02 16:20:00 +07:00,
       CONSTRAINT [foo_pkey] PRIMARY KEY ([id])
   );
   ```

2. Time

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default("1995-05-02T16:20:00+07:00") @db.Time
   }
   ```

   Push works, introspection:

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default(dbgenerated("1995-05-02 16:20:00 +07:00")) @db.Time
   }
   ```

   Push again:

   ```
   Error: Incorrect syntax near '16'.
   ```

   Faulty DDL:

   ```
   CREATE TABLE [dbo].[foo] (
       [id] INT NOT NULL IDENTITY(1,1),
       [a] TIME NOT NULL CONSTRAINT [foo_a_df] DEFAULT 1995-05-02 16:20:00 +07:00,
       CONSTRAINT [foo_pkey] PRIMARY KEY ([id])
   );
   ```

3. DateTimeOffset

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default("1995-05-02T16:20:00+07:00") @db.DateTimeOffset
   }
   ```

   Push works, introspect:

   ```
   model foo {
     id Int      @id @default(autoincrement())
     a  DateTime @default(dbgenerated("1995-05-02 16:20:00 +07:00")) @db.DateTimeOffset
   }
   ```

   Second push:

   ```
   Error: Incorrect syntax near '16'.
   ```

   DDL:

   ```
   CREATE TABLE [dbo].[foo] (
       [id] INT NOT NULL IDENTITY(1,1),
       [a] DATETIMEOFFSET NOT NULL CONSTRAINT [foo_a_df] DEFAULT 1995-05-02 16:20:00 +07:0
       CONSTRAINT [foo_pkey] PRIMARY KEY ([id])
   );
   ```

Default With Current Timestamp

Prisma allows a function `now()` in the PSL field `@default` attribute:

```
model foo {
  id Int      @id @default(autoincrement())
  a  DateTime @default(now())
}
```

This in general works the same in all databases. The generated DDL:

```
CREATE TABLE `foo` (
    `id` INTEGER NOT NULL AUTO_INCREMENT,
    `a` DATETIME(3) NOT NULL DEFAULT CURRENT_TIMESTAMP(3),

    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Client creates one `foo` without any parameters:

```
await client.create({})
```

Query Engine adds the default as UTC:

```
INSERT INTO `prisma`.`foo` (`a`) VALUES (?)
params=[2022-01-20 17:35:11.270 UTC]
```

There is no way to change the timezone.

Changing the native type allows using `now()`, but the resulting DDL is not very often accepted by the database. We miss validations in these cases.

Default with updatedAt

Another Prisma specialty in the PSL syntax is the `@updatedAt` attribute:

```
model foo {
  id Int      @id @default(autoincrement())
  a  DateTime @updatedAt
}
```

This is not reflected at all in the DDL:

```sql
CREATE TABLE `foo` (
    `id` INTEGER NOT NULL AUTO_INCREMENT,
    `a` DATETIME(3) NOT NULL,

    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

So it's a completely client-side feature. As expected, the Query Engine inserts the current time in UTC when using the feature.

```sql
INSERT INTO `prisma`.`foo` (`a`) VALUES (?)
params=[2022-01-20 17:47:10.197774387 UTC]
```

The `updatedAt` can be used in any datetime types. The resulting value will just lose precision depending on the type.

How Our DateTime Handling is Problematic in MySQL

MySQL does not store the timezone to any of the datetime columns it supports: `DATE` for dates, `TIME` for times, `DATETIME` for combined dates and times and `TIMESTAMP` for (micro)seconds since 1970.

The user has a few different ways to define the timezone of inserted datetimes:

- When starting the server, either implicitly from the system `locale`, using a parameter, using a configuration value or an environment variable.
- With `SET GLOBAL time_zone = ...` as an admin user.
- Whenever connecting with `SET time_zone = ...`, defining it for the whole lifetime of the connection.
- From version `8.0.19` forward, the time zone can be defined when inserting. This requires support from the driver.

This means the only way to define the default value is by using the `dbgenerated` escape hatch, leading to lots of problems described in this document.

When writing a value with Prisma, the Query Engine converts all datetimes to UTC timezone, even if using a default value that is defined in different zone in the PSL. If the server is started in other zone than UTC, this means Prisma will write the time in UTC, the server thinks otherwise and this leads to interesting issues with users outside of the western hemisphere.

If a user writes the following definition in the PSL:

```
model A {
  id  Int      @id @default(autoincrement())
  val DateTime @default("1996-12-19T16:39:57-08:00")
}
```

This is the only correct way of writing a default value without using `dbgenerated`. It will lead to a few problems. First comes from the Query Engine. We can store a new record with no data to get the default value:

```
mutation {
  createOneA(data: { }) {
    id
    val
  }
}
```

Surprisingly what the user gets back is the value converted to UTC:

```
{
  "data": {
    "createOneA": {
      "id": 1,
      "val": "1996-12-20T00:39:57.000Z"
    }
  }
}
```

Same happens when the user creates a `Date` object in a non-UTC timezone. We will convert the time to UTC, lose the timezone information from the value and leave the user very confused.

Suggested Changes

Changes in Migrations and Introspection

Introduce new ways to express datetimes in `@default`

With MySQL versions earlier than 8.0.19 and MariaDB, using the RFC-3339 format in the default value will lead to a migration error due to the database not knowing what to do with the timezone:

```
Error: Incorrect datetime value: '1996-12-19T16:39:57-08:00' for column 'val' at row 1
```

To support datetime default values in a more standardized way, we should allow defining all different forms of values directly in the `@default` attribute. This requires Introspection Engine to detect the format of the stored value, create a corresponding type in Rust and enable a correct diffing in the Migration Engine.

In the PSL definition, we then enable more different ways of defining a datetime, including the necessary validations. Let's see an example of a MySQL model with all possible datetime combinations:

```
model A {
  id Int      @id
  // This is the default native type. The fraction is `3`, so we include
  // milliseconds (should be optional).
  a  DateTime @default("1996-12-19 16:39:57.000") @db.DateTime(3)
  // No time stored.
  b  DateTime @default("1996-12-19") @db.Date
  // No date stored. The fraction is `3` so we include (optional) milliseconds.
  c  DateTime @default("16:20:00.000") @db.Time(3)
  // Here the value is always considered to be in UTC, because we store
  // (milli)seconds from 1.1.1970 00:00. Again with fraction, so we can
  // optionally include the milliseconds in the default.
  d  DateTime @default("1996-12-19 16:39:57.000") @db.Timestamp(3)
  // Additionally a timestamp could be a (signed) float.
  e  DateTime @default(0.0) @db.Timestamp
}
```

PostgreSQL:

```
model A {
  id Int      @id
  // This is the default native type. The fraction is `3`, so we include
  // milliseconds (should be optional).
  a  DateTime @default("1996-12-19 16:39:57.000") @db.Timestamp(3)
  // Additionally a timestamp could be a (signed) float with optional fraction.
  b  DateTime @default(0.0) @db.Timestamp
  // No time stored.
  c  DateTime @default("1996-12-19") @db.Date
  // No date stored. The fraction is `3` so we include (optional) milliseconds.
  d  DateTime @default("16:20:00.000") @db.Time(3)
  // The weird PostgreSQL type without date, but with a time zone.
  e  DateTime @default("16:20:00.000+06:00") @db.Timetz(3)
  // Timestamp and a timezone.
  f  DateTime @default("1996-12-19T16:39:57.000+06:00") @db.Timestamptz(3)
  // Additionally a timestamp with time zone could be a (signed) float with
  // optional fraction.
  g  DateTime @default(-100.00) @db.Timestamptz(3)
}
```

SQL Server:

```
model A {
  id Int      @id
  // This is the default native type. Optional fraction included.
  a  DateTime @default("1996-12-19 16:39:57.000") @db.DateTime2(3)
  // Legacy datetime type with less precision. Optional fraction included.
  b  DateTime @default("1996-12-19 16:39:57.997") @db.DateTime
  // Legacy datetime type with even less precision.
  c  DateTime @default("16:20:00") @db.SmallDateTime
  // No time stored.
  d  DateTime @default("1996-12-19") @db.Date
  // No date stored. The fraction is `3` so we include (optional) milliseconds.
  e  DateTime @default("16:20:00.000") @db.Time(3)
  // Date, time and the timezone all in one column. Optional fracion.
  g  DateTime @default("1996-12-19T16:39:57.000+06:00") @db.DateTimeOffset(3)
}
```

MongoDB:

```
model A {
  id Int      @id
  // Timestamp is the default
  a  DateTime @default("1996-12-19 16:39:57.000")
  // We could additionally have this as a float.
  b  DateTime @default(0.0)
}
```

SQLite:

Numeric or string storage. Can be stored in two formats:

```
model A {
  id Int      @id
  a  DateTime @default("1996-12-19T16:39:57-08:00")
  b  DateTime @default("Tue, 1 Jul 2003 10:52:37 +0200")
}
```

Changes in the Query Engine

The Query Engine should not break client workflows when defining the `@default` attribute directly. Especially in systems such as MySQL where converting to UTC would lead to wrong default values being written. This issue must be addressed before the new defaults can be used in the Migration and Introspection engines accordingly.

Solution #1: Stop Handling Defaults in the Query Engine

If the database supports default values, the Query Engine should stop adding them to the queries, leaving it for the database. In the scope of dates and times, we should just remove the client side defaults for the datetime values; still having them for databases such as MongoDB which doesn't support default values.

Solution #2: Adding More Internal DateTime Types

Internally in the Query Engine, we'd add new variants to the `Value` enum:

```rust
enum Value {
    Text(String),
    Int(i64),
    Date(NaiveDate),
    Time(NaiveTime),
    DateTime(NaiveDateTime),
    DateTimeOffset(DateTime<FixedOffset>),
    ...
}
```

This means the Query Engine needs to convert the full date to the corresponding type in the conversion from JavaScript to Rust. In this way the default value gets no conversion to UTC and we write the value the user expects us to insert.

Solution #3: More Types in PSL, Conversion to DateTimeOffset for

Instead of changing the `Value` for Query Engine, Migration Engine and Introspection Engine, we could have a separate `Value` implementation for the migrations and introspection, keeping the `Value` implementation of Query Engine as-is. This means for the PSL and schema side of things we have more granularity (see the `Value` definition in the Solution #2).

When communicating the AST from the PSL side to the Query Engine, we convert the more granular datetime value to a `DateTime<FixedOffset>`. The Query Engine must either take care to inform the database in what timezone the values are written. This can be handled per connection:

```sql
SET time_zone = timezone;
```

In the first phase, Quaint should just initialize the connection with `UTC` timezone.

As a second step, we can provide a way for the user to set the timezone in the connection string. That leads to certain changes:

- JavaScript stringifies the `Date` with the given timezone
- Query Engine respects the given timezone
- Quaint initializes the connection with the given timezone.

This approach follows how the official MySQL client handles timezones.

Breaking Changes?

- The PSL validations would not allow the RFC-3339 values on some of the datetime native types anymore.
- The defaults would be introspected without `dbgenerated`. ***(not sure if breaking)***
- Client Solution #1: If somebody was relying on the default to be converted to UTC, it would now be written as-defined.
- Client Solution #2: Depends on if we have more client types. If not, the conversions should be handled accordingly without breaking in the Query Engine.
- Client Solution #3: Should not be a breaking change. The Query Engine layer would not change, and the conversion between PSL and Query Engine would make sure the engine works as before.