Quering with SQL-like syntax [CRUD]

Drizzle ORM provide you the most SQL-like way to query your relational database.
We natively support mostly every query feature capability of every dialect and whatever we do not yet support - can be done with our powerful `sql` operator

When you declare schema(see docs) in typescript - you can instantly use our typed query builder to insert, update, select and delete.
All types are infered instantly without any need for code generation.

```
import { pgTable, serial, text, varchar } from 'drizzle-orm/pg-core';
import { drizzle } from 'drizzle-orm/postgresjs';
import { InferModel } from 'drizzle-orm';

export const users = pgTable('users', {
  id: serial('id').primaryKey(),
  name: text('full_name'),
});

type User = InferModel<typeof users, "select">;
type NewUser = InferModel<typeof users, "insert">;

const db = drizzle(...);
```

SQL Select

Basic and partial select

Getting a list of all users and you will have a typed result set
TODO: why we return array and there's no `.findOne`, `.findMany`

```
const result: User[] = await db.select().from(users);

result[0].id;
```

```
result[0].name;
```

```
select * from 'users';
```

Whenever you have SQL table with many columns you might not wanna select all of them for either performance or security reasons.
You can omit them by using our partial query syntax which will generate partial SQL select and automatically map results

```
const result = await db.select({
    field1: users.id,
    field2: users.name,
  }).from(users);

const { field1, field2 } = result[0];
```

```
select "user"."id" as "field1", "user"."name" as "field2" from "users";
```

With partial select you can apply sql transformations with `sql` operator

```
const result = await db.select({
    id: users.id,
    lowerName: sql`lower(${users.name})`,
  }).from(users);
```

```
select "user"."id", lower("user"."name") as "lowerName" from "users";
```

You can also select fields conditionally

```
async function selectUsers(withName: boolean) {
  return db
    .select({
      id: users.id,
      ...(withName ? { name: users.name } : {}),
    })
    .from(users);
}
```

```
const users = await selectUsers(true);
```

Select filters

You can filter SQL results with our list of [filter operators]

```
import { eq, lt, gte, ne } from "drizzle-orm";

await db.select().from(users).where(eq(users.id, 42));
await db.select().from(users).where(lt(users.id, 42));
await db.select().from(users).where(gte(users.id, 42));
await db.select().from(users).where(ne(users.id, 42));
...
```

```
select * from 'users' where 'id' = 42;
select * from 'users' where 'id' < 42;
select * from 'users' where 'id' >= 42;
select * from 'users' where 'id' <> 42;
```

Any filter operator is a `sql` operator under the hood, for full SQL potential you can utilise it directly and build type safe and future safe queries
You can safely alter schema, rename tables and columns and it will automatically reflect in queries, as opposed to having regular string raw SQL queries

```
import { sql } from "drizzle-orm";

await db.select().from(users).where(sql`${users.id} < 42`);
await db.select().from(users).where(sql`${users.id} = 42`);
await db.select().from(users).where(sql`${users.id} >= 42`);
await db.select().from(users).where(sql`${users.id} <> 42`);
await db.select().from(users).where(sql`lower(${users.name}) = "aaron"`);
```

```
select * from 'users' where 'id' = 42;
select * from 'users' where 'id' < 42;
select * from 'users' where 'id' <> 42;
select * from 'users' where 'id' >= 42;
select * from 'users' where lower('name') = "aaron";
```

Inverting condition with a `not` operator

```
import { eq, not, sql } from "drizzle-orm";

await db.select().from(users).where(not(eq(users.id, 42)));
await db.select().from(users).where(sql`not ${users.id} = 42`);
```

```
select * from 'users' where not 'id' = 42;
select * from 'users' where not 'id' = 42;
```

Combining filters

You can logically combile filter operators with conditional `and` and `or` operators

```
import { eq, and, sql } from "drizzle-orm";

await db.select().from(users).where(
  and(
    eq(users.id, 42),
    eq(users.name, 'Dan')
  )
);
await db.select().from(users).where(sql`${users.id} = 42 and ${users.name} = "Dan"`);
```

```
select * from 'users' where 'id' = 42 and 'name' = "Dan";
select * from 'users' where 'id' = 42 and 'name' = "Dan";
```

```
import { eq, or, sql } from "drizzle-orm";

await db.select().from(users).where(
  or(
    eq(users.id, 42),
    eq(users.name, 'Dan')
  )
);
await db.select().from(users).where(sql`${users.id} = 42 or ${users.name} = "Dan"`);
```

```
select * from 'users' where 'id' = 42 or 'name' = "Dan";
select * from 'users' where 'id' = 42 or 'name' = "Dan";
```

Limit & Offset

You can apply `limit` and `offset` to the query

```
await db.select().from(users).limit(10);
await db.select().from(users).limit(10).offset(10);
```

```
select * from "users" limit 10;
select * from "users" limit 10 offset 10;
```

Order By

You can sort results with `orderBy` operator

```
import { asc, desc } from "drizzle-orm";

await db.select().from(users).orderBy(users.name);
await db.select().from(users).orderBy(desc(users.name));

// you can pass multiple order args
await db.select().from(users).orderBy(users.name, users.name2);
await db.select().from(users).orderBy(asc(users.name), desc(users.name2));
```

```
select * from "users" order by "name";
select * from "users" order by "name" desc;

select * from "users" order by "name" "name2";
select * from "users" order by "name" asc "name2" desc;
```

WITH clause

SQL `with` clause - is a statement scoped view, helpful to organise complex queries

```
const sq = db.$with('sq').as(db.select().from(users).where(eq(users.id, 42)));

const result = await db.with(sq).select().from(sq);
```

```
with sq as (select * from "users" where "users"."id" = 42)
select * from sq;
```

To select raw `sql` in a WITH subquery and reference that field in other queries, you must add an alias to it

```
const sq = db.$with('sq').as(db.select({
    name: sql<string>`upper(${users.name})`.as('name')
  })
  .from(users));

const result = await db.with(sq).select({ name: sq.name }).from(sq);
```

If you don't provide an alias - field type will become `DrizzleTypeError` and you won't be able to reference it in other queries. If you ignore the type error and still try to reference the field, you will get a runtime error, since there's no way to reference that field without an alias.

Select from subquery

Just like in SQL - you can embed SQL queries into other SQL queries by using `subquery` API

```
const sq = db.select().from(users).where(eq(users.id, 42)).as('sq');
const result = await db.select().from(sq);
```

```
select * from (select * from "user" where "user"."id" = 42) "sq";
```

You can also use subqueries in joins

```
const result = await db.select().from(users).leftJoin(sq, eq(users.id, sq.id));
```

Aggregations

With Drizzle ORM you can do aggregations with functions like `sum`, `count`, `avg`, etc. by grouping and filtering with `groupBy` and `having` respectfully, just like you do in SQL.

With our powerful `sql` operator you can infer aggregations functions return types using `sql<number>` syntax

PostgreSQL MySQL SQLite

```
import { pgTable, serial, text, doublePrecision } from 'drizzle-orm/pg-core';
import { gte } from 'drizzle-orm';

export const product = pgTable('product', {
  id: serial('id').primaryKey(),
  name: text('name'),
  unitPrice: doublePrecision("unit_price")
});
```

```
const result = await db.select({ count: sql<number>`count(*)` }).from(product);
result[0].count // will be number type
```

```
await db.select({ count: sql<number>`count(*)` }).from(product).where(gte(product.unitP
```

```
select count(*) from "product";
select count(*) from "product" where "unit_price" >= 4;
```

Lets have a quick look on how to group and filter grouped using a `having`

PostgreSQL MySQL SQLite

```
import { pgTable, serial, text } from 'drizzle-orm/pg-core';

export const user = pgTable('user', {
  id: serial('id').primaryKey(),
  name: text('name'),
```

```
  city: text("city"),
});
```

```
await db.select({ count: sql<number>`count(${user.id})`, city: user.city })
  .from(user)
  .groupBy(({ city }) => city)
```

```
await db.select({ count: sql<number>`count(${user.id})`, city: user.city })
  .from(user)
  .groupBy(({ city }) => city)
  .having(({ count }) => count)
```

```
select count("id"), "city" from "user" group by "user"."city";
select count("id"), "city" from "user" group by "user"."city" having count("user"."id")
```

Here's a more advanced example

```
const orders = sqliteTable('order', {
  id: integer('id').primaryKey(),
  orderDate: integer('order_date', { mode: 'timestamp' }).notNull(),
  requiredDate: integer('required_date', { mode: 'timestamp' }).notNull(),
  shippedDate: integer('shipped_date', { mode: 'timestamp' }),
  shipVia: integer('ship_via').notNull(),
  freight: numeric('freight').notNull(),
  shipName: text('ship_name').notNull(),
  shipCity: text('ship_city').notNull(),
  shipRegion: text('ship_region'),
  shipPostalCode: text('ship_postal_code'),
  shipCountry: text('ship_country').notNull(),
  customerId: text('customer_id').notNull(),
  employeeId: integer('employee_id').notNull(),
});

const details = sqliteTable('order_detail', {
  unitPrice: numeric('unit_price').notNull(),
  quantity: integer('quantity').notNull(),
  discount: numeric('discount').notNull(),
  orderId: integer('order_id').notNull(),
  productId: integer('product_id').notNull(),
});


db
  .select({
    id: orders.id,
```

```
      shippedDate: orders.shippedDate,
      shipName: orders.shipName,
      shipCity: orders.shipCity,
      shipCountry: orders.shipCountry,
      productsCount: sql<number>`count(${details.productId})`,
      quantitySum: sql<number>`sum(${details.quantity})`,
      totalPrice: sql<number>`sum(${details.quantity} * ${details.unitPrice})`,
    })
    .from(orders)
    .leftJoin(details, eq(orders.id, details.orderId))
    .groupBy(orders.id)
    .orderBy(asc(orders.id))
    .all();
```

SQL Insert

Drizzle ORM provides you the most SQL-like way to insert rows into the database tables

Insert one row

Inserting data with Drizzle is extremely straightfoward and sql-like

```
  await db.insert(users).values({ name: 'Andrew' });
```

```
  insert into "users" ("name") values ("Andrew");
```

If you need insert type for a particular table - you can use `InferModel<typeof table, "insert">` syntax

```
  import { InferModel } from "drizzle-orm";

  type NewUser = InferModel<typeof users, "insert">;

  const insertUser = async (user: NewUser) => {
    return db.insert(users).values(user);
  }

  const newUser: NewUser = { name: "Alef" };
```

```
  await insertUser(newUser);
```

Insert returning

✓ PostgreSQL ✓ SQLite × MySQL

You can insert a row and get it back in PostgreSQL and SQLite

```
  await db.insert(users).values({ name: "Dan" }).returning();

  // partial return
  await db.insert(users).value({ name: "Partial Dan" }).returning({ insertedId: users.id
```

Insert multiple rows

```
  await db.insert(users).values([{ name: 'Andrew' }, { name: 'Dan' }]);
```

OnConflict and Upsert [insert or update]

You can run insert statements with on conflict clause to `do nothing` or `update`

```
  await db.insert(users)
    .values({ id: 1, name: 'John' })
    .onConflictDoNothing();

  // explicitly specify conflict target
  await db.insert(users)
    .values({ id: 1, name: 'John' })
    .onConflictDoNothing({ target: users.id });
```

This is how you upsert with `onConflictDoUpdate`

```
await db.insert(users)
  .values({ id: 1, name: 'Dan' })
  .onConflictDoUpdate({ target: users.id, set: { name: 'John' } });
```

Upsert with where statement

```
await db.insert(users)
  .values({ id: 1, name: 'John' })
  .onConflictDoUpdate({
    target: users.id,
    set: { name: 'John1' },
    where: sql`${users.createdAt} > '2023-01-01'::date`,
  });
```

SQL Update

Drizzle ORM supports SQL-like update syntax

Basic update

```
await db.update(users)
  .set({ name: 'Mr. Dan' })
  .where(eq(users.name, 'Dan'));
```

Update with returning

✓ PostgreSQL ✓ SQLite × MySQL

You can update a row and get it back in PostgreSQL and SQLite

```
const updatedUserId: { updatedId: number }[] = await db.update(users)
  .set({ name: 'Mr. Dan' })
  .where(eq(users.name, 'Dan'))
  .returning({ updatedId: users.id });
```

SQL Delete

Drizzle ORM supports SQL-like delete syntax

Basic detele

You can delete all rows in the table

```
await db.delete(users);
```

And you can delete with filters and conditions

```
await db.delete(users).where(eq(users.name, 'Dan'));
```

Delete with return

✓ PostgreSQL ✓ SQLite × MySQL

You can delete a row and get it back in PostgreSQL and SQLite

```
const deletedUser = await db.delete(users)
  .where(eq(users.name, 'Dan'))
  .returning();

// partial return
const deletedUserIds: { deletedId: number }[] = await db.delete(users)
  .where(eq(users.name, 'Dan'))
  .returning({ deletedId: users.id });
```