5. Data Types — Pick Wisely

Data types in T-SQL are not just labels for columns. They shape performance, storage, correctness, and the elegance of your future queries. A well-chosen type is like picking the right tool from a cosmic toolbox. You can hammer a screw, but the universe frowns upon it.

Why Data Types Matter

A data type determines how SQL Server stores your data, how fast it can index or compare it, and whether your ETL pipelines run like a gentle river or a clogged drain.

Small choices ripple outward. A wide VARCHAR might waste memory. A careless FLOAT might quietly mutate your numbers. Picking precisely signals mastery.

Major Families of Data Types

SQL Server groups its types into broad families, each built for a different kind of truth.

Numbers

Numbers live in two domains: exact and approximate.

Exact numeric types— `int`, `bigint`, `tinyint`, `decimal`, and `numeric` —store values without drifting. Use these when the numbers must stay crisp.

Approximate numeric types— `float` and `real` —trade precision for range. They are handy for scientific or statistical data but risky for financial calculations. A float will occasionally give you a tiny rounding wobble, like a cosmic hiccup.

Text

Text types shape how characters are stored.

`char` and `varchar` handle traditional characters. `nchar` and `nvarchar` store Unicode. Unicode matters when your data wanders beyond ASCII territory.

`char` types are fixed-length, which can waste space. `varchar` types are variable-length, which breathe a little easier.

Date and Time

SQL Server's date and time types aim for clarity.

`date` stores a whole day. `time` stores the clock. `datetime` and the newer `datetime2` combine both. `datetime2` is usually the wiser choice because it uses storage more efficiently and offers better precision.

There's also `datetimeoffset`, which keeps track of time zones—a useful ally in distributed systems.

Binary

Binary types—`binary` and `varbinary`—hold sequences of bytes. They carry encrypted data, file chunks, or anything that swims beneath human readability.

Special Types

A few types are almost whimsical.

`uniqueidentifier` stores GUIDs—128-bit globally unique IDs. These are great for distributed systems but can fragment indexes.

`sql_variant` can hold many different types, but its flexibility can cause chaos if used carelessly.

Choosing the Right Type

Choosing well is a balancing act: size, precision, readability, and indexing all matter.

Smaller types reduce storage and speed up scans. Using a `tinyint` instead of an `int` doesn't sound dramatic, but over millions of rows it becomes meaningful.

Use `decimal` for anything involving money. Money types (`money` and `smallmoney`) exist, but their rounding rules can be surprising.

If your data includes multiple languages or emoji—choosing a non-Unicode type will lead to lost or corrupted characters.

Real-World Patterns

A customer's age fits in a `tinyint` because the human lifespan sits comfortably within 0–255. A product's name naturally becomes `nvarchar(100)` because names vary wildly in length.

Sensor readings often fit well in `float`, but billing rates belong in `decimal(10,2)`.

Log tables often use `datetime2(7)` to capture high-precision timestamps for debugging or replaying events.

ETL pipelines involving global systems thrive with `datetimeoffset` to avoid mismatched time zones.

Example: Crafting a Table With Intent

```
CREATE TABLE dbo.CustomerProfile (
    CustomerID       int IDENTITY PRIMARY KEY,
    FullName         nvarchar(200) NOT NULL,
    Email            varchar(255) NOT NULL,
    DateOfBirth      date NOT NULL,
    CreatedAt        datetime2(3) NOT NULL DEFAULT SYSDATETIME(),
    LoyaltyPoints    int NOT NULL DEFAULT 0,
    IsActive         bit NOT NULL,
    ProfileGUID      uniqueidentifier NOT NULL DEFAULT NEWID()
);
```

Every column has a story: FullName is Unicode. Email is not. CreatedAt uses `datetime2` for precision. LoyaltyPoints are integers. IsActive is a `bit` because it's binary truth. ProfileGUID is globally unique.

This chapter sets up the deeper exploration of how types affect indexing, joins, and storage. The next concept flows naturally into constraints and table design philosophy.