Fondasi Basis Data: Tipe Data SQL dalam Perspektif Tipe Primitif Rust dan Zig

Muhammad Riduwan Khafidi
Program Studi Teknik Rekayasa Perangkat Lunak
Politeknik Negeri Batam
4342611034@students.polibatam.ac.id

November 2, 2026

Abstrak

Tipe data SQL sering diajarkan sebagai daftar hafalan yang terpisah dari dunia pemrograman sistem. Padahal, bagi seorang pengembang yang akrab dengan tipe primitif seperti `u8`, `i32`, `f32`, atau `f64` di Rust dan Zig, tipe data SQL dapat dipahami sebagai *padanan tingkat penyimpanan* dari tipe-tipe tersebut. Makalah ini menyajikan pemetaan sistematis dan visual antara tipe primitif Rust dan Zig dengan tipe data SQL, mulai dari integer bertanda dan tak bertanda, bilangan pecahan IEEE-754, boolean, string, hingga data biner tetap. Setiap konsep dilengkapi ilustrasi bit-level, langkah per langkah, dan contoh output nyata agar pembaca yang belum pernah menyentuh database dapat memahami *mengapa* setiap tipe ada, bukan hanya *apa* namanya. Pembaca akan memperoleh kerangka mental yang utuh untuk memilih tipe data secara sadar.

1 Pendahuluan: Mengapa Harus Peduli pada Tipe Data?

1.1 Cerita Pembuka: Warung Nasi dan Buku Kas

Bayangkan kamu punya warung nasi. Setiap hari kamu mencatat pesanan pelanggan. Ada dua pilihan:

- *Cara A:* mencatat semuanya di satu buku tulis besar, tanpa kolom, tanpa format. Cepat menulis, tetapi saat ingin menghitung total pendapatan hari ini, kamu harus membaca ulang semuanya secara manual.
- *Cara B:* mencatat di buku kas dengan kolom rapi: Tanggal, Nama Menu, Jumlah, Harga, Total. Lebih lambat saat menulis, tetapi saat butuh laporan, kamu bisa langsung menghitung.

Basis data memaksa kita memilih Cara B. Kolom Jumlah hanya boleh diisi angka, bukan teks "dua piring". Kolom Tanggal hanya boleh diisi tanggal, bukan "kemarin sore". Aturan inilah yang disebut tipe data.

1.2 Tipe Data sebagai Kontrak

Ketika kamu menulis:

```
CREATE TABLE pesanan (
    jumlah INT,
    harga  DECIMAL(10,2)
);
```

kamu sedang membuat kontrak dengan database: kolom `jumlah` hanya akan menerima bilangan bulat, dan jika ada yang mencoba memasukkan teks, database akan menolak. Kontrak ini punya tiga manfaat nyata:

1. Mencegah data sampah. Tidak ada yang bisa memasukkan `'dua piring'` ke kolom `jumlah`.
2. Menghemat ruang. Angka `25` disimpan sebagai 4 byte, sedangkan teks `'dua puluh lima'` butuh 15 byte.
3. Mempercepat perhitungan. Database bisa langsung menjumlahkan `jumlah * harga` tanpa perlu mengubah teks menjadi angka.

1.3 Jembatan ke Rust dan Zig

Di Rust, kamu menulis:

```
let usia: u8 = 25;
let saldo: i64 = -150000;
let rasio: f64 = 0.75;
```

Tiga baris itu membuat tiga kontrak yang persis sama dengan SQL: `u8` berarti bilangan bulat 0–255, `i64` berarti bilangan bulat 64-bit bertanda, `f64` berarti bilangan pecahan 64-bit. Yang berbeda hanyalah tempat kontrak itu ditulis. Di Rust, kontrak diperiksa saat kompilasi. Di SQL, kontrak disimpan di katalog database dan diperiksa saat `INSERT`.

Tesis makalah ini: tipe data SQL adalah tipe primitif yang disimpan secara permanen. Memahami satu sisi mempermudah memahami sisi lainnya.

2 Peta Keluarga Tipe Data SQL

Secara garis besar, tipe data SQL terbagi menjadi lima keluarga. Gambar 1 memetakan keluarga tersebut secara visual.

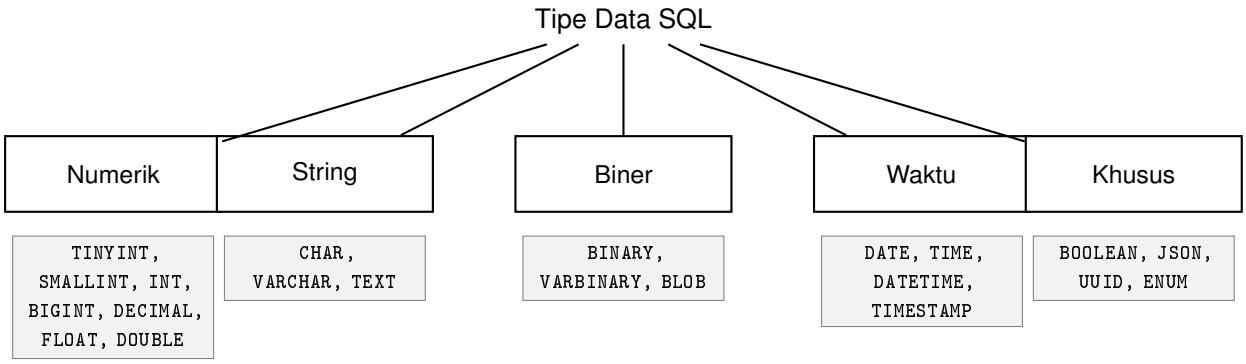


Figure 1. Peta keluarga tipe data SQL.

3 Integer: Bilangan Bulat dengan Ukuran Bertingkat

3.1 Analogi Kotak Berukuran Berbeda

Bayangkan kamu punya lima jenis kotak: kotak mungil (1 byte), kotak kecil (2 byte), kotak sedang (3 byte), kotak besar (4 byte), dan kotak raksasa (8 byte). Kalau kamu hanya perlu menyimpan satu baju, memakai kotak raksasa berarti membuang ruang sembilan belas baju. Sebaliknya, memakai kotak mungil untuk menyimpan sepuluh baju akan membuat baju tidak muat.

Inilah alasan SQL menyediakan banyak ukuran integer: agar kita bisa memilih kotak yang pas.

3.2 Representasi Bit-Level

Sebelum masuk ke tabel, mari lihat bagaimana komputer menyimpan angka. Satu byte terdiri dari 8 bit, setiap bit bernilai 0 atau 1.

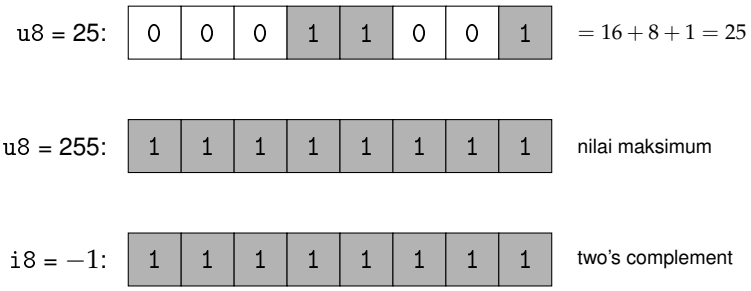


Figure 2. Representasi bit-level untuk `u8` dan `i8`.

Perhatikan Gambar 2: `u8` bernilai 25 disimpan sebagai `00011001`. Kalau bobotnya dijumlahkan, $16 + 8 + 1 = 25$. Nilai maksimum `u8` adalah 255 karena semua bit terisi satu. Sementara `i8` menyimpan -1 sebagai `11111111` karena memakai representasi *two's complement*, di mana bit paling kiri (paling signifikan) menandakan negatif.

Alasan `u8` berhenti di 255 dan `i8` berhenti di 127 adalah karena pada `i8` satu bit "dibuang" untuk

menyimpan tanda. Rentang negatifnya adalah -128 hingga -1, dan positifnya 0 hingga 127.

3.3 Tabel Lengkap Ukuran Integer

Table 1. Rentang nilai tipe integer di MariaDB dan MySQL.

Tipe SQL	Ukuran	Signed	Unsigned
`TINYINT`	1 byte	-128 s/d 127	0 s/d 255
`SMALLINT`	2 byte	-32.768 s/d 32.767	0 s/d 65.535
`MEDIUMINT`	3 byte	$-8.388.608$ s/d 8.388.607	0 s/d 16.777.215
`INT`	4 byte	$-2{,}1 \times 10^9$ s/d $2{,}1 \times 10^9$	0 s/d $4{,}2 \times 10^9$
`BIGINT`	8 byte	$\pm 9{,}2 \times 10^{18}$	0 s/d $1{,}84 \times 10^{19}$

3.4 Pemetaan ke Rust dan Zig

Table 2. Pemetaan tipe integer Rust dan Zig ke tipe SQL.

Rust / Zig	Padanan SQL	Rentang	Kapan Dipakai
`u8`	`TINYINT UNSIGNED`	0–255	Status 0/1, bendera
`i8`	`TINYINT`	-128–127	Suhu, selisih kecil
`u16`	`SMALLINT UNSIGNED`	0–65.535	Kode pos, tahun
`i16`	`SMALLINT`	± 32.767	—
`u32`	`INT UNSIGNED`	0–$4{,}2 \times 10^9$	ID umum
`i32`	`INT`	$\pm 2{,}1 \times 10^9$	ID, jumlah transaksi
`u64`	`BIGINT UNSIGNED`	0–$1{,}84 \times 10^{19}$	ID besar, `UUID_SHORT()`
`i64`	`BIGINT`	$\pm 9{,}2 \times 10^{18}$	Saldo, epoch milidetik
`u128`	`DECIMAL(39,0)`	0–$3{,}4 \times 10^{38}$	Tidak native

3.5 Dampak Ukuran Integer pada Performa

Ini pertanyaan yang sering terlewat: mengapa tidak memakai `BIGINT` untuk semua hal saja, toh 8 byte tidak seberapa?

Jawabannya: setiap indeks sekunder menyalin primary key. Gambar 3 mengilustrasikannya.

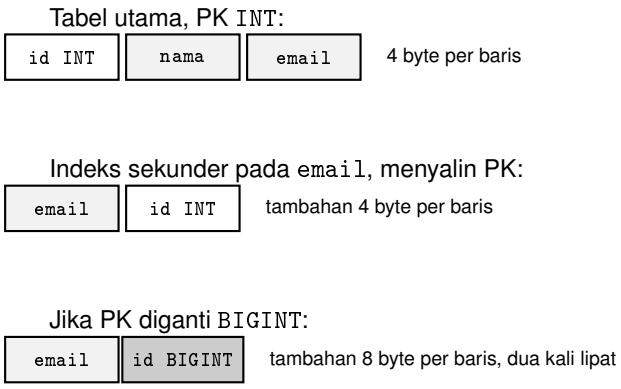


Figure 3. Ukuran primary key diduplikasi pada setiap indeks sekunder.

Pada tabel sepuluh juta baris dengan tiga indeks sekunder, PK `INT` membutuhkan tambahan sekitar 120 MB, sedangkan PK `BIGINT` membutuhkan sekitar 240 MB. Selisih 120 MB di disk. Aturan praktisnya: pilih tipe integer terkecil yang masih menampung seluruh kemungkinan nilai selama umur sistem.

4 Bilangan Pecahan: `FLOAT`, `DOUBLE`, dan `DECIMAL`

4.1 Mengapa `0.1 + 0.2` Tidak Selalu Sama dengan `0.3`

Ini adalah kejutan pertama yang dialami hampir semua programmer. Bilangan desimal seperti `0.1` sebenarnya tidak bisa direpresentasikan secara persis dalam biner. Sama seperti $1/3 = 0.3333\ldots$ tidak bisa ditulis lengkap dalam desimal, `0.1` dalam biner adalah `0.0001100110011...` yang berulang tanpa henti.

Ketika disimpan sebagai `f32` dengan 23 bit mantissa, angka itu dipotong di bit ke-23. Ketika dijumlahkan dengan `0.2` yang juga dipotong, hasilnya bukan `0.3` persis, melainkan `0.30000001...`

`FLOAT` dan `DOUBLE`, approximate numeric.
Mengikuti IEEE-754. Nilai disimpan sebagai kombinasi tanda, mantissa, dan eksponen.
Masalah: `0.1 + 0.2` bisa menghasilkan `0.30000000000000004`.
Analogi: mengukur meja dengan penggaris kayu.

`DECIMAL(p, s)`, exact numeric.
Nilai disimpan sebagai digit desimal dengan presisi dan skala tetap.
Solusi: `10.10` selalu tersimpan sebagai `10.10`.
Analogi: menghitung uang dengan kalkulator.

Figure 4. Perbandingan filosofis `FLOAT`/`DOUBLE` dengan `DECIMAL`.

4.2 Layout Bit IEEE-754

Bagaimana `f32` menyimpan `0.1`? Gambar 5 membedah tiga puluh dua bit menjadi tiga bagian.



Figure 5. Layout bit IEEE-754 single precision (`f32` atau `FLOAT`).

Untuk `f64` atau `DOUBLE`, layoutnya sama tetapi ukurannya berbeda: 1 bit tanda, 11 bit eksponen, dan 52 bit mantissa, memberi akurasi sekitar enam belas digit signifikan.

4.3 Demonstrasi di MariaDB

```
CREATE TABLE demo_uang (
    harga_float   FLOAT,
    harga_double  DOUBLE,
    harga_decimal DECIMAL(12,2)
);
INSERT INTO demo_uang VALUES (10.10, 10.10, 10.10);
SELECT harga_float * 100,
       harga_double * 100,
       harga_decimal * 100
FROM demo_uang;
```

Output yang mungkin muncul:

```
+-------------------+--------------------+----------------------+
| harga_float * 100 | harga_double * 100 | harga_decimal * 100 |
+-------------------+--------------------+----------------------+
|        1009.99994 | 1009.9999999999999 |              1010.00 |
+-------------------+--------------------+----------------------+
```

`FLOAT` menghasilkan `1009.99994`, `DOUBLE` menghasilkan `1009.9999999999999`, hanya `DECIMAL` yang menghasilkan `1010.00` dengan tepat. Pada aplikasi keuangan, selisih enam per seratus ribu per transaksi bisa menjadi miliaran rupiah per tahun. Aturan industri: uang wajib `DECIMAL`.

4.4 Pemetaan ke Rust dan Zig

Table 3. Pemetaan tipe float Rust dan Zig ke tipe SQL.

Rust / Zig	Bita	Padanan SQL	Akurasi	Kegunaan
`f16`	2	tidak ada	sekitar 3 digit	—
`f32`	4	`FLOAT`	sekitar 7 digit	Grafika, ML ringan
`f64`	8	`DOUBLE`	sekitar 16 digit	Ilmu fisika, kalkulasi
`f128`	16	tidak ada	sekitar 34 digit	—
—	—	`DECIMAL(p,s)`	persis	Uang, harga, rasio

5 Boolean: Tipe yang Menyamar

Baik Rust maupun Zig memiliki `bool` sebagai tipe satu byte. Di MariaDB, `BOOLEAN` hanyalah alias untuk `TINYINT(1)`.

Di Rust dan Zig:
`let aktif: bool = true;`
Ukuran satu byte. Nilai `true` atau `false`.

Di MariaDB:
`is_active BOOLEAN`
Sebenarnya `TINYINT(1)`. Nilai `1` atau `0`.

Figure 6. Perbedaan representasi boolean di Rust, Zig, dan MariaDB.

Jebakan yang perlu diwaspadai: MariaDB memperlakukan nilai apa pun yang bukan nol sebagai `TRUE`.

```
INSERT INTO users (is_active) VALUES (5);
SELECT is_active FROM users;
```

Outputnya adalah `1`. MariaDB mengubah `5` menjadi `1` karena `BOOLEAN` hanya peduli nol atau bukan nol. Ini berbeda dengan Rust yang akan menolak `let aktif: bool = 5;` saat kompilasi.

6 String: Mengapa `CHAR` dan `VARCHAR` Berbeda

6.1 Analogi Lemari Sepatu dan Rak Sepatu

Bayangkan kamu menyimpan sepatu. Ada dua cara:

- Lemari sepatu (`CHAR`): setiap kotak berukuran sama, misalnya 30 kali 30 sentimeter. Sepatu kecil tetap memakan kotak penuh. Rapi, mudah dicari, tetapi boros.
- Rak sepatu (`VARCHAR`): setiap sepatu menempati ruang sesuai ukurannya. Hemat, tetapi butuh label tambahan untuk menandai panjang sepatu.

Gambar 7 mengilustrasikan penyimpanan kata "du" pada kedua tipe.

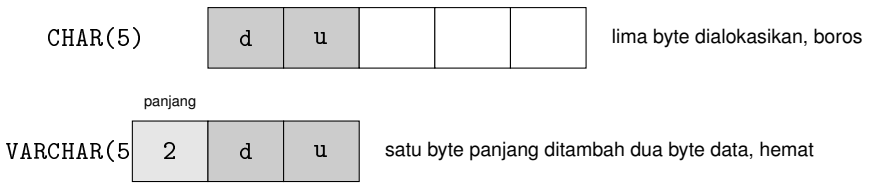


Figure 7. Ilustrasi alokasi memori `CHAR(5)` dan `VARCHAR(5)`.

`CHAR(5)` selalu memakai lima byte, sedangkan `VARCHAR(5)` memakai satu byte panjang ditambah jumlah byte aktual. Untuk "du" yang berukuran dua byte, total `VARCHAR` adalah tiga byte.

6.2 Aturan Emas

- `CHAR(n)`: gunakan untuk data dengan panjang pasti, seperti kode negara `'ID'`, kode pos `'29461'`, atau hash hex dengan panjang tetap.
- `VARCHAR(n)`: gunakan untuk data dengan panjang bervariasi, seperti nama, email, dan alamat.

6.3 Pemetaan ke Rust dan Zig

Table 4. Pemetaan tipe string Rust, Zig, dan SQL.

Rust	Zig	SQL	Kapan Dipakai
`String`, `&str` pendek	`[]u8`, `[]const u8`	`VARCHAR(255)`	Nama, email, judul
`String` panjang	`[]u8` panjang	`TEXT`	Deskripsi, artikel
`char`	`u21`	`CHAR(1)`	Satu codepoint
`[char; N]`	`[N]u8` teks	`CHAR(N)`	Kode tetap
`&[u8]`	`[]u8` biner	`VARBINARY(n)`	Byte mentah

7 Data Biner: Untuk Hash, UUID, dan Gambar

7.1 Kapan Butuh Biner, Bukan Teks

Terkadang data bukan teks, bukan angka, melainkan byte mentah. Contohnya hash password SHA-256 sepanjang tiga puluh dua byte, UUID biner sepanjang enam belas byte, gambar mini, atau kunci enkripsi.

Menyimpan hash sebagai teks hex enam puluh empat karakter, yaitu `CHAR(64)`, memakan enam puluh empat byte. Menyimpannya sebagai `BINARY(32)` hanya memakan tiga puluh dua byte. Setengahnya.

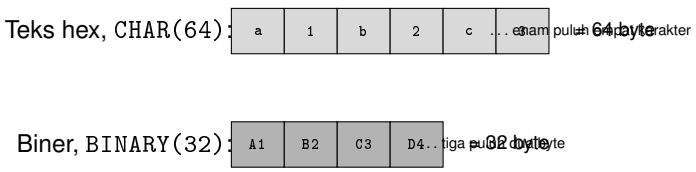


Figure 8. Perbandingan ukuran hash SHA-256 dalam teks hex dan biner.

7.2 Pemetaan ke Rust dan Zig

Table 5. Pemetaan tipe biner.

Rust	Zig	SQL	Ukuran
`[u8; N]`	`[N]u8`	`BINARY(N)`	N byte tetap
`Vec<u8>`	`[]u8`	`VARBINARY(n)`	variabel, maksimum n
`Vec<u8>` besar	`[]u8`	`BLOB`	hingga 65.535 byte
`[u8; 16]` UUID	`[16]u8`	`BINARY(16)`	UUID
`[u8; 32]` SHA-256	`[32]u8`	`BINARY(32)`	hash

7.3 Contoh UUID Biner

```
CREATE TABLE users (
    user_id BINARY(16) PRIMARY KEY,
    nama VARCHAR(100)
);
INSERT INTO users VALUES
    (UNHEX('a1b2c3d4e5f67890a1b2c3d4e5f67890'), 'Budi');
SELECT HEX(user_id) AS user_id_hex, nama FROM users;
```

Output:

```
+----------------------------------+-------+
| user_id_hex                      | nama  |
+----------------------------------+-------+
| A1B2C3D4E5F67890A1B2C3D4E5F67890 | Budi  |
+----------------------------------+-------+
```

Fungsi `UNHEX()` mengubah teks hex menjadi byte mentah saat `INSERT`. Fungsi `HEX()` mengubah byte mentah kembali menjadi teks hex saat `SELECT`. Ini setara dengan `hex::encode()` dan `hex::decode()` di Rust.

8 Tanggal dan Waktu

8.1 Mengapa Tanggal Tidak Boleh Disimpan sebagai Teks

Pertanyaan ini muncul karena teks tampak lebih mudah dibaca. Tetapi menyimpan `'2026-11-02'` sebagai `VARCHAR(10)` membawa tiga konsekuensi. Pertama, operasi seperti "berapa hari lagi sampai tanggal itu" harus melalui parsing ulang setiap baris. Kedua, mengurutkan tanggal berdasarkan teks hanya bekerja untuk format ISO, dan gagal untuk format `'02/11/2026'`. Ketiga, indeks menjadi lebih besar, sepuluh byte versus tiga byte untuk `DATE`.

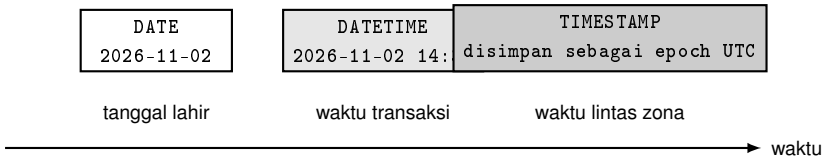


Figure 9. Pemetaan tipe waktu pada garis waktu.

8.2 Pemetaan ke Rust dan Zig

Table 6. Pemetaan tipe waktu ke Rust dan Zig.

Konsep	SQL	Rust	Zig
Tanggal saja	`DATE`	`chrono::NaiveDate`	`std.time.epoch`
Jam saja	`TIME`	`chrono::NaiveTime`	—
Tanggal dan jam	`DATETIME`	`chrono::NaiveDateTime`	—
Tanggal dan jam UTC	`TIMESTAMP`	`chrono::DateTime<Utc>`	`std.time.milliTimestamp`
Epoch saja	`BIGINT`	`i64`	`i64`

9 Tipe Data Khusus Modern

9.1 JSON

MariaDB 10.2 ke atas mendukung tipe `JSON` yang memvalidasi struktur dan memungkinkan query dengan `JSON_EXTRACT`. Di Rust padanannya `serde_json::Value`, di Zig `std.json.Value`.

```
CREATE TABLE konfigurasi (
    user_id BIGINT UNSIGNED PRIMARY KEY,
    preferensi JSON
);
INSERT INTO konfigurasi VALUES
    (1, '{"tema":"gelap","bahasa":"id"}');
SELECT JSON_EXTRACT(preferensi, '$.tema') FROM konfigurasi;
```

Meskipun JSON memudahkan, praktik terbaiknya tetap menormalkan data ke kolom terpisah kecuali strukturnya benar-benar dinamis. JSON tidak bisa diindeks secara langsung kecuali melalui generated column, sehingga query pada JSON lebih lambat daripada query pada kolom biasa.

9.2 UUID dan Alternatifnya

`UUID` adalah seratus dua puluh delapan bit angka yang secara praktis unik. Di MariaDB, UUID tidak memiliki tipe native, sehingga disimpan sebagai `CHAR(36)` atau `BINARY(16)`.

Alternatif yang lebih baik untuk primary key adalah `UUID_SHORT()`, fungsi MariaDB yang

mengembalikan enam puluh empat bit monotonik. Keunggulan monotonik: nilai baru selalu bertambah, sehingga penambahan data baru selalu di ujung indeks dan menghindari fragmentasi.

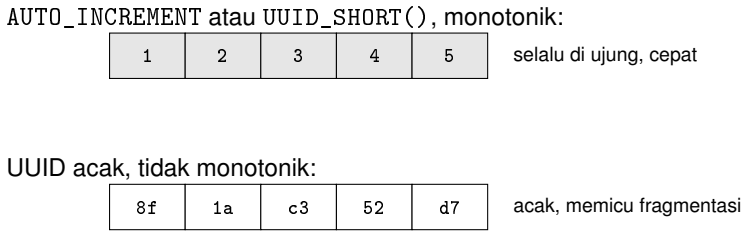


Figure 10. Perbandingan pola penambahan ID monotonik dan acak.

10 Tabel Referensi Silang Lengkap

Table 7. Tabel referensi silang lengkap Rust, Zig, dan SQL.

Rust	Zig	SQL	Keterangan
`u8`	`u8`	`TINYINT UNSIGNED`	1 byte, 0–255
`i8`	`i8`	`TINYINT`	1 byte, −128–127
`u16`	`u16`	`SMALLINT UNSIGNED`	2 byte
`i16`	`i16`	`SMALLINT`	2 byte
`u32`	`u32`	`INT UNSIGNED`	4 byte
`i32`	`i32`	`INT`	4 byte
`u64`	`u64`	`BIGINT UNSIGNED`	8 byte
`i64`	`i64`	`BIGINT`	8 byte
`u128`	`u128`	`DECIMAL(39,0)`	tidak native
`f32`	`f32`	`FLOAT`	IEEE-754 32 bit
`f64`	`f64`	`DOUBLE`	IEEE-754 64 bit
`bool`	`bool`	`BOOLEAN`	alias `TINYINT(1)`
`char`	`u21`	`CHAR(1)`	satu codepoint
`String`, `&str`	`[]u8`	`VARCHAR(n)`	teks variabel
`String` panjang	`[]u8` panjang	`TEXT`	teks panjang
`[u8; N]`	`[N]u8`	`BINARY(N)`	byte tetap
`Vec<u8>`	`[]u8`	`VARBINARY(n)`	byte variabel
`Option<T>`	`?T`	`NULL`	kolom nullable
`chrono::NaiveDate`	`std.time.epoch`	`DATE`	tanggal
`chrono::NaiveDateTime`	—	`DATETIME`	tanggal dan jam
`serde_json::Value`	`std.json.Value`	`JSON`	data semi-terstruktur
`uuid::Uuid`	`[16]u8`	`BINARY(16)`	UUID biner

11 Pohon Keputusan

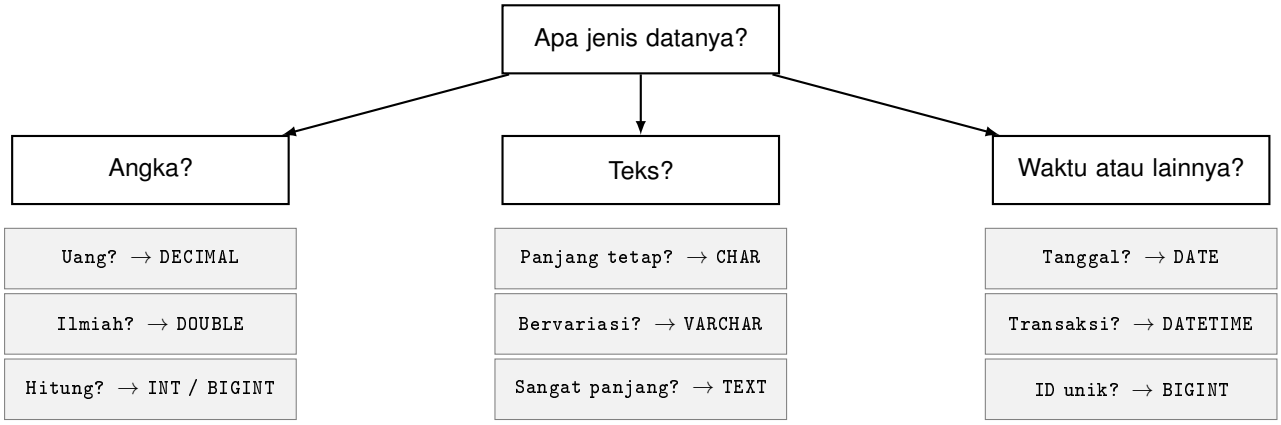


Figure 11. Pohon keputusan pemilihan tipe data.

12 Kesalahan Umum

Table 8. Kesalahan umum dalam memilih tipe data.

Kesalahan	Akibat	Perbaikan
`FLOAT` untuk uang	selisih sen menumpuk menjadi juta	gunakan `DECIMAL`
`CHAR(255)` untuk nama	boros 253 byte per baris	gunakan `VARCHAR(50)`
`BIGINT` untuk umur	indeks membengkak dua kali	gunakan `TINYINT UNSIGNED`
tanggal sebagai `VARCHAR`	sortir dan aritmatika lambat	gunakan `DATE` atau `DATETIME`
`CHAR(36)` untuk UUID	boros 20 byte per baris	gunakan `BINARY(16)`
hash password di `CHAR(64)`	boros 32 byte per baris	gunakan `BINARY(32)`
`TEXT` untuk kolom filter	indeks tidak bisa dipakai penuh	gunakan `VARCHAR(255)`
`AUTO_INCREMENT` terbuka	ID mudah ditebak, rawan scraping	gunakan `UUID_SHORT()`

13 Studi Kasus: Merancang Tabel `users`

Mari rangkai seluruh konsep dalam satu tabel produksi. Setiap kolom dipilih dengan alasan eksplisit.

```
CREATE TABLE users (
    user_id       BIGINT UNSIGNED DEFAULT (UUID_SHORT()) PRIMARY KEY,
    username      VARCHAR(50)     NOT NULL UNIQUE,
    email         VARCHAR(255)    NOT NULL UNIQUE,
    password_hash BINARY(32)      NOT NULL,
    tanggal_lahir DATE,
    is_active     BOOLEAN         NOT NULL DEFAULT TRUE,
    saldo         DECIMAL(15,2)   NOT NULL DEFAULT 0.00,
    metadata      JSON,
    created_at    DATETIME        DEFAULT CURRENT_TIMESTAMP,
    updated_at    DATETIME        DEFAULT CURRENT_TIMESTAMP
                                  ON UPDATE CURRENT_TIMESTAMP
```

```
);
```

Bedah pilihan per kolom:

- `user_id BIGINT UNSIGNED`: setara `u64`. `UUID_SHORT()` monotonik, anti-fragmentasi, lebih cepat dari `CHAR(36)`.
- `username VARCHAR(50)`: panjang bervariasi, wajib diisi, harus unik.
- `email VARCHAR(255)`: batas 255 sesuai RFC 5321.
- `password_hash BINARY(32)`: hash SHA-256 biner, setara `[u8; 32]`. Hemat 32 byte dibanding `CHAR(64)`.
- `tanggal_lahir DATE`: hanya tanggal, tanpa jam.
- `is_active BOOLEAN`: alias `TINYINT(1)`.
- `saldo DECIMAL(15,2)`: uang, wajib exact numeric.
- `metadata JSON`: data semi-terstruktur opsional.
- `created_at DATETIME`: otomatis saat `INSERT`.
- `updated_at DATETIME ON UPDATE`: otomatis berubah saat `UPDATE`.

Coba `INSERT` tanpa menyebut kolom yang punya `DEFAULT`:

```
INSERT INTO users (username, email, password_hash)
VALUES ('budi', 'budi@email.com', UNHEX(REPEAT('ab', 32)));
```

Semua kolom lain akan terisi otomatis: `user_id`, `is_active`, `saldo`, `created_at`, dan `updated_at`. Inilah kekuatan `DEFAULT` yang dikombinasikan dengan ekspresi.

14 Lembar Rujukan Cepat

Table 9. Lembar rujukan cepat memilih tipe data.

Kebutuhan	Tipe yang Tepat
status aktif atau nonaktif	`BOOLEAN`, alias `TINYINT(1)`
umur, jumlah anak	`TINYINT UNSIGNED`
jumlah barang	`INT UNSIGNED`
ID pengguna	`BIGINT UNSIGNED` dengan `UUID_SHORT()`
harga, saldo, pajak	`DECIMAL(15,2)`
koordinat GPS	`DECIMAL(9,6)`
berat badan	`DECIMAL(5,2)`
nama orang	`VARCHAR(100)`
email	`VARCHAR(255)`
alamat	`VARCHAR(500)`
deskripsi produk	`TEXT`
kode negara	`CHAR(2)`
kode pos Indonesia	`CHAR(5)`
tanggal lahir	`DATE`
waktu transaksi	`DATETIME`
waktu lintas zona waktu	`TIMESTAMP`
hash password	`BINARY(32)`
UUID	`BINARY(16)`
konfigurasi aplikasi	`JSON`
gambar kecil di bawah 64 KB	`BLOB`

15 Kesimpulan

Tipe data SQL bukanlah daftar hafalan yang terpisah dari dunia pemrograman sistem. Ia adalah cermin dari tipe primitif yang sudah dikenal di Rust dan Zig. Tiga prinsip penutup:

1. Integer: pilih yang terkecil yang cukup. `INT` lebih baik daripada `BIGINT` jika nilai tidak akan melewati empat miliar, karena setiap indeks sekunder menyalin ukuran tersebut.
2. Uang: wajib `DECIMAL`. Jangan pernah menggunakan `FLOAT` atau `DOUBLE` untuk nilai moneter.
3. Teks dan biner: `CHAR` untuk panjang tetap, `VARCHAR` untuk variabel, `BINARY(n)` untuk byte mentah. Untuk hash dan UUID, gunakan bentuk biner, bukan teks hex.

Dengan kerangka mental ini, keputusan memilih tipe data menjadi keputusan arsitektural yang sadar, bukan sekadar "yang penting bisa jalan".

Referensi

References

[1] The Rust Project Developers. (2026). *Data Types — The Rust Programming Language*. https://doc.rust-lang.org/book/ch03-02-data-types.html.

[2] Zig Software Foundation. (2026). *Language Reference — Primitive Types*. https://ziglang.org/documentation/master/.

[3] MariaDB Foundation. (2026). *Data Types*. MariaDB Knowledge Base. https://mariadb.com/kb/en/data-types/.

[4] Oracle Corporation. (2026). *MySQL 8.0 Reference Manual — Data Types*. https://dev.mysql.com/doc/refman/8.0/en/data-types.html.

[5] The PostgreSQL Global Development Group. (2026). *Chapter 8. Data Types*. https://www.postgresql.org/docs/current/datatype.html.

[6] Microsoft. (2026). *Data Types (Transact-SQL)*. Microsoft Learn. https://learn.microsoft.com/en-us/sql/t-sql/data-types/.

[7] IEEE. (2019). *IEEE Standard for Floating-Point Arithmetic (IEEE 754-2019)*. Institute of Electrical and Electronics Engineers.

[8] Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill Education.

[9] Date, C. J. (2003). *An Introduction to Database Systems* (8th ed.). Addison-Wesley.

[10] Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.

[11] Leach, P., Mealling, M., & Salz, R. (2005). *A Universally Unique IDentifier (UUID) URN Namespace* (RFC 4122). IETF.

[12] Klensin, J. (2008). *Simple Mail Transfer Protocol* (RFC 5321). IETF.