PANDUAN DEFINITIF UNTUK POWER QUERY (M)

Dr. Phil. Dony Novaliendry, S. Kom., M. Kom

Buku "Panduan Definitif untuk Power Query (M) – Jilid 2" melanjutkan pembahasan dari jilid sebelumnya dengan fokus yang lebih mendalam pada tipe data, nilai terstruktur, dan konsep lanjutan dalam bahasa M. Buku ini dirancang untuk membantu pembaca memahami bagaimana data direpresentasikan, dikelola, dan dimanipulasi secara efisien di dalam Power Query.

Pembahasan dimulai dengan pemahaman tipe data—baik primitif maupun khusus—serta pentingnya konsistensi, validasi, dan konversi tipe dalam proses transformasi data. Selanjutnya, buku menguraikan secara rinci tentang nilai terstruktur seperti list, record, dan tabel, lengkap dengan metode pembuatan, manipulasi, serta penetapan tipe data pada masing-masing struktur. Pada bagian lanjutan, buku ini membahas konseptualisasi M, termasuk ruang lingkup, lingkungan global, closures, hingga pengelolaan metadata. Topik terakhir memperdalam kemampuan pembaca dalam bekerja dengan struktur bersarang (nested structures), mencakup pengolahan list, record, tabel, hingga kombinasi struktur yang lebih kompleks.

Dengan disertai kasus pemantik berpikir kritis, tes formatif, glosarium, dan lampiran, jilid kedua ini memberikan wawasan praktis sekaligus teoritis, sehingga menjadi panduan berharga bagi mahasiswa, dosen, peneliti, maupun praktisi data yang ingin menguasai Power Query secara komprehensif.

PENERBITAN & PERCETAKAN UNP PRESS
Jln. Prof. Dr. Hamka Air Tawar Padang
Sumatera Barat

PANDUAN DEFINITIF UNTUK POWER QUERY (M)
Dr. Phil. Dony Novaliendry, S. Kom., M. Kom

PANDUAN DEFINITIF UNTUK POWER QUERY (M)

JILID 2

Penerbitan & Percetakan
UNP PRESS

Dr. Phil. Dony Novaliendry, S. Kom., M. Kom

PANDUAN DEFINITIF UNTUK POWER QUERY (M) JILID 2

Dr. Phil. Dony Novaliendry, S. Kom., M. Kom

DUMMY
Penerbitan & Percetakan
UNP PRESS

UNDANG-UNDANG REPUBLIK INDONESIA
NO 19 TAHUN 2002
TENTANG HAK CIPTA
PASAL 72
KETENTUAN PIDANA SANGSI PELANGGARAN

1. Barang siapa dengan sengaja dan tanpa hak mengumumkan atau memperbanyak suatu Ciptaan atau memberi izin untuk itu, dipidana dengan pidana penjara paling singkat 1 (satu) bulan dan denda paling sedikit Rp 1.000.000, 00 (satu juta rupiah), atau pidana penjara paling lama 7 (tujuh) tahun dan denda paling banyak Rp 5.000.000.000, 00 (lima milyar rupiah)
2. Barang siapa dengan sengaja menyerahkan, menyiarkan, memamerkan, mengedarkan, atau menjual kepada umum suatu Ciptaan atau barang hasil pelanggaran Hak Cipta atau Hak Terkait sebagaimana dimaksud dalam ayat (1), dipidana dengan pidana penjara paling lama 5 (lima) tahun dan denda paling banyak Rp 500.000.000, 00 (lima ratus juta rupiah).

PANDUAN DEFINITIF UNTUK POWER QUERY (M) JILID 2

Dr. Phil. Dony Novaliendry, S. Kom., M. Kom

DUMMY
Penerbitan & Percetakan
UNP PRESS

UNP PRESS

2025

PANDUAN DEFINITIF UNTUK POWER QUERY (M) JILID 2
editor, Tim editor UNP Press
Penerbit UNP Press, Padang, 2025
1 (satu) jilid; 17.6 x 25 cm (B5)
Jumlah Halaman xiii + 309 Halaman Buku

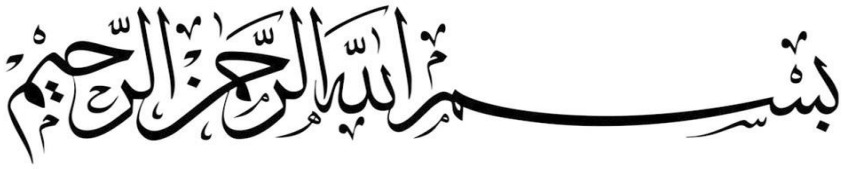
ISBN:

DUMMY
Penerbitan & Percetakan
UNP PRESS

PANDUAN DEFINITIF UNTUK POWER QUERY (M)

Penyusun: Dr. Phil. Dony Novaliendry, S. Kom., M. Kom
Editor Substansi: Fadhillah Majid Saragih, S.Pd., M.Pd.T
Editor Bahasa: Prof. Dr. Harris Effendi Thahar, M.Pd.
Desain Sampul & Layout: Mukhlis Zaki Insani

KATA PENGANTAR



Puji syukur kehadirat Allah SWT atas berkat limpahan rahmat dan karunia-Nya sehingga Buku ajar Panduan Definitif untuk Power Query (M) dapat di buat. Buku ajar Panduan Definitif untuk Power Query (M) adalah buku ajar yang membahas tentang Menguasai Transformasi Data Kompleks dengan Power Query yang bisa di manfaatkan oleh peserta didik khususnya dan bagi semua pihak dari segala lapisan yang membutuhkan sebagai referensi untuk belajar mengunakan Buku ajar Panduan Definitif untuk Power Query (M).

Kami mengucapkan terima kasih kepada semua pihak yang telah membantu dalam proses penyelesain buku ajar ini.

Kami menyadari masih terdapat banyak kekurangan dalam buku ajar ini untuk itu kritik dan saran yang membangun demi penyempurnaan buku ajar ini sangat diharapkan. Dan semoga buku ini dapat memberikan maanfaat bagi para peserta didik khususnya dan bagi semua pihak dari segala lapisan yang membutuhkan.

Padang, Juni 2025

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DUMMY
Penerbitan & Percetakan
UNP PRESS

DAFTAR TABEL

DUMMY
Penerbitan & Percetakan
UNP PRESS

Pendahuluan

Dalam era digital yang semakin berkembang, data telah menjadi salah satu aset terpenting bagi organisasi dan individu. Kemampuan untuk mengolah, menganalisis, dan memvisualisasikan data dengan efektif menjadi kunci untuk mengambil keputusan yang tepat dan strategis. Salah satu alat yang sangat berguna dalam proses ini adalah Power Query, sebuah fitur yang tersedia dalam Microsoft Excel dan Power BI. Power Query memungkinkan pengguna untuk mengimpor, membersihkan, dan mengubah data dari berbagai sumber dengan cara yang intuitif dan efisien.

Buku ajar ini hadir sebagai panduan definitif untuk memahami dan memanfaatkan Power Query secara maksimal. Dengan pendekatan yang sistematis, buku ini dirancang untuk membantu pembaca dari berbagai latar belakang, baik pemula maupun yang sudah berpengalaman, dalam menguasai bahasa pemrograman M yang menjadi dasar dari Power Query. Melalui penjelasan yang jelas dan contoh-contoh praktis, pembaca akan diajak untuk menjelajahi berbagai fitur dan kemampuan yang ditawarkan oleh Power Query.

Dalam setiap bab, pembaca akan menemukan langkah-langkah yang terperinci untuk melakukan berbagai tugas, mulai dari pengambilan data hingga transformasi yang kompleks. Buku ini juga akan membahas berbagai teknik dan trik yang dapat meningkatkan efisiensi kerja, serta cara mengatasi tantangan yang sering dihadapi saat bekerja dengan data. Dengan demikian, pembaca tidak hanya akan belajar cara menggunakan Power Query, tetapi juga memahami prinsip-prinsip dasar yang mendasari pengolahan data.

Akhirnya, kita berharap buku ini dapat menjadi sumber referensi yang berguna dan inspiratif bagi siapa saja yang ingin meningkatkan keterampilan analisis data mereka. Dengan menguasai Power Query, pembaca akan memiliki

alat yang kuat untuk mengubah data menjadi informasi yang berharga, mendukung pengambilan keputusan yang lebih baik, dan pada akhirnya, mencapai tujuan yang diinginkan dalam dunia yang semakin didorong oleh data. Selamat membaca dan selamat berpetualang dalam dunia Power Query!

1. **Unduh File Kode Contoh**

   Kumpulan kode untuk buku ini dihosting di GitHub di https://github.com/PacktPublishing/The- Definitive-Guide-to-Power-Query-M-/. Kita juga memiliki kumpulan kode lain dari katalog buku dan video kita yang lengkap yang tersedia di https://github.com/PacktPublishing/. Lihatlah!

2. **Unduh Gambar Berwarna**

   Kita juga menyediakan berkas PDF yang berisi gambar berwarna dari cuplikan layar/diagram yang digunakan dalam buku ini. Anda dapat mengunduhnya di sini: https://packt.link/gbp/9781835089729.

3. **Konvensi yang Digunakan**

   Ada sejumlah konvensi teks yang digunakan di seluruh buku ini. *CodeInText*: Menunjukkan kata kode dalam teks, nama tabel basis data, nama folder, nama file, ekstensi file, nama jalur, URL tiruan, input pengguna, dan akun Twitter. Misalnya: “Arahkan ke folder /ClientApp/src/app/cities.” Blok kode ditetapkan sebagai berikut:

```
#date(
  year as number,
  month as number,
  day as number,
) as date
```

   Bila kita ingin menarik perhatian Anda ke bagian tertentu dari blok kode, baris atau item yang relevan disorot:

```
#date(
  year as number,
  month as number,
  day as number,
) as date
```

Tebal: Menunjukkan istilah baru, kata penting, atau kata-kata yang Anda lihat di layar. Misalnya, kata-kata dalam menu atau kotak dialog muncul dalam teks seperti ini. Misalnya: “Arahkan ke tab Beranda pada pita, klik menu tarik-turun di bawah tombol Ubah data, dan pilih Edit parameter.”

DUMMY
Penerbitan & Percetakan
UNP PRESS

BAB 1

Memahami Tipe Data

Topik Bab

Pada bab ini, topik yang akan dibahas adalah:

- Pentingnya tipe data
- Tipe data tersedia dalam M
- Deteksi tipe
- Konversi tipe
- Sisi
- Atribusi tipe
- Kesetaraan dan kompatibilitas tipe

A. Pendahuluan

Pada bab sebelumnya, kita melihat peran nilai dalam membentuk ekspresi dan kueri dalam bahasa M. Kita mempelajari bagaimana nilai berinteraksi dengan berbagai operator dan fungsi serta karakteristik uniknya. Berdasarkan pengetahuan ini, mari kita lanjutkan perjalanan ini dengan mengalihkan perhatian kita ke tipe data.

Tipe data berfungsi sebagai sistem klasifikasi untuk nilai, yang menyediakan informasi tentang struktur dan penggunaannya dalam M. Memahami tipe data sangat penting untuk menggunakan M secara efektif dan menyimpan nilai secara efisien. Dalam bab ini, kita menyediakan ikhtisar komprehensif tentang tipe data, signifikansinya, dan penerapannya dalam skenario praktis.

Kita akan mulai dengan memeriksa mengapa tipe data penting. Anda akan menemukan berbagai tipe data dalam M dan konteks penggunaannya. Kita juga menjelaskan cara mengidentifikasi dan memahami berbagai tipe data, dengan fokus pada konversi tipe. Selain itu, bab ini memperkenalkan aspek dan menjelaskan cara menetapkan tipe data ke nilai. Berikut ikhtisar singkat bab ini:

- Pentingnya tipe data
- Tipe data tersedia dalam M
- Deteksi tipe
- Konversi tipe
- Sisi
- Atribusi tipe
- Kesetaraan dan kompatibilitas tipe

1. Kasus Pemantik Berfikir Kritis: Analisis Data Penjualan

Sebuah perusahaan retail ingin menganalisis data penjualan mereka untuk meningkatkan strategi pemasaran. Data penjualan disimpan dalam sebuah file Excel dan mencakup kolom-kolom berikut:

- Tanggal Penjualan

- Nama Produk
- Jumlah Terjual
- Total Pendapatan

Setelah mengimpor data ini ke Power Query, tim menemukan beberapa masalah sebagai berikut:

- **Format Tanggal yang Berbeda**: Beberapa entri di kolom Tanggal Penjualan menggunakan format yang berbeda (misalnya, ada yang dalam format DD/MM/YYYY dan ada yang MM/DD/YYYY).
- **Nilai yang Hilang**: Kolom Jumlah Terjual memiliki beberapa entri yang kosong, dan kolom Total Pendapatan berisi nilai yang tidak konsisten (beberapa nilai adalah teks).
- **Kesalahan Tipe Data**: Kolom Jumlah Terjual seharusnya bertipe angka, tetapi ada beberapa entri yang berisi teks atau karakter yang tidak valid.

Pertanyaan untuk Diskusi:

1) Bagaimana Anda akan mengatasi masalah format tanggal yang berbeda untuk memastikan konsistensi data?
2) Apa langkah-langkah yang dapat diambil untuk menangani nilai yang hilang di kolom Jumlah Terjual?
3) Bagaimana Anda akan memastikan bahwa kolom Jumlah Terjual memiliki tipe data yang benar? Apa yang akan Anda lakukan jika ada entri yang tidak valid?
4) Mengapa penting untuk menetapkan tipe data yang tepat untuk setiap kolom dalam dataset ini? Apa konsekuensi jika tidak dilakukan?
5) Diskusikan bagaimana pengaturan lokal (locale) dapat mempengaruhi cara Anda menangani dan menganalisis data tersebut.

B. Apa Itu Tipe Data?

Bahasa M memiliki nilai dan tipe data. Dalam dokumentasi resmi, tipe data secara formal disebut sebagai **types** karena tidak hanya mengklasifikasikan data tetapi juga mengklasifikasikan fungsi dan tipe data itu sendiri. Di seluruh buku ini, kita menggunakan istilah **tipe data** dan **types** agar tidak membingungkan saat membandingkannya dengan tipe nilai. Jadi, apa sebenarnya tipe data, dan bagaimana keduanya dibandingkan dengan nilai?

1. Sistem Tipe

Sistem tipe di Power Query membantu mengklasifikasikan nilai, menawarkan informasi tentang struktur data Anda. Saat membuat fungsi kustom, tipe data menentukan nilai yang diperlukan. Selain itu, tipe data menyampaikan informasi penting ke sistem mana pun tempat data dimuat. Mari kita mulai dengan mempelajari sebuah contoh.

Setiap jenis nilai dalam bahasa M memiliki tipe data. Tipe data adalah jenis nilai khusus yang mencirikan jenis nilai dan membawa metadata tambahan yang khusus untuk bentuk nilai tersebut. Ini mungkin terdengar rumit saat ini, tetapi tunggu dulu; kita akan mengilustrasikan semuanya dengan contoh yang jelas.

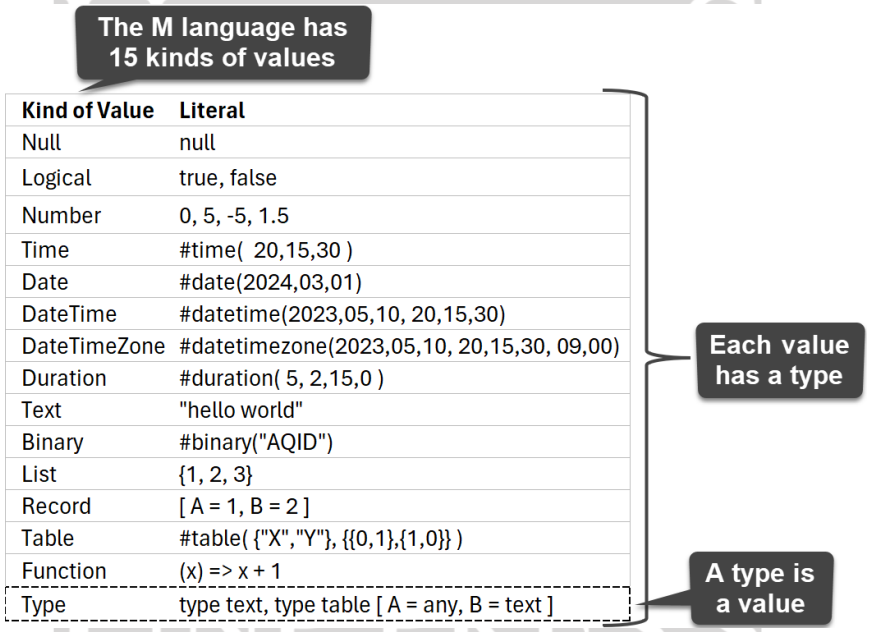
Tipe data adalah nilai, sama seperti angka adalah nilai. Untuk mengilustrasikan ini, mari kita lihat contoh sederhana. Misalkan Anda perlu menambahkan kolom baru ke tabel Anda. Anda dapat melakukannya dengan pernyataan seperti ini:

```
Table.AddColumn(
  #"Changed Type",   // A table value
  "Addition",        // a text value
  each [Value] + 10, // a function
  value type number  // a type value
)
```

Dalam ekspresi ini, setiap argumen fungsi Table.AddColumn menerima nilai dengan tipe yang berbeda:

1) Argumen pertama merujuk ke tabel dalam langkah yang disebut Tipe yang Diubah.
2) Argumen kedua berisi nilai teks dengan nama kolom baru.
3) Argumen ketiga mendefinisikan fungsi yang menghasilkan nilai untuk mengisi kolom baru.
4) Argumen keempat mendefinisikan tipe data untuk kolom baru.

Anda dapat menemukan total 15 jenis nilai dalam bahasa M, dan tipe juga merupakan nilai. Dengan menggunakan sintaksis literal (notasi) yang terkait dengan setiap nilai, Power Query membedakan antara nilai yang berbeda. Gambar 1.1 memperlihatkan ikhtisar semua nilai dan sintaksis literalnya.



Kind of Value	Literal
Null	null
Logical	true, false
Number	0, 5, -5, 1.5
Time	#time(20,15,30)
Date	#date(2024,03,01)
DateTime	#datetime(2023,05,10, 20,15,30)
DateTimeZone	#datetimezone(2023,05,10, 20,15,30, 09,00)
Duration	#duration(5, 2,15,0)
Text	"hello world"
Binary	#binary("AQID")
List	{1, 2, 3}
Record	[A = 1, B = 2]
Table	#table({"X","Y"}, {{0,1},{1,0}})
Function	(x) => x + 1
Type	type text, type table [A = any, B = text]

Gambar 1. 1 Berbagai jenis nilai dalam bahasa M

Seperti yang ditunjukkan pada tabel, ada banyak jenis nilai yang berbeda. Perhatikan bagaimana ekspresi berikut menghasilkan empat nilai yang berbeda:

```
"MyText"      // Returns a text value
2024          // Returns a number value
{ 1, 2, 3 }   // Returns a list value
type list     // Returns a type value
```

Fokuskan perhatian Anda pada contoh terakhir yang menunjukkan list tipe. Sama seperti nilai lainnya, tipe (data) juga merupakan nilai dan ada sintaksis khusus untuk mengembalikan tipe. Contoh tipe adalah:

```
type number // returns a type value holding numbers
type binary // returns a type value holding binary values
type text   // returns a type value holding text values
```

Ini menunjukkan bahwa ada beberapa jenis yang tersedia. Jadi, jika suatu nilai memiliki jenis yang terkait dengannya, kapan nuansa ini menjadi relevan?

2. Kolom dengan Tipe Campuran

Saat Anda berurusan dengan tabel di Power Query M, perbedaan antara tipe data dan nilai menjadi sangat jelas. Kolom memiliki tipe data yang menjelaskan jenis nilai yang terdapat dalam kolom tersebut.

Untuk mengikuti contoh berikut, Anda dapat membuka file latihan yang disertakan dalam buku ini. Kita menganjurkan Anda untuk mencoba berbagai transformasi dan menguji pengetahuan Anda.

Bayangkan situasi di mana tipe data kolom tidak mudah. Data Anda berasal dari sumber yang tidak terstruktur (seperti Excel), dan Anda akan menemukan bahwa data tersebut dapat berisi berbagai jenis nilai. Dalam contoh khusus ini, kolom Anda berisi berbagai nilai, termasuk teks, angka, nilai logika, dan tanggal. Anda dapat membuat kolom seperti itu dengan menulis:

```
Table.FromColumns(
    { { "abc", 123, true, #date(2024,1,1) } } ,
    {"Mixed Data"}
 )
```

Kode ini membuat kolom yang disebut **Data Campuran (Mixed Data)** dengan empat nilai unik. Karena kita tidak menentukan tipe data, Power Query secara otomatis menetapkan tipe apa pun ke kolom ini, yang diwakili oleh **ABC123** di tajuk kolom:

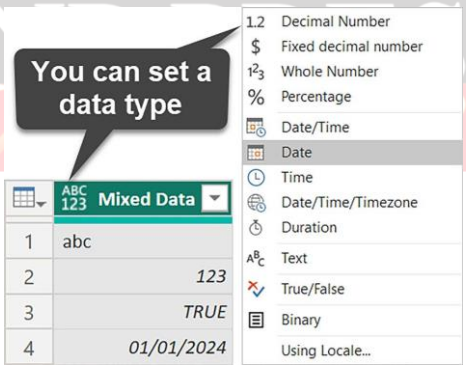
	Mixed Data
1	abc
2	*123*
3	*TRUE*
4	*01/01/2024*

Gambar 1. 2 Kolom dengan campuran nilai yang berbeda

Menetapkan tipe data yang lebih spesifik ke kolom tersebut bisa jadi sulit karena kolom tersebut berisi berbagai jenis nilai. Bila kita tidak menetapkan tipe data dalam ekspresi kita, Power Query, oleh karena itu, akan menggunakan tipe data any secara default. Tipe any memungkinkan berbagai tipe nilai dalam kolom yang sama. Namun, pilihan tipe data pada akhirnya bergantung pada kebutuhan spesifik Anda dan kalkulasi atau transformasi yang ingin Anda terapkan.

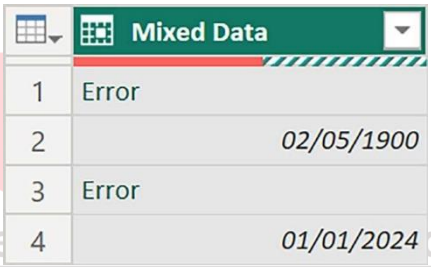
Misalnya, jika Anda ingin bekerja secara eksklusif dengan nilai tanggal, Anda dapat mempertimbangkan untuk menetapkan tipe data kolom ke tanggal. Namun, nilai dalam kolom Anda juga harus mendukung hal ini. Jadi, bagaimana Anda akan menetapkan tipe data tersebut?

Untuk menentukan tipe data kolom, klik ikon tipe data yang terletak di sebelah kiri tajuk kolom. Contoh berikut menetapkan tipe data kolom **Data Campuran** ke tipe **Tanggal (Date)**:



Gambar 1. 3 Tetapkan tipe data untuk kolom

Melalui operasi ini, Power Query akan mencoba mengubah nilai kolom menjadi tipe **Date**, dan memeriksa apakah nilai yang mendasarinya sesuai dengan tipe data ini. Jika, karena beberapa alasan, mesin tidak dapat mengubah nilai ke tipe data baru, kesalahan akan ditampilkan. Misalnya, mengubah tipe data kolom menjadi **Date** memberikan hasil berikut:



Gambar 1. 4 Mengonversi kolom yang berisi berbagai tipe data dapat menyebabkan kesalahan

Dalam contoh ini, mesin berhasil mengonversi nilai 123 dan tanggal 01/01/2024 ke nilai tanggal tetapi menghasilkan kesalahan untuk teks dan nilai logika. Hal ini memberi tahu kita bahwa M tidak mendukung transformasi semua nilai ke tipe tanggal dan dalam kasus nilai campuran ini, tipe data any lebih cocok. Ini menunjukkan bahwa meskipun setiap nilai secara individual mungkin bertipe tertentu, kolom menerima campuran nilai dengan tipe any. Nanti di bab ini, kita memberikan ikhtisar tentang transformasi antara tipe data mana yang didukung dalam bahasa M.

Contoh sebelumnya dimaksudkan untuk mengilustrasikan bahwa tipe data dapat berbeda tergantung pada nilai apa yang dideskripsikannya. Dengan campuran nilai, tipe any mungkin paling sesuai.

Sekarang, mari beralih ke contoh lain di mana kita akan melihat perbedaan yang halus tetapi penting antara tipe data dan tipe nilai. Perbedaan itu penting karena kolom tipe tertentu dapat berisi nilai tipe lain. Mari pelajari apa artinya.

3. Tipe Data Kolom Versus Tipe Nilai

Tipe (data) berbeda dari nilai yang dijelaskannya; lagipula, setiap nilai memiliki tipe. Pertimbangkan tabel yang menyertakan kolom untuk **Invoice ID, Date**, dan **Sales Amount**. Anda dapat mengikutinya dengan membuka file latihan yang disertakan dalam buku. Misalkan Anda ingin menambahkan kolom baru untuk menunjukkan diskon 5% di semua baris.

Untuk menyisipkan kolom baru ini, Anda harus:

- Arahkan ke **Add Columns** dan pilih **Custom Column**.
- Masukkan nilai 0,05 dan tetapkan nama kolom Diskon.

Ini akan membuat tabel dalam gambar berikut:

```
= Table.AddColumn(#"Changed Type", "Discount", each 0.05)
```

	Invoice ID	Date	Sales Amount	Discount
1	INV-11302	*01/01/2024*	*195*	*0.05*
2	INV-11303	*15/01/2024*	*925*	*0.05*
3	INV-11304	*31/01/2024*	*250*	*0.05*
4	INV-11305	*01/02/2024*	*500*	*0.05*

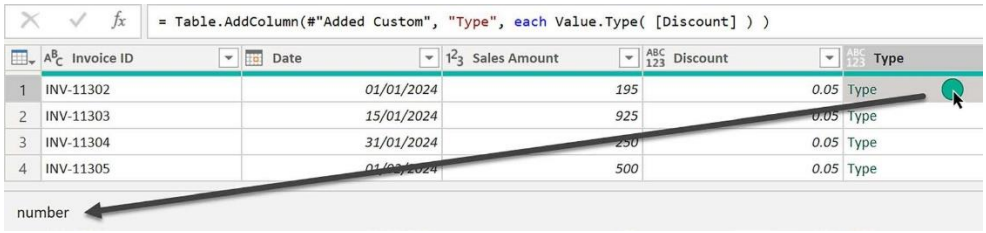
Gambar 1. 5 Tabel dengan kolom Diskon bertipe apa pun

Saat kita menambahkan kolom Diskon, Power Query secara otomatis memberinya tipe data apa pun karena kita belum menentukan tipe tertentu untuknya. Namun, di sinilah letak kesulitannya: kolom Diskon diberi label sebagai tipe apa pun, tetapi nilai sebenarnya di kolom tersebut adalah nilai angka. Kolom tersebut tidak memiliki label yang menyebutkan nilai-nilai tersebut.

Kita dapat menunjukkannya dengan menggunakan fungsi *Value.Type* pada nilai-nilai kolom **Discount**. Untuk melakukannya, ikuti langkah-langkah berikut:

1) Navigasi ke **Add Column** dan pilih **Custom Column**.
2) Kali ini, masukkan rumus: *Value.Type([Diskon])*.
3) Tekan **Ok**.

Anda akan mendapatkan tabel berikut:



Gambar 1. 6 Power Query dapat mengenali jenis nilai bahkan tanpa tipe data yang ditetapkan

Dengan menerapkan fungsi *Value.Type* pada setiap nilai di kolom **Discount**, kita akan mendapatkan kolom yang untuk setiap baris menunjukkan jenis nilai yang dikandungnya. Untuk melihat pratinjau jenisnya, Anda dapat mengeklik spasi kosong di dalam sel, di mana Anda dapat melihat bahwa nilai tersebut berjenis *Number*.

Nah, ini menarik. Jadi, bahkan tanpa tipe data yang ditetapkan ke kolom, fungsi tersebut tetap mengenali jenis nilai yang sedang kita kerjakan. Bagaimana cara kerjanya?

Ingatkah cara kita memasukkan nilai diskon sebelumnya? Kita meneruskan nilai 0,05 ke fungsi *Table.AddColumn*. Power Query mengenali sintaksis literal (notasi) yang digunakan untuk memasukkan nilai diskon. Ini memungkinkan mesin mengenali jenis nilai yang sedang kita kerjakan.

Itu berarti bahwa memiliki tipe, seperti yang lainnya, di tajuk kolom, tidak berarti nilai kolom yang mendasarinya bertipe sama. Itu hanya menunjukkan bahwa nilai tersebut sesuai dengan tipe data kolom tersebut. Seperti yang telah kita lihat sejauh ini, M dapat bekerja dengan nilai bertipe any dan mengenali nilai apa yang sedang kita kerjakan. Pertanyaan yang wajar untuk ditanyakan pada titik ini adalah: jika M mengenali jenis nilai apa yang sedang kita kerjakan, mengapa tidak menyederhanakannya saja dan menggunakan tipe data Any untuk seluruh kumpulan data? Itulah yang akan kita bahas di bagian berikutnya.

C. Pentingnya Tipe

Bahasa M adalah bahasa kueri yang diketik secara dinamis. Artinya, Anda tidak perlu mendeklarasikan variabel dan tipe (data) variabel tersebut sebelum dapat menggunakannya. Dan, seperti yang baru saja Anda pelajari, Power Query dapat mengenali jenis data yang diterimanya, tetapi tidak menjelaskan tipe data secara eksplisit berisiko dalam data. Saat kolom Anda diberi label sebagai tipe apa pun, ini menandakan bahwa kolom tersebut dapat berisi nilai apa pun. Saat Anda kemudian melakukan operasi yang hanya berfungsi pada tipe nilai tertentu, operasi Anda dapat mengakibatkan kesalahan.

Di bagian ini, kita akan membahas mengapa lebih baik untuk menjelaskan tipe data Anda dengan jelas. Anggap tipe data seperti memberi label pada kotak saat Anda memindahkan; ini memerlukan perhatian saat menyimpan, tetapi akan menghemat banyak kebingungan di kemudian hari. Jadi, mengapa tipe data penting?

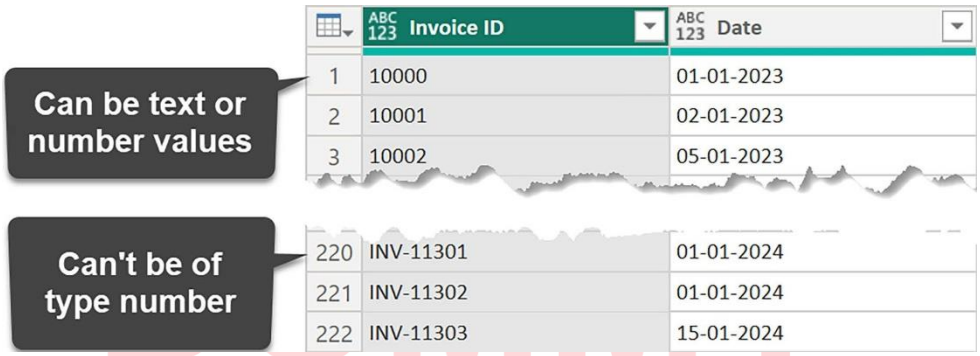
1. Kejelasan dan Konsistensi

Tipe data berperan penting dalam meningkatkan kejelasan dan memastikan konsistensi dalam kumpulan data. Tipe data melakukannya dengan memberi sinyal yang jelas tentang sifat nilai yang terkandung dalam kolom. Saat Anda menentukan tipe data untuk kolom, Anda menetapkan ekspektasi dan aturan yang jelas untuk data yang dikandungnya. Data yang tidak sesuai dengan tipe data tersebut akan menimbulkan kesalahan, menandai ketidakkonsistenan, dan membantu menjaga integritas data.

Bayangkan Anda sedang menangani sekumpulan faktur. Sebelumnya, perusahaan Anda menggunakan urutan numerik sederhana untuk nomor faktur: 10000, 10001, 10002, dan seterusnya. Dengan tahun baru, format diperbarui untuk menyertakan awalan dan tanda hubung, seperti INV-11000, INV-11001, dan INV-11002.

Tanpa menentukan tipe data, sekilas pandang pada kumpulan data mungkin membuat Anda mengira bahwa Anda sedang berhadapan dengan data

numerik. Lagi pula, format data baru baru akan terlihat saat Anda mencapai baris 220:



Gambar 1. 7 Kumpulan data yang pemindaian 200 baris teratasnya dapat menyebabkan asumsi tipe data yang salah

Contoh ini menunjukkan bahwa tidak adanya tipe data yang ditetapkan dapat menyebabkan asumsi yang salah tentang data Anda dan mungkin mengakibatkan kesalahan pemrosesan. Jika Anda kemudian menerapkan fungsi yang hanya kompatibel dengan angka, Anda akan mengalami kesalahan. Bagaimanapun, seluruh kumpulan data diproses saat memuat data Anda ke tujuan seperti Power BI atau Excel. Setelah mesin mencoba memproses nilai teks, operasi angka kemudian gagal.

Sebaliknya, tipe data yang ditetapkan dengan jelas untuk kolom mencegah masalah tersebut dengan mengomunikasikan sifat data yang tepat, memastikan semua nilai sesuai dengan format yang ditentukan. Kejelasan ini, bersama dengan konsistensi yang dibawanya, memungkinkan Anda bekerja dengan percaya diri dengan data Anda. Namun, itu bukan satu-satunya alasan untuk menetapkan tipe data. Tipe data juga berguna untuk memvalidasi data Anda, seperti yang akan kita bahas di bagian berikutnya.

2. Validasi Data

Tipe data memainkan peran penting dalam validasi data dan pencegahan kesalahan. Dengan menetapkan tipe data, Anda dapat memastikan bahwa data

sesuai dengan tipe tertentu. Ini penting karena banyak operasi bergantung pada penerimaan tipe data yang benar.

Ambil contoh, tindakan menambahkan nilai bersama-sama. Tidak masuk akal untuk menjumlahkan nilai teks. Demikian pula, saat Anda menggabungkan tabel, operasi Gabungkan Kolom mengharuskan kolom yang digunakan untuk penggabungan memiliki tipe data yang cocok. Misalkan penggabungan Anda memberikan hasil yang tidak diharapkan. Dalam skenario itu, menentukan masalah bisa jadi sulit jika Anda tidak menentukan tipe data Anda.

Mungkin contoh paling jelas mengapa tipe data penting adalah saat membuat fungsi kustom, topik yang kita bahas dalam Bab 9, Parameter dan Fungsi Kustom. Fungsi kustom menerima parameter, melakukan operasi dengannya, dan mengembalikan hasil. Saat Anda menentukan fungsi, disarankan untuk mendeklarasikan tipe data yang diharapkan untuk setiap parameter. Misalnya, fungsi berikut mengharapkan nilai tipe data teks sebagai input dan mengembalikan nilai tipe teks sebagai output:

```
( myText1 as text, myText2 as text ) as text => myText1 & myText2
```

Fungsi ini mengambil dua nilai teks sebagai input, menggabungkannya, dan mengembalikan nilai gabungan tersebut. Menetapkan tipe data secara eksplisit mengomunikasikan ke mesin bahwa setiap nilai yang diteruskan ke fungsi harus sesuai dengan tipe data yang dideklarasikan, dan memvalidasi input. Dalam skenario saat fungsi menerima nilai input dengan tipe yang berbeda, misalnya *number*, fungsi akan menampilkan kesalahan dan memberi sinyal kepada pengguna bahwa ada yang salah. Umpan balik langsung ini membantu memastikan fungsi digunakan seperti yang diharapkan. Selain dua alasan yang baru saja kita bahas, ada juga alasan lain untuk menyediakan tipe data, yang akan kita bahas di bagian berikutnya.

3. Alasan Lain

Selain memberikan kejelasan, konsistensi, dan validasi data, tipe data penting karena alasan berikut:

- **Performa**: Menetapkan tipe data dapat mempercepat kueri Anda. Saat Anda menetapkan tipe untuk kolom, Power Query mengoptimalkan cara penyimpanan dan pengambilan data tersebut, sehingga mengurangi penggunaan memori. Di sisi lain, jika Anda tidak menetapkan tipe, Power Query akan mencoba menyimpulkan (menebak) tipe tersebut. Inferensi tipe menghabiskan sumber daya komputasi tambahan, sehingga memperlambat kueri Anda. Untuk mempelajari lebih lanjut tentang peningkatan performa kueri, rujuk Bab 15, Mengoptimalkan Performa.
- **Penanganan kesalahan**: Dengan menetapkan tipe data, Power Query dapat menangani kesalahan dengan lebih baik. Jika Power Query menemukan nilai yang tidak sesuai dengan tipe yang ditetapkan, Power Query akan menolaknya atau mengonversinya, tergantung pada ekspresi Anda. Berbagai teknik penanganan kesalahan dibahas dalam Bab 12, Menangani Kesalahan dan Mendebug.
- **Kompatibilitas**: Tipe data dapat menjadi penting saat mengintegrasikan dengan sistem lain atau saat mengekspor data. Sistem tertentu memerlukan tipe data tertentu untuk operasi atau antarmukanya. Dengan menggunakan tipe data, Power Query memastikan kompatibilitas dengan sistem ini.

  Misalnya, meskipun Anda dapat memanggil skrip R dari dalam Power Query untuk melakukan analisis statistik tingkat lanjut, jika data Anda dinyatakan sebagai mata uang (benar-benar valid dalam M), data tersebut akan dibaca ke dalam R sebagai teks, dan sebagian besar fungsi statistik yang dipanggil dalam skrip tidak akan berfungsi.

Seperti yang dibahas sebelumnya, tipe data memiliki lebih dari sekadar tujuan deskriptif; tipe data memainkan peran penting dalam menjaga

konsistensi dan integritas data, meningkatkan kinerja, penanganan kesalahan, memastikan kompatibilitas sistem, dan meningkatkan kejelasan secara keseluruhan.

Di bagian berikutnya, kita akan membahas lebih dalam berbagai jenis tipe data yang tersedia dalam bahasa M. Kita akan memperkenalkan tipe-tipe yang tersedia, menjelaskan propertinya, dan menunjukkan cara menggunakannya.

D. Tipe Data Tersedia Dalam M

Bahasa M memiliki berbagai tipe (data) yang berbeda. Tipe-tipe tersebut menyediakan cara untuk mengklasifikasikan nilai dan terkadang membatasi jenis data yang diizinkan dalam fungsi kustom. Anda dapat mengatakan tipe data mendefinisikan bentuk nilai dan menunjukkan operasi yang dapat dilakukan padanya.

Sistem tipe dalam M dapat dilihat sebagai hierarki. Pada dasarnya, semua nilai sesuai dengan tipe apa pun. Dari sini, kita dapat beralih ke tipe yang lebih spesifik.

Pada lapisan di bawah hierarki, Anda dapat menemukan tipe yang lebih spesifik. Pikirkan tipe primitif, serta konstruksi yang lebih kompleks seperti record, list, dan tabel. Sistem tipe juga memungkinkan definisi tipe kustom, yang memberi pengguna fleksibilitas untuk menentukan struktur data kustom.

Kita juga dapat membedakan antara nilai yang dapat dan tidak dapat menyimpan nilai *null* melalui tipe yang dapat *nullable types*. Perbedaan ini berguna saat menangani sumber data di mana tidak adanya nilai, yang diwakili oleh null, merupakan suatu kemungkinan dan harus ditangani dengan benar. Jadi, apa saja semua tipe yang berbeda ini dan bagaimana Anda dapat menggunakannya?

1. Tipe Primitif

Tipe primitif adalah tipe data fundamental yang menjadi dasar pembuatan semua tipe data lainnya. Tipe primitif dapat digunakan untuk

mengklasifikasikan nilai primitif, tetapi juga sebagai blok penyusun tipe data yang lebih kompleks seperti record, tabel, atau fungsi.

Anda dapat menemukan total 18 tipe primitif dalam bahasa M. Enam di antaranya adalah tipe abstrak yang mengklasifikasikan lebih dari satu nilai dan empat dapat digunakan sebagai tipe kustom:

Primitive Types	Type Qualifiers	Value
binary		Binary
date		Date
datetime		DateTime
datetimezone		DateTimeZone
duration		Duration
list	Custom	List
logical		Logical
null		Null
number		Number
record	Abstract, Custom	Record
text		Text
time		Time
type		Type
function	Abstract, Custom	Function
table	Abstract, Custom	Table
any	Abstract	
anynonnull	Abstract	
none	Abstract	

Abstract: type is considered an abstract type
Custom: type can also be used as custom type

Gambar 1. 8 Tinjauan umum tipe primitif di M

Pada tabel sebelumnya, Anda dapat menemukan informasi berikut:

- Kolom pertama, **Primitive Types**, mencantumkan semua tipe primitif yang tersedia dalam bahasa M, yang mencakup tipe abstrak.
- Kolom **Type Qualifiers** mengidentifikasi tipe primitif mana yang abstrak dan juga menunjukkan tipe primitif mana yang dapat digunakan sebagai tipe kustom. Kita akan membahas lebih lanjut istilah-istilah ini nanti di bab ini.

- Kolom **Value** di posisi ketiga menampilkan nilai-nilai spesifik yang diklasifikasikan menurut setiap tipe. Kolom ini mengungkapkan bahwa tipe-tipe tertentu (seperti *any*, *anynonnull*, dan *none*) tidak sesuai dengan satu nilai, sehingga sel-sel ini dibiarkan kosong.

Catatan: Tabel tidak menyertakan tipe seperti *Int64.Type* (bilangan bulat), *Currency.Type* (bilangan desimal tetap), dan *Number.Type* (bilangan desimal). Alasannya adalah karena ini adalah Klaim Tipe, yang termasuk dalam kategori aspek. Masing-masing memiliki tipe dasar yang disertakan dalam tabel yang disediakan sebelumnya. Untuk eksplorasi lebih rinci tentang aspek ini dan aspek lainnya, lihat bagian Aspek dalam bab ini.

Salah satu alasan untuk mempelajari tipe data adalah karena berbagai jenis operasi didukung oleh nilai yang berbeda. Memahami dan mengidentifikasi tipe data yang Anda gunakan menandakan apa yang dapat Anda lakukan dengan data tersebut. Berikut adalah lima contoh nilai dan sintaksnya:

```
2024                  // Returns a value of type number
[ a = 10, b = 20 ]  // Returns a value of type record
#date( 2024, 12, 31 ) // Returns a value of type date
{ 1 }                 // Returns a value of type list
null                  // Returns a value of type null
```

Nah, contoh-contoh ini menunjukkan berbagai jenis nilai. Jika sebaliknya, Anda ingin mengembalikan tipe (data) yang Anda hadapi, Anda dapat menggunakan fungsi *Value.Type*. Fungsi ini menerima nilai dan mengembalikan tipe datanya. Misalnya:

```
Value.Type( 2024 )                  // Returns 'type number'
Value.Type( [ a = 10, b = 20 ] )    // Returns 'type record'
Value.Type( #date( 2024, 12, 31 ) ) // Returns 'type date'
Value.Type( { 1 } )                 // Returns 'type list'
Value.Type( null )                  // Returns 'type null'
```

Yang menarik adalah fungsi *Value.Type* juga dapat mengembalikan tipe. Misalnya:

```
Value.Type( type function ) // Returns 'type type'
```

Output ini menunjukkan bahwa kita berurusan dengan nilai tipe, jenis nilai khusus yang digunakan untuk mengklasifikasikan nilai lain. Di Power Query, tampilannya seperti berikut:

Gambar 1. 9 Fungsi Value.Type mengembalikan representasi tekstual dari suatu tipe

Meskipun output dari ekspresi ini tampak seperti teks, ingatlah bahwa ini hanyalah representasi tekstual dari suatu tipe, bukan nilai aktual atau propertinya. Untuk mengekstrak informasi tambahan tentang suatu tipe, Power Query menyediakan serangkaian fungsi yang diawali dengan Tipe. Fungsi-fungsi ini dirancang untuk menyelidiki secara spesifik dan karakteristik tipe.

Sekarang setelah Anda mengetahui tentang tipe primitif umum dan cara mengenalinya, mari kita lihat lebih dekat tipe primitif abstrak.

Tipe Primitif Abstrak

Dalam bahasa M, tipe primitif dibagi menjadi dua kategori: tipe abstrak dan non-abstrak. Ada 6 tipe abstrak dan 12 tipe non-abstrak dan perbedaan utamanya terletak pada cara tipe-tipe ini mengklasifikasikan nilai. Tipe primitif non-abstrak bersifat spesifik; tipe ini secara unik mengidentifikasi satu nilai spesifik. Sebaliknya, tipe abstrak lebih luas dan kurang spesifik. Tipe ini tidak sepenuhnya mengklasifikasikan satu nilai unik.

Ambil fungsi tipe sebagai contoh. Meskipun semua fungsi dalam M sesuai dengan tipe ini, fungsi tipe saja tidak cukup untuk menjelaskan detail fungsi, seperti parameter dan tipe datanya. Untuk klasifikasi yang lengkap dan spesifik, tipe kustom diperlukan. Tipe kustom dibahas nanti dalam bab ini.

Konsep ini juga berlaku untuk tabel tipe dan record tipe. Tipe ini secara luas mengklasifikasikan tabel dan record, tetapi tidak menjelaskan hal-hal spesifik seperti nama kolom atau bidang, atau tipe data kolom atau bidang. Tipe kustom diperlukan untuk klasifikasi terperinci tersebut.

Pertimbangkan juga tipe any dan *anynonnull*. Setiap nilai yang mungkin dalam M sesuai dengan tipe any, dan setiap nilai bukan-null sesuai dengan tipe *anynonnull*. Akan tetapi, tidak satu pun dari tipe ini yang menetapkan nilai yang unik dan individual.

Terakhir, ada tipe none, yang unik karena tidak mungkin menghasilkan nilai yang sesuai dengannya. Ekspresi tipe none pasti akan menghasilkan kesalahan (kesalahan tidak dianggap sebagai nilai) atau memasuki loop tanpa akhir. Dengan penalaran ini, tipe none tidak mengklasifikasikan nilai yang unik karena tidak ada nilai yang sesuai dengannya. Oleh karena itu, tipe none bukanlah tipe yang akan Anda gunakan saat menulis kode.

Poin pentingnya adalah bahwa setiap kali Anda mendeskripsikan suatu nilai dengan salah satu tipe data abstrak, itu bukanlah klasifikasi nilai yang tepat.

Klasifikasi menggunakan tipe abstrak tidak merepresentasikan nilai yang mendasarinya secara akurat atau klasifikasi tersebut tidak mengklasifikasikan suatu tipe sama sekali (seperti halnya tipe none). Secara umum, tipe abstrak memerlukan penanganan yang lebih bernuansa karena dapat mendeskripsikan berbagai nilai atau tidak ada nilai sama sekali.

Untuk kolom yang mengharapkan beberapa tipe nilai, Anda dapat menggunakan tipe umum seperti any atau *anynonnull*. Namun, ada konsep lain yang berguna yang membuat tipe data lain mendukung nilai null. Anda dapat melakukannya dengan membuatnya dapat bernilai null, yang akan kita bahas selanjutnya.

Tipe Primitif yang Dapat Dibatalkan

Tipe primitif tidak hanya terbatas pada bentuk dasar yang telah kita bahas sebelumnya. Ada variasi penting dari tipe ini yaitu tipe **nullable primitive type**. Tipe ini dibangun di atas tipe primitif standar dengan menggabungkan kemampuan tambahan: kemampuan untuk merepresentasikan ketiadaan nilai dengan null.

Bayangkan **nullable primitive type** sebagai sistem yang mampu menampung dua jenis nilai. Jenis pertama mirip dengan jenis yang tidak dapat diubah menjadi null, yang menyimpan nilai tertentu seperti angka atau teks. Jenis kedua adalah kemampuan untuk menampung nilai null. Nilai null ini menandakan ketiadaan nilai. Anda dapat menganggap tipe primitif yang dapat diubah menjadi null sebagai nilai abstrak, karena tipe ini dapat memuat **nullable primitive type**, *null*.

Pentingnya tipe **nullable types** menjadi jelas dalam skenario di mana operasi memerlukan jenis data tertentu, tetapi juga memerlukan fleksibilitas untuk menangani contoh nilai yang hilang. Untuk lebih memahami konsep ini, mari kita bahas sebuah contoh.

Pertimbangkan fungsi Power Query: *Number.IsOdd* dan *Number.Sign*. Fungsi-fungsi tersebut memiliki sintaksis berikut:

```
Number.IsOdd( number as number ) as logical

Number.Sign( number as nullable number ) as nullable number
```

Fungsi *Number.IsOdd* mudah dipahami. Fungsi ini mengambil angka dan mengembalikan true jika angka tersebut ganjil, dan *false* jika tidak. Fungsi ini tidak menerima nilai *null*, tetapi hanya membutuhkan nilai bertipe angka.

Di sisi lain, *Number.Sign* lebih mudah dipahami. Fungsi ini menerima angka yang dapat bernilai *null*, artinya fungsi ini menerima angka atau nilai *null*. Hal ini penting karena fungsi ini memungkinkan fungsi ini diterapkan pada kumpulan data yang beberapa nilainya mungkin hilang.

Mari kita pertimbangkan contoh praktis: Anda memiliki kumpulan data dengan kolom Jumlah yang berisi nilai numerik, tetapi beberapa entri hilang dan direpresentasikan sebagai null. Saat menerapkan fungsi *Number.IsOdd* dan *Number.Sign* pada kolom ini, perilaku yang berbeda muncul, seperti yang ditunjukkan pada gambar layar berikut:

	Amount	Number.IsOdd	Number.Sign
1	-10	FALSE	-1
2	-5	TRUE	-1
3	null	Error	null
4	0	FALSE	0
5	null	Error	null
6	3	TRUE	1
7	8	FALSE	1

Gambar 1. 10 Perilaku fungsi dengan dan tanpa tipe nullable

Gambar 1.10 menunjukkan kepada kita bahwa:

- Dengan *Number.IsOdd*, fungsi tersebut hanya memproses entri dengan nilai numerik aktual. Entri yang berisi nilai null akan menghasilkan kesalahan, karena *Number.IsOdd* tidak menerima nilai null.
- Sebaliknya, fungsi *Number.Sign* dapat menangani semua baris. Untuk angka, fungsi tersebut akan menghasilkan 1, 0, atau -1, tergantung pada apakah angka tersebut positif, 0, atau negatif. Untuk nilai null, fungsi tersebut hanya akan menghasilkan *null*, yang mengakui tidak adanya nilai tanpa menyebabkan kesalahan.

Konsep tipe **nullable types** sangat penting saat menangani fungsi. Untuk fungsi, Anda ingin mengetahui apakah fungsi tersebut menerima null sebagai input atau mungkin menghasilkan null sebagai hasilnya. Memahami cara menggunakan tipe **nullable types** membantu mencegah kesalahan. Dalam Bab 1 Jilid 3, Parameter dan Fungsi Kustom, kita akan mempelajari topik ini lebih dalam. Anda akan mempelajari cara menggunakan tipe yang dapat

menghasilkan null untuk parameter fungsi, yang membuat fungsi kustom Anda lebih fleksibel dan tidak mudah mengalami kesalahan.

Menerapkan Kata Kunci Nullable Pada Tipe

Sekarang setelah kita menetapkan pentingnya tipe **nullable types**, mari kita periksa dampak penerapan kata kunci nullable pada tipe data dalam bahasa M. Aturan praktisnya sederhana: memberi awalan pada tipe data apa pun dengan kata kunci nullable memungkinkan tipe tersebut mengakomodasi **primitive value** dan **null**. Berikut ini tampilannya dalam tindakan:

```
nullable text  // describes both text and null values
nullable table // describes both table and null values
```

Dalam beberapa kasus, penggunaan nullable dengan suatu tipe data akan mengakibatkan transformasi tipe. Hal ini khususnya berlaku untuk tipe *anynonnull* dan *none*:

```
nullable anynonnull // Returns type any
nullable none       // Returns type null
```

Akan tetapi, ada pengecualian untuk aturan transformasi ini. Ketika *nullable* diterapkan pada tipe yang sudah mengakomodasi null, tipe tersebut tetap tidak berubah. Misalnya, menerapkan nullable pada tipe text mengubahnya menjadi *nullable text*, yang menggambarkan nilai yang dapat berisi nilai text atau null. Namun, menerapkan nullable pada tipe any, yang sudah menerima nilai apa pun termasuk null, tidak mengubah tipe tersebut. Contoh berikut tetap sama ketika menerapkan kata kunci *nullable*:

```
nullable any  // returns type any
nullable null // returns type null
```

Contoh-contoh berikut menunjukkan bagaimana kata kunci nullable dapat mengubah nilai tipe untuk mendukung null. Anda juga dapat melakukan kebalikannya dengan menggunakan fungsi *Type.NonNullable*:

```
Type.NonNullable( type nullable number )
```

Setelah membahas semua tipe primitif, sekarang kita siap untuk memeriksa penerapan praktisnya.

Mengatur Tipe Data Untuk Kolom

Kasus penggunaan yang paling umum untuk tipe adalah pengaturan tipe data untuk kolom. Mari kita lihat cara kerjanya. Bayangkan Anda memiliki kumpulan data dengan kolom seperti *Date*, *Product*, dan *Sales*. Kolom-kolom ini berisi nilai primitif, tetapi saat ini tidak memiliki kumpulan tipe data. Dalam kasus tersebut, Power Query secara default menggunakan tipe apa pun, yang diwakili oleh ikon **ABC123** di tajuk kolom:

	ABC 123 Date	ABC 123 Product	ABC 123 Sales
1	2024-01-05	Bread	2.50
2	2024-02-10	Milk	1.99
3	2024-03-15	Cereal	3.75
4	2024-04-20	Pasta	1.29

Gambar 1. 11 Tabel dengan kolom bertipe apa pun

Misalnya, kita ingin menetapkan tipe data untuk kolom-kolom ini. Untuk kolom *Date*, *Product*, dan *Sales*, Anda ingin menetapkan tipe data ke tanggal, teks, dan angka. Sebelum menetapkan tipe data, pastikan untuk meninjau nilai kolom guna memastikan bahwa nilai tersebut sesuai dengan tipe yang diinginkan guna menghindari kesalahan transformasi.

Setelah memeriksa data, Anda dapat menentukan tipe data dengan mengeklik ikon di sisi kiri setiap tajuk kolom dan memilih tipe yang diinginkan. Memilih tipe yang relevan untuk setiap kolom menghasilkan kode berikut:

```
Table.TransformColumnTypes(
  Source,
{ {"Date",     type date},
  {"Product", type text},
    {"Sales",    type number} } )
```

Kode ini menggunakan fungsi *Table.TransformColumnTypes*, yang melakukan dua hal. Fungsi ini menentukan tipe data untuk setiap kolom dan ketika suatu nilai tidak sesuai dengan tipe baru, fungsi ini mencoba mengubah nilai yang relevan menjadi tipe baru. Hal ini dapat terjadi, misalnya, ketika mengubah kolom dengan nilai angka menjadi *text*. Nanti di bab ini, kita akan membahas lebih lanjut tentang apa yang dimaksud dengan mengubah tipe data di bagian Konversi tipe data.

Menggunakan Tipe Primitif Untuk Penyaringan Data

Sebelum melanjutkan, mari kita lihat skenario lain di mana Anda dapat memperoleh manfaat dari mengetahui tipe data primitif Anda. Pertimbangkan skenario di mana Anda memiliki tabel dengan nilai campuran, termasuk Boolean, tanggal, waktu, dan angka. Anda ingin menyimpan hanya nilai angka. Sayangnya, mengubah seluruh kolom menjadi tipe angka tidak akan berhasil karena akan mengubah tipe lain juga, seperti yang ditunjukkan pada gambar berikut:

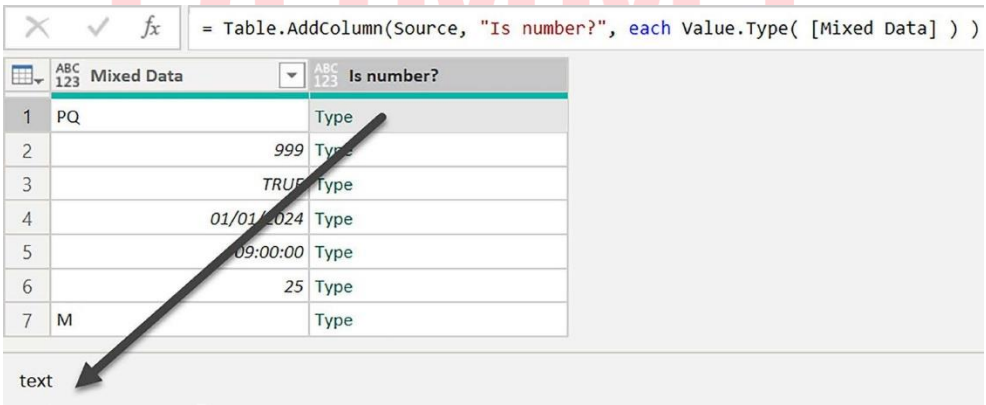
	Mixed Data
1	PQ
2	*999*
3	*TRUE*
4	*01/01/2024*
5	*09:00:00*
6	*25*
7	M

Gambar 1. 12 Tabel dengan berbagai jenis nilai

Untuk mengatasi tantangan ini, bahasa M menyediakan fungsi yang dapat mengidentifikasi tipe setiap nilai, sehingga Anda dapat memfilter tipe yang relevan. Fungsi *Value.Type* sangat berguna di sini. Fungsi ini menerima nilai sebagai input dan mengeluarkan tipe datanya. Di kolom kustom, Anda dapat menggunakan ekspresi seperti berikut:

```
Value.Type( [Mixed Date] )
```

Ini menghasilkan kolom baru yang diisi dengan tipe data setiap nilai:



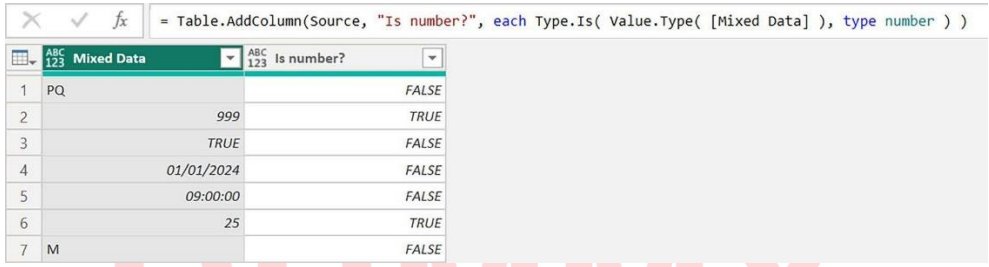
Gambar 1. 13 Fungsi Value.Type mengembalikan tipe (data) suatu nilai

Dengan mengklik spasi di sekitar sel, Anda dapat menampilkan pratinjau yang menunjukkan tipe data untuk setiap sel. Penting untuk dicatat bahwa tipe data yang dikembalikan ini merupakan deskriptor tipe nilai, bukan nilai itu sendiri.

Selanjutnya, kita ingin menguji apakah kita berurusan dengan nilai angka. Untuk melakukannya, kita dapat menggunakan fungsi lain bernama *Type.Is*. Fungsi ini menentukan apakah nilai yang ditentukan dalam argumen pertama sesuai dengan tipe data yang ditunjukkan dalam argumen kedua. Dengan menerapkan ini pada skenario kita, Anda dapat memeriksa apakah kolom tipe baru cocok dengan tipe angka dengan menggunakan ekspresi berikut:

```
Type.Is( Value.Type( [Mixed Data] ), type number )
```

Memasukkan ekspresi ke dalam kolom khusus memberi Anda metode mudah untuk memfilter secara eksklusif untuk nilai angka:



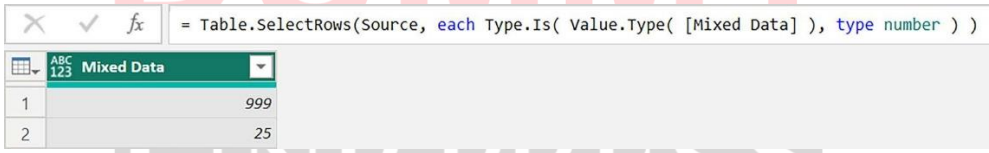
	Mixed Data	Is number?
1	PQ	FALSE
2	999	TRUE
3	TRUE	FALSE
4	01/01/2024	FALSE
5	09:00:00	FALSE
6	25	TRUE
7	M	FALSE

Gambar 1. 14 Ekspresi yang memvalidasi apakah suatu nilai bertipe tertentu

Kolom yang dihasilkan berisi nilai Boolean yang mengembalikan true untuk nilai angka. Dari sini, Anda memiliki dua opsi untuk memfilter. Salah satunya adalah langsung menerapkan filter ke kolom kustom baru. Pendekatan lain yang lebih ringkas adalah menggunakan ekspresi dalam fungsi *Table.SelectRows*. Sintaks untuk ini adalah:

```
Table.SelectRows(
  Source,

  each Type.Is( Value.Type( [Mixed Data] ), type number ) )
```

Menerapkan operasi itu membuat kita memiliki tabel yang hanya berisi nilai angka:



	Mixed Data
1	999
2	25

Gambar 1. 15 Ekspresi dapat memfilter nilai tipe tertentu

Sebagai alternatif, Anda juga dapat menggunakan sintaksis yang disederhanakan menggunakan operator *is*:

```
Table.SelectRows(
  Source,

  each [Mixed Data] is number )
```

Alternatif ini tidak hanya mengurangi jumlah kode tetapi juga meningkatkan keterbacaan. Seperti yang ditunjukkan contoh ini, dengan fungsi

yang tepat, memfilter nilai dari tipe tertentu relatif mudah. Kita akan membahas lebih dalam penggunaan sintaksis ini dan implikasinya saat membahas kesetaraan tipe nanti di bab ini.

Setelah membahas tipe primitif di bagian ini, sekarang kita akan membahas **tipe khusus**. Tipe ini lebih kompleks tetapi sangat berguna untuk menggambarkan nilai dan fungsi terstruktur Anda secara akurat.

2. Tipe Khusus

Tipe kustom bukan bagian dari himpunan tipe primitif tertutup. Sementara tipe primitif mendeskripsikan nilai sederhana seperti teks atau angka, tipe kustom mendeskripsikan nilai yang lebih kompleks. Tipe ini relevan untuk nilai terstruktur dan fungsi kustom. Untuk mendeskripsikan secara tepat apa yang terkandung dalam nilai-nilai ini, kita memerlukan kerangka kerja yang menggabungkan satu atau beberapa tipe primitif.

Bahasa M mengenal empat tipe kustom: list, record, tabel, dan fungsi. Tipe-tipe ini dapat dibangun menggunakan ekspresi tipe yang memungkinkan Anda mengklasifikasikan struktur yang lebih kompleks. Untuk tipe kustom, Anda dapat memikirkan tipe untuk:

- *List*: Mengklasifikasikan tipe data elemen dalam list.
- *Record*: Menentukan nama dan tipe data untuk setiap bidang dalam record.
- *Tables*: Menentukan nama dan tipe data untuk setiap kolom dalam tabel.
- *Functions*: Menentukan nama dan tipe data parameter fungsi, serta menentukan tipe data nilai pengembalian fungsi.

Tipe kustom ini dibangun di atas tipe primitif dan abstrak tetapi menambahkan batasan pada strukturnya. Meskipun tipe primitif penting, tipe kustom juga banyak digunakan di Power Query. Sama seperti tipe primitif yang mengklasifikasikan nilai primitif, tipe kustom menambahkan lapisan klasifikasi tambahan untuk nilai yang lebih kompleks. Untuk mengilustrasikan penerapan tipe kustom, kita akan menggunakan *Table.AddColumn*. Fungsi ini memungkinkan Anda untuk menetapkan tipe data ke kolom.

Menetapkan tipe seperti mendeklarasikan bahwa suatu nilai sesuai dengan tipe tertentu tanpa melakukan pemeriksaan apa pun. Dalam hal itu, bahasa M tidak memverifikasi apakah nilai dalam kolom cocok dengan tipe yang ditetapkan. Ini berarti Anda mungkin secara tidak sengaja menetapkan tipe yang tidak sesuai dengan nilai dasar dalam kolom. Untuk saat ini, cukup menyadari perilaku ini; kita akan mempelajari konsep penetapan nanti dalam bab ini saat membahas penetapan tipe.

Mari kita pelajari lebih dalam setiap tipe kustom, dimulai dengan list tipe.

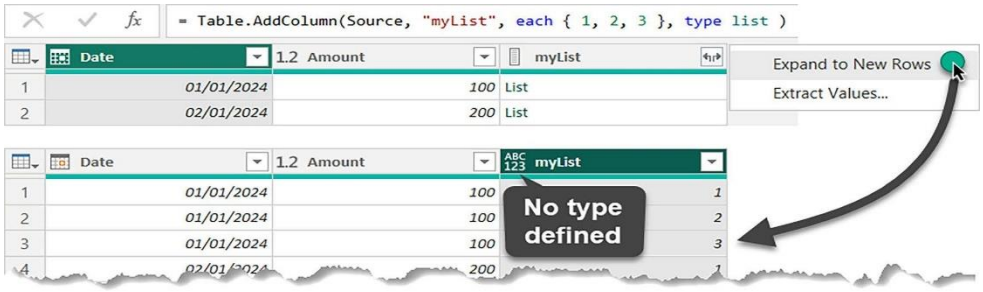
Tipe List

Tipe list kustom dapat digunakan untuk mengklasifikasikan nilai list. Dengan menggunakan tipe list kustom, Anda dapat menentukan nilai bertipe list tetapi juga menyertakan tipe data dari nilai yang dikandungnya. Mari kita lihat cara kerjanya.

Nilai list adalah kumpulan item yang dapat berupa tipe data apa pun. Anda dapat membuat nilai tersebut menggunakan tanda kurung kurawal, seperti {1, 2, 3}, list yang hanya berisi angka. Membuat kolom dan menyetel tipe datanya ke tipe list tidak akan menentukan tipe nilai yang terkandung di dalamnya.

Misalnya, gambar berikut menunjukkan cara membuat kolom baru yang berisi list angka. Argumen keempat dari *Table.AddColumn* memungkinkan Anda untuk menetapkan tipe data kolom; dalam kasus ini, kita menggunakan tipe primitif *list*.

Nah, di sinilah hal-hal menjadi menarik: ketika Anda memperluas kolom *mylist* dengan ikon di header dan memilih **Expand to New Rows**, baris baru tidak memiliki tipe data tertentu yang ditentukan:



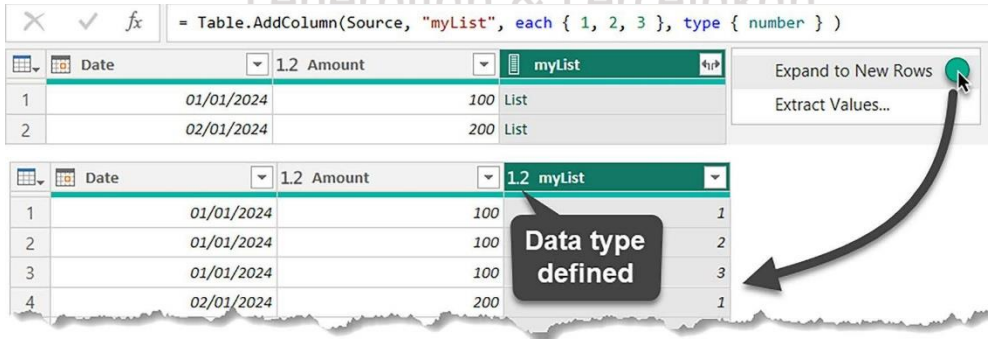
Gambar 1. 16 Tipe hilang saat memperluas kolom dengan tipe list primitif

Hal ini karena list tipe bersifat generik dan tidak menentukan tipe nilai yang dikandungnya. Namun, Anda dapat mengklasifikasikan list dengan tipe kustom yang lebih spesifik untuk menunjukkan bahwa semua elemennya sesuai dengan tipe data tertentu.

Misalnya, untuk mengklasifikasikan tipe yang berbeda, Anda dapat menggunakan:

```
type { text }     // Returns a list type containing text values
type { number }   // Returns a list type containing number values
```

Untuk menentukan tipe kustom ini, Anda memulai dengan kata tipe, diikuti oleh tipe dalam tanda kurung kurawal. Menerapkan tipe yang lebih kompleks ini, yang juga mengklasifikasikan nilai dalam list, memastikan Power Query dapat menghasilkan tabel dengan tipe yang relevan:



Gambar 1. 17 Tipe dipertahankan saat memperluas kolom dengan tipe list kustom

Dengan cara ini, tipe kustom memungkinkan Anda untuk mengatur tipe data dan menjaganya tetap utuh saat mengembangkannya. Seperti yang ditunjukkan contoh sebelumnya, tajuk kolom tidak membedakan antara list tipe primitif dan list tipe kustom; ikon listnya identik.

Jika Anda berurusan dengan tipe list kustom dan ingin menguji jenis nilai yang diterima list, Anda dapat menggunakan fungsi *Type.ListItem*:

```
Type.ListItem( type { number } ) // Output: type number
```

Hal ini memungkinkan Anda untuk mengekstrak nilai tipe yang sedang Anda hadapi. Seperti tipe list, tipe record juga merupakan tipe kustom yang memungkinkan Anda untuk mendeskripsikan record, yang akan kita bahas selanjutnya.

Tipe *Record*

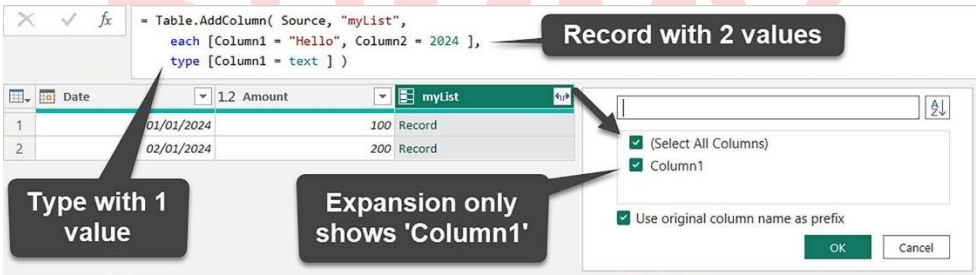
Record di Power Query M seperti baris dalam tabel. Record terdiri dari satu atau beberapa bidang, masing-masing dengan nama dan tipe data yang sesuai. Tipe *record*, mirip dengan tipe *list*, secara umum mengklasifikasikan nilai tetapi tidak memberikan informasi apa pun tentang tipe yang dapat dimiliki nilai. Untuk menjelaskan tipe nilai apa yang diharapkan untuk setiap nama bidang, Anda dapat menggunakan tipe record kustom. Sintaks untuk menentukan record mungkin terlihat seperti ini:

```
type record       // primitive type

type [ Date = date ] // custom type
```

Tipe kustom menyertakan nama bidang dan tipe dalam tanda kurung siku, sedangkan tipe primitif tidak menyediakan detail seperti itu. Jadi, bagaimana kita dapat menggunakannya? Misalkan Anda ingin membuat tipe record dengan dua bidang: Kolom1 dengan nilai teks dan Kolom2 dengan nilai numerik. Anda dapat merepresentasikannya sebagai:

```
type [ Column1 = text, Column2 = number ]
```

Saat Anda menambahkan kolom baru dengan nilai record dan mengembangkannya, tipe data hanya dipertahankan jika ditetapkan secara eksplisit dalam tipe record kustom. Selain itu, jika Anda memiliki kolom dalam record yang tidak ditentukan dalam definisi tipenya, operasi Perluas Record tidak menyarankan kolom yang hilang ini, seperti yang ditunjukkan pada gambar berikut:



Gambar 1. 18 Bidang yang hilang dalam jenis record kustom tidak ditampilkan saat memperluas kolom

Setiap kali Anda kehilangan kolom saat memperluas kolom record, pastikan untuk memeriksa definisi tipe. Kemungkinan ada beberapa kolom yang hilang. Selain itu, saat membuat tipe *custom record*, tidak perlu menentukan tipe setiap kolom. Misalnya, sangatlah sah untuk menentukan record yang sama dan hanya menyediakan tipe data pada *column1*:

```
type [ Column1 = text, Column2 ]
```

Perlu diingat bahwa saat memperluas kolom record, bidang apa pun yang jenisnya tidak ditentukan akan menerima jenis apa pun.

Sintaksis menarik lain yang dapat Anda gunakan adalah record terbuka. Penanda record terbuka, yang diwakili oleh tiga titik (…), memungkinkan Anda menentukan record yang memungkinkan bidang baru (yang tidak diketahui) di masa mendatang. Misalnya, bayangkan Anda memiliki record dengan rentang nama bidang. Anda hanya ingin menentukan jenis Tanggal dan Jumlah; yang lainnya dapat bertipe apa pun. Anda dapat mencapainya dengan menggunakan penanda record terbuka sebagai berikut:

```
type [ Date = type date, Amount = type number, ... ]
```

Tipe record kustom ini menetapkan ekspektasi bahwa record tersebut mungkin atau mungkin tidak berisi lebih banyak nilai daripada yang dijelaskan. Definisi record tipe primitif setara dengan record terbuka. Oleh karena itu, pernyataan berikut mengembalikan *true*:

```
type [...] = type record // returns true
```

Ini menunjukkan bahwa record tipe dianggap sebagai record terbuka. Setelah melihat **list list** dan **tipe record**, mari beralih ke salah satu tipe kustom yang paling penting: **tipe tabel**. Karena sebagian besar transformasi di Power Query berputar di sekitar tabel, Anda biasanya akan menemukan tipe kustom ini. Mari kita lihat cara kerjanya di bagian berikutnya.

Tipe Tabel (*Type Table*)

Tipe umum lainnya yang akan Anda temukan adalah tipe tabel. Anda dapat menganggap tabel sebagai list record dengan struktur yang ditentukan. Setiap kolom dalam tabel memiliki tipe data yang ditentukan, dan semua record (baris) dalam tabel sesuai dengan struktur ini. Tipe tabel adalah tipe dasar dari semua tabel. Tipe ini ditentukan sebagai:

```
type table
```

Ini menunjukkan tipe baris dengan record terbuka yang kosong, identik dengan:

```
type table []
```

Ini juga satu-satunya waktu di mana tipe tabel diizinkan memiliki tipe baris terbuka. Tipe tabel kustom harus selalu memiliki bentuk record tertutup.

Definisi *type table* kustom mirip dengan record tipe. Skenario umum di mana Anda akan menemukannya digunakan adalah saat menggunakan fitur **Group By**. Saat melakukan operasi tersebut, mesin M secara otomatis

menghasilkan tipe tabel kustom untuk menentukan nama dan tipe kolom tabel yang relevan. Contoh dasar tipe tabel dengan kolom teks dan angka adalah:

```
type table [ Column1 = text, Column2 = number ]
```

Tipe ini menunjukkan semua baris dalam tabel sesuai dengan nilai-nilai ini, di mana *Column1* selalu berisi data teks, dan *Column2* berisi data numerik. Jika Anda mau, Anda bahkan dapat menggunakan tipe record kustom dan menggunakannya untuk menyediakan informasi untuk tipe tabel Anda:

```
let
  myRecordType = type [ Column1 = number, Column2 = text ],
  myTableType = type table myRecordType
in
  myTableType
```

Kode tersebut mengembalikan tipe tabel, yang, sebagai nilai tersendiri, tidak sering digunakan. Tipe tabel sering ditemukan dalam fungsi tabel untuk menentukan detail kolom dan tipe data tabel. Contoh ini menggambarkan bahwa tipe tabel dapat menyertakan tipe record dalam definisinya. Namun, ada perbedaan yang tidak kentara antara tipe record dan tipe tabel. Agar valid, tipe record yang digunakan dalam mendefinisikan tipe tabel kustom harus bertipe tertutup. Ini berarti tipe tersebut tidak dapat memiliki kolom opsional; semua kolom dalam record harus didefinisikan secara eksplisit dan diperlukan agar tipe tabel dikenali dengan benar. Akan berguna jika dapat menyediakan tipe tabel terbuka sehingga Anda dapat menentukan tipe kolom tertentu dan menunjukkan bahwa Anda tidak yakin tentang tipe kolom lainnya. Sayangnya, ini tidak diperbolehkan.

Sejalan dengan apa yang diilustrasikan untuk tipe list dan tipe record, saat Anda menambahkan tipe tabel kustom ke kolom dan tidak menyertakan kolom tertentu, kolom tersebut tidak akan muncul di kotak dialog **Perluas Kolom**

(Expand Columns). Kita akan mempelajari lebih lanjut tentang nilai terstruktur di Bab 6.

Sesuatu yang unik pada tipe tabel adalah bahwa tipe tersebut dapat menyertakan definisi kunci tabel. Power Query menggunakan informasi kunci untuk meningkatkan kinerja fungsionalitas, seperti operasi penggabungan. Anda dapat menggunakan Type untuk menetapkan kunci pada fungsi *type.AddTableKey* tabel. Misalnya, kode berikut menetapkan *Column1* sebagai kunci utama untuk tipe tabel:

```
let
  myType = type table [ Column1 = text, Column2 = number ],
  setKeys = Type.AddTableKey( myType, {"Column1"}, true )
in
  setKeys
```

Output dari kode ini adalah tipe tabel dengan informasi kunci. Sekali lagi, tipe ini digunakan untuk mengklasifikasikan tabel, yang kemudian dapat digunakan untuk operasi berikutnya. Setelah kunci ditetapkan, Anda dapat mengambil yang sudah ada menggunakan fungsi *Type.TableKeys* atau menggantinya menggunakan *Type.ReplaceTableKeys*. Kita akan mempelajari lebih lanjut tipe tabel kustom di Bab 6, Nilai Terstruktur. Setelah membahas sebagian besar tipe kustom, selanjutnya adalah yang terakhir: ***type function***.

Tipe Fungsi (Type Function)

Kita juga dapat menentukan tipe fungsi kustom. Anda dapat menetapkan fungsi dengan tipe primitif, dan ekspresi apa pun akan berfungsi. Namun, ini tidak memberlakukan batasan apa pun pada tipe parameter input atau tipe yang dapat dikembalikan oleh suatu fungsi. Menentukan tipe fungsi kustom memungkinkan Anda mengatasi batasan ini.

Tipe fungsi terdiri dari tipe pengembalian dan list yang berisi nol atau lebih parameter fungsi. Tipe fungsi kompatibel dengan tipe fungsi lain jika tipe pengembaliannya kompatibel, dan tipe parameternya identik.

Menentukan **tipe fungsi kustom** dapat dilakukan dengan sintaksis seperti berikut:

```
type function ( date as date, days as number) as date
```

Yang menarik adalah Anda dapat menentukan **nilai fungsi (function valur)**, tanpa menentukan tipe:

```
( date, days ) => Date.AddDays( date, days )
```

Dalam kasus tersebut, nilai secara otomatis menerima **jenis fungsi kustom**:

```
type function ( date as any, days as any ) as any
```

Hal ini menunjukkan bahwa pernyataan tipe bersifat opsional saat menentukan nilai fungsi. Namun, saat menentukan tipe fungsi kustom, Anda diharuskan menyertakan pernyataan tipe. Pernyataan berikut menghasilkan kesalahan sintaksis:

```
type function ( date, days ) as date
```

Saat menentukan tipe data untuk parameter dalam fungsi kustom, Anda hanya dapat menggunakan tipe primitif. Memberikan tipe kustom akan mengakibatkan kesalahan. Kita akan membahas lebih lanjut tentang fungsi kustom dan sintaksisnya di Bab 9, Parameter dan Fungsi Kustom.

Untuk kasus saat Anda perlu menerapkan tipe fungsi kustom ke suatu fungsi, Anda hanya dapat melakukannya menggunakan tipe yang ditetapkan menggunakan fungsi *Value.ReplaceType*.

Sejauh ini dalam bab ini, kita telah melihat tipe data yang tersedia dan cara menentukannya menggunakan tipe primitif dan kustom. Itu sering kali merupakan proses yang membosankan yang memerlukan pekerjaan manual. Namun, Anda tidak selalu harus mengidentifikasi tipe data Anda secara manual. Power Query memiliki metode berbeda untuk melakukannya secara otomatis, yang akan kita bahas selanjutnya.

E. Deteksi Tipe (*Type Detection*)

Interaksi pertama Anda dengan tipe data di Power Query kemungkinan terjadi saat Anda mengimpor data. Data Anda sering kali berbentuk tabel, dan kolom untuk tabel tersebut dapat memiliki tipe data. Saat menggunakan pengaturan default, Power Query secara otomatis mengenali tipe data dari tabel mana pun yang Anda impor. Namun, cara Power Query mendeteksi tipe data dari nilai dalam kumpulan data Anda akan bergantung pada sumber data yang digunakan dan pengaturan Power Query Anda. Ada dua metode yang tersedia:

- Mengambil tipe data dari sumber data
- Mendeteksi tipe secara otomatis

Bagian berikut membahas cara kerja kedua metode ini.

1. Mengambil Tipe Data Dari Sumber Data

Saat Anda mengimpor data dari sumber data terstruktur (misalnya, SQL, Oracle, Azure Data Lake, umpan OData, dsb.), Power Query secara otomatis mengakses skema tabel dari sumber-sumber ini. Skema tabel ini hanya tersedia di sumber data terstruktur dan menentukan struktur tabel dalam database. Skema ini berisi informasi tentang berbagai tipe data yang ditemukan di setiap kolom.

Jika sistem sumber berisi informasi tentang tipe data kolom yang diimpor, Power Query menetapkan tipe-tipe tersebut sesuai dengan definisi ini. Namun, mesin tidak dapat mengandalkan skema apa pun jika Anda mengimpor data dari sumber data tidak terstruktur. Alasannya adalah karena sumber data tersebut tidak disertai skema tabel. Dalam kasus tersebut, mesin menggunakan pendekatan lain, yang akan kita bahas selanjutnya.

2. Mendeteksi Tipe Secara Otomatis

Power Query memiliki fitur yang dirancang untuk mengidentifikasi tipe data kolom dalam set data Anda secara otomatis. Fitur ini sangat berguna saat menangani sumber data tak terstruktur seperti file Excel, CSV, atau file teks,

yang tidak memiliki skema yang telah ditetapkan sebelumnya untuk memandu tipe data untuk berbagai kolom. Fitur ini bekerja dengan memeriksa 200 baris pertama set data Anda untuk menentukan tipe data yang paling sesuai untuk setiap kolom.

Misalkan Anda bekerja dengan tabel tanpa tipe data tertentu, seperti yang ditunjukkan pada gambar berikut:

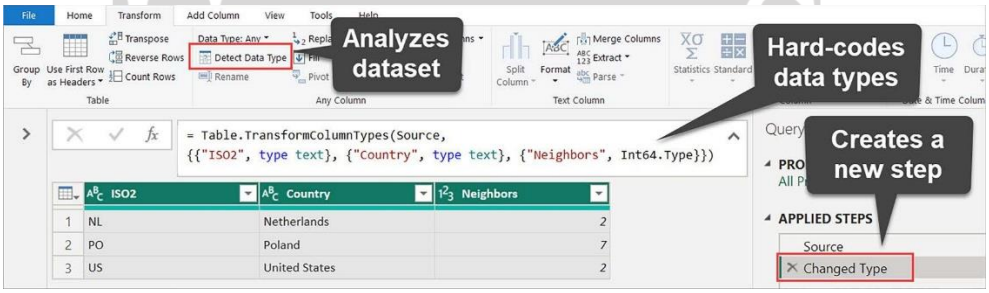
	ABC 123 ISO2	ABC 123 Country	ABC 123 Neighbors
1	NL	Netherlands	2
2	PO	Poland	7
3	US	United States	2

Gambar 1. 19 Tabel tanpa informasi tipe apa pun

Anda dapat menggunakan operasi **Deteksi Tipe Data** untuk menetapkan tipe data secara otomatis ke kolom-kolom ini. Ini melibatkan langkah-langkah berikut:

1) Memilih semua kolom di tabel Anda.
2) Menavigasi ke tab **Transform**.
3) Mengklik **Detect Data Type**.

Power Query kemudian menganalisis data Anda dan membuat tebakan cerdas tentang tipe data yang sesuai, mengonversi kolom-kolom yang sesuai. Proses ini divisualisasikan dalam gambar berikut:

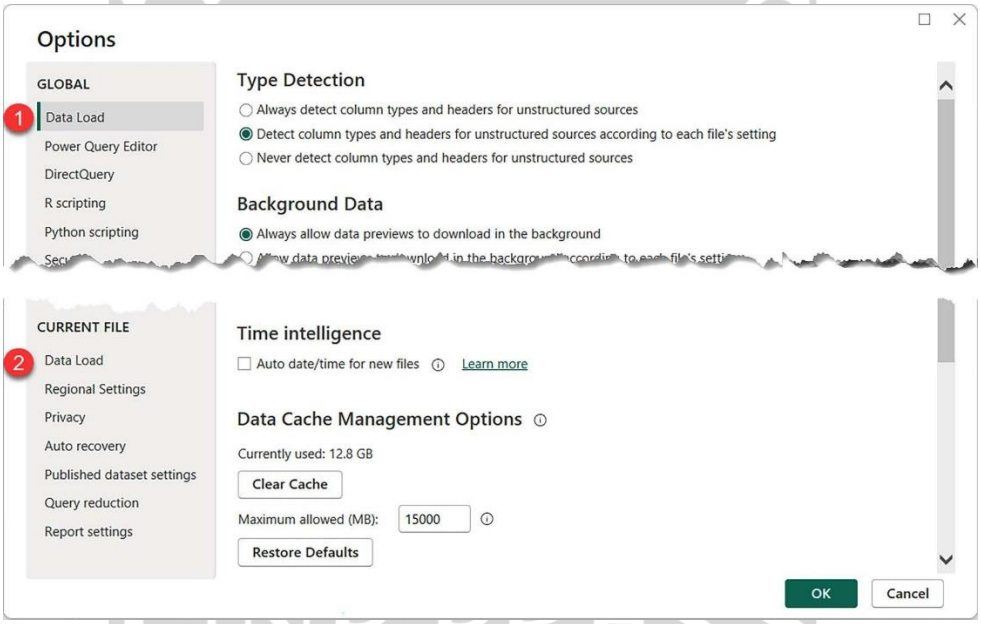


Gambar 1. 20 Operasi Deteksi Tipe Data mengkodekan tipe data secara permanen pada langkah baru

Secara default, Power Query menjalankan operasi **Deteksi Tipe Data** saat Anda terhubung ke sumber data tanpa skema terstruktur. Namun, Anda memiliki opsi untuk mengubah pengaturan ini di pengaturan Power Query. Ini dapat dilakukan dengan melakukan hal berikut:

1. Masuk ke menu **File**.
2. Pilih **Options** dan **settings**.
3. Klik **Options**.

Di jendela pengaturan, Anda dapat memilih cara data Anda dimuat di bagian **Data Load** pada opsi **Global**. Proses pemilihan ini diilustrasikan dalam gambar berikut:



Gambar 1. 21 Menu Opsi Power Query memungkinkan Anda untuk mengatur preferensi Detection Type

Dalam hal pengaturan untuk menangani **Detection Type**, Power Query menawarkan tiga opsi:

- **Always**: Ini memeriksa 200 baris pertama setiap kolom di setiap file. Power Query kemudian menggunakan algoritme pengenalan pola untuk menyimpulkan tipe data yang paling mungkin dari setiap kolom.

- **According to each file's setting**: Penggunaan deteksi tipe otomatis bergantung pada pilihan pengguna dalam pengaturan **Current File Data Load**.
- **Never**: Ini mencegah Power Query memasukkan langkah tipe yang diubah secara otomatis untuk mengidentifikasi tipe kolom.

Bahkan jika deteksi tipe otomatis dinonaktifkan, baik dalam pengaturan laporan global maupun individual, pengguna masih dapat menerapkan fitur **Detect Data Type** secara manual. Ini dilakukan dengan memilih kolom yang diinginkan, menavigasi ke tab **Transform**, dan memilih **Detect Data Type** dari pita Power Query.

Saat memutuskan pengaturan mana yang akan digunakan di Power Query, pertimbangkan kebutuhan dan keadaan spesifik Anda. Meskipun fitur deteksi otomatis umumnya efektif dalam memprediksi jenis kolom, fitur ini hanya mendasarkan analisisnya pada 200 baris pertama. Oleh karena itu, baik Anda mengaktifkan deteksi otomatis di tingkat global maupun tingkat laporan, meninjau setiap jenis kolom secara menyeluruh selalu merupakan ide yang bagus. Memverifikasi atau menyesuaikan jenis data yang terdeteksi menggunakan pemahaman Anda tentang data merupakan praktik terbaik untuk memastikan keakuratan.

Sejauh ini, kita telah melihat berbagai tipe data dalam bahasa M dan mengapa tipe-tipe tersebut penting sebagai sebuah konsep. Yang belum kita lihat adalah bahwa nilai yang berbeda mendukung operasi yang berbeda. Oleh karena itu, Anda sering kali perlu mengonversi suatu nilai menjadi tipe nilai lain agar dapat bekerja dengannya. Itulah yang akan kita bahas selanjutnya.

F. Mendeteksi Tipe Secara Otomatis

Baik Anda mengimpor atau membuat data sendiri, pengaturan tipe data yang benar sangatlah penting. Terkadang, Anda perlu mengubah tipe data untuk

sementara waktu agar semuanya berfungsi—seperti memastikan sepotong data diterima oleh fungsi tertentu, atau saat Anda mencampur berbagai tipe data.

Misalnya Anda memiliki angka yang telah dimasukkan sebagai teks. Untuk melakukan perhitungan matematika dengannya, Anda perlu mengubahnya dari teks ke format numerik. Atau, jika Anda mencoba menambahkan tanggal ke pesan teks, Anda harus mengubah tanggal ke format teks terlebih dahulu. Jenis perubahan ini sering kali diperlukan saat data Anda disimpan dalam tabel.

Untuk situasi ini, bahasa M memiliki dua cara utama untuk mengonversi nilai Anda ke tipe lain. Anda dapat:

- Membuat kolom baru dan menggunakan fungsi konversi dalam ekspresi
- Mengubah seluruh kolom ke tipe lain

Kita akan melihat kedua pendekatan di bagian berikut.

1. Mengonversi Tipe Nilai

Pertama, mari kita lihat serangkaian fungsi yang memungkinkan Anda mengonversi nilai dari satu tipe ke tipe lain. Fungsi-fungsi tersebut terbagi dalam dua kategori:

- **Fungsi ekstraksi tipe**: Fungsi-fungsi ini mengekstrak tipe nilai tertentu dari input yang lebih umum, yang bisa berupa tipe apa pun. Tujuannya adalah untuk mengidentifikasi tipe tertentu dari tipe data yang mungkin tercampur atau tidak terdefinisi. Contoh umum fungsi-fungsi ini meliputi Date.From, Text.From, dan Number.From. Fungsi-fungsi tersebut tersedia untuk berbagai tipe data, seperti *binary, date, duration, function, logical, number, time, datetime, datetimezone,* dan *text*.
- **Fungsi konversi tipe**: Kategori ini berisi fungsi-fungsi yang mengonversi data antara tipe-tipe tertentu. Fungsi-fungsi tersebut mampu menangani konversi di kedua arah. Misalnya, *Date.ToText* dan *Binary.ToText* mengubah data tanggal dan biner menjadi format teks. Di sisi lain, *Date.FromText* dan *Binary.FromText* membalikkan konversi ini, mengubah teks menjadi nilai tanggal dan teks menjadi data biner.

Fungsi-fungsi yang mengonversi atau mengekstrak tipe biasanya digunakan selama evaluasi suatu ekspresi. Misalnya, ekspresi berikut mengekstrak nilai tanggal dan angka:

```
Date.From( "31 dec 2024" ) // Output: #date( 2024, 12, 31 )
Date.From( 45657 )         // Output: #date( 2024, 12, 31 )
Number.From( "550" )       // Output: 550
Number.From( true )        // Output: 1
```

Seperti yang dapat Anda lihat, fungsi ekstraksi memiliki tingkat fleksibilitas dalam nilai yang diizinkan. Di sisi lain, fungsi konversi memerlukan tipe yang lebih spesifik. Misalnya:

```
Number.ToText( 125 )              // Output: "125"

Record.ToList( [A = 1, B = 2, C = 3] ) // Output: {1, 2, 3 }
```

Saat Anda bekerja dengan tabel, penggunaan fungsi konversi dalam ekspresi memiliki dua manfaat:

- **Mempertahankan kolom asli**: Struktur asli tabel dapat tetap tidak berubah, mempertahankan tipe data aslinya.
- **Mencegah kekacauan**: Tidak diperlukan langkah tambahan untuk mengubah tipe data, mencegah kueri Anda dari kekacauan.

Untuk latihan berikutnya, kita akan mulai dengan tabel berikut:

	Date	Product Name	Price
1	01/01/2024	Chef's Knife	45.00
2	02/01/2024	Cast Iron Skillet	30.50
3	03/01/2024	Cutting Board	15.75
4	04/01/2024	Digital Kitchen Scale	22.00
5	05/01/2024	Dutch Oven	55.99
6	06/01/2024	Silicone Spatula Set	10.25
7	07/01/2024	Stainless Steel Cookware Set	120.00
8	08/01/2024	Programmable Slow Cooker	75.00

Gambar 1. 22 Tabel yang berisi tanggal, teks, dan nilai angka

Untuk mengikutinya, Anda dapat mengunduh berkas latihan di halaman GitHub buku tersebut (https://github.com/PacktPublishing/The-Definitive-Guide-to-Power-Query-M-). Kumpulan data sebelumnya memiliki tiga kolom dengan tipe data yang berbeda: *Date*, *Text*, dan *Number*. Tujuan Anda adalah menggabungkan informasi dari kolom-kolom ini menjadi satu kalimat yang koheren. Namun, dalam bahasa M, Anda harus terlebih dahulu mengonversi nilai menjadi teks sebelum menggabungkannya menjadi kalimat. Di sinilah fungsi konversi berperan. Misalnya Anda ingin membuat kalimat seperti:

```
"On [Date] we sold [Price] worth of [Product Name]s."
```

Untuk menggabungkan nilai-nilai yang berbeda, Anda harus memastikan bahwa nilai *Date* dan *Price* diubah menjadi teks. Fungsi *Text.From* sangat cocok untuk tugas ini. Fungsi ini mengubah nilai Tanggal dan Harga menjadi teks, menyediakan format yang benar untuk penggabungan.

Setelah mengubah nilai-nilai tersebut, Anda dapat menggunakan operator penggabungan (&) untuk menyusun kalimat kustom Anda. Kode berikut akan mencapainya:

```
"On "
  & Text.From( [Date] )
```

```
  & " we sold "
  & Text.From( [Price] )
  & " worth of "
  & [Product Name]
  & "s"
```

Jika Anda menambahkan ini ke kolom **Custom**, Anda akan mengalami situasi berikut:

```
= Table.AddColumn(#"Changed Type", "Custom",
each "On " & Text.From( [Date] ) & " we sold " & Text.From( [Price] ) & " worth of "& [Product Name] & "s"  )
```

	Date	Product Name	Price	Custom
1	01/01/2024	Chef's Knife	45.00	On 01/01/2024 we sold 45 worth of Chef's Knifes
2	02/01/2024	Cast Iron Skillet	30.50	On 02/01/2024 we sold 30.5 worth of Cast Iron Skillets
3	03/01/2024	Cutting Board	15.75	On 03/01/2024 we sold 15.75 worth of Cutting Boards
4	04/01/2024	Digital Kitchen Scale	22.00	On 04/01/2024 we sold 22 worth of Digital Kitchen Scales
5	05/01/2024	Dutch Oven	55.99	On 05/01/2024 we sold 55.99 worth of Dutch Ovens
6	06/01/2024	Silicone Spatula Set	10.25	On 06/01/2024 we sold 10.25 worth of Silicone Spatula Sets
7	07/01/2024	Stainless Steel Cookware Set	120.00	On 07/01/2024 we sold 120 worth of Stainless Steel Cookware Sets
8	08/01/2024	Programmable Slow Cooker	75.00	On 08/01/2024 we sold 75 worth of Programmable Slow Cookers

Gambar 1. 23 Fungsi konversi memungkinkan Anda mengubah nilai menjadi teks sebelum menggabungkannya

Contoh ini mengilustrasikan cara Anda dapat menggunakan fungsi konversi dalam kolom kustom untuk membuat kalimat kustom. Dengan menggunakan fungsi konversi, tipe kolom tabel tetap utuh untuk operasi apa pun di masa mendatang.

Contoh sebelumnya menunjukkan konversi yang berhasil dari berbagai tipe data ke teks. Namun, bahasa M tidak mendukung konversi tipe data menjadi sembarang tipe. Beberapa kombinasi menghasilkan kesalahan. Matriks berikut menunjukkan konversi mana yang didukung. Nilai di sisi kiri menunjukkan nilai saat ini dan ikon menunjukkan apakah Anda dapat mengubah nilai menjadi tipe yang ditunjukkan di baris atas:

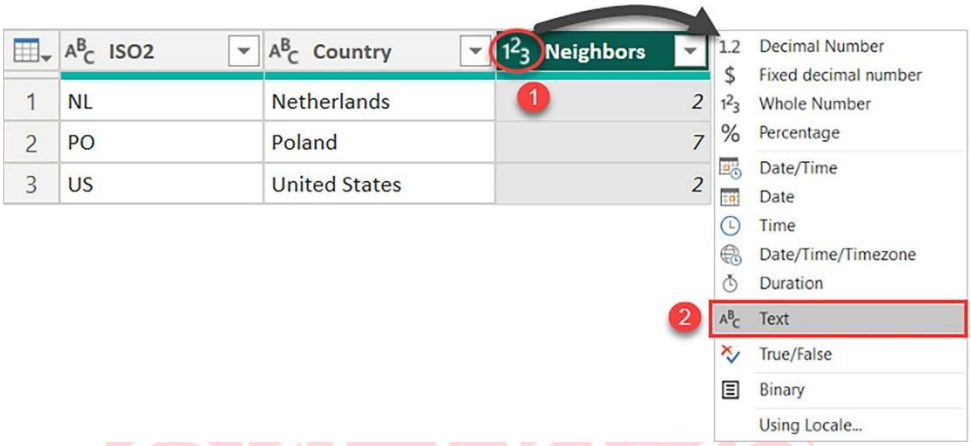
Gambar 1. 24 Matriks konversi tipe data

Setelah mempelajari bagaimana fungsi konversi memungkinkan Anda mengubah nilai tanpa mengubah tipe kolom asli, mari beralih ke pendekatan lain. Bagian berikutnya membahas transformasi tipe kolom, dan dengan itu, semua nilai kolom.

2. Mengonversi Tipe Kolom

Selanjutnya, kita akan membahas proses konversi tipe kolom secara keseluruhan. Tidak seperti konversi nilai individual yang telah kita bahas sebelumnya, metode ini melibatkan perubahan tipe data dari seluruh kolom. Pendekatan ini khususnya berguna ketika Anda ingin memuat tipe data yang sesuai ke dalam host seperti Power BI atau ketika Anda memerlukan hasil transformasi ini untuk beberapa perhitungan lainnya. Jadi, bagaimana cara kerjanya?

Gambar berikut menyajikan kumpulan data kecil dengan kolom **Neighbours** yang diformat sebagai *Whole Number*:



Gambar 1. 25 Ubah kolom ke jenis yang diinginkan dengan memilihnya di tajuk kolom

Misalkan di kolom **Neighbors**, kita ingin nilai bertipe teks, bukan bertipe angka. Cara mudah untuk melakukan perubahan ini adalah dengan mengklik ikon kolom di samping nama kolom dan memilih tipe **Text**. Operasi ini mengubah kolom Anda dari bilangan bulat menjadi tipe teks dengan

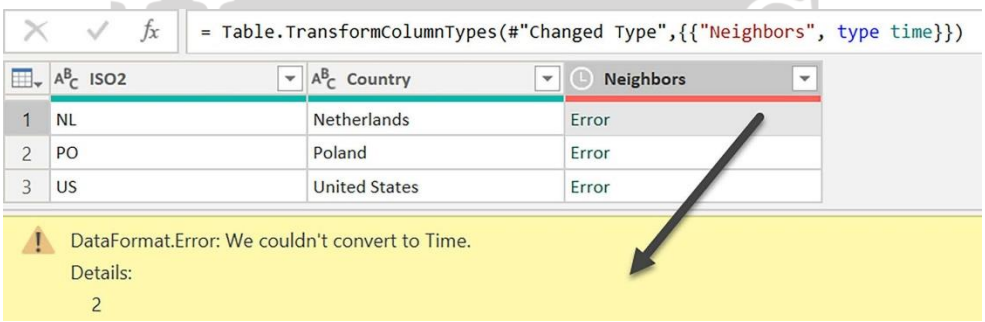
menggunakan fungsi *Table.TransformColumnTypes*. Fungsi ini menggunakan ekspresi berikut:

```
Table.TransformColumnTypes(
  #"Changed Type",

  { {"Neighbors", type text} } )
```

Jadi, bagaimana cara kerja fungsi ini? Table.TransformColumnTypes memerlukan dua argumen. Yang pertama adalah tabel yang ingin Anda ubah, dalam hal ini, *#"Changed Type"*. Argumen kedua adalah list yang setiap elemennya berpasangan: nama kolom dan tipe data baru. Misalnya, *{ {"Neighbors", type text} }* menunjukkan bahwa tipe data kolom *Neighbors* akan diubah menjadi text. Fungsi ini menjalankan dua operasi utama:

1) **Penetapan tipe data**: Fungsi ini menetapkan tipe data baru ke kolom yang ditentukan, seperti yang diarahkan. Header kolom tabel akan mencerminkan tipe data baru ini.
2) **Konversi nilai**: Fungsi ini mencoba mengonversi nilai yang ada di kolom ke tipe data baru.

Menetapkan tipe data kolom selalu berfungsi karena ikon di header akan berubah. Namun, konversi nilai tidak selalu berhasil. Misalnya, jika kita mencoba mengonversi kolom dari bilangan bulat ke tipe data waktu, bukan teks, hasilnya adalah galat berikut:



Gambar 1. 26 Mengonversi nilai ke nilai yang tidak didukung menghasilkan DataFormat.Error

Alasan terjadinya galat ini adalah karena angka tidak dapat diubah menjadi nilai waktu. Anda dapat merujuk ke Gambar 5.24 untuk memeriksa konversi mana yang didukung dalam bahasa M.

Seperti yang ditunjukkan bagian ini, Anda dapat mengonversi tipe kolom, sama seperti Anda dapat mengonversi nilai. Anda telah mempelajari konversi tipe data mana yang didukung dan kapan Anda dapat mengalami galat. Namun, ada satu topik penting yang masih perlu ditambahkan. Meskipun konversi berhasil, beberapa konversi dapat menyebabkan hilangnya data. Bagian berikutnya membahas lebih lanjut kapan hal ini terjadi.

3. Menghindari Kehilangan Data Selama Konversi

Saat Anda mengubah nilai dari satu tipe data ke tipe data lain, penting untuk berhati-hati dan memahami sepenuhnya apa arti perubahan tersebut. Selama konversi tipe, Power Query akan mencoba melakukan konversi jika memungkinkan, tetapi perlu diingat bahwa hal ini dapat menyebabkan hilangnya atau bertambahnya data atau presisi dalam proses tersebut. Proses ini memengaruhi cara data disimpan dan akhirnya diberikan ke aplikasi host seperti Power BI.

Sebaliknya, praktik penerapan string format dalam lapisan visualisasi Power BI Desktop beroperasi secara berbeda. Tidak seperti konversi tipe di Power Query, yang secara langsung memengaruhi penyimpanan data, masker format hanya menyesuaikan penyajian nilai tanpa mengubah penyimpanan data yang mendasarinya. Perbedaan ini membantu dalam memahami bagaimana penyesuaian data memengaruhi backend (penyimpanan) versus frontend (tampilan) di Power BI. Mari kita lihat contoh di mana kehilangan data menjadi masalah.

Pertimbangkan kumpulan data parkir berbayar harian di Seattle, *Wasbington*. Tipe data didefinisikan menurut kamus data yang menyertai kumpulan data tersebut:

	Transaction ID	Meter Code	Transaction Date	Payment Mean	Amount Paid
1	1250162207	12028002	7/11/2023	PHONE	0.26
2	1250162278	19232002	7/11/2023	PHONE	1.52
3	1250131414	19127010	7/11/2023	PHONE	9.67
4	1250134465	5073002	7/11/2023	CREDIT CARD	4
5	1250134860	19161010	7/11/2023	PHONE	6.73
6	1250134876	5019002	7/11/2023	PHONE	0.3
7	1250134941	19126010	7/11/2023	PHONE	8.77

Gambar 1. 27 Tabel dengan kolom Jumlah yang Dibayar berisi angka desimal

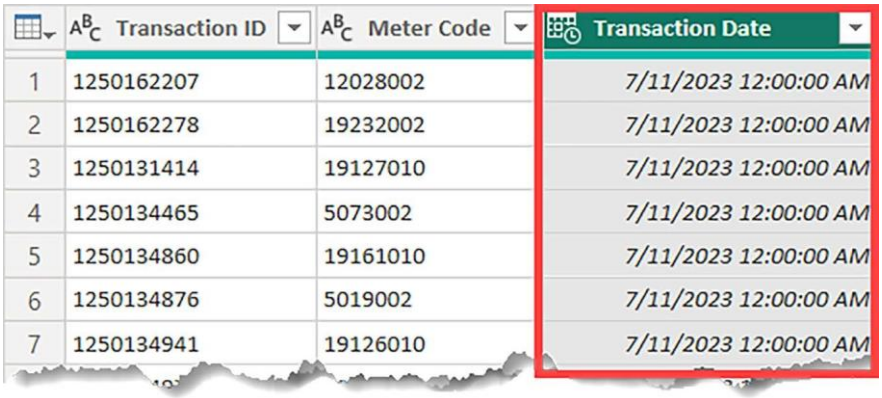
Sekarang, anggaplah kita tertarik untuk melakukan beberapa analisis terkait bagian dolar dari kolom **Amount Paid** (dan untuk saat ini tidak peduli dengan sen yang tersisa). Jika kita mengubah kolom tersebut menjadi angka bulat menggunakan UI, maka akan berfungsi:

	Transaction ID	Meter Code	Transaction Date	Payment Mean	Amount Paid
1	1250162207	12028002	7/11/2023	PHONE	0
2	1250162278	19232002	7/11/2023	PHONE	2
3	1250131414	19127010	7/11/2023	PHONE	10
4	1250134465	5073002	7/11/2023	CREDIT CARD	4
5	1250134860	19161010	7/11/2023	PHONE	7
6	1250134876	5019002	7/11/2023	PHONE	0
7	1250134941	19126010	7/11/2023	PHONE	9

Gambar 1. 28 Tabel dengan kolom Jumlah yang Dibayar diubah menjadi bilangan bulat

Namun, mengembalikan Jumlah yang Dibayar ke tipe desimal untuk menyertakan sen akan memulihkan tipe bidang tetapi kehilangan nilai sen asli. Ini menyoroti poin penting: mengubah tipe data bidang memengaruhi penyimpanannya dan dapat mengurangi ketelitian data. Di sisi lain, mengubah format melalui lapisan visualisasi Power BI hanya mengubah cara data muncul dalam visual, bukan format penyimpanannya.

Dalam beberapa kasus, mengubah tipe data tampaknya dapat meningkatkan detail data, tetapi ini dapat menyesatkan. Misalnya, mengonversi kolom **Transaction Date** ke tipe datetime menambahkan nilai waktu ke tanggal:



	Transaction ID	Meter Code	Transaction Date
1	1250162207	12028002	7/11/2023 12:00:00 AM
2	1250162278	19232002	7/11/2023 12:00:00 AM
3	1250131414	19127010	7/11/2023 12:00:00 AM
4	1250134465	5073002	7/11/2023 12:00:00 AM
5	1250134860	19161010	7/11/2023 12:00:00 AM
6	1250134876	5019002	7/11/2023 12:00:00 AM
7	1250134941	19126010	7/11/2023 12:00:00 AM

Gambar 1. 29 Mengonversi tanggal ke nilai datetime menambahkan waktu default 12 AM

Namun, waktu tambahan tersebut tidak sesuai dengan kolom Waktu Transaksi yang sebenarnya. Sebaliknya, kolom tersebut menambahkan pukul 12.00 secara default, yang menggambarkan bahwa meskipun data mungkin tampak lebih terperinci, data tersebut mungkin tidak akurat.

Contoh-contoh ini menunjukkan pentingnya memahami sepenuhnya implikasi konversi tipe data sebelum Anda menerapkannya. Seperti yang ditunjukkan sebelumnya, Power Query tidak akan menghentikan Anda dari menjalankan konversi yang akan mengakibatkan hilangnya data. Konversi tipe data dapat memiliki satu dari empat hasil yang berbeda:

- **Success**: Perubahan tipe data diterapkan tanpa perolehan atau kehilangan data.
- **Success dengan hilangnya data**: Perubahan tipe data diterapkan tetapi dengan beberapa tingkat kehilangan data.
- **Success dengan perolehan data**: Perubahan tipe data diterapkan dengan beberapa tingkat perolehan data. Data yang diperoleh mungkin benar atau mungkin tidak benar.
- **Failure**: Perubahan tipe data tidak diterapkan karena ketidakcocokan tipe input dan output.

Perlu dicatat bahwa meskipun kehilangan data karena konversi tipe data biasanya merupakan hasil yang harus dihindari, dalam beberapa kasus, hal itu dapat disengaja. Misalnya, mengonversi nilai datetime ke tanggal menghapus komponen waktu tetapi memiliki dampak positif pada kompresi saat memuat data Anda ke Power BI.

Sejauh ini, kita telah melihat bahwa mengonversi nilai dari satu tipe ke tipe lain dapat menyebabkan kehilangan data. Kita juga telah melihat bahwa transformasi hanya didukung antara tipe tertentu. Topik lain yang relevan dengan konversi adalah efek lokal. Bergantung pada konvensi pemformatan lokal, satu nilai dapat ditafsirkan secara berbeda. Itulah yang akan kita bahas di bagian berikutnya.

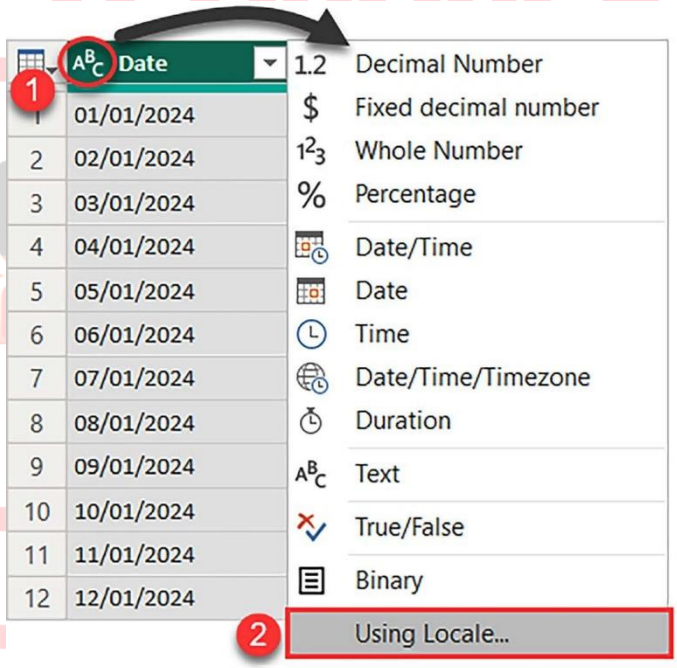
4. Pengaruh Lokal/Budaya

Berbagai wilayah di seluruh dunia mengikuti berbagai konvensi data. Ini termasuk pilihan seperti pemisah mana yang akan digunakan dalam angka (koma atau titik), format untuk tanggal, dan apakah akan menggunakan penanda AM/PM. Konvensi regional ini memiliki pengaruh besar pada bagaimana tipe data diubah karena proses konversi bergantung pada pengenalan format nilai yang mendasarinya. Bagian ini menunjukkan bagaimana pengaturan regional dapat memengaruhi hasil transformasi data Anda.

Bayangkan situasi di mana Anda memiliki kolom nilai teks yang mewakili tanggal dari 1 Januari hingga 12 Januari 2024. Bergantung pada konvensi yang digunakan, tanggal-tanggal ini dapat ditafsirkan dengan berbagai cara. Tanggal-tanggal tersebut dapat dilihat sebagai tanggal dari 1 Januari hingga 12 Januari 2024, atau sebagai hari pertama setiap bulan sepanjang tahun 2024. Pendekatan yang tepat bergantung pada konvensi yang digunakan saat menyediakan data. Power Query mengatasi ambiguitas ini menggunakan pengaturan Lokal. Secara default, pengaturan ini cocok dengan pengaturan bahasa dan regional komputer Anda. Jadi, jika komputer Anda diatur untuk

menginterpretasikan tanggal dalam format *MM/DD/YYYY*, Power Query akan mengikuti format ini saat mengonversi nilai teks ke tanggal.

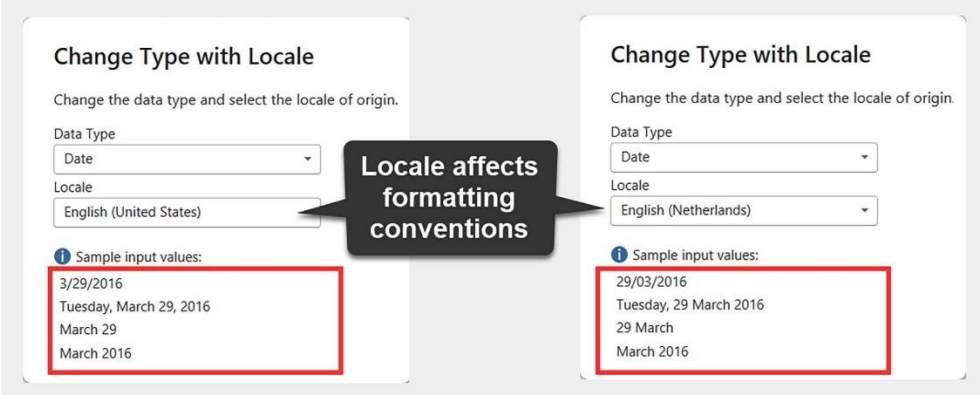
Namun, bagaimana jika Anda perlu mengonversi nilai teks ini ke format tanggal, dan pengaturan Anda berbeda dari pengaturan rekan kerja? Bagaimana Anda memastikan pengenalan tanggal yang konsisten di berbagai sistem? Kuncinya adalah menggunakan pengaturan Lokal untuk memandu transformasi nilai. Jadi, bagaimana cara kerjanya?



Gambar 1. 30 UI memungkinkan Anda mengubah tipe data menggunakan opsi Menggunakan Lokal…

Saat Anda mengklik ikon tipe data yang terletak di tajuk kolom, menu tarik-turun akan muncul. Menu ini menawarkan berbagai opsi untuk *data dtype*. Opsi terakhir dalam menu tarik-turun ini memungkinkan Anda untuk mengubah tipe data sesuai dengan pengaturan **Locate** tertentu, seperti yang diilustrasikan dalam gambar di atas. Memilih opsi ini akan membuka kotak dialog tempat Anda dapat memilih tipe data yang diinginkan dan **Locate**. Fitur ini dirancang untuk mengakomodasi konvensi pemformatan lokal selama proses transformasi tipe data.

Misalnya, saat Anda memilih **Date** sebagai tipe data dalam kotak dialog ini, pratinjau akan ditampilkan. Pratinjau ini memperlihatkan nilai input apa yang diharapkan Power Query menggunakan Lokal yang dipilih. Artinya, negara yang Anda pilih mewakili format data sumber dan bukan format yang Anda inginkan untuk output. Output akan sesuai dengan pengaturan regional Anda saat ini. Pratinjau ini khususnya berguna untuk memverifikasi bahwa data sesuai dengan format yang diharapkan untuk **Locate** yang dipilih. Anda dapat melihat contoh dalam gambar berikut:



Gambar 1. 31 Mengubah tipe menggunakan Lokal menunjukkan pratinjau konvensi pemformatan yang diharapkan dari input

Mengonfirmasi transformasi menggunakan lokal **English (Netherlands)** akan mengubah tanggal dan menggunakan rumus berikut:

```
Table.TransformColumnTypes( Source, {{"Date", type date}}, "en-NL")
```

Dalam fungsi ini, argumen ketiga adalah kode budaya opsional. Lokal en-NL mengharapkan tanggal dalam format *dd/mm/yyyy*.

Jika Anda perlu menggunakan lokal Inggris (Amerika Serikat) sebagai gantinya, mengharapkan *format m/d/yyyy*, kode budaya berubah menjadi en-US:

```
Table.TransformColumnTypes( Source, {{"Date", type date}}, "en-US")
```

Pendekatan ini menawarkan cara yang mudah digunakan untuk mengubah tipe kolom melalui antarmuka. Selain itu, Power Query menyertakan beberapa

fungsi yang menerima kode kultur untuk memandu proses konversi. Fungsi-fungsi ini sering digunakan untuk mengekstrak atau mengonversi tipe data, serta memformat nilai sebagai teks.

Misalnya, penafsiran tanggal bervariasi berdasarkan kode budaya dalam fungsi *Date.From*:

```
Date.From( "01-12-2024", "nl-NL" )// Output: #date(2023,12,1)
Date.From( "01-12-2024", "en-US" )// Output: #date(2023,1,12)
```

Selain itu, fungsi *Text.Upper* menunjukkan perilaku yang berbeda dengan lokal yang berbeda. Saat menggunakan Lokal Turki *tr-TR* dibandingkan dengan lokal Inggris en-US, keluaran kapitalisasi bervariasi:

```
Text.Upper( "i am lucky", "tr-TR" )    // Output: "İ AM LUCKY"
Text.Upper( "i am lucky", "en-US" )    // Output: "I AM LUCKY"
```

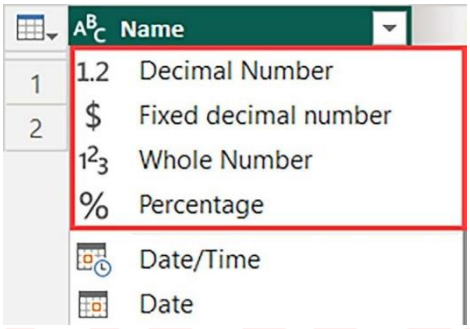
Itu dapat berarti perbedaan besar bagi kebutuhan transformasi Anda.

Kiat profesional: Setiap kali dokumentasi fungsi menyebutkan parameter kultur opsional, ini menunjukkan bahwa Anda dapat menggunakan kode kultur untuk memengaruhi output. Jika parameter kultur tersedia dalam suatu fungsi, kita sarankan untuk menggunakannya. Itu memastikan output Anda dapat diprediksi terlepas dari bahasa mesin pengguna Anda.

Setelah menjelajahi berbagai tipe data, deteksi tipe, dan konversi nilai di Power Query, sekarang kita beralih ke bagian berikutnya. Bagian ini akan memperkenalkan aspek, sebuah konsep yang memungkinkan Anda memberikan detail tambahan tentang tipe data Anda.

G. Aspek

Sejauh ini, kita telah melihat tipe data dasar dalam bahasa M, baik yang primitif maupun yang kustom. Namun, Anda mungkin menemukan nilai yang tampak seperti tipe data, tetapi dengan notasi yang berbeda. Misalnya, saat memilih tipe data baru, Anda akan melihat popup berikut:



Gambar 1. 32 Menu drop-down untuk mengubah jenis kolom

Perhatikan bagaimana menu menampilkan empat cara berbeda untuk memberi label pada nilai angka. Sekarang, misalkan kita mengubah kolom menjadi **Decimal Number** dan kolom menjadi **Fixed decimal number**. Antarmuka pengguna menghasilkan kode berikut:

```
Table.TransformColumnTypes( Source ,
  { { "Name",  type number   },
    { "Value", Currency.Type } } )
```

Transformasi tipe pertama merujuk pada nomor tipe yang sudah dikenal. Akan tetapi, apa sebenarnya Currency.Type itu? Jawabannya adalah ini adalah salah satu Facet Tipe, yang umumnya disebut sebagai facet.

Facet Tipe menyediakan informasi tambahan tentang tipe data Anda. Facet menjadi berguna saat Power Query perlu berkomunikasi dengan sistem lain, seperti Power BI atau aplikasi lain yang menerima data dari Power Query. Meskipun facet tidak mengubah cara kerja Power Query dengan data itu sendiri, facet menyediakan informasi berharga bagi sistem lain.

Misalnya, facet memungkinkan Power Query memberi tahu sistem jenis data yang diharapkan, seperti menggunakan facet Int64.Type untuk mengatakan, "Ini semua adalah bilangan bulat 64-bit." Aplikasi yang menerima data ini dapat menggunakan informasi tipe untuk memesan jumlah ruang yang tepat dan menangani data dengan benar.

Facet juga dapat memberikan petunjuk tentang data, seperti seberapa panjang kolom teks atau seberapa tepat angka. Meskipun Anda dapat menggunakan fungsi pustaka default untuk bekerja dengan informasi faset, Anda tidak akan banyak melihat faset bekerja di dalam Power Query. Faset cenderung hanya digunakan saat menerima data dari atau mentransfer data ke sistem lain. Aplikasi lain ini sering kali memiliki sistem tipe yang lebih kompleks dibandingkan dengan Power Query.

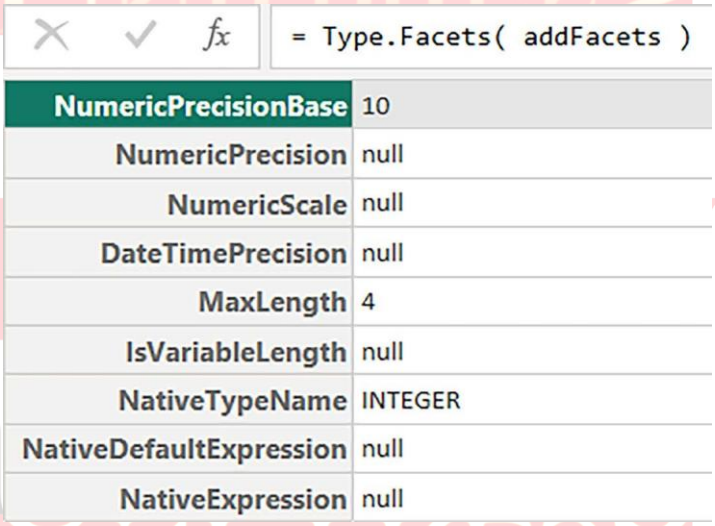
Meskipun banyak aspek tidak penting untuk penggunaan sehari-hari, **Type Claims** merupakan pengecualian. Aspek khusus ini penting, karena sering muncul, bahkan di UI, dan terlihat di banyak fungsi. Bagi sebagian besar pengguna, pemahaman terperinci tentang semua aspek tidak diperlukan untuk menjadi ahli dalam Power Query. Namun, karena Anda kemungkinan akan menemukan **Type Claims** (jenis aspek tertentu), bahkan saat menggunakan UI, kita akan membahas topik tersebut secara singkat di bagian berikutnya.

Bergantung pada cara Anda mengimpor data, tabel Anda akan atau tidak akan memiliki informasi aspek yang terkait dengannya. Jika database SQL Anda berisi informasi aspek dan Anda terhubung ke tabel, Power Query akan mengambil informasi tipe untuk tabel Anda.

Misalnya, anggaplah Anda memiliki tipe tabel dasar. Secara default, tidak akan ada aspek yang terkait dengan nilai tersebut. Anda dapat menambahkan beberapa informasi kustom ke aspek Anda menggunakan fungsi *Type.ReplaceFacets*. Kemudian untuk mengembalikan aspek yang terkait dengan tipe Anda, gunakan fungsi *Type.Facets*:

```
let
  myType = type table,
  myFacets =
    [
      NumericPrecisionBase = 10,
      MaxLength            = 4,
      NativeTypeName       = "INTEGER"
    ],
  addFacets = Type.ReplaceFacets( myType, myFacets ),
  showFacets = Type.Facets( addFacets )
in
  showFacets
```

Ekspresi sebelumnya menyediakan informasi faset untuk *NumericPrecisionBase*, *MaxLength*, dan *NativeTypeName* dan menambahkannya ke tipe tabel. Untuk memeriksa informasi faset, kita dapat menggunakan fungsi *Type.Facets* dan merujuk ke tipe tabel yang menyertakan informasi faset, seperti yang ditunjukkan pada gambar berikut:



Gambar 1. 33 Sisi sederhana yang terkait dengan tipe tabel

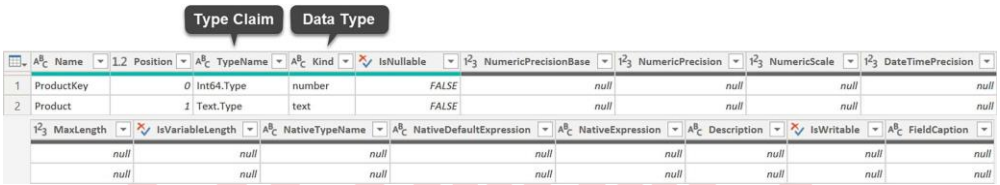
Seperti yang Anda lihat, ada banyak informasi tambahan yang dapat disimpan oleh suatu tipe (data). Facet Tipe yang diperlihatkan pada gambar sebelumnya dianggap sebagai **Simple Facets**. Anda biasanya tidak akan menambahkan sendiri nilai-nilai semacam ini, tetapi ada baiknya untuk mengetahui bahwa menggunakannya dalam ekspresi Anda sangatlah sah. Ingatlah bahwa aplikasi eksternal yang bekerja dengan Power Query akan sering

menggunakan informasi ini untuk membantu meningkatkan penyimpanan dan penanganan data.

Contoh sebelumnya memberi kita beberapa informasi tentang tipe tabel umum, tetapi Anda mungkin ingin mengambil informasi ini untuk setiap kolom dalam tabel Anda. Untuk melakukannya, Anda dapat menggunakan fungsi *Table.Schema*. Contoh berikut membuat nilai tabel lalu mengembalikan informasi facetnya:

```
let
  myTable = #table(
     type table[ ProductKey = Int64.Type, Product = Text.Type ],
     {{ 1, "Apple" },{ 2, "Prume" }} ),
  tableFacets = Table.Schema( myTable)
```

Hasil ekspresi ini adalah tabel yang menyediakan informasi skema untuk setiap kolom, seperti yang ditunjukkan berikut ini:



	Name	Position	TypeName	Kind	IsNullable	NumericPrecisionBase	NumericPrecision	NumericScale	DateTimePrecision
1	ProductKey	0	Int64.Type	number	FALSE	null	null	null	null
2	Product	1	Text.Type	text	FALSE	null	null	null	null

MaxLength	IsVariableLength	NativeTypeName	NativeDefaultExpression	NativeExpression	Description	IsWritable	FieldCaption
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null

Gambar 1. 34 Ketik informasi Facet untuk kolom yang dikembalikan oleh Table.Schema

Anda dapat menemukan *Type Claim* dan *Data Type* sebagai kolom ketiga dan keempat. Tepat setelah kedua kolom ini, Anda dapat menemukan informasi aspek lainnya.

Cara alternatif untuk menampilkan informasi yang sama adalah dengan mengambilnya dari tipe tabel, bukan dari tabel itu sendiri. Anda dapat melakukannya menggunakan fungsi *Type.TableSchema* sebagai berikut:

```
let
  myTableType =
    type table[ ProductKey = Int64.Type, Product = Text.Type ],
  typeFacets = Type.TableSchema( myTableType)
in
  typeFacets
```

Hasil yang diperoleh dengan menggunakan potongan kode ini identik dengan gambar sebelumnya. Perbedaan utamanya adalah *Table.Schema* diterapkan pada tabel, sedangkan *Type.TableSchema* dapat digunakan pada tipe tabel.

Seperti yang telah Anda pelajari sejauh ini, faset adalah cara Power Query menyimpan informasi tipe tambahan yang diterima dari sistem eksternal. Power Query kemudian dapat meneruskan informasi ini ke sistem eksternal yang menyediakan data. Sekarang faset Sederhana (yang dikembalikan oleh *Type.Facets*), sering kali spesifik sistem dan tidak memungkinkan pertukaran yang mudah antar aplikasi.

Namun, beberapa faset lebih umum dan dapat digunakan untuk mengomunikasikan informasi tipe antar sistem. Ini adalah Klaim Tipe. Klaim Tipe adalah faset yang kemungkinan besar akan Anda gunakan. Karena alasan itu, bagian berikutnya berfokus pada nilai-nilai ini.

1. Type Claims

Sebelumnya dalam bab ini, kita telah menjelajahi berbagai tipe data primitif dan kustom yang tersedia dalam bahasa M, seperti tipe date untuk tanggal dan tipe logika untuk nilai Boolean. Sementara mesin mashup Power Query terutama menggunakan tipe data standar ini, aplikasi eksternal dapat bekerja dengan klasifikasi tipe yang lebih spesifik. Di sinilah Klaim Tipe berperan.

Klaim Tipe termasuk dalam kategori aspek dalam bahasa M. Klaim ini menyediakan cara untuk menentukan informasi yang lebih terperinci tentang

tipe data, di luar apa yang biasanya digunakan dalam Power Query. Misalnya, sementara Power Query mungkin memperlakukan angka secara seragam, basis data mungkin membedakan antara *numeric*, *decimal*, *8*, *16*, *32*, atau *64-bit*, untuk menyebutkan beberapa. Spesifikasi terperinci ini tidak penting dalam Power Query itu sendiri, tetapi sangat berharga untuk aplikasi eksternal yang memproses dan menyimpan data yang diterima dari Power Query. Anggap Klaim Tipe sebagai bahasa universal untuk menyampaikan informasi tipe data dari satu sistem ke sistem lainnya. Sekarang mari kita lihat klaim tipe apa saja yang tersedia dalam bahasa M.

Tipe Klaim yang Tersedia

Dalam bahasa M, berbagai **Type Claims** sesuai dengan tipe data tertentu. Tabel yang ditunjukkan pada gambar berikut mencantumkan **Type Claims** ini dan tipe dasarnya. Misalnya, *Currency.Type* dan *Decimal.Type* keduanya memiliki tipe number sebagai tipe dasar, sedangkan *Text.Type* dan *Password.Type* termasuk dalam tipe text. Jadi, **Type Claims** mana yang tersedia?

Type Claim	Base Type	Description
Any.Type	type any	Represents all values
Binary.Type	type binary	Represents all binary values
Date.Type	type date	Represents all date values
DateTime.Type	type datetime	Represents all date and time values
DateTimeZone.Type	type datetimezone	Represents all date and time values relative to a timezone
Duration.Type	type duration	Represents all duration values
Function.Type	type function	Represents all functions
List.Type	type list	Represents all lists
Logical.Type	type logical	Represents all logical values
None.Type	type none	Represents no values
Null.Type	type null	Represents null
Byte.Type	type number	Represents all bytes
Currency.Type	type number	Represents currency value
Decimal.Type	type number	Represents fixed-point decimal number
Double.Type	type number	Represents double precision floating point number
Int16.Type	type number	Represents signed 16 bit integer
Int32.Type	type number	Represents signed 32 bit integer
Int64.Type	type number	Represents signed 64 bit integer
Int8.Type	type number	Represents signed 8 bit integer
Number.Type	type number	Represents all numbers
Percentage.Type	type number	Represents percentage value
Single.Type	type number	Represents single precision floating point number
Record.Type	type record	Represents all records
Table.Type	type table	Represents all tables
Character.Type	type text	Represents all characters
Guid.Type	type text	Represents a GUID value
Password.Type	type text	Represents a text password
Text.Type	type text	Represents all text values
Uri.Type	type text	Represents a text URI
Time.Type	type time	Represents all time values
Type.Type	type type	Represents all types

Gambar 1. 35 Tinjauan Umum Klaim Jenis dan tipe data dasar yang sesuai

Penting untuk dicatat bahwa Klaim Tipe bukanlah tipe data itu sendiri. Sebaliknya, klaim tersebut berfungsi dengan menetapkan tipe dasar ke suatu nilai dan melengkapinya dengan informasi aspek tambahan. Meskipun demikian, saat menangani fungsi yang memerlukan tipe data sebagai input, valid untuk memberikan tipe dasar atau Klaim Tipe sebagai argumen. Karena Anda sekarang mengetahui klaim tipe apa saja yang tersedia, mari kita lihat bagaimana kita dapat menggunakannya untuk mengonversi nilai.

Mengonversi Nilai Menggunakan Klaim Tipe

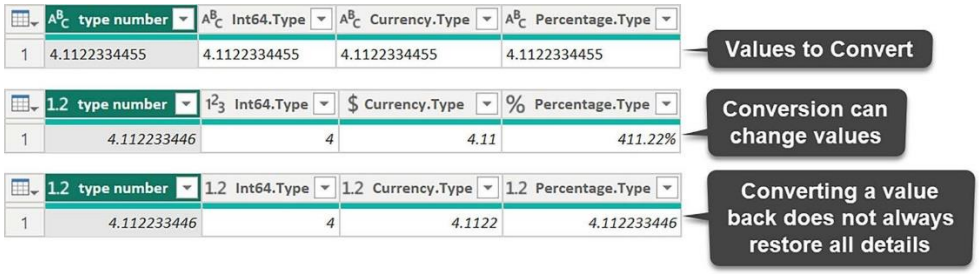
Pertimbangkan skenario saat Anda perlu mengonversi kolom menjadi tipe angka. Anda dapat menggunakan tipe dasar atau Klaim Tipe:

```
Table.TransformColumnTypes( Source, {{"Sales", type number}} )
Table.TransformColumnTypes( Source, {{"Sales", Int64.Type}} )
```

Kedua ekspresi tersebut valid, tetapi penggunaan tipe dasar atau Klaim Tipe dapat menghasilkan perilaku yang sedikit berbeda dengan tipe angka berikut:

- **Currency type**: *Currency.Type* mengembalikan angka dengan hingga 4 tempat desimal.
- **Integer types***: Int8, Int16, Int32,* dan *Int64* mengembalikan angka bulat, memangkas titik desimal apa pun.

Mari kita periksa dampak penerapan Klaim Tipe untuk konversi nilai di Power Query. Pada gambar berikut, kita menggunakan tabel dengan empat kolom sebagai contoh, seperti yang diperlihatkan pada gambar berikut. Setiap kolom ini berisi nilai teks yang identik. Tujuan kita adalah untuk mengamati dan memahami perubahan yang terjadi saat nilai teks ini mengalami konversi menggunakan Klaim Tipe yang berbeda di Power Query.



Gambar 1. 36 Konversi tipe dapat mengakibatkan hilangnya presisi

Baris tengah tabel menunjukkan seperti apa tampilan nilai-nilai yang berbeda setelah konversi, berdasarkan Klaim Tipe yang ditentukan dalam tajuk kolom masing-masing. Aspek penting yang perlu diperhatikan selama proses konversi ini adalah perubahan presisi data. Misalnya, saat nilai teks dikonversi ke *Int64.Type*, proses tersebut menghilangkan desimal apa pun, sehingga menghasilkan bilangan bulat. Sebaliknya, mengonversi nilai ke *Currency.Type* mempertahankan presisi hingga empat tempat desimal. Ini menggambarkan bagaimana Klaim Tipe yang berbeda memengaruhi presisi dan format data yang dikonversi.

Sekarang, perhatikan baik-baik baris bawah. Saat kita mengonversi tipe data yang berbeda kembali ke tipe desimal, yang dapat menampung sejumlah besar desimal, kita dapat melihat hilangnya presisi untuk kolom ***Int64.Type*** dan *Currency.Type*.

Penting untuk dipahami bahwa penggunaan Klaim Tipe untuk mengonversi tipe nilai dapat mengakibatkan hilangnya presisi nilai tersebut yang tidak dapat dipulihkan. Ingat, setelah nilai berubah, satu-satunya cara untuk kembali ke presisi awal adalah dengan membatalkan langkah konversi. Sebelumnya dalam bab ini, Anda melihat cara mengembalikan tipe data suatu nilai. Namun, bagaimana cara kerjanya untuk Klaim Tipe?

Memeriksa Klaim Jenis

Dalam bahasa M, Klaim Tipe terutama digunakan dalam tipe tabel. Karena penggunaan khusus ini, bahasa M hanya memungkinkan kita untuk

memeriksanya menggunakan fungsi tabel *Table.Schema*. Namun, mungkin ada situasi di mana Anda mungkin perlu menjelajahi aspek yang terkait dengan nilai tertentu.

Pada tulisan ini, metode paling mudah untuk menyelidiki aspek ini adalah dengan menanamkan nilai dalam tabel. Pendekatan ini memungkinkan Anda untuk menggunakan fungsi ***Table.Schema*** untuk mengungkap detail Klaim Tipe yang mendasarinya.

Pertimbangkan contoh berikut, di mana kita membuat tabel khusus untuk memeriksa tipe tertentu – *Character.Type*:

```
let
  myType   = Character.Type,
  mySchema = Table.Schema( #table(type table [Col1 = myType], {} ) ),
  typeInfo = mySchema[[TypeName], [Kind]]
in
  typeInfo
```

Dalam kode ini, kita melakukan hal berikut:

- Kita mendefinisikan *myType* sebagai *Character.Type*.
- Kita membuat tabel kosong dengan satu kolom, dengan menentukan *Character.Type* sebagai tipe kolom.
- Kita menggunakan fungsi *Table.Schema* untuk mengekstrak informasi skema tabel ini.
- Terakhir, kita mengekstrak kolom *TypeName* dan *Kind*.

Ekspresi sebelumnya mengembalikan record yang menampilkan nama tipe dan jenisnya:

```
[ TypeName = Character.Type, Kind = text ]
```

Output ini menunjukkan bahwa *Character.Type* termasuk dalam teks tipe dasar.

Sebagai kesimpulan, Anda sekarang telah mempelajari bahwa aspek menyediakan informasi tambahan untuk tipe data. Informasi tambahan ini

khususnya berguna saat mengomunikasikan data ke dan dari berbagai sistem eksternal yang berinteraksi dengan Power Query. Skenario paling umum saat analis menemukan aspek, khususnya klaim tipe, adalah saat menetapkan tipe data. Pastikan untuk memperhatikan Klaim Tipe yang mengubah ketepatan nilai karena klaim tersebut dapat mengubah nilai Anda tanpa Anda sadari.

Ke depannya, fokus kita akan beralih ke konsep atribusi tipe. Atribusi dapat menggunakan tipe data dan Klaim Tipe, dan tidak seperti konversi nilai sederhana, atribusi menggunakan pendekatan yang lebih berani – pendekatan yang memerlukan kehati-hatian karena potensi risikonya. Mari kita cari tahu cara kerjanya.

H. Mengatribusikan Tipe

Di bagian ini, kita akan membahas tujuan atribusi dan cara menggunakannya. Anda juga akan mempelajari risiko dan skenario apa saja yang dapat menyebabkan kesalahan.

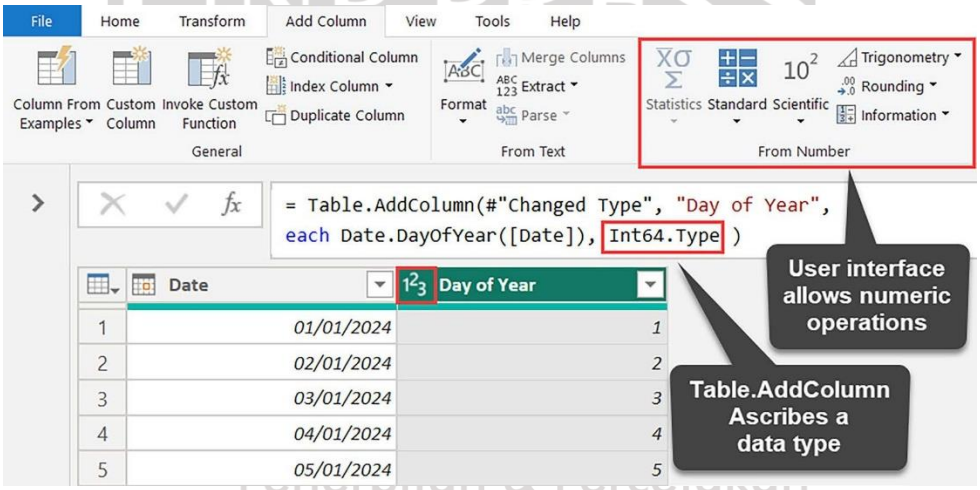
1. Apa Itu Atribusi?

Menentukan tipe adalah proses saat Anda mendeklarasikan bahwa suatu nilai sesuai dengan tipe tertentu. Hingga taraf tertentu, proses ini memerintahkan mesin untuk menerima penilaian Anda terhadap tipe nilai tersebut. Berbeda dengan konversi tipe data, saat Power Query secara aktif mengonversi nilai ke tipe yang ditentukan, menentukan tipe memberi label tipe nilai tanpa mengubah nilai itu sendiri. Selama penentuan, pemeriksaan kesesuaian terbatas dilakukan. Pemeriksaan ini memverifikasi bahwa tipe yang Anda deklarasikan kompatibel dengan tipe primitif intrinsik nilai tersebut.

Namun, pendekatan ini dapat menyebabkan beberapa kerumitan. Misalnya, jika Anda menentukan tipe teks ke kolom yang berisi bilangan bulat, Power Query akan memberi label tipe kolom sebagai teks, tetapi nilainya disimpan secara numerik. Akibatnya, Anda tidak akan dapat menerapkan fungsi khusus teks ke nilai-nilai ini, karena masih berupa angka.

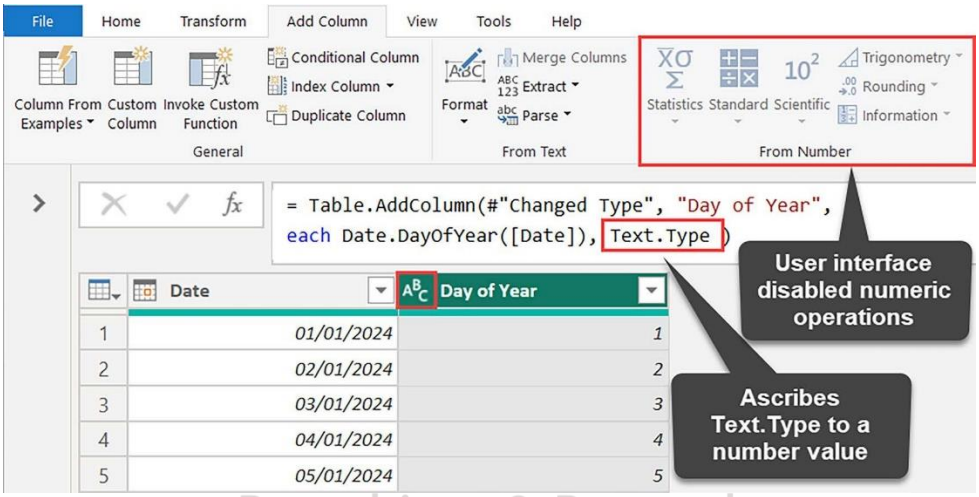
Hal ini berbeda secara signifikan dari mengonversi nilai. Saat Anda mengonversi tipe data kolom, sistem memberi label nilai, memeriksa kompatibilitas, dan mengubahnya menjadi tipe data baru. Jika suatu nilai tidak dapat dikonversi, Power Query akan menandai kesalahan. Ini memastikan bahwa nilai yang dihasilkan benar-benar cocok dengan tipe data yang dipilih atau memberikan kesalahan. Namun, selama proses pemberian nilai, Anda tidak menerima kesalahan ini.

Mari kita lihat bagaimana Anda dapat memberikan nilai. Kita lanjutkan contoh kita dengan tabel yang hanya berisi kolom tanggal. Saat kita memilih kolom, lalu menavigasi ke **Add Column** di pita, pilih **Date**, lalu **Day of year**, kita akan mendapatkan situasi berikut:



Gambar 1. 37 Table.AddColumn menetapkan Int64.Type ke kolom dengan nilai angka

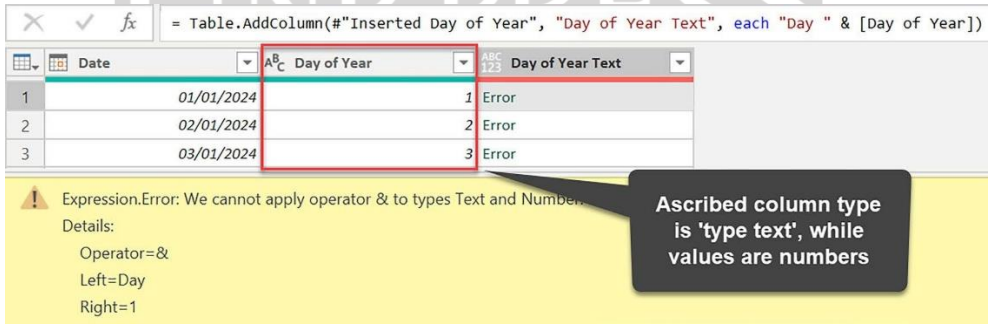
Operasi ini membuat kolom baru yang disebut **Day of year** dan secara otomatis menerima bilangan bulat (*In64.Type*) sebagai tipe kolom. Fungsi *Table.AddColumn* menerima tipe data dalam argumen keempat. Perhatikan bahwa fungsi ini tidak mengubah tipe kolom tetapi menetapkannya. Itu berarti tidak ada konversi nilai yang terjadi. Sekarang, mari kita sesuaikan rumus dan masukkan tipe yang tidak kompatibel dengan nilai angka, seperti tipe teks:



Gambar 1. 38 *Table.AddColumn* menetapkan *Text.Type* ke kolom dengan nilai angka

Dengan menyesuaikan rumus dari *Int64.Type* ke *Text.Type*, Anda dapat melihat ikon jenis teks di tajuk kolom. Perhatikan juga bahwa kolom tersebut tidak lagi mendukung operasi numerik. Nah, inilah bagian yang membingungkan. Meskipun tajuk kolom menyatakan kolom tersebut bertipe teks, nilai sebenarnya adalah numerik. Mencoba ekspresi yang memerlukan nilai teks tetap akan menghasilkan kesalahan.

Kita dapat mengujinya untuk melihat apa yang terjadi. Jika nilainya bertipe teks, kita seharusnya dapat menggabungkan nilai seperti Day ke dalamnya. Namun, gambar berikut menunjukkan bahwa ini menghasilkan kesalahan:



Gambar 1. 39 Tipe kolom yang ditetapkan adalah *Type.Text* tetapi nilainya masih bertipe angka

Itu memberi Anda gambaran tentang apa artinya mengatribusikan data. Contoh ini memberi label kolom sebagai tipe tertentu tanpa melakukan pemeriksaan tambahan. Jadi, mengapa bermanfaat untuk memahami atribusi? Ada beberapa alasan mengapa penting untuk memahami atribusi:

- **Performance**: Mengatribusikan tipe data tidak mengharuskan mesin memindai semua data Anda. Dengan mendeklarasikan tipe, Power Query memercayai kebenarannya, yang berpotensi meningkatkan performa karena melewati validasi tipe data yang ekstensif.
- **Manajemen risiko**: Aspek penting dari atribusi adalah risiko yang terlibat. Jika Anda mengatribusikan tipe data ke kolom yang berisi nilai yang tidak kompatibel dengan tipe ini, kesalahan dapat terjadi selama pemuatan data ke aplikasi seperti Power BI. Ini karena Power BI, selama proses pemuatan, mengharapkan data sesuai dengan tipe yang diatribusikan dan dapat mengembalikan kesalahan jika ada perbedaan.
- **Integrasi dengan aplikasi host**: Mengatribusikan tipe sangat relevan saat memuat data Anda ke tujuan seperti Power BI atau Excel. Aplikasi ini dapat menggunakan informasi tipe data yang ditetapkan untuk kebutuhan pemrosesan dan penyimpanannya. Pencantuman yang salah dapat menyebabkan masalah kompatibilitas dan kesalahan saat memuat data ke dalam aplikasi ini.

Sekarang setelah Anda mengetahui kegunaan pencantuman dan mengapa penting untuk memahaminya, Anda mungkin bertanya-tanya fungsi mana yang dapat mencantumkan tipe data ke suatu nilai. Mari kita cari tahu.

2. Fungsi yang Mendukung Penetapan Tipe

Bagian ini dimaksudkan untuk memperdalam pemahaman Anda tentang atribusi. Bagian ini memperkenalkan Anda pada fungsi-fungsi yang memungkinkan Anda untuk mengatribusikan suatu tipe dan akan membuat Anda lebih memahami sintaksisnya.

Menetapkan Tipe Saat Membuat Record

Records di Power Query dapat memiliki beberapa bidang, masing-masing dengan tipe data tertentu. Untuk menentukan tipe data ini, fungsi seperti *Record.FromList* mendukung tipe record kustom. Penyertaan tipe record kustom menyediakan fungsi dengan nama bidang kustom dan tipe terkaitnya. Berikut contohnya:

```
Record.FromList(
  { 2024, "M-Language", true },
  type [ Year = number, Topic = text, LearningM = logical ]
)
```

Dalam kasus ini, *Record.FromList* digunakan untuk membuat *record* di mana *Year* adalah angka, *Topic* adalah teks, dan *LearningM* adalah nilai logika.

Menentukan Tipe Saat Membuat Tabel

Beberapa fungsi memungkinkan pembuatan tabel dengan tipe data kustom. Yang paling umum adalah *#table, Table.FromRecords, Table.FromList, Table.FromColumns, Table.FromRows,* dan *Table.FromValue*. Fungsi-fungsi ini memungkinkan Anda menentukan nama kolom dan tipe datanya.

Misalnya, kode M berikut membuat tabel dengan menentukan nama dan nilai kolom:

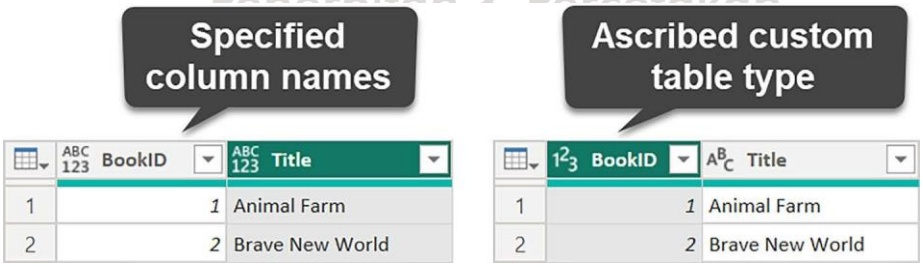
```
#table(
    { "BookID", "Title" },
    { { 1, "Animal Farm" }, { 2, "Brave New World" } }
)
```

Dalam contoh ini, kita menetapkan nama kolom *BookID* dan *Title* pada argumen pertama, dan kolom tabel secara otomatis ditetapkan ke tipe any. Ini menunjukkan bahwa penyediaan tipe kolom tidak diperlukan.

Namun, dalam kasus di mana Anda ingin lebih spesifik, Anda dapat menyediakan fungsi #table dengan tipe tabel kustom untuk menetapkan kolom *BookID* sebagai integer dan *Title* sebagai *text*. Untuk menetapkan nama dan tipe kolom yang relevan, Anda dapat menggunakan kode berikut:

```
#table(
    type table[ BookID = Int64.Type, Title = text ],
    { { 1, "Animal Farm" }, { 2, "Brave New World" } }
)
```

Hasil dari kedua potongan kode ini ditunjukkan pada gambar berikut:



Gambar 1. 40 Dua tabel di mana satu menerima nama kolom dan yang lainnya juga mendapatkan jenis kolom

Gambar ini menunjukkan dua skenario: tabel yang hanya mencantumkan nama kolom dan tabel lain dengan tipe kolom yang ditentukan. Meskipun sintaksisnya berbeda di antara berbagai fungsi pembuat tabel, konsep dasar penetapan tipe data adalah seragam. Oleh karena itu, bab ini tidak akan merinci setiap fungsi, tetapi berfokus pada prinsip inti penetapan tipe.

Menentukan Tipe Saat Memodifikasi Tabel

Selain membuat tabel, Power Query menawarkan fungsi untuk mengubah tabel dan menetapkan tipe pada tabel tersebut. Fungsi yang paling umum digunakan untuk tujuan ini adalah *Table.TransformColumns*, *Table.AddColumn*, dan *Table.Group*.

Pertimbangkan tabel seperti yang disediakan sebelumnya, di mana kolom Judul berisi nilai teks. Untuk mengubah kolom ini menggunakan

Table.TransformColumns, Anda dapat menggunakan UI editor Power Query. Misalnya, dengan memilih kolom Judul, menavigasi ke tab **Transform**, memilih **Format**, lalu memilih **UPPERCASE**, kode berikut akan dihasilkan:

```
Table.TransformColumns(
  Source,

  {{"Title", Text.Upper, type text}} )
```

Dalam fungsi ini, argumen pertama menentukan tabel yang akan diubah. Argumen kedua adalah list yang merinci operasi yang dilakukan pada kolom, diformat sebagai *{"column name", transformation, new column type}*. Format ini tidak hanya mengubah kolom tetapi juga memungkinkan Anda untuk menetapkan tipe tertentu padanya.

Menetapkan Tipe ke Nilai Apapun

Contoh terakhir di bagian ini adalah *Value.ReplaceType*, yang merupakan fungsi yang didedikasikan untuk mengganti tipe data dari nilai tertentu. Fungsi ini memerlukan dua argumen: nilai itu sendiri dan tipe yang ingin Anda tetapkan. Misalnya:

```
Value.ReplaceType( 5, type number )
```

Pernyataan sebelumnya memberikan nomor tipe pada suatu nilai. Namun, fungsi tersebut tidak hanya terbatas pada pemberian tipe primitif. Fungsi tersebut juga dapat menangani tipe kustom yang lebih kompleks.

```
Value.ReplaceType( { "a", "b", "c" }, type { text } )
```

Sejauh ini, semuanya tampak dapat diprediksi. Sekarang, mari kita buat beberapa penyesuaian yang mungkin mengejutkan. Pertimbangkan contoh berikut:

```
Value.ReplaceType( { "a", "b", "c" }, type { number } )
Value.ReplaceType( [ a = 1, b = true ], type [ a = date, b = text ] )
```

Perhatikan bahwa pada baris pertama, list item teks menerima tipe data yang menjelaskan list angka. Pada baris kedua, record ditetapkan sebagai tipe record kustom, tetapi sekali lagi dengan tipe data yang tidak cocok. Menariknya, UI Power Query tidak mengembalikan kesalahan.

Setelah meninjau contoh-contoh sebelumnya, Anda mungkin berpikir bahwa mungkin untuk menetapkan tipe apa pun yang dipilih ke suatu nilai. Namun, ini tidak selalu memungkinkan. Ada skenario di mana upaya untuk menetapkan tipe dapat menyebabkan kesalahan. Di bagian berikutnya, kita akan mempelajari situasi tertentu di mana penetapan tipe mungkin tidak berfungsi sebagaimana mestinya dan akhirnya mengakibatkan kesalahan.

3. Kesalahan Saat Menetapkan Tipe

Pemberian atribusi merupakan penyebab umum kesalahan dalam Power Query, tetapi kesalahan tidak terjadi dalam semua situasi. Jadi, mengapa kita mendapatkan kesalahan dalam beberapa kasus dan tidak dalam kasus lain? Ada tiga situasi di mana pemberian atribusi tipe ke nilai dapat menghasilkan kesalahan. Kesalahan pemberian atribusi tipe terjadi dalam situasi berikut:

- Tipe dasar dari tipe yang diberikan tidak kompatibel dengan nilai yang ditetapkan.
- Tipe dasar kompatibel, tetapi Klaim Tipe itu sendiri tidak sesuai dengan nilai.
- Tipe yang diberikan diterapkan ke nilai terstruktur yang berisi nilai yang tidak kompatibel dengan tipe tersebut.

Mari kita lihat masing-masing penyebabnya lebih dekat.

Tipe Dasar Tidak Kompatibel Dengan Nilai

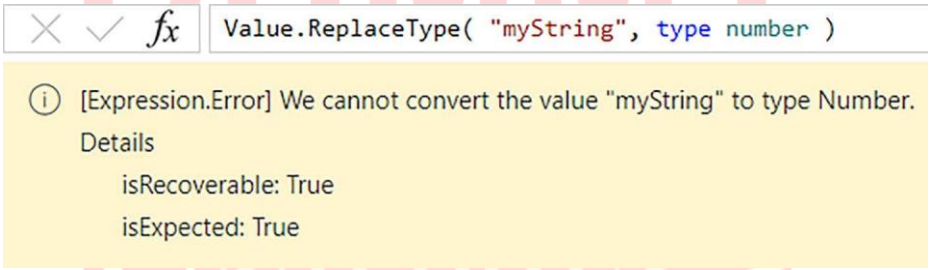
Setiap kali Anda menetapkan Klaim Tipe ke suatu nilai, tipe dasar harus kompatibel dengan nilai yang Anda gunakan. Dengan kata lain, struktur nilai dan tipe Anda harus sama. Jika keduanya tidak kompatibel, Anda akan menerima galat. Untuk tinjauan umum Klaim Tipe yang berbeda dan tipe

dasarnya, silakan lihat bagian Klaim Tipe yang Tersedia dalam bab ini, serta Gambar 5.35.

Berikut ini beberapa contoh yang memunculkan galat karena tipe yang tidak kompatibel:

```
Value.ReplaceType( "myString", type number )
Value.ReplaceType ( #date( 2024, 12, 31 ), type logical )
Value.ReplaceType( 123, type text )
```

Contoh kode pertama mengembalikan kesalahan berikut:



Gambar 1. 41 Saat menetapkan tipe, nilai dasarnya harus sesuai dengan nilai

Mengubah tipe data ke tipe yang kompatibel dengan struktur nilai akan menyelesaikan kesalahan ini.

Klaim Jenis Tidak Sesuai Dengan Nilai

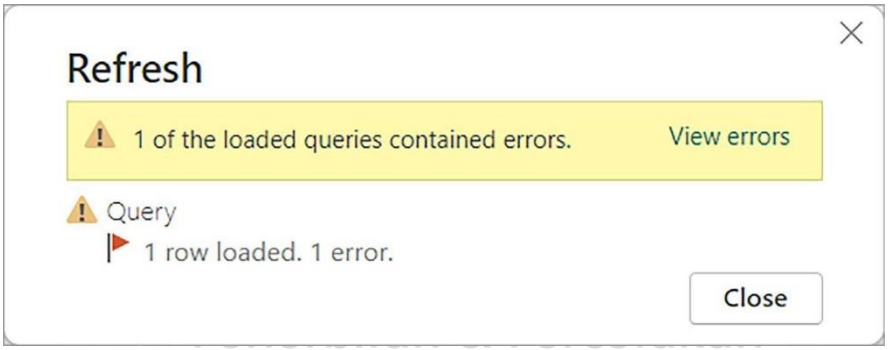
Skenario lain yang dapat menyebabkan kesalahan adalah saat Anda menetapkan tipe yang tipe dasarnya sesuai dengan nilai, tetapi Klaim Tipe tidak. Misalnya, pertimbangkan skenario saat Anda memiliki nilai dengan titik desimal, seperti *5,33353*. Power Query memungkinkan Anda menetapkan nilai ini dengan *Int64.Type* yang mewakili bilangan bulat. Anda dapat melakukannya dengan kode berikut:

```
Value.ReplaceType( 5.33353, Int64.Type )
```

Ini mengembalikan nilai angka:

Gambar 1. 42 Ketidakcocokan antara Klaim Jenis dan nilai

Nilai dasar yang mendasarinya, yang merupakan nomor tipe, awalnya tampak kompatibel; dengan demikian Power Query tidak menimbulkan kesalahan. Penting untuk dicatat bahwa Power Query hanya melakukan pemeriksaan dasar selama atribusi. Pemeriksaan ini hanya memverifikasi apakah tipe dasar nilai tersebut selaras dengan tipe yang ditetapkan. Namun, komplikasi muncul saat memuat kueri ke tujuan seperti Power BI. Di sini, Power BI memeriksa informasi Klaim Tipe yang ditentukan dan mencoba memproses nilai sebagai bilangan bulat, yang menyebabkan kesalahan berikut:



Gambar 1. 43 Menetapkan Tipe Klaim yang tidak sesuai dengan nilai akan menghasilkan kesalahan

Yang membuat frustrasi, saat Anda mengeklik **View errors**, sebuah jendela muncul dengan tulisan: **Unable to get errors for your queries. Please refresh your queries and try again**.

Skenario ini dapat membingungkan karena Power BI menandai kesalahan yang tidak terlihat di editor Power Query. Jadi, berikut beberapa saran: jika Anda menemukan kesalahan dan Power Query tidak dapat menunjukkannya

pada baris tertentu, sebaiknya tinjau kueri Anda dengan saksama. Anda mungkin telah menetapkan tipe yang salah pada suatu kolom.

Menetapkan Tipe yang Tidak Kompatibel ke Nilai Terstruktur

Kesalahan dalam Power Query juga dapat muncul akibat penetapan tipe yang tidak tepat pada elemen dalam nilai terstruktur. Kesalahan ini hanya muncul setelah memuat data Anda ke Power BI. Mari kita bahas apa artinya ini.

Anda dapat menetapkan Klaim Tipe pada nilai terstruktur seperti list, record, tabel, dan fungsi. Power Query menilai apakah nilai tersebut sesuai dengan tipe dasar Klaim Tipe yang ditetapkan. Oleh karena itu, jika nilai tertentu tidak sesuai dengan tipe dasar Klaim Tipe, nilai tersebut akan menghasilkan kesalahan, seperti yang ditunjukkan oleh dua contoh berikut:

```
Value.ReplaceType( [A = 1, B = 2 ], Int64.Type )
Value.ReplaceType( {1, 2, 3}, type text )
```

Dalam kedua ekspresi, tipe dasar tidak kompatibel dengan nilai yang mendasarinya. Sebuah record bukanlah bilangan bulat; list juga bukan nilai teks. Menyesuaikan tipe sebagai berikut akan membuat kedua ekspresi valid:

```
Value.ReplaceType( [A = 1, B = 2 ], type [ A = number, B = number ] )
Value.ReplaceType( {1, 2, 3} , type { number } )
```

Dalam kasus ini, kita menyediakan tipe list dan tipe record yang benar. Namun, penting untuk dipahami bahwa saat Anda menetapkan tipe ke nilai terstruktur, nilai yang terdapat dalam nilai terstruktur tersebut tidak divalidasi untuk kompatibilitas.

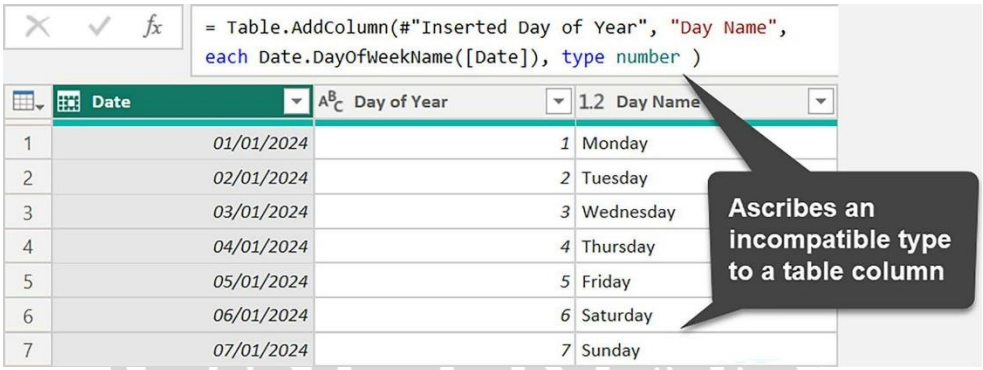
Ini berarti Anda dapat menetapkan tipe list kompleks ke nilai yang menunjukkan bahwa nilai dalam list tersebut bertipe berbeda dari yang sebenarnya. Selain itu, operasi ini tidak akan menimbulkan kesalahan. Misalnya, lihat ekspresi berikut:

```
Value.ReplaceType( { 1, 2, 3} , type { date } )
```

Meskipun list tersebut hanya berisi nilai angka, Anda menetapkan list yang berisi tanggal tanpa menerima kesalahan di editor Power Query. Yaitu, hingga Anda memuat data Anda ke aplikasi host seperti Power BI. Saat Power BI menerima informasi jenis, ia akan menggunakannya untuk menyerap dan mengompresi data. Saat informasi jenis tersebut kemudian bertentangan dengan nilai sebenarnya, Power BI mengembalikan kesalahan.

Hal yang sama terjadi pada record, tabel, dan jenis fungsi. Anda dapat menunjukkan dalam nilai jenis bahwa Anda mengharapkan nilai jenis tertentu, tetapi Power Query tidak memeriksa apakah klaim tersebut benar. Anda mungkin mengalami kesalahan hanya saat Anda memuat data ke Power BI.

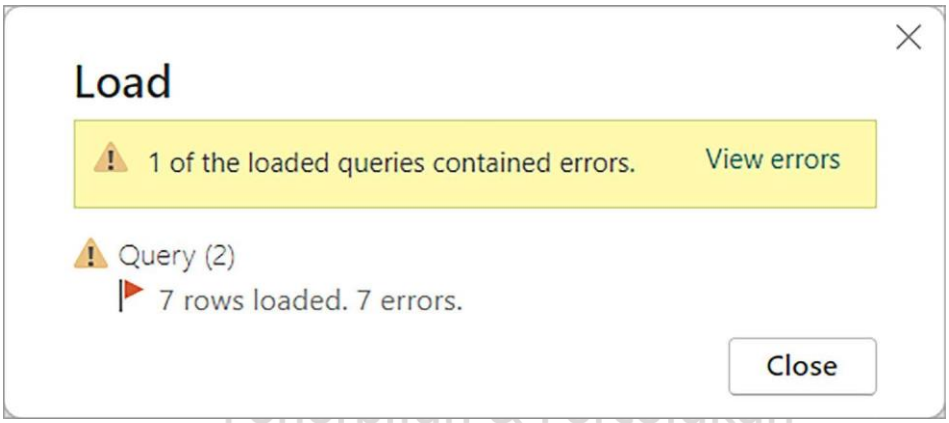
Mengetahui hal ini, mari kita bahas skenario paling umum di mana Anda akan mengalami masalah ini. Misalkan Anda ingin menambahkan kolom kustom ke tabel Anda. Untuk mencegah langkah Ubah Jenis tambahan, Anda dapat menetapkan jenis ke kolom yang baru dibuat menggunakan argumen keempat *Table.AddColumn*:



Gambar 1. 44 *Table.AddColumn* memungkinkan Anda untuk menetapkan tipe yang tidak kompatibel dengan nilai kolom

Menetapkan tipe dalam ekspresi ini sangat praktis untuk mencegah langkah-langkah tambahan. Ingatlah bahwa menetapkan tipe kolom ke nilai terstruktur (seperti tabel) akan melewati pemeriksaan kesesuaian nilai yang mendasarinya. Ia hanya memeriksa apakah tipe dasar (tabel tipe) sesuai dengan

nilai tersebut. Saat Anda memuat kueri ini ke Power BI, Anda akan mengalami galat berikut:



Gambar 1. 45 Kesalahan ini muncul saat memuat tabel dengan tipe yang tidak kompatibel ke Power BI

Saat Anda mengeklik **View errors**, kueri baru akan terbuka yang menunjukkan kesalahan jenis. Hanya setelah Anda memperbaiki jenis kolom yang salah ditetapkan, Anda dapat menyegarkan kueri dengan sukses.

Untuk memberi Anda contoh umum lain yang mengalami masalah yang sama, mari kita lihat fungsi *Table.Group*. Misalkan Anda memiliki tabel berikut:

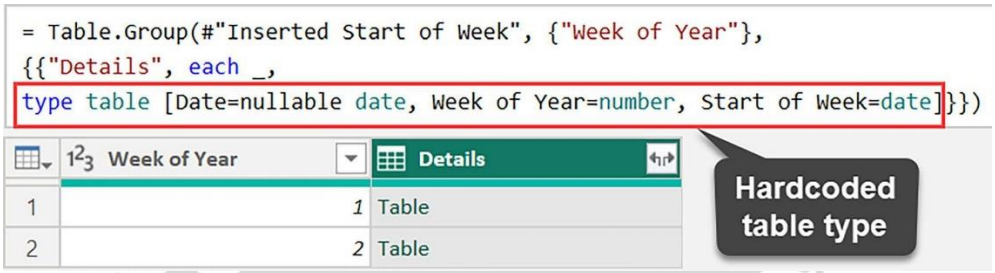
	Date	Week of Year	Start of Week
1	03/01/2024	1	01/01/2024
2	04/01/2024	1	01/01/2024
3	05/01/2024	1	01/01/2024
4	06/01/2024	1	01/01/2024
5	07/01/2024	1	01/01/2024
6	08/01/2024	2	08/01/2024
7	09/01/2024	2	08/01/2024

Gambar 1. 46 Dataset yang akan kita gunakan untuk meringkas nilai

Sekarang, jika Anda ingin meringkas tabel berdasarkan Minggu dalam Setahun, Anda dapat melakukannya dengan:

1) Memilih kolom **Week of Year**.
2) Masuk ke **Home** di pita dan pilih **Group By**.
3) Membuat **New column** bernama **Details** dan memilih **All Rows operation**.

Anda berakhir dengan situasi berikut:



Gambar 1. 47 Pengelompokan data Anda dengan operasi Semua Baris mengkodekan tipe data secara permanen

Operasi ini menggunakan fungsi *Table.Group* dan mengkodekan tipe tabel kustom dalam ekspresinya. Karena tipe tersebut sesuai dengan nilai yang ada, tidak ada masalah. Misalkan Anda ingin membuat perubahan lebih awal dalam kueri Anda. Anda mengubah nilai kolom *Start of Week* dari tanggal menjadi nama hari (teks). Ekspresi aslinya adalah:

```
Table.AddColumn( Source, "Start of Week",
  each Date.StartOfWeek([Date]), type date )
```

Untuk mengubahnya menjadi nama hari pada masing-masing tanggal, Anda menggunakan ekspresi berikut:

```
Table.AddColumn( Source, "Start of Week",
  each Date.DayOfWeekName( Date.StartOfWeek([Date]) ), type text)
```

Di sinilah masalahnya dimulai. Operasi **Group By** merujuk pada kolom *Start of Week* tetapi mengkodekan tipe data Date secara permanen. Kita menyesuaikan kolom kita sebelumnya dalam kueri, yang sekarang bertipe teks. Efeknya adalah saat Anda memperluas kolom details, kolom tersebut akan diberi tipe data yang tidak lagi sesuai dengan nilainya. Memuat data ini ke Power BI kemudian menghasilkan kesalahan.

Untuk mengatasi situasi ini, Anda memiliki beberapa pilihan. Pertama-tama, Anda dapat menyesuaikan fungsi *Table.Group* untuk mencerminkan tipe data yang benar. Ini adalah tugas manual yang berisiko lupa mengubah tipe tabel. Atau, Anda dapat mengambil tipe tabel langkah sebelumnya secara dinamis menggunakan fungsi *Value.Type*. Ekspresi berikut mengambil tipe tabel dari langkah sebelumnya yang disebut *Inserted Start of Week*:

```
Table.Group(
  #"Inserted Start of Week",
  {"Week of Year"},
  {{"Details", each _, Value.Type( #"Inserted Start of Week" )}}
)
```

Setiap perubahan mendatang yang muncul lebih awal dalam kueri akan secara otomatis diambil, memastikan tabel Anda selalu memiliki tipe data yang benar yang terkait dengannya.

Kesimpulannya, memahami dan menggunakan atribusi tipe dengan benar sangat penting untuk memastikan kueri Anda bebas dari kesalahan. Atribusi tipe dengan benar memungkinkan Anda mengklasifikasikan data secara efisien, sehingga Power Query tidak perlu memindai semua data Anda. Namun, tugas ini memerlukan akurasi dan perhatian terhadap detail, karena kesalahan dapat menyebabkan kesalahan yang mungkin tidak terlihat hingga data dimuat ke aplikasi eksternal. Jadi, meskipun atribusi tipe membantu mengklasifikasikan nilai, atribusi harus dilakukan dengan pertimbangan cermat untuk mencegah kesalahan.

Karena Anda sekarang tahu cara mengatribusi nilai, mari kita lihat cara memeriksa apakah tipe tersebut sama atau kompatibel.

I. Kesetaraan Tipe, Kesesuaian, dan Pernyataan

Mengetahui apakah suatu nilai sama atau kompatibel dengan tipe lain berguna untuk beberapa operasi. Ini akan memungkinkan Anda untuk

menyiapkan pernyataan kondisional yang menangani tipe data berbeda dengan operasi yang sesuai. Bagian ini membahas topik kesetaraan dan kompatibilitas tipe. Mengetahui lebih banyak tentang topik ini akan membekali Anda dengan pengetahuan untuk membuat logika pada topik tersebut.

1. Tipe Kesetaraan

Katakanlah Anda ingin memeriksa apakah dua tipe data sama; pertimbangkan perbandingan berikut:

```
type text = type text // Output: true
```

Perbandingan langsung ini tampaknya mengonfirmasi kesetaraan; lagipula, hasilnya benar. Namun, membandingkan tipe yang lebih kompleks dapat memberikan hasil yang berbeda:

```
type [ a = text ] = type [ a = text ] // Output: false
```

Hasilnya salah dalam kasus ini, meskipun konten yang dibandingkan tampak identik. Bagaimana mungkin? Alasannya adalah bahwa hasil perbandingan tipe dalam Power Query M tidak dijamin konsisten atau dapat diprediksi. Ini karena kesetaraan tipe tidak didefinisikan dalam bahasa M itu sendiri.

Akibatnya, implementasi M yang berbeda mungkin menerapkan aturannya sendiri saat membandingkan tipe. Variabilitas ini berarti perbandingan tipe yang sama dapat menghasilkan hasil yang berbeda tergantung pada lingkungan spesifik Anda, seperti Excel atau Power BI. Meskipun versi saat ini mungkin setuju bahwa perbandingan tertentu menghasilkan true, tidak adanya definisi formal untuk kesetaraan tipe dalam bahasa tersebut berarti perilaku ini dapat berubah dalam pembaruan atau versi mendatang.

Daripada membandingkan nilai tipe secara langsung, bahasa M menawarkan berbagai fungsi pustaka yang dirancang untuk menangani nilai tipe. Contoh penting adalah fungsi *Type.Is*. Fungsi ini memeriksa apakah tipe

yang diberikan sebagai argumen pertama kompatibel dengan tipe yang diberikan sebagai argumen kedua. Misalnya:

```
Type.Is(type date, type nullable date)  // true
Type.Is(type nullable date, type date)  // false
Type.Is(type text, type number)    // false
```

Fungsi *Type.Is* dapat menerima tipe apa pun, termasuk tipe kustom, sebagai argumen pertamanya. Namun, argumen kedua harus berupa tipe primitif yang dapat bernilai null. Misalnya:

```
Type.Is(type [ A = number], type [ A = number] ) // Output false
Type.Is(type [ A = number], type record )        // Output true
```

Contoh pertama menunjukkan bahwa penggunaan tipe kustom sebagai argumen kedua dalam fungsi *Type.Is* secara konsisten menghasilkan false. Sebaliknya, contoh kedua menunjukkan kemampuan fungsi untuk menguji apakah nilai tipe pertama sesuai dengan yang kedua. Karena tipe record kustom secara inheren sesuai dengan record tipe, perbandingan khusus ini menghasilkan *true*.

Hal ini menunjukkan bahwa adalah mungkin untuk memeriksa apakah tipe kustom cocok dengan tipe dasar tertentu menggunakan *Type.Is*. Namun, penting untuk dicatat bahwa bahasa M saat ini tidak memiliki fungsi bawaan khusus untuk menilai kompatibilitas antara tipe standar dan tipe kustom.

Meskipun demikian, pustaka standar dalam M mencakup fungsi yang dapat mengidentifikasi karakteristik spesifik dari tipe kustom, yang memungkinkan pembuatan uji kompatibilitas yang disesuaikan. Contoh fungsi tersebut meliputi:

```
Type.IsNullable( type nullable text )  // Output: true
Type.NonNullable( type nullable text ) // Output: false
Type.ListItem( type { number } )       // Output: type
```

Fungsi-fungsi ini hanyalah sekilas fungsi tipe yang tersedia dalam bahasa M. Secara total, ada 22 fungsi tipe, yang masing-masing memungkinkan Anda

untuk melakukan operasi yang berbeda dengan tipe. Kita menganjurkan Anda untuk memeriksa opsi yang tersedia dan sintaksnya di https://powerquery.how/type-functions/.

Selain memeriksa apakah suatu tipe sama dengan tipe lainnya, ada beberapa kasus di mana memeriksa apakah suatu nilai sesuai dengan tipe lainnya berguna. Kita akan membahas cara melakukannya selanjutnya.

2. Tipe Kesesuaian

Selain memeriksa apakah suatu tipe sesuai dengan tipe lain, Anda dapat melakukan jenis pemeriksaan serupa untuk nilai. Bila Anda memiliki nilai dan tipe data dan ingin menentukan apakah nilai tersebut konsisten dengan (yaitu, "sesuai dengan") tipe data tersebut, Anda dapat menggunakan fungsi *Value.Is*. Fungsi ini mengambil semua jenis nilai sebagai input dan semua tipe data dan mengembalikan true atau *false*, tergantung pada apakah nilai input sesuai dengan tipe data input. Proses ini membantu dalam memvalidasi data, mencegah kesalahan, dan menjaga konsistensi dalam kueri Anda. Sintaksnya adalah sebagai berikut:

```
Value.Is(value as any, type as type) as logical
```

Saat Anda menerapkan fungsi ke berbagai jenis nilai, fungsi tersebut akan mengembalikan nilai benar atau salah yang menunjukkan apakah nilai tersebut termasuk jenis tertentu:

```
Value.Is(9, Number.Type)  // returns true because 9 is a number.
Value.Is("126", Number.Type) // returns false: "126" is a text value
Value.Is(36, Value.Type(36))//  returns true: self-conformance.
```

Dengan menggunakan *Value.Is*, Anda memiliki cara mudah untuk menguji apakah nilai Anda sesuai dengan tipe tertentu. Perbandingan ini juga dapat dilakukan dengan cara yang disederhanakan dengan menggunakan operator is. Ekspresi berikut menguji apakah 49 bertipe angka:

```
49  is number
```

Ini pada dasarnya adalah perbandingan yang sama tetapi ditulis dengan cara yang lebih sederhana. Perhatikan bahwa metode yang disederhanakan yang ditunjukkan sebelumnya hanya berfungsi dengan nilai primitif. Anda tidak dapat menggunakan Klaim Tipe, seperti *Number.Type* atau *Int64.Type* di sini. Dengan demikian, ekspresi berikut keduanya bernilai benar:

```
Value.Is( "Hello",
Text.Type) "Hello" is
```

Mengetahui apakah suatu nilai sesuai dengan tipe tertentu sangatlah berguna. Bahkan, hal ini memungkinkan Anda untuk memfilter data hingga ke baris yang sesuai dengan tipe tertentu. Dalam situasi lain, jika Anda ingin menerapkan tipe data, Anda juga dapat menggunakan pernyataan tipe, topik yang akan kita bahas selanjutnya.

3. Tipe Pernyataan

Penegasan tipe di Power Query diimplementasikan melalui fungsi *Value.As*. Fungsi ini digunakan untuk secara eksplisit memberlakukan tipe data tertentu pada nilai tertentu. Fungsi ini bertindak sebagai titik pemeriksaan untuk memverifikasi apakah suatu nilai sesuai dengan tipe data yang diinginkan, memberi Anda opsi lain untuk memastikan integritas data dalam kueri Anda.

Fungsi *Value.As* menerima dua parameter: nilai dari tipe apa pun dan tipe data primitif untuk diperiksa. Peran utamanya adalah mengembalikan nilai asli jika cocok dengan tipe yang ditentukan. Namun, jika ada ketidakcocokan antara nilai dan tipe, fungsi tersebut menghasilkan kesalahan. Perilaku ini berguna untuk menjaga konsistensi data dan mencegah kesalahan yang tidak terduga. Berikut sintaksis dasar *Value.As*:

```
Value.As(value as any, type as type) as any
```

Anda dapat menerapkan Value.As dalam beberapa cara, misalnya:

```
Value.As( 36, Number.Type ) // 36
Value.As( "36", Text.Type ) // 36
Value.As( 36, Text.Type )   // Expression.Error: We cannot convert the
                            // value 36 to type Text
```

Ekspresi alternatif dari konsep ini adalah pengecoran tipe langsung, seperti:

```
36 as number
```

Ekspresi ini benar ketika *Value.Is(36, Number.Type)* dibandingkan, yang memvalidasi bahwa 36 memang angka. Fitur *Value.As* yang bermanfaat adalah mengembalikan kesalahan sebagai nilai. Karakteristik ini memungkinkan pengembangan strategi penanganan kesalahan yang rumit, yang penting ketika nilai tidak sesuai dengan tipe yang diharapkan. Dengan menggunakan fungsi ini, Anda dapat membuat logika kustom untuk mengelola kesalahan. Ini dapat berkisar dari pencatatan kesalahan sederhana hingga menyediakan nilai fallback, yang memastikan bahwa pemrosesan data Anda tetap tidak terganggu. Eksplorasi terperinci dari teknik penanganan kesalahan ini akan menjadi fokus Bab 12.

Kesimpulannya, *Value.As* adalah fungsi yang berguna yang memastikan bahwa nilai sesuai dengan tipe yang ditentukan. Ditambah dengan konsep yang lebih luas tentang kesetaraan dan kesesuaian tipe, kini Anda memiliki berbagai opsi untuk memastikan integritas tipe dan membuat kueri yang andal.

J. Ringkasan

Saat kita mengakhiri bab ini, mari kita renungkan apa yang telah kita pelajari tentang tipe data dalam bahasa M. Kita mulai dengan memperkenalkan konsep dasar tipe data, sebuah konsep penting untuk memahami dan menggunakan bahasa M secara efektif.

Kita menjelajahi tipe data primitif dan kustom. Kita mempelajari bagaimana berbagai fungsi dalam M menangani tipe data ini, dengan memperhatikan

nuansa antara fungsi yang menerima tipe yang dapat bernilai null dan fungsi yang memerlukan kesesuaian tipe yang lebih ketat. Perbedaan ini penting untuk menyusun kueri yang tepat dan fungsional.

Kita juga menyelidiki potensi jebakan yang terkait dengan tipe data. Anda melihat bagaimana tipe data dapat menyebabkan berbagai kesalahan. Sementara beberapa, seperti kesalahan konversi, langsung terlihat, yang lain lebih halus. Khususnya, kesalahan yang diakibatkan oleh penetapan tipe yang tidak kompatibel dapat menjadi tantangan untuk diidentifikasi dan diperbaiki.

Sekarang, dengan dasar dalam tipe data ini, Anda diperlengkapi dengan baik untuk mengintegrasikannya ke dalam kueri, fungsi, dan ekspresi Anda dengan lebih efektif. Meskipun bab ini memberikan contoh-contoh spesifik, bab ini hanya menyentuh permukaan dari apa yang mungkin dilakukan dengan tipe data di M. Dalam bab-bab berikutnya, kita akan menguraikan konsep-konsep ini, menawarkan contoh-contoh tambahan tentang cara menggabungkannya saat bekerja dengan nilai-nilai terstruktur, saat membuat fungsi-fungsi kustom, dan bagaimana fungsi-fungsi ini berguna saat menulis kode yang efisien menggunakan *List.Generate*.

K. Penutup

1. Tes Formatif

No	Soal	Bobot
1.	Jelaskan apa yang dimaksud dengan tipe data dalam konteks bahasa M. Mengapa pemahaman tentang tipe data sangat penting dalam pengolahan data?	10
2.	Diskusikan perbedaan antara tipe data primitif dan tipe data kustom dalam M. Berikan contoh untuk masing-masing tipe dan situasi penggunaannya.	10
3.	Apa yang dimaksud dengan deteksi tipe data otomatis dalam Power Query? Bagaimana proses ini dilakukan dan apa manfaatnya?	10
4.	Uraikan langkah-langkah yang perlu diambil untuk mengonversi tipe data kolom di Power Query. Apa saja tantangan yang mungkin dihadapi selama proses ini?	10

5.	Jelaskan bagaimana tipe data dapat mempengaruhi proses transformasi dan analisis data dalam kueri M. Berikan contoh konkret.	10
6.	Diskusikan pentingnya atribusi tipe dalam Power Query. Apa risiko yang terlibat jika atribusi tipe tidak dilakukan dengan benar?	10
7.	Apa yang dimaksud dengan nullable types dalam bahasa M? Mengapa penggunaan nullable types penting dalam konteks pengolahan data?	10
8.	Jelaskan bagaimana cara menetapkan tipe data untuk kolom dalam tabel di Power Query. Apa saja metode yang dapat digunakan untuk memastikan akurasi tipe data?	10
9.	Bagaimana pengaruh lokal (locale) dapat memengaruhi deteksi dan konversi tipe data di Power Query. Berikan contoh situasi di mana hal ini terjadi.	10
10.	Bagaimana cara mengatasi kesalahan yang muncul akibat konversi tipe data yang tidak sesuai di Power Query? Berikan pendekatan yang dapat diambil untuk memperbaiki masalah ini.	10

BAB 2

Nilai Terstruktur

Topik Bab

Pada bab ini, topik yang akan dibahas adalah:

- Memperkenalkan nilai terstruktur
- List (List)
- Records (Record)
- Tables (Tabel)

A. Pendahuluan

Dalam bab ini, kita fokus pada konsep penting dalam bahasa M Power Query: nilai terstruktur. Sementara kita memperkenalkan nilai dan tipe data dalam Gbapter 4 dan 5, nilai terstruktur lebih kompleks dan memerlukan perhatian tambahan. Tidak seperti nilai sederhana atau primitif, nilai terstruktur dapat mencakup beberapa nilai lain di dalamnya. Kompleksitas ini memungkinkan nilai terstruktur untuk menampung berbagai nilai primitif, atau bahkan nilai terstruktur lainnya, yang membuka berbagai kemungkinan untuk manipulasi dan analisis data.

Pentingnya nilai terstruktur dalam bahasa M sedemikian rupa sehingga kita mendedikasikan seluruh bab untuk membahasnya. Konstruksi yang akan Anda pelajari tidak hanya berkontribusi pada kode yang lebih efisien dan efektif, tetapi juga membentuk tulang punggung banyak operasi dalam bahasa M. Bab ini membahas topik-topik berikut:

- Memperkenalkan nilai terstruktur
- List
- Record
- Tabel

Di setiap bagian, kita akan memberikan penjelasan yang jelas tentang konsep-konsep tersebut, mengklarifikasinya dengan contoh-contoh yang sesuai, dan menyoroti kasus-kasus penggunaannya. Pendekatan terstruktur ini akan memudahkan Anda memahami konsep-konsep ini, melihat bagaimana konsep-konsep tersebut saling berhubungan, dan menerapkannya secara efektif dalam pekerjaan Anda.

1. Kasus Pemantik Berfikir Kritis: Analisis Data Penjualan

Sebuah perusahaan retail mengumpulkan data penjualan dalam bentuk tabel yang berisi kolom-kolom berikut:

- ID Produk (Teks)
- Nama Produk (Teks)

- Jumlah Terjual (Angka)
- Tanggal Penjualan (Tanggal)

Situasi: Tim analisis data menemukan bahwa beberapa kolom dalam tabel tersebut memiliki format yang tidak konsisten. Misalnya, kolom Tanggal Penjualan ada yang menggunakan format DD/MM/YYYY dan ada juga yang menggunakan MM/DD/YYYY. Selain itu, beberapa nilai di kolom Jumlah Terjual berisi teks yang tidak valid.

1) Identifikasi Masalah: Apa saja masalah yang dihadapi tim analisis data dalam tabel ini? Mengapa masalah ini penting untuk diatasi?
2) Penyelesaian Masalah: Bagaimana Anda akan mengatasi masalah format tanggal yang berbeda? Apa langkah-langkah yang perlu diambil?
3) Validasi Data: Bagaimana Anda memastikan bahwa kolom Jumlah Terjual hanya berisi nilai angka? Apa yang harus dilakukan jika ditemukan nilai yang tidak valid?
4) Penggunaan Nilai Terstruktur: Bagaimana penggunaan nilai terstruktur (list, record, tabel) dapat membantu dalam menganalisis dan membersihkan data penjualan ini?
5) Keputusan Berdasarkan Data: Setelah membersihkan data, keputusan apa yang dapat diambil oleh manajemen untuk meningkatkan strategi penjualan?

Diskusikan kasus ini dalam kelompok. Identifikasi langkah-langkah konkret yang dapat diambil untuk membersihkan dan menganalisis data, serta bagaimana data yang bersih dapat mendukung pengambilan keputusan yang lebih baik.

B. Memperkenalkan Nilai Terstruktur

Saat kita mempelajari lebih jauh bahasa M, kita sampai pada salah satu konsep terpenting: nilai terstruktur. Nilai-nilai ini membentuk fondasi pembentukan dan manipulasi data dalam Power Query M. Namun, apa sebenarnya yang kita maksud ketika kita mengatakan nilai terstruktur? Anda dapat menganggapnya sebagai wadah atau paket yang dapat berisi satu atau beberapa nilai. Wadah ini disusun sedemikian rupa sehingga memungkinkan kita melakukan transformasi pada wadah tersebut. Sebagai pengembang, mendapatkan pemahaman menyeluruh tentang cara menggunakan nilai-nilai ini memungkinkan Anda melakukan transformasi data yang menantang dengan lebih mudah.

Nilai terstruktur memungkinkan kita menangani kompleksitas dengan membagi data menjadi potongan-potongan yang dapat dikelola. Setiap nilai terstruktur – baik itu list, record, tabel, atau fungsi – memiliki karakteristik dan aplikasinya sendiri yang unik, dan memahami kapan harus menggunakan masing-masing membantu dalam menulis kode yang efektif.

Saat kita mempelajari bab ini, Anda mungkin menemukan beberapa konsep yang sedikit lebih menantang daripada yang lain. Namun, dengan meluangkan waktu untuk memahami konsep-konsep mendasar ini, Anda akan meletakkan dasar yang kokoh yang memungkinkan Anda untuk menangani tugas-tugas yang lebih rumit dengan percaya diri di masa mendatang.

Bab ini bertujuan untuk memperjelas nilai-nilai terstruktur dan memberi Anda pemahaman yang komprehensif tentang peran dan penerapannya di Power Query M. Kita akan mempelajari secara spesifik list, record, tabel, dan fungsi, membahas apa itu, bagaimana cara kerjanya, dan pentingnya mereka dalam tugas-tugas transformasi data.

C. List (List)

Memulai penjelajahan kita tentang nilai terstruktur, pertama-tama kita akan menemukan list. Jadi, apa itu list?

1. Pengantar List

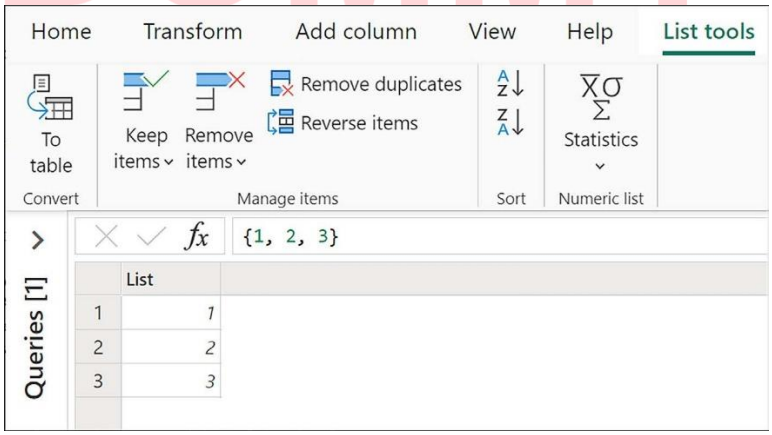
List berisi serangkaian nilai yang dipisahkan koma dari jenis apa pun. Itu termasuk nilai primitif (seperti teks, angka, tanggal, atau waktu) dan nilai terstruktur (list, record, atau tabel).

Jadi, mengapa list menarik minat Anda? Anda akan menemukan bahwa list digunakan secara luas di seluruh Power Query. Misalnya, memilih kolom mengembalikan nilainya dalam bentuk list. Demikian pula, ketika fungsi mengambil beberapa nilai sebagai input (seperti *List.Count*) atau mengembalikan beberapa nilai sebagai output (seperti *Table.ColumnNames*), Anda akan sering menemukan bahwa fungsi ini melibatkan list. Mari kita lihat bagaimana Anda dapat membuat list.

Membuat list adalah proses yang mudah. Anda dapat menentukan nilai list dengan melampirkan nilai dalam tanda kurung kurawal { }, yang secara formal dikenal sebagai inisialisasi list. Setiap nilai dalam list dipisahkan oleh koma. Misalnya, Anda dapat membuat list tiga nilai angka sebagai berikut:

```
{1, 2, 3}
```

List ini kemudian dapat dilihat, seperti yang ditunjukkan pada gambar berikut:



Gambar 2. 1 Inisialisasi list di Power Query menggunakan { }

List juga dapat berisi campuran nilai. Pernyataan berikut membuat list yang berisi angka, teks, dan nilai logika:

```
{1, "Hello", true}
```

Dengan cara yang sama, Anda dapat menyertakan nilai terstruktur lainnya di dalam list. Ekspresi berikut mengembalikan list yang berisi nilai teks, list, record, dan tabel:

```
{
  "a",
  {4,5,6},
  [Column1 = 1, Column2 = "v"],
  #table( { "ID", "Product" }, {{ 1, "Apple" }} )
}
```

Kita akan membahas catatan dan tabel di bab ini nanti. Jika suatu list tidak berisi item apa pun, maka list tersebut dianggap sebagai list kosong. Untuk mengembalikan list kosong, Anda dapat menulis:

```
{} // Output: empty list
```

Untuk memeriksa jumlah item dalam list, Anda dapat menggunakan fungsi seperti *List.Count*:

```
List.Count( {1, 2, 3} ) // Output: 3
List.Count( {} )        // Output: 0
```

Untuk mempelajari lebih lanjut tentang fungsi *List* yang tersedia dan tipe data input serta output yang diharapkan, kita sarankan untuk melihat ikhtisar fungsi *List* di https://powerquery.how/list-functions/.

2. List Operator

Saat kita memperdalam pemahaman kita tentang list, mari kita alihkan perhatian kita ke operator. Kita secara singkat memperkenalkan operator di Gbapter 4 dan memberikan gambaran umum tentang nilai mana yang mendukung operator mana. Bagian ini membahas operator yang bekerja dengan list.

List mendukung operator berikut:

Tabel 2. 1 Operator yang didukung oleh list

Operator	Notasi	Deskripsi
Equal	=	Memeriksa apakah dua list identik.
Not equal	<>	Memeriksa apakah dua list berbeda.
Concatenation	&	Menambahkan dua list.
Coalesce	??	Mengembalikan nilai bukan nol pertama dari dua nilai.

Equal (Setara)

Operator sama dengan (=) memeriksa apakah dua list identik, baik dari segi urutan kemunculannya maupun apakah nilai dalam list tersebut sama.

Saat membandingkan list berikut, operator mengembalikan *true* karena kedua list tersebut cocok secara sempurna:

```
{1, 2, 3} = {1, 2, 3}
```

Namun, penataan ulang item seperti yang ditunjukkan di bawah ini akan menghasilkan hasil yang *false*:

```
{1, 2, 3} = {3, 2, 1}
```

Ini menunjukkan bahwa urutan elemen penting saat membandingkan list. Operator sama dengan berfungsi, terlepas dari apakah list tersebut berisi nilai primitif atau terstruktur (seperti list bersarang). Misalnya:

```
{ {1,"a","b"}, 2, 3} = { {1,"a","b"}, 2, 3} // Output: true
```

Salah satu cara yang dapat kita gunakan adalah saat membandingkan nama kolom dalam dua tabel. Saat menambahkan tabel menggunakan fungsi *Table.Combine*, kolom dengan nama kolom yang sama akan digabungkan. Misalkan Anda ingin memverifikasi apakah nama kolom dalam dua tabel tersebut identik. Untuk mengambil nama kolom dalam tabel, Anda dapat menggunakan:

```
Table.ColumnNames( YourTable )
```

Seperti yang baru saja kita lihat, saat menguji apakah list itu sama, urutan nilai penting. Oleh karena itu, kita harus mengurutkan list sebelum membandingkannya. Anda dapat melakukannya dengan kode berikut:

```
let
  ColumnNamesTable1 =
    List.Sort( Table.ColumnNames( Table1 )),
  ColumnNamesTable2 =
    List.Sort( Table.ColumnNames( Table2 )),
  ColumnsAreEqual = ColumnNamesTable1 = ColumnNamesTable2
in
  ColumnsAreEqual
```

Output dari kueri ini adalah nilai Boolean yang menunjukkan apakah kedua list tersebut sama.

Not Equal (Tidak Setara)

Operator not equal melakukan hal yang sebaliknya dari operator equal. Operator ini memverifikasi apakah dua list berbeda. Contoh berikut sedikit berbeda dari yang sebelumnya:

```
{1, 2, 3} <> {3, 2, 1} // Output: true
```

Meskipun berisi unsur-unsur yang identik, perbedaan urutan list membuatnya tidak sama, sehingga outputnya benar.

Concatenate (Menggabungkan)

Operator penggabungan digunakan untuk menggabungkan dua list, pada dasarnya menambahkan satu list ke akhir list lainnya. Misalnya, Anda dapat menggabungkan dua list sebagai berikut:

```
{1, 2} & {4, 5} // Output: {1, 2, 3, 4}
```

Ini adalah cara yang sangat baik untuk menggabungkan list Anda. Daripada menggabungkan list secara manual, Anda juga dapat menggunakan

fungsi *List.Combine*. *List.Combine* mengharuskan Anda untuk menyediakan list yang akan digabungkan menjadi list lain; yaitu, dua list bagian dalam di dalam list bagian luar, seperti ini:

```
List.Combine( { {1, 2}, {4, 5} } ) // Output: {1, 2, 3, 4}
```

Penggabungan list berguna dalam berbagai skenario. Misalnya, Anda mungkin ingin memilih semua angka dan huruf dari suatu string. Daripada melakukannya secara manual, Anda dapat membuat serangkaian nilai dan menggabungkannya. Misalkan Anda memiliki string berikut:

```
"inv-1006-!**!-(act)"
```

Untuk mengekstrak hanya teks dan angka, Anda dapat membuat dua list nilai sebagai berikut:

```
{ "a".."z" } // Generates a list of lowercase letters
{ "0".."9" } // Generates a list of numbers
```

Kedua pernyataan ini menggunakan konstruksi AB, yang akan kita bahas lebih mendalam nanti di bab ini. Untuk menggabungkan list yang dihasilkan dan mengekstrak nilai dari string, Anda dapat menggunakan kode berikut:

```
let
  LowercaseLetters = { "a".."z" }
  Numbers = { "0".."9" }

  CombinedCharacters = LowercaseLetters & Numbers
  SelectCharacters =

    Text.Select( "inv-1006-!**!-(act)", CombinedCharacters )
in
  SelectCharacters
```

Operator & digunakan untuk menggabungkan list huruf kecil dan angka. *Text.Select* kemudian memfilter karakter yang diinginkan dari string yang diberikan, menghasilkan: inv1006act.

Coalesce (Bersatu)

Operator coalesce efektif untuk menangani nilai null dalam data. Operator ini ditempatkan di antara dua nilai: nilai utama (di sebelah kiri) dan nilai fallback (di sebelah kanan). Operator mengevaluasi nilai utama terlebih dahulu. Jika nilai ini bukan null, maka nilai ini akan dikembalikan sebagai hasilnya. Jika nilai utama adalah null, operator akan mengembalikan nilai fallback sebagai gantinya. Pertimbangkan contoh ini, yang mengembalikan {1, 2, 3}:

```
{ 1, 2, 3 } ?? null
```

Dalam kasus ini, karena list pertama tidak null, operator coalesce mempertahankannya. Namun, jika urutannya dibalik seperti berikut:

```
null ?? {1, 2, 3}
```

Ekspresi tersebut mengembalikan {1, 2, 3}. Perilaku ini menjamin hasil yang tidak nol, yang berguna dalam skenario dengan potensi nilai nol.

Untuk aplikasi praktis, bayangkan katalog produk dengan kolom Fitur yang mencantumkan fitur produk. Jika beberapa produk tidak memiliki list fitur tertentu, Anda dapat memberikan list default menggunakan kode ini:

```
each [Features] ?? {"Feature 1", " Feature 2"}
```

Dengan cara ini, menemukan null di kolom Features akan menghasilkan list fitur standar yang dikembalikan. Pendekatan ini menjamin keseragaman dalam deskripsi produk, menyediakan fitur default jika list tertentu tidak ada.

Sekarang kita tahu cara membuat list dan operator apa yang dapat bekerja pada list tersebut, mari kita lihat cara mudah membuat list nilai.

3. Metode Untuk Membuat List

Sejauh ini, Anda telah melihat dasar-dasar list dalam hal cara menggunakan operator pada list tersebut. Sekarang mari kita tinjau berbagai cara untuk membuat list.

Membuat List Menggunakan Inisialisasi List

Metode pertama untuk membuat list adalah dengan menggunakan inisialisasi list, seperti yang ditunjukkan sebelumnya. Kesederhanaan dalam membuat list secara manual merupakan daya tarik terbesarnya. Berikut ini cara menggabungkan berbagai nilai untuk membentuk list:

```
{ 1, true, #date( 2023, 12, 31 ) }
```

Membuat List Menggunakan Fungsi

Cara mudah lainnya untuk membuat list nilai adalah dengan menggunakan fungsi generator. Fungsi berikut menghasilkan list dari awal:

- *List.Dates*
- *List.DateTimeZones*
- *List.DateTimes*
- *List.Durations*
- *List.Numbers*
- *List.Times*

Fungsi-fungsi ini mengikuti pola serupa yang membutuhkan tiga unsur utama:

- Nilai awal untuk list.
- Jumlah nilai yang akan dibuat dalam list.
- Penambahan yang ditambahkan ke setiap nilai untuk setiap langkah.

Membuat list angka mudah dilakukan dengan fungsi *List.Numbers*. Dengan 1 sebagai nilai awal, mari buat 5 nilai, dan tambahkan masing-masing sebesar 1:

```
List.Numbers( 1, 5, 1 ) // Returns { 1, 2, 3, 4, 5 }
```

Dengan kenaikan 2 dan nilai awal serta hitungan yang sama, Anda memperoleh:

```
List.Numbers( 1, 5, 2 )  // Returns { 1, 3, 5, 7, 9 }
```

Keajaiban list juga berlaku untuk tanggal. Fungsi *List.Dates* memungkinkan Anda membuat list tanggal, seperti lima tanggal yang dimulai dari 31 Desember 2023, dengan menambahkan setiap tanggal satu hari:

```
List.Dates( #date(2023,12,31), 5, #duration(1,0,0,0) )
```

Jika Anda ingin membuat list angka acak, Anda dapat menggunakan *List.Random*. Misalnya, ekspresi berikut membuat list dengan 3 angka acak:

```
List.Random( 3 )
```

Ketika kita menjalankan kueri ini, output mengembalikan list ini:

```
{0.42290808419832404,0.76420168008850964,0.75750946894172089}
```

Sebagai alternatif, ada banyak fungsi terdefinisi lainnya dalam bahasa M yang mengembalikan list, misalnya:

- **Nama bidang**: Anda dapat menemukan fungsi yang mengembalikan nama bidang yang ditemukan dalam tabel (*Table.ColumnNames* atau *Table.ColumnsOfType*) atau dalam record (*Record.FieldNames*).
- **Membentuk ulang nilai terstruktur**: Fungsi lain mengubah nilai terstruktur menjadi list. Anda dapat membentuk ulang tabel menjadi list dengan fungsi seperti Table.ToColumns, *Table.ToRecord*, *Table.ToRows*, atau *Table.ToList* atau membentuk ulang record dengan *Record.ToList*.
- **Memisahkan teks**: Saat bekerja dengan string, fungsi *Text.Split* dan *Text.SplitAny* dapat membagi string berdasarkan pembatas yang ditentukan dan mengembalikan hasilnya sebagai list.

Selain yang umum ini, dua fungsi hebat lainnya membuat list dengan logika khusus. Fungsi ini adalah *List.Accumulate* dan *List.Generate*. Kita akan membahas keduanya di Bab 13, Iterasi dan Rekursi.

Merujuk ke Kolom Tabel

Cara elegan lain untuk mengembalikan list adalah dengan merujuk ke kolom tabel. Bayangkan Anda memiliki tabel Produk seperti yang ditunjukkan pada Gambar 6.2, dengan **Product** sebagai kolom pertama:

	Product	Size	Category
1	Shoe	Medium	Clothes
2	Hat	Large	Clothes
3	Shirt	Medium	Clothes
4	Belt	Extra Small	Accessories

Gambar 2. 2 Dataset dengan tiga kolom

Jika langkah Power Query yang mengembalikan tabel ini disebut Sumber, Anda dapat mengembalikan nilai kolom **Product** dalam bentuk list dengan menggunakan ekspresi berikut:

```
Source[Product] // Output: {"Shoe", "Hat", "Shirt", "Belt"}
```

Proses ini, yang dikenal sebagai **pemilihan bidang (field-selection)**, akan dibahas lebih lanjut di bagian berikutnya tentang catatan dan tabel.

Menggunakan Bentuk a..b

Fungsionalitas yang berguna untuk list adalah kemampuan untuk membuat list item berurutan menggunakan bentuk a..b. Ini dapat sangat berguna ketika Anda ingin membuat list angka, huruf, atau karakter khusus yang berurutan.

Dalam bentuk yang paling sederhana, sintaksis a..b menghasilkan serangkaian nilai dalam rentang angka tertentu. Misalnya, untuk membuat list angka dari 1 hingga 10, Anda dapat menggunakan:

```
{1..10} // Output: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Ini memberi tahu Power Query untuk membuat list bilangan bulat, dimulai dari 1 dan berakhir pada 10. List tersebut dapat berguna untuk tugas seperti perulangan, pengulangan, pembuatan data, atau bahkan pembuatan kalender.

Anda juga dapat menggunakan konstruksi ini untuk membuat serangkaian nilai yang tidak berkesinambungan. Berikut adalah dua contoh:

```
{ 1..3, 11..13 }    // Output: { 1, 2, 3, 11, 12, 13}
{ -1..1, 5, 8..10 } // Output: { -1, 0, 1, 5, 8, 9, 10 }
```

Namun, penting untuk dicatat bahwa fungsi ini dirancang untuk menambah list. Jika Anda mencoba membuat list yang mengurangi dengan sintaks ini, Power Query akan mengembalikan list kosong:

```
{10..1} // Output: {}
```

Untuk membuat list menurun dari 10 hingga 1, ada dua metode mudah yang melibatkan penggunaan fungsi *List.Reverse* atau *List.Numbers*:

```
List.Reverse( {1..10} )
List.Numbers( 10, 10, -1 )
```

Sama seperti Anda membuat urutan angka, Anda juga dapat membuat urutan huruf. Misalnya, untuk membuat list dengan 6 huruf pertama alfabet, Anda dapat menggunakan:

```
{"a".."f"} // Output: {"a", "b", "c", "d", "e", "f}
```

Namun perlu diingat, bahasa M peka terhadap huruf besar/kecil. Oleh karena itu, mengubah huruf besar/kecil menjadi {"A".."F"} akan menghasilkan list 6 huruf kapital pertama dalam alfabet:

```
{"A".."F"} // Output: {"A", "B", "C", "D", "E", "F}
```

Anda mungkin bertanya-tanya apakah ada hal lain pada formulir a..b. Memang ada. Untuk memahami cara kerja fitur ini, penting untuk memahami apa yang terjadi di balik layar.

Saat Power Query membuat list ini, ia menggunakan karakter Unicode, baik Anda menentukan angka, huruf, atau huruf kapital.

Pertimbangkan contoh berikut. Jika Anda membuat nilai mulai dari huruf "Y" hingga "a", Anda akan mendapatkan:

```
{ "Y" .. "a"}
// Output: { "Y", "Z", "[", "\", "]", "^", "_", "`", "a" }
```

Apa yang terjadi di sini? Huruf "Y" sesuai dengan desimal Unicode 89, sedangkan "a" sama dengan 98. Anda dapat memverifikasi ini dengan menulis:

```
Character.ToNumber( "Y" ) // Output: 89
Character.ToNumber( "a" ) // Output: 97
```

Jika Anda hanya memiliki desimal Unicode, Anda dapat melakukan kebalikannya dengan menggunakan:

```
Character.FromNumber( 97 ) // Output: "a"
```

Oleh karena itu, untuk memahami sepenuhnya urutan pembuatan karakter, ada baiknya merujuk ke Standar Unicode: https://en.wikipedia.org/wiki/List_of_Unicode_characters#Latin_script.

4. Mengakses Item Dalam List

Setelah Anda memahami cara membuat list, langkah berikutnya adalah mempelajari cara mengakses nilainya, sebuah proses yang dikenal sebagai **pemilihan (Selection)**.

Bahasa M menggunakan tanda kurung kurawal yang sama yang digunakan dalam mendefinisikan list untuk memilih item, sebuah fakta yang sering membingungkan. Untuk mengekstrak item dari list berdasarkan posisinya, terapkan nomor indeks di dalam tanda kurung kurawal setelah list. Ingatlah hal berikut:

- List dalam bahasa M Power Query mengikuti sistem pengindeksan berbasis nol, yang berarti item pertama dalam list Anda memiliki indeks 0.
- Mencoba memilih item pada posisi yang tidak ada akan mengakibatkan kesalahan. Mari kita ilustrasikan kedua poin tersebut dengan beberapa contoh.

Mengakses Nilai List Berdasarkan Indeks

Misalkan kita memiliki list yang bertambah dalam langkah 10, mulai dari 10 hingga 50: {10, 20, 30, 40, 50}. Mengakses suatu nilai semudah menentukan indeksnya. Misalnya, untuk mendapatkan elemen pertama, Anda akan menggunakan *myList{0}*, yang menghasilkan 10. Demikian pula, *myList{4}* akan mengembalikan 50, elemen terakhir list. Berikut ini cara Anda menggunakan item-access-expression dalam kode:

```
{10, 20, 30, 40, 50}{0} // Output: 10
{10, 20, 30, 40, 50}{4} // Output: 50
```

Tanda kurung kurawal memiliki peran ganda di sini. Set awal menyusun list, sedangkan set kedua mengidentifikasi indeks nilai yang ingin kita ekstrak. Perlu diingat bahwa pernyataan pemilihan selalu mewakili satu nilai numerik – indeks.

Kita dapat memperoleh hasil yang sama dengan fungsi *List.Range*. Pernyataan berikut mengambil nilai kedua dari list:

```
List.Range( { 1, 2, 3}, 1, // Output:
```

Konsep pemilihan bahkan lebih jauh lagi. Saat Anda bekerja dengan list bersarang (list dalam list), Anda dapat memilih nilai dari beberapa lapisan. Pertimbangkan list yang memiliki list sebagai item pertama. Jika Anda ingin mengembalikan item pertama, yang merupakan list itu sendiri, Anda akan menggunakan kode berikut:

```
{{1,2,3}, "a", "b"}{0}    // Output: {1, 2, 3}
```

Namun bagaimana jika kita ingin mengekstrak nilai kedua dalam list bertingkat tersebut? Nah, tidak ada yang menghentikan Anda untuk memilih nilai tersebut dengan menambahkan serangkaian kurung kurawal lainnya:

```
{{1,2,3}, "a", "b"}{0}{1} // Output: 2
```

{1} yang ditambahkan di sini menandakan bahwa kita memilih nilai kedua dari list pertama dalam list induk kita. Ingat, kita menggunakan pengindeksan berbasis nol.

Menangani Posisi Indeks yang Tidak Ada

Sekarang, mari kita telusuri apa yang terjadi ketika kita mencoba mengambil item pada posisi indeks yang tidak ada dalam list kita. Bayangkan kita memiliki beberapa kode yang mencoba mengakses nilai keenam dalam list yang terdiri dari lima elemen:

```
{10, 20, 30, 40, 50}{5}
```

Power Query akan mengembalikan kesalahan ekspresi yang menyatakan: *"There weren't enough elements in the enumeration to complete the operation"*.

Untuk mencegah kesalahan ini, bahasa M menawarkan operator yang menginstruksikan pemilihan item opsional. Ini dapat dicapai dengan menambahkan tanda tanya di akhir ekspresi yang mencoba mengakses list. Jika posisi indeks tidak ada dalam list, maka akan mengembalikan nilai null:

```
{10, 20, 30, 40, 50}{4}? // Output: 50
{10, 20, 30, 40, 50}{5} // Output: Expression error
{10, 20, 30, 40, 50}{5}? // Output: null
```

Jadi, sekarang Anda tahu cara membuat list dan mengakses nilainya, kapan list berguna?

5. Operasi Umum Dengan List

List memainkan peran penting dalam berbagai operasi. Untuk mengilustrasikan penggunaannya, mari kita bahas beberapa contoh konkret yang menunjukkan penerapan list dalam tugas transformasi data Anda. Misalkan Anda bekerja dengan tabel yang terdiri dari tiga kolom.

Anda ingin menentukan nilai maksimum di seluruh kolom ini. Anda dapat memberikan list ke fungsi *List.Max* untuk mencapainya:

```
List.Max( { [Column1], [Column2], [Column3] } )
```

Itu mengembalikan:

```
Table.AddColumn(Source, "Max", each List.Max( { [Column1], [Column2], [Column3] } ))
```

	Column1	Column2	Column3	Max
1	5	20	12	5
2	26	8	10	8
3	14	20	90	90

Gambar 2. 3 Anda dapat menggunakan List.Max untuk mengembalikan jumlah maksimum beberapa kolom

Dalam kode ini, list tersebut mengelilingi kolom yang Anda minati, yang memungkinkan Anda untuk mengekstrak nilai maksimum di semua kolom tersebut.

Sekarang, bagaimana jika Anda ingin mengevaluasi apakah suatu nilai di kolom **Color** cocok dengan salah satu dari tiga warna tertentu? Bahasa M Power Query tidak memiliki operator IN tradisional seperti yang terlihat di SQL atau DAX. Namun, Anda dapat mereplikasinya dengan menyediakan list ke fungsi *List.Contains*:

```
List.Contains( { "Blue", "Green", "orange" }, [Color] )
```

Hasilnya terlihat sebagai berikut:

```
Table.AddColumn(Source, "IsMyColor", each List.Contains( { "Blue", "Green", "Orange" }, [Color] ))
```

	Product ID	Color	IsMyColor
1	1	Orange	TRUE
2	2	Pink	FALSE
3	3	Blue	TRUE
4	4	Green	TRUE
5	5	White	FALSE

Gambar 2. 4 *List.Contains* adalah alternatif yang bagus untuk meniru operator *IN*

Kode di atas menguji apakah kolom [*Color*] berisi setidaknya satu warna yang ditentukan. Ini adalah contoh bagus tentang bagaimana list memfasilitasi logika kondisional.

Pertimbangkan skenario lain saat Anda berhadapan dengan string: ISBN: 978-3-16-148410-0. Anda hanya ingin mengekstrak angka dan tanda hubung. Anda dapat menangani ini secara efektif dengan menggunakan formulir a..b untuk menghasilkan karakter yang dibutuhkan:

```
Text.Select( "ISBN: 978-3-16-148410-0", {"0".."9","-"} )
```

Dalam contoh ini, list membantu menentukan set karakter yang ingin kita pertahankan, dengan hanya mengekstrak teks yang diinginkan.

Kasus penggunaan lain untuk list adalah untuk menyediakan nilai guna membuat tabel. Misalnya, Anda dapat membuat tabel dengan satu kolom yang berisi tiga nilai sebagai berikut:

```
Table.FromList( { "Apple", "Pear", "Banana" } )
```

Terakhir, fungsi *Table.Group* adalah contoh lain di mana list digunakan secara luas. Fungsi ini menggunakan list untuk menentukan kolom untuk pengelompokan dan untuk mendefinisikan agregasi:

```
Table.Group(    Source,
   {"Column1", "Column2"},
   {{"Count", each Table.RowCount(_), Int64.Type}} )
```

Dalam kode ini, list digunakan untuk mengelompokkan data (berdasarkan *Column1* dan *Column2*) dan untuk memberikan detail guna membuat kolom agregasi dengan jumlah baris. Ini menghasilkan hal berikut:

```
Table.Group(    Source,
    {"Column1", "Column2"},
    {{"Count", each Table.RowCount(_), Int64.Type}})
```

	Column1	Column2	Count
1	A	Ecuador	3
2	B	Taiwan	3
3	B	Siberia	1

Gambar 2. 5 Fungsi seperti Table.Group memerlukan list sebagai input untuk argumennya

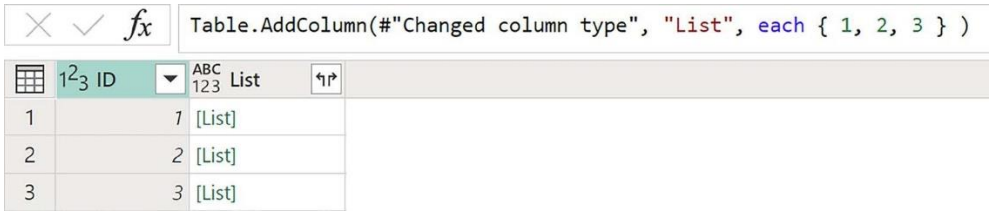
Contoh-contoh ini menunjukkan peran serbaguna list dalam bahasa M. Memahami cara memanipulasi dan menggunakan list merupakan keterampilan penting untuk menjadi pembuat kode M yang lebih baik.

Seperti yang ditunjukkan dalam bab ini, list dapat berisi nilai apa pun. Sekarang, opsi apa yang Anda miliki saat menentukan tipe data untuk list? Itulah yang akan kita bahas selanjutnya.

6. Menetapkan Tipe Data ke Sebuah List

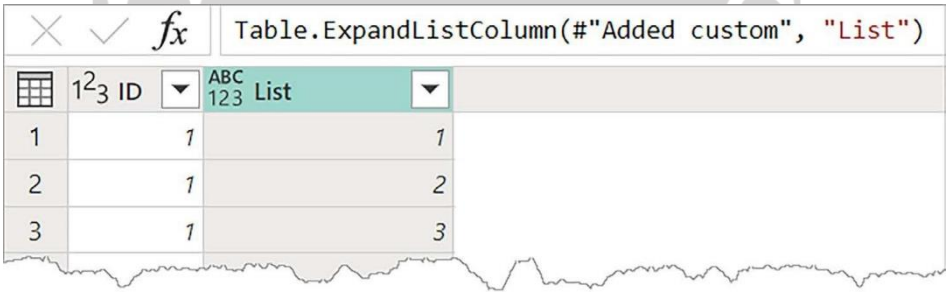
Seperti yang disebutkan dalam Bab 5, mengenai tipe data, setiap nilai memiliki tipe data yang terkait dengannya. Hal ini berlaku untuk nilai primitif, tetapi juga untuk list dan isinya. Mari kita kembangkan pengetahuan yang diperoleh dalam bab ini sejauh ini dan gali lebih dalam tipe data yang termasuk dalam list.

Bila list berisi nilai yang termasuk dalam satu tipe data, Anda dapat menentukan tipe data tersebut saat membuat kolom. Sebagai contoh, mari kita ingat kembali langkah-langkah untuk membuat list dari awal bab ini. Setelah Anda memiliki list ini, Anda dapat menggabungkannya ke dalam kolom kustom menggunakan fungsi *Table.AddColumn*, seperti yang ditunjukkan di bawah ini:



Gambar 2. 6 Menambahkan nilai list ke dalam kolom tanpa menetapkan tipe data

Pada kode di atas, fungsi *Table.AddColumns* tidak menerima instruksi tipe data tertentu, sehingga membuat kolom **List** dengan tipe *any*. Akibatnya, saat Anda memperluas nilai kolom, nilai tersebut tetap bertipe any, seperti yang ditunjukkan pada gambar berikut:

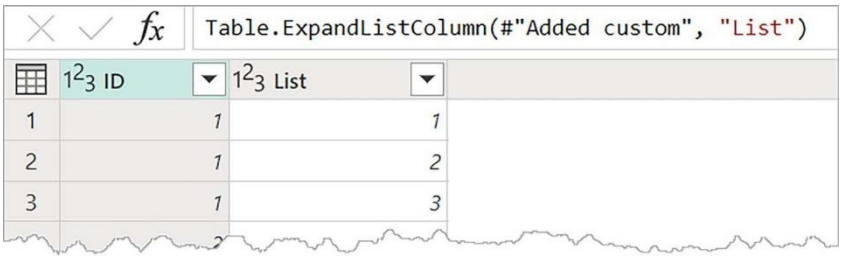


Gambar 2. 7 Kolom tetap bertipe "apa pun" saat diperluas tanpa tipe data yang ditetapkan sebelumnya

Kabar baiknya adalah Anda dapat menetapkan tipe data ke nilai list Anda. Untuk mencapainya, Anda perlu mengisi argumen *columnType* opsional dari fungsi *Table.AddColumn*. Dengan list, seperti nilai lainnya, Anda dapat menentukan tipe menggunakan nilai tipe data apa pun yang tersedia. Satu-satunya perbedaan untuk list adalah Anda harus membungkus tipe tersebut dalam tanda kurung kurawal. Jadi, jika Anda ingin menentukan list dengan bilangan bulat, Anda akan menulis:

Gambar 2. 8 Menambahkan nilai list ke dalam kolom dengan tipe data yang ditetapkan

Saat Anda memperluas kolom yang baru dibuat ini, kolom tersebut akan muncul sebagai kolom dengan tipe Bilangan Bulat (*Whole Number*). Menentukan tipe data akan mencegah Anda melakukan langkah terpisah untuk mengubah tipe data kolom, seperti yang ditunjukkan pada gambar berikut:



Gambar 2. 9 Saat diperluas, kolom *List* menyertakan tipe data yang ditetapkan

Cuplikan layar sebelumnya didasarkan pada penetapan tipe data ke list selama operasi *Table.AddColumn*. Namun, bagaimana jika Anda memiliki tabel yang sudah berisi list yang ingin Anda tetapkan tipe datanya?

Misalkan kita memiliki situasi yang sama seperti sebelumnya, tetapi kita tidak menetapkan tipe data untuk kolom List. Anda juga dapat mengonversi tipe kolom dengan nilai list. Misalnya, untuk mengubah kolom yang ada menjadi list nilai bertipe Bilangan Bulat, Anda dapat menggunakan fungsi *Table.TransformColumns*. Kode berikut mencapai hal ini:

```
Table.TransformColumns(
  Source,

  {{"List", each _ ??_, type {Int64.Type}}} )
```

Yang penting untuk diperhatikan di sini adalah bahwa fungsi *Table.TransformColumn* mengharuskan Anda untuk melakukan operasi pada list yang mendasarinya. Gagal melakukannya menyebabkan *Table.TransformColumns* tidak menetapkan tipe data. Dalam contoh di atas, kita menerapkan operator coalesce ke nilai yang mendasarinya. Karena operasi itu, fungsi tersebut juga berhasil menetapkan tipe data.

Sebagai kesimpulan, manfaat menetapkan tipe ke list adalah bahwa ketika Anda memperluas list ke baris baru, list tersebut akan memiliki tipe data yang ditetapkan ke kolom. Itu menghilangkan kebutuhan untuk menambahkan langkah perubahan tipe terpisah untuk menentukan tipe data.

Sekarang setelah kita menjelajahi list, selanjutnya mari kita jelajahi tipe data terstruktur lainnya, yaitu record.

D. Record

Berikutnya dalam perjalanan kita melalui nilai terstruktur dalam bahasa M Power Query, kita menemukan record.

1. Pengantar Record

Catatan adalah list nilai yang diberi nama. Sementara list adalah kumpulan nilai yang sederhana dan teratur, catatan mengaitkan nama bidang atau kunci ke setiap nilai. Ini mengarah ke struktur data yang lebih kompleks, namun terorganisir.

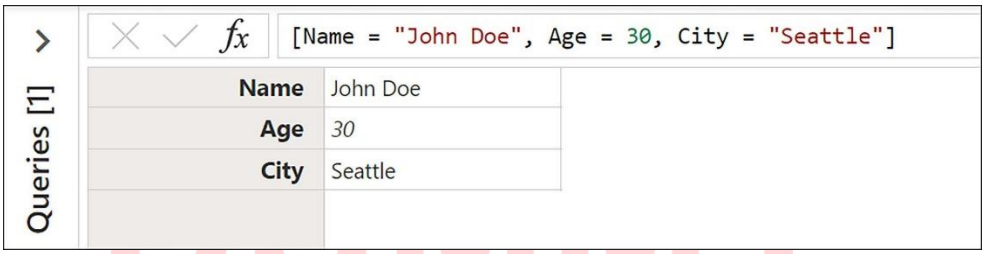
Anda dapat menganggap catatan sebagai satu baris dalam tabel, di mana setiap bidang dalam baris memiliki nama kolom dan nilai yang unik. Sama seperti list, catatan dapat berisi jenis nilai apa pun, baik primitif (seperti teks, angka, atau tanggal), atau terstruktur (seperti list, catatan, atau tabel).

Anda dapat membuat record dengan operator *record initialization*. Ini melibatkan penentuan pasangan kunci dan nilai dalam tanda kurung siku [], yang juga dikenal sebagai inisialisasi record. Yang penting di sini adalah mengingat bahwa:

- Setiap record diapit oleh tanda kurung siku.
- Kunci (nama bidang) diikuti oleh tanda sama dengan (=). Record memungkinkan Anda memberikan nama bidang tanpa tanda kutip.
- Nilai bidang mengikuti tanda sama dengan.
- Pasangan kunci dan nilai dipisahkan oleh koma.

Perhatikan contoh pada gambar berikut. Kode ini membuat record dengan detail seseorang:

```
[ Name = "John Doe", Age = 30, City = "Seattle" ]
```



Gambar 2. 10 Inisialisasi record di Power Query dengan menggunakan []

Dalam contoh sebelumnya, *Name*, *Age*, dan *City* adalah kunci (nama kolom) dan *John Doe*, *30*, dan *Seattle* adalah nilai kolom yang sesuai. Perhatikan hal berikut saat membuat record:

- Kombinasi kunci dan nilai dipisahkan dengan koma.
- Setiap nama bidang yang ditentukan harus unik berdasarkan perbandingan peka huruf besar/kecil.

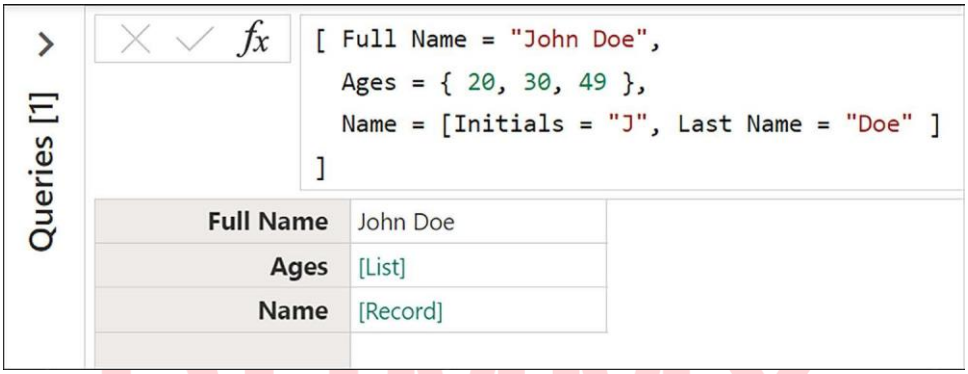
Kesalahan akan muncul jika nama bidang yang sama diulang, seperti pada record yang salah ini:

```
[ Name = "John Doe", Age = 30, Age = 31 ]
```

Pesan kesalahan menyatakan Nama *'Age'* didefinisikan *than* satu kali karena beberapa definisi bidang *Age*. Selalu pastikan untuk memberikan nama bidang yang unik pada catatan Anda.

Catatan dapat berisi nilai primitif atau terstruktur lainnya. Berikut ini contoh catatan dengan list dan catatan bersarang:

```
[ Full Name = "John Doe",
  Ages = { 20, 30, 49 },

  Name = [ Initials = "J", Last Name = "Doe" ]

]
```



Gambar 2. 11 Sebuah record dapat berisi nilai primitif dan terstruktur

Anda dapat mengembalikan record kosong sebagai berikut:

```
[ ] // Output: empty record
```

Untuk menghitung jumlah bidang dalam suatu record, Anda dapat menggunakan fungsi *Record.FieldCount*:

```
Record.FieldCount( [ Name = "John Doe", Age = 30 ] ) // 2
Record.FieldCount( [] )                              // 0
```

Selain menggunakan fungsi, Anda juga dapat menggunakan operator untuk bekerja dengan record, yang akan kita bahas selanjutnya.

2. Operator *Record*

Operator record menyediakan fungsi untuk memanipulasi dan membandingkan record, sama seperti yang mereka lakukan untuk list. Mari kita bahas beberapa operator yang berguna untuk bekerja dengan record.

Tabel 2. 2 Operator yang digunakan dengan records

Operator	Notation	Deskripsi
Equal	=	Memeriksa apakah dua record identik.
Not equal	<>	Memeriksa apakah dua record berbeda.
Concatenation	&	Menambahkan dua catatan.
Coalesce	??	Mengembalikan nilai bukan nol pertama dari dua nilai.

Equal

Operator sama dengan (=) digunakan untuk mengevaluasi apakah dua record identik. Bahasa M menganggap record sama jika:

- Jumlah total nama bidang identik.
- Setiap nama bidang ada di kedua record.
- Nama bidang yang sesuai memiliki nilai identik dengan mempertimbangkan nama bidang dan nilai terkaitnya.

Misalnya, pernyataan berikut mengembalikan nilai *true* karena kedua record identik:

```
[ Name = "John", Age = 32 ] = [ Name = "John", Age = 32 ]
```

Anda dapat dengan bebas memindahkan nama-nama bidang dalam record. Saat menguji kesamaan, perbandingan akan menghasilkan true jika kumpulan nama-nama bidang dan nilai-nilai identik dalam setiap record, apa pun urutannya. Pernyataan berikut akan menghasilkan *true*:

```
[ Name = "John", Age = 32 ] = [ Age = 32, Name = "John" ]
```

Namun jika ada perbedaan pada nama kolom atau nilai, hasilnya akan salah. Misalnya:

```
[ Name = "John", Age = 32 ] = [ Name = "Jane", Age = 32 ]
```

Dalam kasus ini, kolom Nama dari dua record berbeda, yang menyebabkan *false* dikembalikan. Ingat, perbandingan tidak bergantung pada urutan dan peka huruf besar/kecil untuk nama kolom dan nilai.

Jadi, bagaimana Anda dapat menggunakan ini? Pertimbangkan situasi saat Anda perlu menentukan apakah list record identik, tetapi urutan kolom dalam record ini bervariasi. Kesalahpahaman umum mungkin membuat Anda percaya bahwa menyelaraskan kolom dalam urutan yang seragam diperlukan. Namun, Anda baru saja mempelajari bahwa kesetaraan record tidak

bergantung pada urutan, jadi langkah tambahan ini tidak diperlukan. Pertimbangkan contoh berikut:

```
let
  RecordsList =
    { [ID = 1, Product = "Widget"],
      [Product = "Widget", ID = 1],
      [ID = 1, Product = "Widget"]  },
    UniqueRecords = List.Distinct( RecordsList ),
    CountRecords = List.Count( UniqueRecords )
in
    CountRecords
```

Dalam skrip ini, kita mulai dengan list record (*RecordsList*) yang urutan bidangnya tidak konsisten di semua record. Dengan menerapkan fungsi List.*Distinct*, kita menghilangkan semua record duplikat. Langkah terakhir, *CountRecords*, menghitung jumlah total record unik yang tersisa dalam list. Hasil operasi ini adalah 1, yang menunjukkan bahwa urutan bidang dalam record tidak relevan saat menentukan kesetaraan record.

Not Equal

Operator not equal (<>) berfungsi sebagai kebalikan logis dari operator equal. Operator ini mengembalikan true jika data tidak identik dan false jika tidak:

```
[ Name = "John", Age = 32 ] <> [ Name = "Jane", Age = 32 ]
```

Dalam kasus ini, hasilnya benar karena bidang Nama berbeda antara kedua record.

Concatenation (*Rangkaian*)

Operator penggabungan berguna untuk menggabungkan data. Operator ini memungkinkan Anda menggabungkan dua data menjadi satu. Misalnya, kode berikut mengembalikan: *[Name = "John", Age = 32, Gender = "Male"]*

```
[Name = "John", Age = 32 ] & [ Gender = "Male"]
```

Jika terjadi tumpang tindih antar bidang, operasi penggabungan menggunakan bidang dari record di sebelah kanan untuk menimpa nilai bidang dalam record di sebelah kiri. Dengan mengingat hal itu, ekspresi berikut mengembalikan [*Name = "John", Age = 45*]:

```
[ Name = "John", Age = 32 ] & [ Age = 45 ]
```

Anda dapat memperoleh hasil yang sama dengan fungsi *Record.Combine* dengan menyediakan record yang akan digabungkan sebagai list:

```
Record.Combine(
  { [ Name = "John", Age = 32 ],
    [ Age = 45]
  } )
```

Jadi, apa saja kasus penggunaan praktis untuk menggabungkan nilai record? Mari kita pertimbangkan skenario saat Anda perlu mengakses data dari API. Proses ini biasanya dimulai dengan panggilan API untuk mengambil token otorisasi. Untuk permintaan ini, diperlukan informasi header tertentu, yang harus disediakan sebagai record.

Berikut ini contoh yang hanya memperlihatkan bagian header dari kode yang diperlukan untuk permintaan otorisasi awal:

```
MyHeaders =
[ #"Content-Type" = "application/json",
  #"Api-Client-Identifier" = "ccurr",
  #"Api-Client-Token" = "11d8373-9y3f-9agw-piwr" ]
```

Setelah menerima token autentikasi, panggilan API berikutnya biasanya dilakukan. Panggilan ini sering kali memerlukan informasi header yang sama. Karena merupakan praktik yang baik untuk tidak mengulanginya, kode untuk panggilan API merujuk ke variabel *MyHeaders* beberapa kali. Untuk permintaan kedua, kita hanya perlu memperkaya catatan dengan token otorisasi yang baru diterima. Berikut tampilannya:

```
Json.Document(
  Web.Contents("https://myAPI.product.com/api/v4/products/",

    [ Headers =  MyHeaders & [ Authorization = Token ] ]

  ) )
```

Pada baris 3, operator penggabungan menggabungkan record *MyHeaders* dengan record yang berisi token otorisasi. Contoh ini menggambarkan bagaimana penggabungan record dapat berguna saat menyediakan record opsi ke fungsi. Penjelasan logika panggilan API dari contoh ini berada di luar cakupan buku ini. Namun, jika Anda tertarik, dalam Bab 13, Rekursi dan iterasi, kita membahas contoh yang lebih mendasar tentang cara memanggil API.

Coalesce

Sama seperti nilai *list*, operator coalesce (??) berfungsi untuk record. Penggunaannya melibatkan penempatan operator di antara dua nilai: nilai utama (di sebelah kiri) dan nilai fallback (di sebelah kanan). Operator pertama-tama mengevaluasi nilai utama. Jika nilai ini bukan null, maka akan dikembalikan. Jika tidak, maka akan dipilih nilai fallback. Pertimbangkan contoh berikut:

```
null ?? [ Age = 13, Name = "Marc" ]
```

Di sini, operator coalesce memeriksa nilai utama. Karena nilainya *null*, operator mengembalikan catatan fallback [*Age = 13*, *Name = "Marc"*]. Meskipun Anda mungkin tidak sering melihat operator coalesce digunakan dengan catatan, penggunaannya menjadi lebih jelas saat menggunakannya dalam fungsi.

Bayangkan sebuah fungsi yang dirancang untuk menghitung jumlah kolom dalam sebuah catatan:

```
Record.FieldCount( [ Age = 13, Name = "Marc" ]  )
```

Fungsi ini mengembalikan 2, yang menunjukkan dua kolom dalam record. Sekarang, mari kita perluas contoh ini ke kolom yang berisi record dan nilai null, yang bertujuan untuk menghitung kolom di setiap baris. Untuk baris dengan nilai null (yang menunjukkan tidak ada kolom), Anda ingin mengembalikan 0. Namun, kode di bawah ini tidak memenuhi persyaratan tersebut:

```
Record.FieldCount( null )
```

Fungsi *Record.FieldCount* tidak menerima nilai null. Sebaliknya, fungsi ini akan mengembalikan kesalahan berikut:

```
Expression.Error: We cannot convert the value null to type Record.
```

Untuk mengatasi hal ini, operator coalesce menjadi berguna. Asumsikan record dan nilai null berada dalam kolom bernama *myRecords*. Untuk menangani nilai null tanpa kesalahan, masukkan operator coalesce sebagai berikut:

```
Record.FieldCount( [myRecords] ?? [ ] )
```

Dengan pendekatan ini, saat fungsi menemukan nilai null dalam kolom *myRecords*, fungsi tersebut akan secara default menampilkan record kosong. *Record.FieldCount* kemudian dapat menghitung kolom dalam record kosong ini secara akurat, dan mengembalikan nilai 0 untuk nilai *null* dengan benar. Hal ini mencegah terjadinya kesalahan di masa mendatang.

3. Metode Untuk Membuat Record

Bagian ini akan memberi Anda metode untuk membuat dan mengambil record, masing-masing berguna untuk skenario tertentu.

Membuat record Menggunakan Inisialisasi record

Membuat record secara manual sering kali merupakan cara yang paling mudah. Sesuai dengan apa yang telah kita pelajari di bagian sebelumnya,

operator *record initialization* memungkinkan kita membuat record dalam waktu singkat. Perhatikan contoh berikut:

```
[ Name = "John Doe", Age = 30, City = "Seattle" ]
```

Begitu saja, kita punya record nya.

Membuat record menggunakan fungsi

Selain pembuatan manual, Anda juga dapat membuat record menggunakan fungsi tertentu. Pertimbangkan fungsi *Record.FromList*. Fungsi praktis ini memungkinkan kita membuat record dari nilai dan bidang yang diberikan seperti berikut:

```
Record.FromList(
  {"John Doe", 30, "Seattle" },
  { "Name", "Age", "City"    } )
```

Fungsi *Record.FromTable* menawarkan alternatif lain untuk mengubah tabel menjadi record:

```
  Record.FromTable(
#table( { "Name", "Value"    },
 { { "Name", "John Doe" },
   { "Age",  30         },
   { "City", "Seattle"  } }
      ))
```

Hasil operasi ini identik dengan potongan kode pertama. Namun, sintaksnya sedikit berbeda.

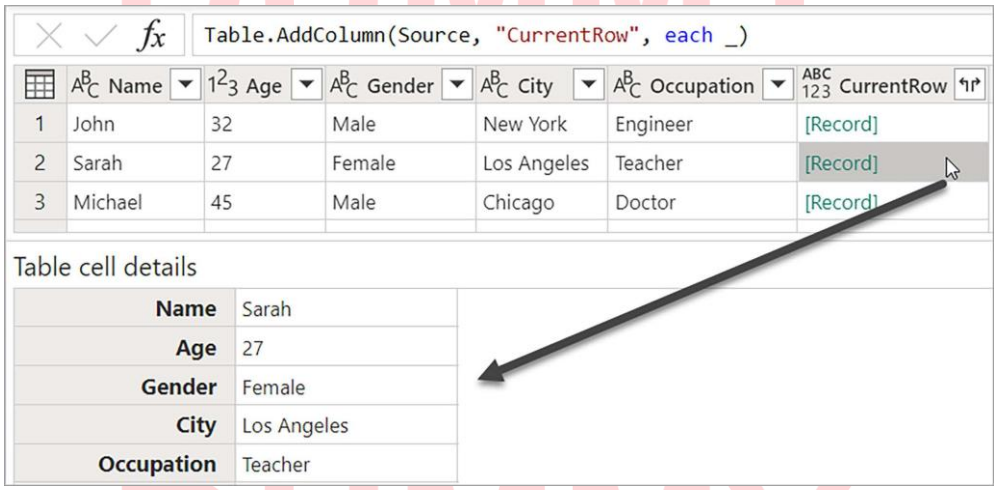
Mengambil Record Dengan Mereferensikan Baris Tabel

Pendekatan ketiga untuk memperoleh record adalah dengan merujuk ke baris tabel, yang secara formal dikenal sebagai **field selection**. Misalnya, jika

Anda memiliki tabel bernama Sumber, baris pertama tabel ini dapat dikembalikan sebagai record dengan ekspresi:

```
Source{0}
```

Alternatifnya, saat Anda menggunakan fungsi *Table.AddColumn*, Anda dapat merujuk ke baris tabel dengan menulis garis bawah (_). Setiap sel di kolom baru akan berisi record yang berisi nilai baris saat ini. Gambar berikut mengilustrasikan hal ini:



Gambar 2. 12 Garis bawah merujuk ke baris saat ini dalam tabel

Tujuan dari bagian ini hanya untuk memperkenalkan pemilihan bidang sebagai metode ketiga. Kita akan mengeksplorasi tabel dan karakteristiknya secara lebih mendalam di bagian berikut.

Power Query memiliki berbagai metode dan fungsi untuk menyelesaikan tugas yang sama. Pilihan di antara metode dan fungsi ini sangat bergantung pada skenario yang ada. Pembuatan record secara manual, menggunakan tanda kurung siku, biasanya merupakan rute yang paling mudah. Namun, dalam situasi saat Anda merujuk ke list atau nilai kolom, fungsi seperti *Record.FromList* dapat berguna. Pemahaman terhadap berbagai metode ini akan memandu Anda ke pilihan yang paling efektif untuk membuat record Anda.

Dengan pemahaman baru tentang pembuatan record ini, mari kita alihkan perhatian kita ke operator.

4. Mengakses Bidang Dalam Record

Catatan dapat berisi satu atau beberapa bidang dan nilai. Untuk mengaksesnya, diperlukan pemahaman tentang dua metode dasar—pemilihan bidang dan proyeksi catatan.

Tabel 2. 3 Operator yang digunakan untuk mengakses bidang dalam record

Operator	Notation	Description
Field selection	`[ A ]`	Selects a field from a record.
Record projection	`[ [A], [B] ]`	Selects multiple fields from a record.

Field Selection

Untuk mengakses satu bidang dalam sebuah record, Anda dapat merujuk kunci yang terkait dengan nilai tersebut. Ini secara formal disebut sebagai **field selection**. Misalnya, jika Anda ingin mengembalikan nilai dari bidang Nama dari sebuah record, Anda akan menulis:

```
[Name = "John Doe", Age = 30, City = "Seattle"][Name]
```

Perhatikan penggunaan tanda kurung siku di sini? Mirip dengan cara Anda mengakses elemen dalam list, tetapi alih-alih posisi indeks dalam tanda kurung kurawal {}, Anda menentukan nama bidang dalam tanda kurung siku.

Anda dapat memperoleh hasil yang sama dengan menggunakan *Record.Field* sebagai berikut:

```
Record.Field(
  [Name = "John Doe", Age = 30, City = "Seattle"],
  "Name"
)
```

Saat catatan Anda menjadi lebih kompleks, Anda mungkin menemukan catatan yang bersarang di dalam catatan lain. Misalnya, catatan kedua dalam bidang Info mungkin menyertakan informasi lebih lanjut seperti Usia atau

Jenis Kelamin. Untuk mengambil data Jenis Kelamin dari struktur bersarang tersebut, Anda akan menuliskannya sebagai:

```
[Info = [Age = 20, Gender = "Male"]][Info][Gender]
```

Perbedaan utama antara membuat dan memilih kolom dalam record terletak pada notasi dalam tanda kurung siku. Definisi record mencakup nama kolom, tanda sama dengan (=), dan nilai, sedangkan pemilihan kolom hanya menentukan nama kolom dalam tanda kurung. Menemukan perbedaan mungkin memerlukan latihan, tetapi Anda akan terbiasa.

Namun, berhati-hatilah saat memilih kolom. Mencoba memilih kolom yang tidak ada dalam record akan mengakibatkan kesalahan, seperti yang diilustrasikan dalam contoh berikut:

```
[Name = "John", Age = 30][City] // error
```

Kode di atas mengembalikan *Expression.Error*: " *The field 'Gender' of the record wasn't found*.". Untuk mencegah pesan kesalahan ini, Anda dapat menggunakan operator pemilihan kolom opsional (?) untuk mengembalikan nilai null dan mengabaikan kesalahan:

```
[Name = "John", Age = 30] [City]  // error
[Name = "John", Age = 30] [City]? // null
```

Ini identik dengan penggunaan fungsi *Record.FieldOrDefault*:

```
Record.FieldOrDefault( [Name = "John", Age = 30], "City", null )
```

Contoh sejauh ini mengembalikan nilai dari suatu bidang. Namun, Records juga mendukung pengembalian sebagian dari record sehingga Anda dapat mengambil beberapa nilai sekaligus, yang akan kita bahas sekarang.

Proyeksi Record

Proyeksi record memperluas konsep pemilihan bidang, yang memungkinkan Anda memanggil lebih dari satu bidang secara bersamaan. Dengan kata lain, proyeksi ini memungkinkan Anda melakukan pemilihan

bidang beberapa kali dan menyatukannya. Ini menciptakan record baru dengan bidang yang dipilih dari record asli.

Untuk mencapainya, cukup cantumkan nama bidang yang ingin Anda ambil, dipisahkan dengan koma dalam tanda kurung siku, seperti:

```
[Ab=1,Bo=2,Ed=3][[Ab],[Ed]] // Output [Ab=1,Ed=3]
[Ab=1,Bo=2,Ed=3][[Ab]]      // Output [Ab=1]
```

Saat Anda mencoba memilih bidang yang tidak ada, Power Query akan memunculkan kesalahan. Untuk mengatasinya, Anda dapat menggunakan operator proyeksi record opsional (?) seperti yang Anda lakukan dengan pemilihan bidang, dan ini akan mengembalikan nilai null untuk bidang yang hilang:

```
[Ab=1,Bo=2,Ed=3][[Ab],[Z]]  // Output error
[Ab=1,Bo=2,Ed=3][[Ab],[Z]]? // Output [Ab=1, Z=null]
```

Anda dapat meniru perilaku ini dengan menggunakan Record.SelectFields:

```
Record.SelectFields([Ab=1,Bo=2,Ed=3], {"Ab", "Z"}, MissingField.UseNull )
```

Di bagian ini, kita mempelajari lebih lanjut mengenai pengaksesan elemen dalam record, tetapi kapan record berguna?

5. Operasi Umum Dengan Record

Sekarang setelah Anda terbiasa membuat catatan dan mengakses bidang dengan pemilihan dan proyeksi, Anda mungkin bertanya-tanya tentang penggunaan praktisnya. Di bagian ini, kita akan menjelajahi aplikasi dan skenario dunia nyata di mana nilai catatan berguna dalam manipulasi data.

Struktur untuk Variabel

Kasus penggunaan pertama untuk catatan adalah menggunakannya sebagai struktur untuk variabel. Dalam DAX, seperti yang Anda ketahui,

variabel diawali dengan kata VAR dan diakhiri dengan teks RETURN. Editor tingkat lanjut bekerja dengan konstruksi *let*..in, yang melakukan hal yang sama. Sekarang, apa hubungannya ini dengan catatan? Pernyataan *let*..in dalam editor tingkat lanjut pada dasarnya adalah catatan, dan Anda dapat menggunakannya untuk keuntungan Anda untuk variabel. Jadi mengapa Anda ingin menggunakan variabel? Karena tiga alasan:

- Ketika Anda menggunakan kembali bagian-bagian kode Anda, Anda memiliki lebih sedikit pengulangan dan mungkin kode yang lebih efisien saat menyimpan hasil ekspresi dalam variabel.
- Dengan membagi pernyataan yang kompleks, kode Anda menjadi lebih mudah dibaca.
- Variabel membantu dalam pemecahan masalah karena Anda dapat mengembalikan kalkulasi sementara sebagai hasilnya.

Mari kita lihat sebuah contoh. Bayangkan Anda memiliki tabel dengan kolom Tanggal yang berisi nilai teks 31-12-2024. Sasaran Anda adalah membuat kode untuk secara dinamis menyatakan pada hari apa Malam Tahun Baru jatuh. Langkah-langkah untuk melakukannya adalah:

i. Ubah nilai teks menjadi tanggal.
ii. Dapatkan tahun dan nama hari dalam seminggu dari tanggal ini.
iii. Buat string untuk menggabungkan informasi ini.

Anda dapat melakukannya secara manual dengan menambahkan kode berikut ke kolom khusus:

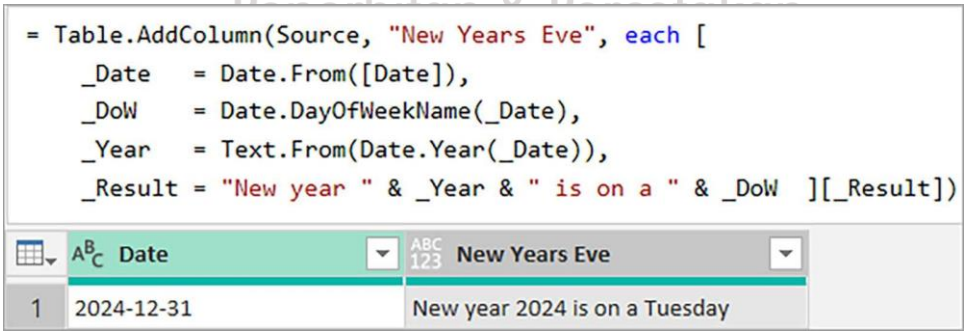
```
"New year "
    & Text.From( Date.Year( Date.From( [Date] )))
    & " is on a "

    & Date.DayOfWeekName( Date.From( [Date] ))
```

Output dari pernyataan ini adalah: *New year 2023 is on a Sunday*. Perhatikan bagaimana pernyataan ini mengonversi nilai teks dari kolom Tanggal ke tipe data tanggal sebanyak dua kali.

Jika Anda ingin menggunakan struktur record dan bekerja dengan variabel, Anda dapat menulis:

```
[
    _Date   = Date.From([Date]),
    _DoW    = Date.DayOfWeekName(_Date),
    _Year   = Text.From(Date.Year(_Date)),
    _Result = "New year " & _Year & " is on a " & _DoW
  ][_Result]
```

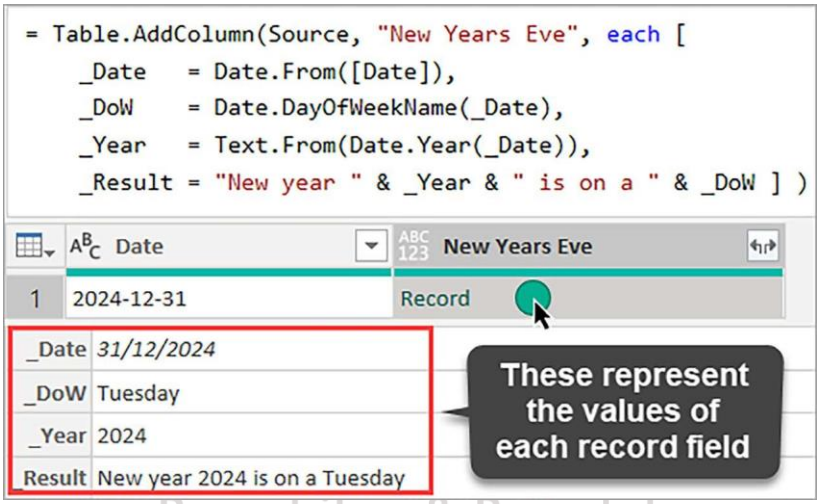
Output dari kode ini terlihat sebagai berikut:



Gambar 2. 13 Catatan adalah cara yang efektif untuk menyimpan variabel

Kode diakhiri dengan mengembalikan kolom bernama *_Result*. Jika terjadi kesalahan, Anda dapat menentukan langkah yang bermasalah dengan menukar baris terakhir dengan kolom lain dalam record.

Anda bahkan dapat mengembalikan seluruh record, seperti yang ditunjukkan pada gambar berikut:



Gambar 2. 14 Dengan mengembalikan seluruh record, Anda dapat memeriksa nilai setiap variabel

Itu akan memungkinkan Anda melihat hasil setiap kolom dalam record Anda. Penggunaan konstruksi record dengan cara ini khususnya bermanfaat untuk pemecahan masalah, karena memungkinkan peninjauan sistematis terhadap nilai variabel pada setiap tahap. Anda dapat menemukan contoh di atas dalam berkas latihan yang disediakan dalam repo GitHub untuk buku ini.

Merujuk ke Baris Saat ini

Skenario lain di mana catatan berguna adalah saat membuat kalkulasi pada nilai baris saat ini. Gambar berikut menggambarkan tabel yang berisi kolom dengan nama lokasi penyimpanan dan semua kolom lain yang berisi bulan dalam setahun.

	Store	January	February	March	April
1	Denver	501	490	477	522
2	Seattle	780	869	840	700
3	New York	1100	1250	1219	1846

Gambar 2. 15 Dataset untuk mengambil baris saat ini sebagai record

Jika Anda ingin menambahkan kolom dengan penjualan rata-rata tanpa mengambil data dari kolom toko, Anda dapat menulis secara manual:

```
List.Average( { [January], [February], [March], [April] } )
```

Namun, struktur record menyediakan cara yang lebih dinamis untuk mereferensikan nilai dari suatu baris. Untuk mendapatkan hasil yang sama tanpa melakukan hardcoding pada kolom bulan, Anda dapat menulis:

```
List.Average(
 Record.ToList(

   Record.RemoveFields( _, "Store" ) ) )
```

Kode ini pertama-tama mengambil semua nilai dari baris saat ini sebagai record menggunakan garis bawah. Fungsi *Record.RemoveFields* kemudian menghapus kolom Store dari record ini. Terakhir, *Record.ToList* mengubah record menjadi list sehingga *List.Average* dapat menghitung rata-rata semua nilai yang tersisa.

Memberikan Opsi Untuk Fungsi

Catatan juga umum digunakan untuk menyediakan opsi dalam parameter fungsi. Menggunakan catatan sebagai input untuk argumen fungsi memungkinkan Anda menentukan input untuk beberapa opsi dengan cara yang mudah.

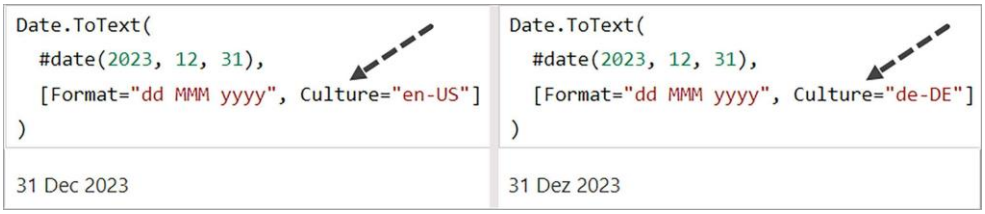
Ambil contoh, fungsi *Date.ToText*. Untuk menentukan format dan budaya output untuk bahasa Inggris, Anda dapat menggunakan:

```
Date.ToText( #date( 2023, 12, 31 ),
  [Format = "dd MMM yyyy", Culture = "en-US" ] )
```

Melakukan hal serupa untuk bahasa Jerman adalah sebagai berikut:

```
Date.ToText( #date( 2023, 12, 31 ),
  [Format = "dd MMM yyyy", Culture = "de-DE" ] )
```

Hasil operasi ini dapat dilihat pada gambar berikut:



Gambar 2. 16 Catatan digunakan dalam argumen fungsi

Perbedaan antara kedua contoh ini dapat ditemukan dalam kode *Culture*. Dalam kasus ini, nama bulan yang disingkat menunjukkan hasil yang berbeda tergantung pada kode budaya yang digunakan.

Singkatnya, kemampuan untuk menyediakan opsi pada suatu fungsi adalah alasan bagus lainnya untuk mempelajari tentang record. Anda akan menemukan sejumlah besar fungsi yang memerlukan record untuk menentukan opsi tambahan.

Melacak Hasil Perantara

Catatan juga memainkan peran penting dalam menyimpan hasil perantara dalam fungsi yang kompleks. Fungsi yang dapat menggunakan catatan untuk perhitungan yang lebih kompleks adalah *List.Generate*.

Fungsi ini menghasilkan list nilai berdasarkan nilai awal, kondisi yang menentukan kapan harus berhenti menghasilkan nilai, dan fungsi yang menjelaskan cara menghasilkan setiap nilai berikutnya. Tanpa menggunakan catatan, fungsi ini dapat, misalnya, menghasilkan list nilai yang meningkat dari 1 hingga 5:

```
List.Generate(
    () => 1,      // the starting value is 1
    each _ <= 5,  // until the value reaches 5
```

```
    each _ + 1    // increment the initial value by 1
)
```

Di sini, fungsi *List.Generate* menghasilkan list nilai. Namun, operasi ini masih cukup sederhana, dan di sinilah catatan berperan. Catatan memainkan peran penting saat menangani iterasi multivariabel di *List.Generate*. Catatan dapat menyimpan status terkini dari semua variabel yang terlibat dalam iterasi. Dengan menghasilkan list catatan, Anda dapat menyimpan lebih dari satu nilai yang diperlukan. Misalnya, contoh berikut membuat total berjalan dengan menambah nilai x sebanyak satu di setiap langkah, lalu menambahkan nilai baru ini ke nilai sebelumnya dalam variabel RT (untuk total berjalan):

```
List.Generate(
     () => [ x = 1, RT = 1 ],
     each [x] <= 5,
     each [ x = [x] + 1, RT = [RT] + [x] + 1 ],
     each [RT]  )
```

Perhatikan bahwa ilustrasi ini dimaksudkan untuk menunjukkan kepada Anda di mana catatan berguna. *List.Generate* adalah fungsi kompleks yang akan kita bahas lebih rinci di Bab 13 saat kita mempelajari rekursi dan iterasi. Berikutnya adalah topik tentang penetapan tipe data ke catatan.

6. Menetapkan Tipe Data ke Record

Record adalah nilai terstruktur yang mampu menampung satu atau beberapa nilai berbeda. Saat Anda membuat record ini, Power Query akan menggunakan tipe apa pun secara default jika tidak ada tipe data tertentu yang ditetapkan, seperti yang diilustrasikan dalam gambar berikut:

```
Table.AddColumn(ChangeColType, "MyRecord", each [ Key = [ID], Product = "Jeans" ] )
```

	1²3 ID	ABC 123 MyRecord
1	1	[Record]
2	2	[Record]
3	3	[Record]

Gambar 2. 17 Membuat kolom Record tanpa mengatur tipe data tertentu

Memperluas kolom **Record** kemudian menambahkan kolom-kolom baru dan menetapkan tipe data apa pun seperti yang diilustrasikan di sini:

```
Table.ExpandRecordColumn(Custom, "MyRecord", {"Key", "Product"}, {"Key", "Product"})
```

	ID	Key	Product
1	1	1	Jeans
2	2	2	Jeans
3	3	3	Jeans

Gambar 2. 18 Memperluas kolom Record yang tidak memiliki tipe data yang ditetapkan

Untuk mengalokasikan tipe data ke suatu record, Anda dapat memasukkan tipe record yang menentukan tipe data untuk setiap bidang record. Sintaks untuk ini mencerminkan pembuatan record, dengan satu-satunya perbedaan adalah bahwa ia menentukan tipe data, bukan nilai bidang, seperti yang ditunjukkan di bawah ini:

```
Table.AddColumn( ChangeColType, "MyRecord",
each [ Key = [ID],       Product = "Jeans"   ],
type [ Key = Int64.Type, Product = Text.Type ] )
```

	ID	MyRecord
1	1	[Record]
2	2	[Record]
3	3	[Record]

Gambar 2. 19 Membuat kolom Record dengan tipe data yang ditetapkan

Pada gambar berikutnya, kita perluas kolom **Record** yang memiliki tipe yang ditetapkan. Ini akan menciptakan kolom baru yang mencerminkan tipe data yang diberikan.

```
Table.ExpandRecordColumn(AddRecord, "MyRecord", {"Key", "Product"}, {"Key", "Product"})
```

	ID	Key	Product
1	1	1	Jeans
2	2	2	Jeans
3	3	3	Jeans

Gambar 2. 20 Memperluas kolom Record ini akan mempertahankan tipe data yang ditetapkan

Seperti yang ditunjukkan bagian ini, catatan menyediakan cara terstruktur untuk menyimpan dan memanipulasi data. Kegunaannya menjadi jelas dalam skenario yang mengharuskan penanganan beberapa titik data terkait secara bersamaan. Dengan mengingat hal ini, kita sekarang sampai pada nilai terstruktur yang tersisa, nilai tabel.

E. Tabel

Setelah kita menyelami lebih dalam list dan record, sekarang saatnya untuk memperluas pemahaman kita ke nilai tabel. Seperti halnya list dan record, tabel memegang peranan penting dalam bahasa M karena perannya dalam menyusun dan mengatur data. Jadi, apa itu tabel?

1. Pengantar tabel

Pada dasarnya, tabel adalah nilai terstruktur yang menyusun data dalam baris dan kolom. Anda juga dapat mengatakan bahwa tabel adalah list record yang setiap recordnya mewakili baris tabel. Karena bentuknya yang dua dimensi, mudah untuk melihat pratinjau data Anda dalam tabel dan Anda akan menemukan bahwa sebagian besar kueri yang dimuat ke Power BI atau Excel memiliki nilai tabel sebagai output. Jadi, bagaimana cara membuat tabel?

Membuat tabel dalam bahasa M Power Query sering kali dilakukan dengan memanggil fungsi pengaksesan data seperti yang dijelaskan dalam Bab 3. Misalnya, saat Anda mengimpor file Excel atau CSV, Power Query secara otomatis membuat tabel untuk Anda.

Namun, untuk mengilustrasikan karakteristik tabel, bagian ini menggunakan fungsi #table yang membuat tabel secara manual. Fungsi ini membutuhkan dua argumen:

- List nama kolom.
- List list, di mana setiap list bagian dalam membentuk baris.

Mari kita lihat contoh ekspresi yang menggunakan #table:

```
#table(
    {"ID", "Name", "Country"},
    { {1, "John Doe", "USA"},
     {2, "Jane Doe", "USA"},
    {3, "Jane Doe", "Canada"}
     })
```

Kode di atas mengembalikan tabel yang ditunjukkan di bawah ini:

	ABC 123 ID	ABC 123 Name	ABC 123 Country
1	1	John Doe	USA
2	2	Jane Doe	USA
3	3	Jane Doe	Canada

Gambar 2. 21 Tabel yang dibuat menggunakan fungsi #table

Tabel yang dibuat memiliki tiga kolom (*ID, Name, Country*) dan tiga baris data. Untuk menentukan tipe data, Anda dapat menentukan tipe tabel kustom seperti yang dibahas di Bab 5, Memahami Tipe Data. Kita akan membahas beberapa aspeknya nanti di bab ini. Selanjutnya, kita akan melihat bagaimana operator yang berbeda berperilaku dengan tabel.

2. Operator Tabel

Saat kita memperdalam pemahaman kita tentang tabel, mari kita alihkan perhatian kita ke operator. Tabel mendukung operator berikut:

Tabel 2. 4 Operator yang didukung oleh tabel

Operator	Notasi	Deskripsi
Equal	=	Memeriksa apakah dua tabel identik.
Not equal	<>	Memeriksa apakah dua tabel berbeda.
Concatenation	&	Menambahkan dua tabel.
Coalesce	??	Mengembalikan nilai bukan nol pertama dari dua opsi.

Equal

Operator sama dengan (=) digunakan untuk menentukan apakah dua tabel sama, baik dalam strukturnya maupun nilai yang dikandungnya. Agar dua tabel dianggap sama, keduanya harus memenuhi kriteria berikut:

- Memiliki jumlah kolom yang sama.
- Setiap nama kolom tersedia di kedua tabel.
- Memiliki jumlah baris yang sama.
- Sel yang sesuai di baris ini memiliki nilai yang sama.

Pertimbangkan contoh berikut di mana dua tabel memiliki nama kolom yang diurutkan secara berbeda:

```
#table( {"ID", "Name", "Country"},
        {{1, "John Doe", "USA"}, {2, "Jane Doe", "USA"}} ) =
#table( {"Name", "Country", "ID"},
        {{"John Doe", "USA", 1}, {"Jane Doe", "USA", 2}} )
```

Ekspresi ini mengembalikan true, yang menunjukkan bahwa urutan nama kolom tidak memengaruhi kesetaraan tabel. Faktor kuncinya adalah bahwa data selaras dengan benar dalam hal nilai dan struktur.

Kasus penggunaan umum untuk jenis perbandingan ini adalah dalam validasi data. Misalnya, anggaplah Anda telah mengimpor tabel anggaran ke Power Query. Untuk memastikan keakuratan, Anda ingin memverifikasi bahwa data yang diunggah cocok dengan file anggaran terbaru yang Anda terima. Saat membandingkan kedua tabel menggunakan operator yang sama, hasil true mengonfirmasi bahwa data di kedua tabel memang identik. Ini memastikan bahwa tidak ada perbedaan antara versi anggaran lama dan file terbaru.

Not Equal

Operator not equal melakukan kebalikan dari operator equal. Operator ini memeriksa apakah dua tabel berbeda dalam hal apa pun. Dengan mengambil contoh sebelumnya, kita dapat mengubah negara dari USA menjadi US:

```
#table( {"ID", "Name", "Country" },
      {{1, "John Doe", "USA"}, {2, "Jane Doe", "USA"}}) =
#table( {"ID", "Name", "Country" },
        {{1, "John Doe", "US"}, {2, "Jane Doe", "USA"}})
```

Ekspresi di atas mengembalikan false karena nama negara di kolom ketiga baris pertama berbeda. Perlu diketahui bahwa operator not equal berfungsi, terlepas dari apakah nilai yang dibandingkan adalah nilai primitif atau terstruktur.

Concatenation

Anda dapat menggunakan operator penggabungan, yang direpresentasikan oleh &, untuk menggabungkan dua tabel. Operasi ini menambahkan satu tabel ke akhir tabel lainnya. Misalnya, kode berikut menggabungkan dua tabel:

```
#table({"ID", "Name", "Country"}, {{1, "John Doe", "USA"}}) &
#table({"ID", "Name", "Country"}, {{2, "Jane Doe", "USA"}})
```

Ekspresi ini mengembalikan tabel berikut:

Gambar 2. 22 Menggabungkan dua tabel dengan kolom yang identik

Di sini, kedua tabel berhasil digabungkan, menyelaraskan baris dari kedua tabel menjadi satu urutan yang berkesinambungan. Dalam kasus di mana tabel

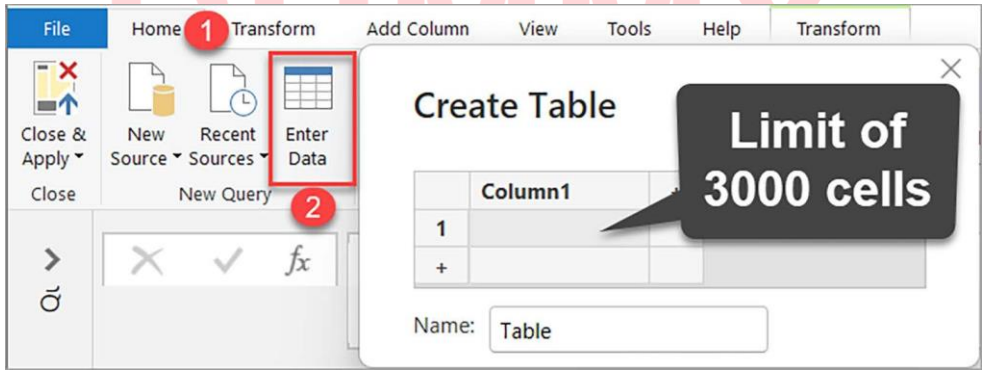
memiliki kolom yang berbeda, operasi penggabungan secara cerdas mengisi data yang hilang dengan nilai nol untuk memastikan integritas tabel yang digabungkan. Anda dapat mencoba ini menggunakan kode berikut:

```
#table({"ID", "Name", "City"}, {{1, "John Doe", "Texas"}}) &
#table({"ID", "Name", "Country"}, {{2, "Jane Doe", "USA"}})
```

Output dari ekspresi ini adalah sebagai berikut:

Gambar 2. 23 Menggabungkan tabel dengan kolom yang berbeda menambahkan nilai null

Skenario lain yang berguna untuk menggabungkan tabel adalah saat menggunakan fungsi **Enter Data**. Fungsi ini memungkinkan Anda untuk menyediakan data secara manual dan mengimpornya ke kueri Anda. Cukup buka tab **Home** dan pilih **Enter Data** seperti yang ditunjukkan di bawah ini:



Gambar 2. 24 Fungsi Masukkan Data dibatasi hingga 3.000 sel

Anda kemudian dapat menempelkan nilai-nilai Anda ke dalam tabel. Namun, ada batasan 3.000 sel yang dapat Anda tempel. Solusi untuk batasan ini adalah dengan membagi tabel Anda menjadi beberapa tabel yang berada dalam batasan 3.000 sel tersebut. Anda kemudian dapat menggunakan operator

penggabungan untuk merujuk ke tabel-tabel yang berbeda dan menggabungkannya menjadi satu tabel.

Atau, jika Anda lebih suka pendekatan fungsional, *Table.Combine* menawarkan hasil yang serupa. Fungsi ini mengharuskan tabel-tabel tersebut digabungkan dalam sebuah list, seperti yang ditunjukkan di bawah ini:

```
Table.Combine( Table1, Table2 )
```

Metode ini sama efektif dan lebih eksplisit, terutama saat menangani banyak tabel.

Coalesce

Operator coalesce (??), yang kita tunjukkan untuk list dan record, sama efektifnya saat diterapkan pada nilai tabel. Operator ini memungkinkan Anda memberikan nilai fallback jika tabel utama mungkin bernilai null. Ini memastikan kueri Anda tetap tangguh dan bebas kesalahan.

Contoh operator coalesce yang beraksi pada nilai tabel adalah sebagai berikut:

```
null ?? #table( {"ID", "Name"}, {{1, "John Doe"}})
```

Dalam pernyataan ini, jika nilai pertama adalah null, operator secara default akan mengembalikan tabel yang ditentukan setelah ??. Akibatnya, nilai tabel dikembalikan.

Anda biasanya akan menemukan operator coalesce yang digunakan dalam kombinasi dengan tabel saat membuat fungsi kustom. Di sini, operator ini bertindak sebagai pengaman, mencegah kesalahan yang dapat muncul dari input null. Dengan menyediakan tabel alternatif saat tabel yang diharapkan adalah null, Anda memastikan bahwa fungsi kustom Anda tangguh dan andal. Anda dapat menemukan detail tentang cara membuat fungsi kustom di Gbapter 9.

Setelah mempelajari pembuatan tabel dan penerapan operator, bagian berikutnya berfokus pada pengaksesan item dalam tabel, keterampilan yang berguna saat mengubah data Anda.

3. Metode untuk membuat tabel

Tabel merupakan konsep dasar dalam bahasa M. Bagian ini akan memandu Anda melalui berbagai metode pembuatan tabel, yang membuka dunia kemungkinan.

Mengambil Data Dari Sumber

Cara paling umum untuk membuat tabel adalah dengan menyambungkannya ke sumber data. Anda dapat menyambungkannya ke berbagai sumber data, termasuk file Excel, file CSV, halaman web, dan database. Setelah tersambung ke sumber data, Power Query M akan secara otomatis membuat tabel dari data tersebut.

Misalnya, untuk membuat tabel dari file Excel, Anda akan mengikuti langkah-langkah berikut:

1) Di editor Power Query, klik tombol **New Source** di pita.
2) Pilih Buku **Excel Workbook**.
3) Di kotak dialog **Open**, pilih file Excel yang ingin Anda sambungkan.
4) Pilih tabel atau lembar yang ingin Anda impor.
5) Klik tombol **OK**.

Power Query M akan secara otomatis membuat tabel dari file Excel menggunakan fungsi *Excel.Workbook* bersama dengan *File.Contents*.

Memasukkan Data Secara Manual ke Dalam Fungsi

Cara kedua untuk membuat nilai tabel adalah dengan membuatnya secara manual menggunakan fungsi pustaka. Fungsi yang memungkinkan Anda membuat tabel secara manual adalah:

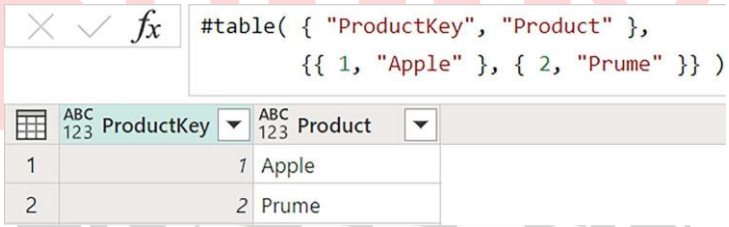
- *#table*

- *Table.FromColumns*
- *Table.FromList*
- *Table.FromRecords*
- *Table.FromRows*
- *Table.FromValue*
- *Table.FromPartitions*
- *Record.ToTable*

Setiap fungsi memiliki serangkaian kemampuannya sendiri ke dalam tabel (dengan maksud tertentu). Fungsi yang paling sering digunakan untuk membuat tabel secara manual adalah fungsi #table. Fungsi ini mengambil list nama kolom sebagai argumen pertama dan list list sebagai argumen kedua untuk membentuk baris. Perhatikan contoh berikut:

```
#table( {"ProductKey", "Product"},
        {{1, "Apple"}, {2, "Prume"}} )
```

Kode sebelumnya menggunakan list sebagai input dan mengembalikan tabel berikut:



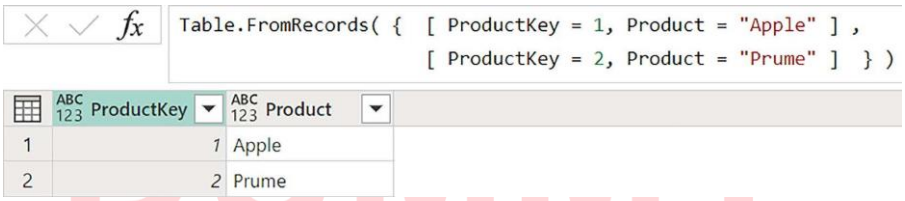
Gambar 2. 25 Fungsi #table membuat tabel

Argumen pertama mengambil list nama kolom, sedangkan argumen kedua mengambil list list untuk baris dalam tabel.

Fungsi populer lainnya, *Table.FromRecords*, menerima list record sebagai input. Record ini menampung nama kolom, yang digambarkan oleh nama bidang record, dan nilai baris, yang diwakili oleh nilai bidang record. Anda dapat menggunakannya sebagai berikut:

```
Table.FromRecords( { [ ProductKey = 1, Product = "Apple"  ],
                     [ ProductKey = 2, Product = "Prume" ] })
```

Kode ini mengembalikan tabel yang identik, tetapi menggunakan record sebagai input:



Gambar 2. 26 Ilustrasi pembuatan tabel menggunakan Table.FromRecords

Jumlah metode untuk membuat tabel di Power Query sangat banyak, dan membahas setiap fungsi di sini akan sangat banyak. Namun, menjadi ahli bahasa M melibatkan pemahaman berbagai metode pembuatan tabel.

Pertimbangkan skenario saat Anda membuat list nilai total yang berjalan menggunakan fungsi *List.Generate*. Output dari fungsi ini adalah nilai list. Untuk menghubungkan nilai dalam list ini ke tabel yang sudah ada, Anda harus menggabungkan beberapa fungsi ini.

Dalam contoh lain, Anda mungkin ditugaskan untuk membuat tabel rekonsiliasi yang melakukan berbagai pemeriksaan dengan menjawab pertanyaan seperti: apakah jumlah baris tabel akhir sama dengan jumlah awal? Apakah total pendapatan Query1 sesuai dengan Query2?

Dalam skenario ini, mengetahui cara menyimpan nilai output pemeriksaan ini dalam tabel terbukti membantu. Dan hal baiknya adalah Anda memiliki berbagai fungsi yang dapat Anda gunakan untuk membuat tabel yang memenuhi kebutuhan spesifik Anda.

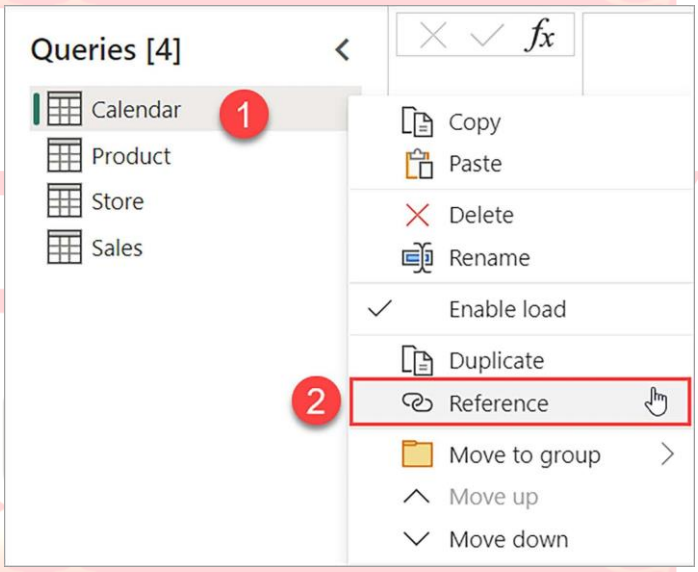
Gunakan Kembali Tabel/Queri yang ada

Kini Anda mengetahui metode untuk membuat tabel secara manual, saatnya untuk menjelajahi dua cara penting lainnya untuk menambahkan tabel di Power Query: mereferensikan dan menduplikasi kueri.

Referensi Tabel/Queri

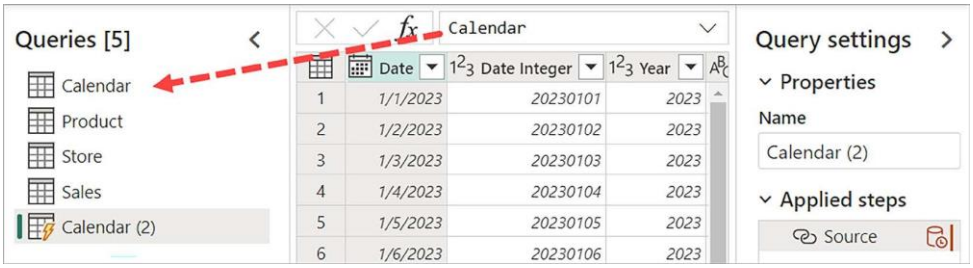
Mereferensikan tabel adalah proses menunjuk ke output tabel atau kueri lain. Pendekatan ini memungkinkan Anda untuk menggunakan tabel atau kueri yang sudah ada tanpa mengubah atau memengaruhi konten aslinya. Anggaplah mereferensikan sebagai pembuatan tautan. Proses ini memungkinkan Anda untuk menautkan ke output tabel atau kueri yang sudah ada dan terus mengerjakannya tanpa gangguan atau perubahan apa pun pada sumbernya. Jadi, bagaimana cara melakukannya?

Mereferensikan tabel atau kueri itu mudah. Misalnya, anggaplah kita memiliki tabel yang disebut **Calendar**. Anda ingin menggunakan output tabel **Calendar** sebagai basis kueri baru. Yang perlu Anda lakukan hanyalah mengklik kanan kueri **Calendar** dan mengklik **Reference**, seperti yang ditunjukkan di bawah ini:



Gambar 2. 27 Ilustrasi operasi kueri referensi

Ini menciptakan kueri, ditunjukkan di bawah, yang merujuk (atau tertaut ke) kueri **Calendar**.



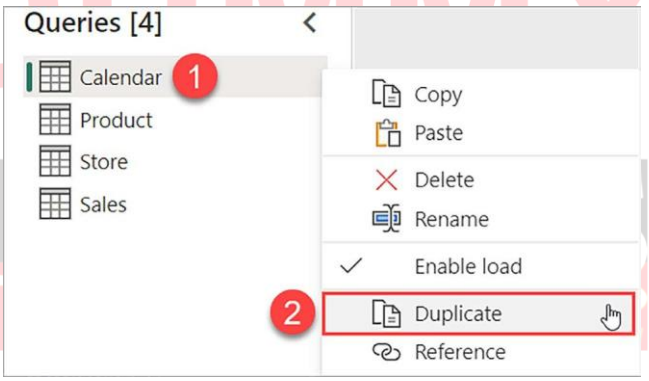
Gambar 2. 28 Operasi kueri referensi menghubungkan ke tabel asli

Setiap pembaruan pada kalender asli akan mengalir ke kueri yang direferensikan. Dengan cara itu, ada serangkaian logika. Selain mereferensikan, Anda juga dapat menduplikasi tabel.

Menduplikasi Tabel/Queri

Menduplikasi tabel di Power Query adalah tindakan membuat salinan persis dari kueri yang sudah ada. Kueri yang diduplikasi memiliki langkah kueri dan konten yang sama seperti aslinya tetapi berfungsi sebagai entitas terpisah.

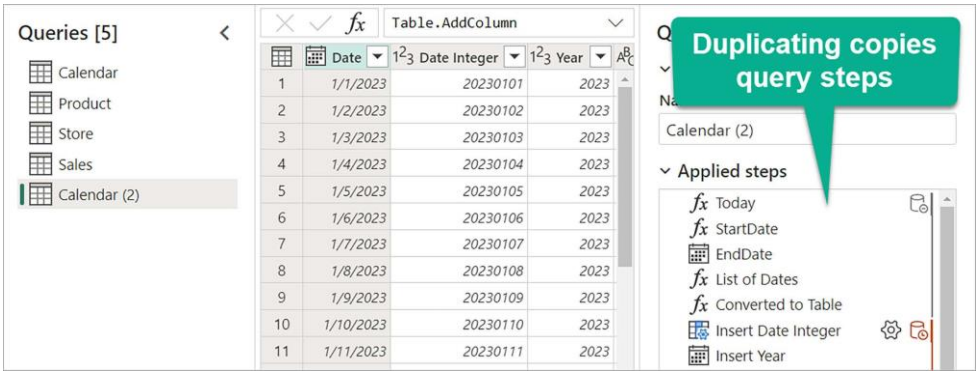
Untuk menduplikasi kueri, Anda klik kanan kueri yang sudah ada dan klik **Duplicate** seperti yang ditunjukkan pada gambar berikut:



Gambar 2. 29 Anda dapat menduplikasi kueri dengan mengklik kanan dan memilih Gandakan

Prosesnya berbeda dengan merujuk ke kueri: saat merujuk membuat tautan ke kueri yang sudah ada, duplikasi membuat kueri yang sama sekali

baru yang hanya menyerupai kueri asli. Salinan ini mencakup semua langkah kueri dalam kueri asli, seperti yang diilustrasikan oleh gambar berikut.



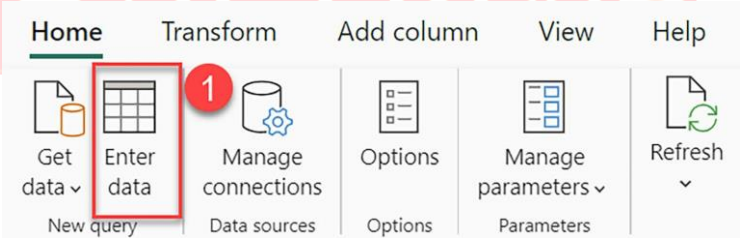
Gambar 2. 30 Operasi kueri Duplikat juga menyalin langkah kueri

Saat mencoba berbagai pendekatan pada data Anda, kita sarankan untuk menduplikasi kueri dan menggunakannya sebagai cadangan. Dengan cara ini, Anda selalu dapat memulihkan kueri asli jika percobaan Anda gagal.

Menggunakan Fungsi Masukkan Data

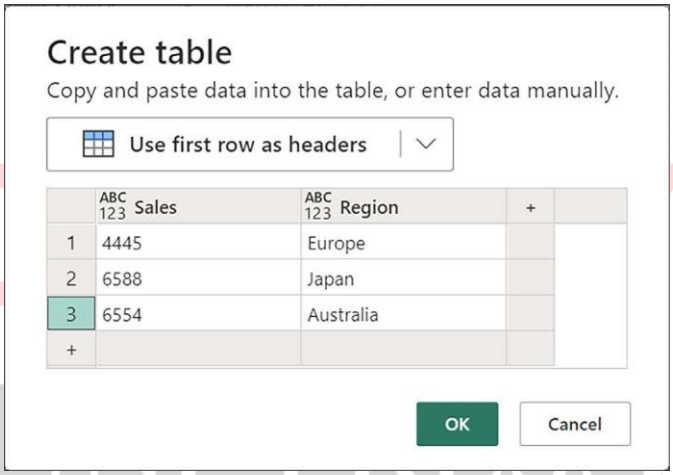
List ini tidak akan lengkap tanpa menyebutkan fungsi **Enter data**. Editor Power Query menyediakan cara yang mudah digunakan untuk membuat tabel secara manual hanya dengan memasukkan data di layar.

Untuk memasukkan data secara manual, navigasikan ke tab **Home** di pita. Di bagian **New query**, Anda akan menemukan tombol **Enter data** seperti yang ditunjukkan pada Gambar 6.31. Di sinilah Anda dapat membuat tabel secara manual tanpa menulis kode M apa pun.



Gambar 2. 31 Tombol masukkan data di tab Beranda

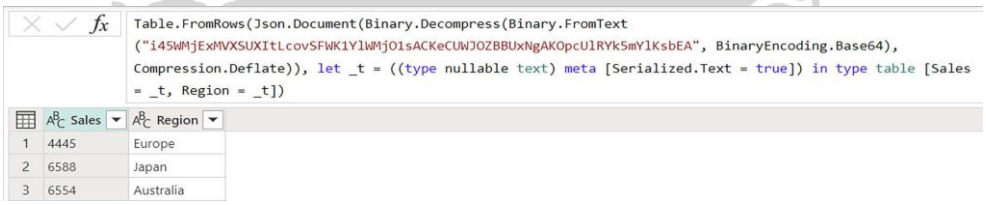
Setelah Anda mengklik **Enter data**, akan muncul jendela berisi tabel kosong yang siap diisi. Nama dan nilai kolom dapat diedit, dan Anda dapat langsung memasukkan data seperti yang ditunjukkan pada gambar berikut:



Gambar 2. 32 Layar dialog Buat tabel

Untuk menambahkan baris, Anda cukup mengeklik sel (disorot dengan warna abu-abu) di akhir baris terakhir, dan baris baru akan muncul. Demikian pula, untuk menambahkan kolom, Anda mengeklik tajuk kolom paling kanan (juga berwarna abu-abu), dan kolom baru akan dibuat.

Setelah Anda mengisi tabel dengan data, klik **OK** di sudut kanan bawah jendela. Tindakan ini akan menutup jendela dan memuat data ke Power Query sebagai tabel baru.



Gambar 2. 33 Kode biner yang dihasilkan dengan fungsi Enter data

Seperti yang ditunjukkan gambar ini, fungsi **Enter data** mengompresi data menjadi kode biner dan menggunakan fungsi Table.FromRows untuk membuat tabel. Karena ini adalah cara yang mudah untuk membuat tabel, kita ingin menyebutkannya secara terpisah. Perlu diketahui bahwa fungsi ini

mendukung hingga 3.000 sel dengan nilai. Selanjutnya, kita akan membahas akses ke berbagai elemen dalam tabel.

4. Mengakses elemen dalam tabel

Seperti halnya list dan record, mengakses elemen dalam tabel merupakan keterampilan yang berguna untuk dimiliki saat bekerja dengan Power Query. Anda mungkin ingin memilih satu kolom, beberapa kolom, satu baris, atau bahkan satu nilai sel. Mari kita lihat bagaimana kita dapat menggunakan prinsip pemilihan dan proyeksi dalam konteks tabel.

Field Item

Dalam konteks tabel, ekspresi akses item digunakan untuk mengembalikan baris. Katakanlah tabel berikut dibuat dalam langkah yang disebut *Source*.

	ABC 123 ID	ABC 123 Name	ABC 123 Country
1	1	John Doe	USA
2	2	Jane Doe	USA
3	3	Jane Doe	Canada

Gambar 2. 34 Tabel untuk mengakses item

Menggunakan operator pemilihan item dalam konteks tabel akan mengembalikan baris dalam bentuk record. Misalnya, Anda dapat mengembalikan baris pertama tabel dengan menulis:

```
Source{0} // Output: [ID=1, Name="John Doe", Country="USA"]
```

Ini identik dengan menggunakan fungsi Table.Range dan menentukan bahwa Anda ingin mengembalikan satu baris:

```
Table.Range( Source, 0, 1 )
```

Sama halnya dengan list, mencoba mengambil item yang posisi yang diminta tidak ada dalam tabel akan memunculkan kesalahan yang dapat Anda cegah dengan operator pemilihan item opsional:

```
Source{3}  // Row 4 does not exist, returns an Error
Source{3}? // Returns null
```

Tabel memungkinkan Anda mengakses baris dengan cara tambahan. Daripada memberikan nomor indeks dalam tanda kurung kurawal, Anda juga dapat mencari satu atau beberapa nilai bidang. Misalnya, untuk mengembalikan baris kedua tabel kita, kita dapat menggunakan salah satu ekspresi berikut:

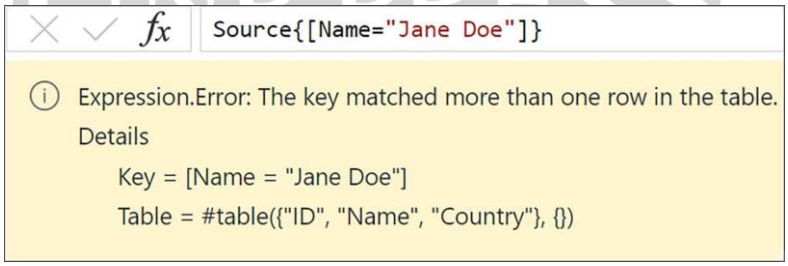
```
Source{1        // Returns the second row
Source{[ID = 2 ]} // Returns the row where ID = 2
```

Anda dapat mempersempit pencarian lebih lanjut dengan memberikan beberapa nilai pencarian. Cukup perluas data Anda dengan kolom tambahan dan tampilkan dalam tanda kurung kurawal, seperti ini:

```
Source{[ID = 2, Name ="Jane Doe" ]}
```

Penting untuk diingat bahwa pemilihan item dengan memberikan record harus menghasilkan baris yang unik. Ekspresi berikut cocok dengan dua baris dan menghasilkan kesalahan, seperti yang ditunjukkan pada Gambar 6.35:

```
Source{[Name ="Jane Doe"]}
```



Gambar 2. 35 Kesalahan saat pemilihan item mengembalikan beberapa baris

Perilaku ini sangat mirip dengan ekspresi berikut, yang mengharapkan satu baris sebagai hasilnya:

```
Table.SingleRow(
  Table.SelectRows( Source, each [Name] = "Jane Doe" ) )
```

Jika Anda ingin mengembalikan beberapa baris, Anda dapat menggunakan fungsi seperti:

```
Table.SelectRows( Source, each [Name] = "Jane Doe" )
```

Field Access

Sementara pemilihan item mengembalikan baris, operator pemilihan bidang memungkinkan Anda mengembalikan nilai dari kolom sebagai list. Untuk mengembalikan nilai dari kolom Nama sebagai list, Anda dapat menggunakan sintaksis yang sama seperti yang Anda lakukan untuk catatan:

```
Source[Name] // Output: { "John Doe", "Jane Doe", "Jane Doe" }
```

Sebagai alternatif, Anda dapat menggunakan fungsi *Table.Column* untuk hasil serupa:

```
Table.Column( Source, "Name" )
```

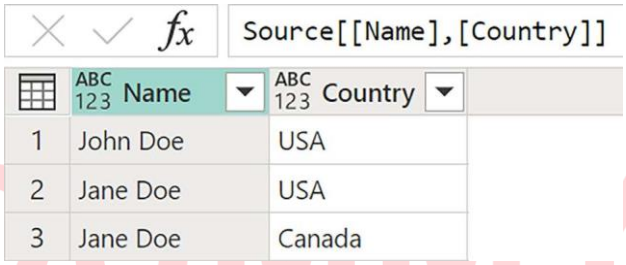
Perilaku ini berguna saat Anda ingin merujuk semua nilai kolom untuk melakukan operasi pada nilai tersebut, mirip dengan saat Anda menggunakan *List.Sum*. Anda kemudian dapat menggunakan hasilnya untuk menyatakan setiap nilai sebagai persentase dari total.

Sekarang, katakanlah Anda ingin mengakses beberapa kolom dari tabel Anda, bagaimana cara Anda mencapainya? Proyeksi bidang memungkinkan Anda memproyeksikan tabel ke tabel dengan kolom yang lebih sedikit.

```
Source[[Name],[Country]] // Returns two columns
```

Ini seperti menyediakan list referensi kolom, tetapi alih-alih mengapit kolom dengan tanda kurung kurawal, Anda menyediakan tanda kurung siku.

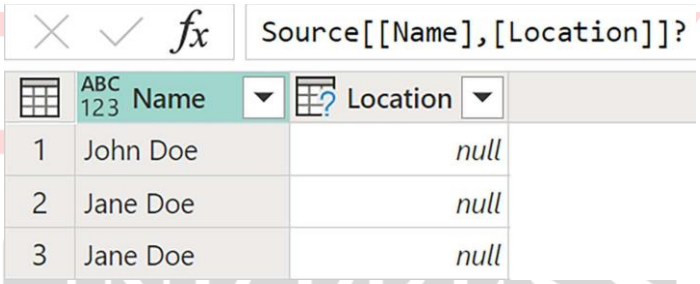
Metode ini berfungsi untuk kolom tunggal dan ganda. Manfaat dari pendekatan ini adalah output menyediakan kolom dalam format tabel, bukan list, seperti yang diilustrasikan di bawah ini:



Gambar 2. 36 Proyeksi bidang yang diperlukan memproyeksikan tabel ke kolom yang lebih sedikit

Kesalahan terjadi saat memilih bidang yang tidak ada, tetapi Anda dapat mencegah kesalahan tersebut dengan menggunakan proyeksi bidang opsional, yang direpresentasikan *by* ?. Misalnya:

```
Source[[Name],[Location]]  // Error
Source[[Name],[Location]]? // Returns null in Location column
```



Gambar 2. 37 Proyeksi bidang opsional mengembalikan null untuk kolom yang hilang

Perilaku ini identik dengan penggunaan fungsi *Table.SelectColumns* bersama dengan enumerasi *MissingField.UseNull*:

```
Table.SelectColumns(
  Source,
  {"Name", "Location"},
  MissingField.UseNull
)
```

Sejauh ini, kita telah melihat cara memilih baris sebagai record (item-selection) dan kita tahu cara mengembalikan satu atau beberapa kolom dari tabel (field-selection dan field-projection). Untuk mengembalikan satu nilai dari tabel, Anda dapat menggabungkan kedua konsep tersebut. Misalnya, Anda ingin mengembalikan nilai dari baris ketiga di kolom Nama – Anda dapat menggunakan salah satu ekspresi berikut:

```
Source[Name]{2}
Source{2}[Name]
```

Ini berguna saat Anda ingin menelusuri satu nilai dalam tabel Anda. Sekarang setelah Anda mengetahui cara membuat tabel dan mengakses item di dalamnya, mari kita lihat beberapa situasi di mana tabel berguna.

5. Operasi umum dengan tabel

Umumnya, tabel adalah hal yang kita cari saat bekerja dengan Power Query. Mungkin ini jelas, tetapi berikut ini beberapa area tempat tabel sering digunakan:

- **Tabel pembantu**: Tabel berguna saat menggabungkan atau menambahkan data. Ini dapat berupa impor dari file atau database lain, tetapi Anda juga dapat membuat tabel pembantu secara manual untuk memperkaya data Anda atau menggunakannya untuk menggabungkan atau menggabungkan.
- **Periksa data dan debug**: Tabel sering kali merupakan struktur paling efektif untuk memeriksa data Anda. Dengan kolom dan baris, Anda dapat melihat banyak sel secara bersamaan. Dan dengan setiap transformasi

yang Anda lakukan, Anda dapat dengan mudah melihat efeknya pada tabel Anda.

- **Integrasikan dengan Power BI dan Excel**: Setelah melakukan transformasi pada data Anda, Anda biasanya akan mendapatkan satu atau beberapa tabel yang bersih. Anda kemudian dapat memuatnya ke tujuan yang Anda inginkan, seperti lembar kerja Excel atau kumpulan data Power BI. Dari sana, Anda dapat terus bekerja dengan data Anda.

Saat ini ada lebih dari 117 fungsi tabel dan ini adalah kategori fungsi terbesar dalam bahasa M. Pada bab-bab berikutnya, Anda akan mempelajari teknik-teknik yang menggabungkan fungsi-fungsi tabel yang paling penting.

Sekarang Anda tahu cara membuat, memanipulasi, dan mengakses nilai-nilai dalam suatu tabel. Yang tersisa adalah mempelajari tentang penetapan tipe data ke dalam tabel, yang akan kita bahas selanjutnya.

6. Menetapkan Tipe Data ke tabel

Nilai tabel adalah nilai terstruktur dengan tipe data yang kompleks. Bagaimanapun, nilai tersebut dapat berisi nilai primitif atau nilai terstruktur lainnya. Namun, Anda tetap dapat menentukan tipe data untuk setiap kolom dalam tabel Anda. Jadi, bagaimana Anda harus menentukan tipe data untuk nilai tabel?

Bayangkan Anda sedang membuat nilai tabel secara manual untuk setiap baris dalam tabel. Anda dapat melakukannya dengan menggunakan fungsi *#table* bersama dengan fungsi *Table.AddColumn*, seperti yang ditunjukkan pada gambar berikut:

```
Table.AddColumn(ChangeColType, "Table",
each #table( {"Key", "Product"}, {{ 1, "Apple" },{ 2, "Prume" }} ))
```

	1²3 ID	ABC 123 Table
1	1	[Table]
2	2	[Table]
3	3	[Table]

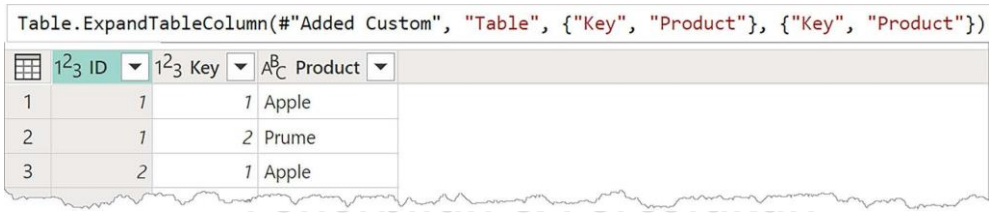
Gambar 2. 38 Membuat nilai tabel menggunakan fungsi #table

Sesuai dengan harapan kita, kolom yang baru dibentuk secara otomatis diberi tipe *any*. Saat Anda memperluas kolom, kolom tersebut akan menghasilkan kolom tambahan, yang semuanya diatur ke tipe data any. Jadi, bagaimana cara kita menetapkan tipe tabel selama proses pembuatan tabel?

Ini sangat mirip dengan cara kita mendefinisikan tipe record, dengan perbedaan utama adalah kali ini, kata tipe diikuti oleh kata tabel, seperti yang ditunjukkan pada gambar berikut.

Gambar 2. 39 Membuat kolom tabel dengan tipe data yang ditetapkan

Memperluas kolom kemudian menambahkan kolom baru dengan tipe data yang ditetapkan, seperti ditunjukkan di bawah ini:

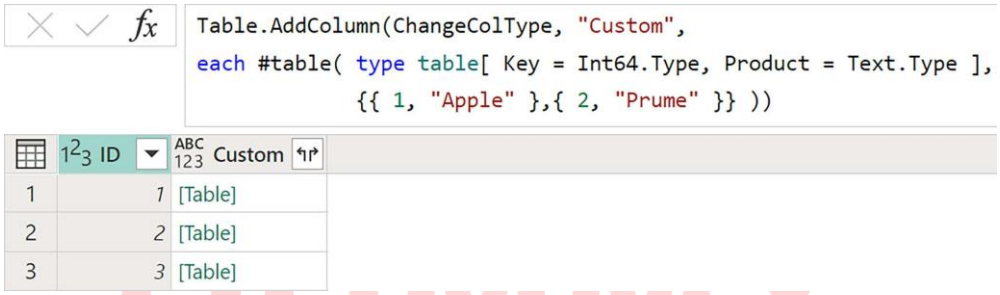


Gambar 2. 40 Memperluas kolom tabel mempertahankan tipe yang disediakan melalui tipe tabel kustom

Perhatikan bahwa Anda juga dapat menyediakan tipe tabel dalam fungsi #table. Saat Anda membuat tabel hanya menggunakan fungsi ini, tanpa menggunakan *Table.AddColumns*, tipe tabel menyediakan informasi tipe data yang diperlukan kepada mesin.

Namun, saat menggunakan fungsi *Table.AddColumn*, menambahkan tipe tabel ke fungsi #table tidak memengaruhi tipe data keluaran. Hanya argumen

keempat yang memengaruhi tipe keluaran, seperti yang ditunjukkan di bawah ini:



```
Table.AddColumn(ChangeColType, "Custom",
each #table( type table[ Key = Int64.Type, Product = Text.Type ],
                {{ 1, "Apple" },{ 2, "Prume" }} ))
```

	1²3 ID	ABC 123 Custom
1	1	[Table]
2	2	[Table]
3	3	[Table]

Gambar 2. 41 Fungsi *Table.AddColumn* menentukan tipe data keluaran

Pada gambar di atas, tipe kolom tidak ditetapkan untuk *Table.AddColumn* tetapi disertakan dalam ekspresi tabel. Oleh karena itu, tipe kolom tidak dipertahankan. Dalam situasi ini, penting untuk menambahkan tipe tabel ke argumen keempat *Table.AddColumn*.

F. Ringkasan

Dalam bab ini, kita menjelajahi list, record, dan tabel, yang menunjukkan betapa pentingnya hal-hal tersebut saat Anda menggunakan bahasa M.

Kita mempelajari bahwa nilai terstruktur berfungsi sebagai wadah yang menampung satu atau lebih nilai primitif atau terstruktur. Mempelajarinya bermanfaat dalam berbagai bidang. Misalnya, list dan record sering digunakan untuk menyediakan beberapa item dalam argumen fungsi dan membantu menyederhanakan kode Anda dengan menggunakannya. Struktur record juga merupakan struktur hebat tempat Anda dapat membuat variabel.

Kita juga menyelidiki cara mengakses item dari berbagai nilai terstruktur melalui pemilihan dan proyeksi, yang memungkinkan Anda mengekstrak nilai dengan mudah. Anda akan menemukan bahwa keterampilan ini membantu Anda memahami kode yang dibuat oleh antarmuka pengguna, tetapi juga membantu Anda membuat kode yang lebih pendek sendiri.

Kita kemudian melihat pembuatan nilai-nilai ini dan bagaimana operator bekerja pada nilai-nilai tersebut. Anda juga mempelajari tentang tipe data kompleks, yang menyoroti pentingnya nilai-nilai tersebut dan cara menetapkannya.

Terakhir, bab ini memberikan contoh dan skenario nyata di dunia nyata. Hal ini berfungsi sebagai demonstrasi praktis untuk menunjukkan kekuatan nilai terstruktur dan bagaimana Anda dapat menanganinya dengan mudah. Sekarang kita akan melanjutkan ke bab berikutnya, di mana kita fokus pada konseptualisasi M dan menggali lebih dalam prinsip-prinsip dasar bahasa M.

G. Penutup

1. Tes Formatif

No	Soal	Bobot
1.	Apa yang dimaksud dengan nilai terstruktur dalam bahasa M? Berikan penjelasan tentang bagaimana nilai terstruktur berbeda dari nilai primitif?	10
2.	Jelaskan cara membuat list dalam M dan berikan contoh kode untuk membuat list yang berisi campuran nilai primitif dan terstruktur.	10
3.	Diskusikan peran record dalam Power Query. Bagaimana record memungkinkan pengorganisasian data yang lebih kompleks dibandingkan dengan list?	10
4.	Jelaskan bagaimana tabel dibentuk dalam bahasa M. Apa saja komponen kunci dari sebuah tabel yang membedakannya dari tipe nilai lainnya?	10
5.	Bagaimana cara mengakses nilai spesifik dalam tabel menggunakan pemilihan bidang? Berikan contoh kode untuk mengakses nilai dari kolom tertentu.	10
6.	Diskusikan fungsi-fungsi yang terkait dengan list dan bagaimana fungsi-fungsi tersebut dapat digunakan untuk memanipulasi data dalam Power Query.	10
7.	Apa itu proyeksi record dan bagaimana cara kerjanya? Berikan contoh skenario di mana proyeksi record dapat digunakan.	10
8	Diskusikan pentingnya menetapkan tipe data untuk kolom dalam tabel. Apa konsekuensi dari tidak menetapkan tipe data yang tepat?	10

9.	Jelaskan bagaimana operator penggabungan (concatenation) bekerja pada record dan list. Berikan contoh kode yang menunjukkan cara penggunaannya.	10
10.	Analisis bagaimana nilai terstruktur meningkatkan efisiensi dalam penanganan data di Power Query. Apa keuntungan utama dari penggunaan nilai terstruktur?	10

BAB 3

Konseptualisasi M

Topik Bab

Pada bab ini, topik yang akan dibahas adalah:

- Memahami ruang lingkup
- Meneliti lingkungan global
- Memahami penutupan (*closures*)
- Mengelola metadata

A. Pendahuluan

Seperti halnya bahasa pemrograman lainnya, penting untuk memahami prinsip abstrak bahasa tersebut serta topik yang lebih praktis, seperti fungsi apa saja yang tersedia dan cara menggunakannya. Bab ini bertujuan untuk memberi Anda pemahaman konseptual yang solid tentang aspek-aspek utama Power Query M yang mungkin lebih abstrak, seperti cakupan, lingkungan global, penutupan, dan metadata. Konsep-konsep ini sangat penting untuk benar-benar menjadi ahli Power Query M.

Kita akan mulai dengan mempelajari konsep cakupan dalam Power Query M. Memahami cakupan sangat penting untuk mengendalikan visibilitas dan

aksesibilitas variabel dan fungsi dalam kueri Anda. Kita akan mengeksplorasi perbedaan antara cakupan lokal dan global, dengan memeriksa bagaimana variabel dan fungsi berinteraksi dalam cakupan yang berbeda untuk menghasilkan transformasi data yang efektif.

Setiap pembahasan tentang cakupan secara alami mengarah ke pembahasan tentang lingkungan global untuk Power Query M dan bagaimana kita dapat membuat lingkungan global kita sendiri melalui penggunaan fungsi Expression.Evaluate. Kita akan mengeksplorasi berbagai aspek lingkungan global dan bagaimana fungsi Expression.Evaluate dapat digunakan untuk membuat kueri dinamis dan memodifikasi logika kueri secara terprogram berdasarkan kondisi atau masukan tertentu.

Penutupan merupakan konsep penting dalam Power Query M yang memperkuat kapabilitas fungsi. Kita akan mengeksplorasi bagaimana penutupan memungkinkan fungsi untuk merangkum data dan memelihara referensi ke variabel dari lingkungan leksikalnya, yang memungkinkan manipulasi data yang canggih dan kalkulasi berulang.

Terakhir, memahami dan memanfaatkan metadata dapat berguna saat bekerja di Power Query M. Kita akan mengeksplorasi bagaimana metadata dapat dimanfaatkan untuk meningkatkan tata kelola data, melacak sumber data, dan memastikan kualitas data.

1. Kasus Pemantik Berfikir Kritis: Evaluasi Kode Dinamis untuk Laporan Bulanan

Seorang analis data di sebuah perusahaan retail diminta membuat laporan penjualan bulanan menggunakan Power BI. Untuk setiap bulan, data akan dimuat dari file Excel dengan nama file seperti Sales_Jan.xlsx, Sales_Feb.xlsx, dan seterusnya.

Manajer meminta agar laporan ini dibuat dinamis, sehingga tidak perlu mengubah kode setiap kali bulan baru dimulai. Analis ingin menggunakan

fungsi Expression.Evaluate untuk menyusun kueri dinamis berdasarkan nama file dan tabel yang ditentukan pengguna.

1) Bagaimana konsep lingkungan global (global environment) dalam M dapat membantu menyusun ekspresi evaluasi kode berdasarkan nama file atau tabel?
2) Apa risiko yang mungkin timbul jika tidak memperhatikan cakupan variabel saat menggunakan Expression.Evaluate?
3) Bagaimana penutupan (closures) dapat digunakan jika pengguna ingin menyesuaikan transformasi untuk setiap bulan secara fleksibel?
4) Jika metadata pada tabel penjualan menyimpan informasi seperti "Sudah Diproses: Ya/Tidak", bagaimana metadata ini dapat dimanfaatkan untuk memfilter data yang belum diproses secara otomatis?
5) Apa kelebihan dan kelemahan penggunaan Expression.Evaluate dibanding menulis langkah transformasi secara manual di Power Query GUI?

B. Persyaratan Teknis

Untuk mengikuti bab ini, Anda perlu menginstal Power BI Desktop.

C. Memahami ruang lingkup

Dalam Power Query M, cakupan menentukan di mana variabel dan fungsi dapat diakses dan digunakan, dan memegang peran penting dalam mengendalikan aliran transformasi data dan operasi manipulasi data:

- **Cakupan global**: Cakupan global merujuk pada variabel dan fungsi yang ditetapkan pada tingkat kueri, di luar langkah atau fungsi kueri tertentu. Variabel dan fungsi dengan cakupan global dapat diakses dan digunakan di

semua langkah dan fungsi dalam keseluruhan kueri. Mereka menyediakan cara untuk menyimpan dan berbagi data atau kalkulasi antara berbagai bagian kueri, memastikan konsistensi dan penggunaan ulang.

- **Cakupan lokal**: Cakupan lokal adalah jenis cakupan yang paling umum dalam Power Query M. Cakupan ini merujuk pada variabel dan fungsi yang ditetapkan dalam langkah kueri tertentu atau definisi fungsi. Variabel dan fungsi dengan cakupan lokal hanya dapat diakses dalam langkah atau fungsi tempat mereka ditetapkan. Mereka tidak dapat dirujuk atau digunakan di luar konteks lokal mereka, sehingga ideal untuk kalkulasi sementara atau hasil antara dalam transformasi tertentu.
- **Menggunakan ekspresi *let* untuk manajemen cakupan**: Dalam Power Query M, ekspresi let adalah alat yang ampuh untuk mengelola cakupan. Ekspresi ini memungkinkan Anda untuk menentukan variabel lokal dalam langkah kueri tertentu sekaligus menjaga cakupan global tetap bersih dan teratur, membuat kode Anda lebih mudah dibaca dan dipelihara dengan memisahkan kalkulasi sementara dan hasil antara dari transformasi data utama.
- **Cakupan dan parameter fungsi**: Fungsi dalam Power Query M memiliki cakupannya sendiri. Parameter fungsi dianggap sebagai variabel lokal dalam cakupan fungsi. Parameter tersebut dapat diakses dan digunakan dalam isi fungsi tetapi tidak memiliki makna di luarnya. Parameter fungsi memungkinkan Anda untuk meneruskan data dan nilai di antara berbagai bagian fungsi, yang memastikan aliran data dan modularitas.
- **Menyelesaikan konflik cakupan**: Ketika variabel atau fungsi memiliki nama yang sama dalam cakupan yang berbeda, Power Query M menggunakan proses yang disebut cakupan leksikal (juga dikenal sebagai cakupan statis) untuk menyelesaikan konflik cakupan. Cakupan leksikal memprioritaskan cakupan lokal daripada cakupan global, yang berarti

bahwa jika variabel atau fungsi dengan nama yang sama ada secara lokal dan global, variabel lokal akan diutamakan dalam konteks spesifiknya.

Mari kita lihat beberapa contoh spesifik cakupan. Pertimbangkan kueri berikut:

```
let
    x = 10,
    y = 20,
    z = 30
in
    x * y * z
```

Mengabaikan lingkungan global (lihat Memeriksa lingkungan global di bab ini), tabel berikut merangkum cakupan kueri dan setiap ekspresi dalam kueri:

Tabel 3. 1 Merangkum cakupan kueri dan ekspresi yang digunakan

Item	Lingkup Global	Lingkup Lokal
Query	`x = 10, y = 20, z = 30`	
X		`y = 20, z = 30`
Y		`x = 10, z = 30`
Z		`x = 10, y = 20`

Seperti yang dapat dilihat pada tabel, pada level kueri, ketiga variabel atau ekspresi, (x, y, z) berada dalam cakupan kueri. Akan tetapi, setiap variabel memiliki cakupan lokalnya sendiri. Dalam cakupan ini, setiap ekspresi dapat melihat variabel global lainnya; akan tetapi, ekspresinya sendiri tidak berada dalam cakupan dan umumnya tidak dapat dirujuk tanpa menyebabkan kesalahan. Misalnya, berikut ini adalah kode yang tidak valid:

```
Let
    x = 10,
    y = 20,
    z = x * y * z
in
    z
```

Hal ini mengakibatkan kesalahan berikut:

```
Expression.Error: The name 'z' wasn't recognized.  Make sure it's
spelled correctly.
```

Jenis kesalahan ini umumnya merupakan hasil dari kesalahpahaman cakupan. Untuk mencegah kesalahan tersebut, verifikasi bahwa variabel dan definisi fungsi Anda didefinisikan dalam cakupan yang tepat dan dengan demikian dapat diakses di tempat yang ingin Anda gunakan.

Dalam kasus khusus ini, kesalahan disebabkan oleh ekspresi *z = x * y * z* karena *z* tidak berada dalam cakupan untuk baris kode tersebut.

Ada kemungkinan ekspresi merujuk diri sendiri menggunakan simbol @ sebagai awalan. Namun, hal ini juga umumnya mengakibatkan kesalahan. Misalnya, kode berikut ini valid secara sintaksis:

```
let
    x = 10,
    y = 20,
    z = x * y * @z
in
    z
```

Namun, kueri ini menghasilkan kesalahan berikut:

```
Expression.Error: A cyclic reference was encountered during evaluation.
```

Kesalahan ini disebabkan oleh ekspresi *z = x * y * @z* karena mesin Power Query M telah menentukan bahwa tidak ada kondisi akhir di mana referensi diri akan berakhir. Referensi diri umumnya hanya berguna, dan tidak akan menyebabkan kesalahan, dalam konteks fungsi rekursif dengan beberapa bentuk

kondisi penghentian. Lihat Bab 13, Iterasi dan Rekursi, untuk informasi lebih lanjut tentang fungsi rekursif.

Melanjutkan contoh kita, mari kita jelajahi kueri yang lebih kompleks yang menunjukkan penggunaan ekspresi let untuk mengontrol cakupan dan menunjukkan cakupan leksikal di mana cakupan lokal diberi prioritas daripada cakupan global:

```
let
    x = 10,
    y =  let
             x = 1,
             z = x
         in
             z
in
    y
```

Hasil dari query ini adalah 1. Mari kita jelaskan bagaimana hal ini terjadi:

Tabel 3. 2 Merangkum cakupan kueri dan ekspresinya

Item	Lingkup Global	Lingkup Lokal
Query	`x = 10, y = ?`	
x		`y = ?`
y		`x = 10`
x		`x = 10, z = ?`
z		`x = 1`

Di sini kita menggunakan ? untuk menunjukkan nilai yang diselesaikan sebagai bagian dari penjelasan yang diberikan di sini dalam paragraf berikut. Seperti yang dibahas sebelumnya, cakupan global untuk kueri mencakup semua variabel atau ekspresi yang didefinisikan dalam kueri (x dan y). Masing-masing variabel ini memiliki cakupan lokal yang mencakup setiap variabel global lainnya tetapi tidak mencakup dirinya sendiri.

Untuk ekspresi variabel y, kita menggunakan ekspresi let untuk mengontrol cakupan. Dalam cakupan ini, kita mendefinisikan variabel x dengan nilai 1.

Cakupan lokal untuk variabel ini tidak dapat melihat dirinya sendiri tetapi dapat melihat variabel cakupan global x yang nilainya 10.

Ia juga dapat melihat variabel lokal z. Variabel z tidak dapat melihat dirinya sendiri tetapi dapat melihat x. Namun, karena prioritas lokal, nilai untuk x yang dilihat z adalah 1 dan bukan 10. Jadi, z menghasilkan 1 dan dengan demikian y menghasilkan 1 dan dengan demikian kueri itu sendiri menghasilkan 1. Sekarang perhatikan contoh berikut:

```
let
    x = z,
    y = 20,
    z =  let
             a = 30
         in
             a
in
    x
```

Kueri ini mengembalikan 30. Hal ini karena z berada dalam cakupan untuk x dan jelas urutan kode tidak menjadi masalah. Fakta bahwa definisi z muncul setelah x tidak relevan karena mesin M mengenali bahwa nilai untuk x bergantung pada nilai z. Pertimbangkan contoh terakhir ini:

```
let
    x = a,
    y = 20,
    z =  let
             a = 30
         in
             a
in
    x
```

Kueri ini menghasilkan kesalahan, "Nama 'a' tidak dikenali". Hal ini karena meskipun x dan y berada dalam cakupan a, hal yang sama tidak berlaku untuk a yang berada dalam cakupan x atau y. Variabel (ekspresi) yang didefinisikan dalam struktur let bersarang tidak tersedia untuk ekspresi yang didefinisikan lebih tinggi dalam hierarki bersarang.

Memahami cakupan sangat penting untuk menulis kode Power Query M yang efisien dan dapat dipelihara. Mengelola cakupan dengan benar memastikan

bahwa variabel dan fungsi digunakan dalam konteks yang tepat dan bahwa perhitungan serta manipulasi data dilakukan secara akurat. Dengan memanfaatkan cakupan lokal dan global secara strategis, Anda dapat membuat proses transformasi data yang fleksibel, modular, dan terorganisasi dalam Power Query M.

Mari kita lanjutkan eksplorasi cakupan dengan membahas tingkat cakupan teratas, lingkungan global.

D. Memeriksa lingkungan global

Dalam Power Query M, lingkungan global merujuk pada cakupan tingkat atas yang mencakup seluruh kueri Power Query. Ini mewakili tingkat visibilitas dan aksesibilitas tertinggi untuk variabel dan fungsi, yang membuatnya tersedia di seluruh kode kueri. Lingkungan global berfungsi sebagai wadah untuk variabel, fungsi, dan pengaturan global yang dapat diakses dan digunakan di seluruh langkah dan fungsi kueri.

Saat membuat kueri dalam Editor Power Query di Power BI Desktop, lingkungan global terdiri dari tiga komponen berikut:

- **Pustaka standar**: Pustaka standar bahasa pemrograman adalah kumpulan sumber daya yang tidak mudah menguap seperti data konfigurasi, dokumentasi, kelas, nilai, tipe, dan kode yang telah ditulis sebelumnya. Pustaka standar M terdiri dari elemen-elemen berikut:
  - Fungsi bawaan
  - Tipe bawaan
  - Enumerasi bawaan
- **Fungsi ekstensi bersama**: Fungsi dalam ekstensi Power Query didefinisikan dengan kata kunci shared. Lihat Bab 16, Mengaktifkan Ekstensi, untuk informasi selengkapnya.
- **Kueri saat ini**: Ini adalah kueri yang didefinisikan dalam sesi saat ini.

Anda sebenarnya dapat melihat lingkungan global ini dengan menggunakan kata kunci #shared. Di Editor Power Query, buat kueri kosong baru. Buka kueri di **Advanced Editor** dan ganti seluruh konten dengan, cukup, #shared. Perhatikan bahwa #shared adalah satu dari sekitar selusin kata kunci/pengenal yang telah ditentukan sebelumnya dalam bahasa inti M yang dimulai dengan tagar (#). Sebagian besar kata kunci/pengenal ini adalah konstruktor untuk tipe data tertentu, seperti *#binary*, *#date*, *#time*, dll. Dua kata kunci/pengenal tambahan adalah *#infinity*, yang mengembalikan simbol untuk tak terhingga, dan *#nan*, yang mengembalikan *NaN*.

Yang dikembalikan adalah tabel dengan kolom Nama dan Nilai. Dengan menggulir list, Anda akan menemukan kueri yang sudah ada yang ditetapkan dalam sesi Editor Power Query Anda dan semua fungsi, tipe, dan enumerasi M standar, serta fungsi untuk konektor pihak ketiga yang disertakan dengan Power BI Desktop.

Dengan memanfaatkan lingkungan global secara efektif dan melengkapinya dengan manajemen cakupan lokal, Anda dapat membuat proses transformasi data yang kuat, fleksibel, dan kohesif di Power Query M, seperti yang akan Anda lihat dalam contoh praktis di seluruh buku ini. Sebelum kita melanjutkan, mari kita bahas dua topik tambahan yang melibatkan lingkungan global, dimulai dengan bagian.

1. **Mempelajari bagian**

   Bagian-bagian dibahas lebih luas di Bab 16, Mengaktifkan Ekstensi. Akan tetapi, kita akan membahas topik ini secara singkat di sini untuk menjelaskan konsep keseluruhan tanpa membahas detailnya. Bagian-bagian di Power Query M adalah wadah dari *tbing*, seperti kueri, parameter, fungsi, ekspresi, dan sebagainya. Tbing-tbing ini disebut sebagai **anggota**. **Bagian** tidak dapat dibuat di editor Power Query, tetapi harus diimplementasikan sebagai **ekstensi** ke lingkungan global, yang merupakan subjek Bab 16, Mengaktifkan Ekstensi.

Meskipun bagian tidak dapat dibuat di Editor Power Query, kita dapat melihat bagian yang tersedia dalam lingkungan global dengan menggunakan kata kunci #sections. Buat kueri kosong lalu gunakan Editor Lanjutan untuk mengganti seluruh konten kueri dengan kata kunci: *#sections*.

Sebuah record dikembalikan dengan satu bidang bernama *Section1* yang berisi record. *Section1* adalah bagian default yang dibuat secara otomatis oleh Power Query. Sekarang, ubah kueri Anda sehingga kodenya seperti berikut:

```
#sections[Section1]
```

Sekarang kita memiliki record dengan kolom untuk setiap kueri yang saat ini ada di sesi Editor Power Query kita. Misalkan salah satu kueri ini adalah Kueri1. Kita dapat mengakses kueri ini menggunakan sintaks berikut (dalam kueri yang berbeda dari Kueri1):

```
#sections[Section1][Query1]
```

Kode ini mengembalikan hasil dari Query1. Namun, sintaksis yang lebih disukai untuk mencapai hal yang sama adalah:

```
Section1!Query1
```

Di sini, kita merujuk langsung ke *Section1* tanpa kata kunci *#sections* dan menggunakan tanda seru (!) sebagai jalan pintas untuk mengakses Kueri1.

Untuk setiap sesi Editor Power Query, lingkungan global berisi satu bagian, *Section1*, yang berisi semua kueri dalam Editor Power Query sebagai anggota Bagian1. Inilah yang secara efektif memungkinkan kita untuk merujuk kueri dari kueri lain. Ini memungkinkan kita untuk membuat kueri perantara yang tidak dimuat ke dalam model semantik, seperti kueri perantara, parameter, dan fungsi yang dibuat saat menggabungkan file dalam folder (lihat Bab 3, Mengakses dan Menggabungkan Data).

Menariknya, jika Anda menjalankan ekspresi *#sections* dari dalam bagian dan mengembalikan hasilnya, Anda akan mendapatkan hasil yang sangat berbeda. Alih-alih satu bagian, *Section1*, ekspresi tersebut akan

mengembalikan bagian untuk setiap ekstensi yang disertakan dengan Power BI Desktop serta ekstensi kustom apa pun.

Serupa dengan itu, jika seseorang menjalankan ekspresi #shared dari dalam bagian dan mengembalikan hasilnya, Anda juga akan mendapatkan hasil yang berbeda daripada menjalankan #shared dalam lingkungan global default. Dalam kasus ini, record yang dikembalikan hanya akan menyertakan ekspresi yang ditandai dengan kata kunci yang dibagikan dalam bagian tersebut serta semua fungsi, tipe, dan enumerasi pustaka standar M.

Dengan pembahasan singkat tentang bagian-bagian di belakang kita, selanjutnya mari kita bahas cara membuat lingkungan global kita sendiri, yang memungkinkan Anda mengontrol variabel dan fungsi yang tersedia dalam ekspresi Anda secara tepat.

2. Menciptakan lingkungan global Anda sendiri

Lingkungan global yang baru saja kita bahas adalah lingkungan global standar tempat kueri beroperasi di Editor Power Query. Namun, Anda dapat membuat lingkungan global sendiri untuk menjalankan kode Power Query M. Ini dilakukan dengan menggunakan fungsi *Expression.Evaluate*.

Fungsi Expression.Evaluate memiliki dua parameter. Parameter pertama adalah parameter dokumen. Ini adalah ekspresi M yang akan dievaluasi. Parameter kedua adalah parameter lingkungan opsional sebagai record. Ini adalah lingkungan tempat mengevaluasi ekspresi M.

Secara default, parameter kedua ini adalah *null*. Jadi, kueri berikut mengembalikan record null:

```
Expression.Evaluate("#shared")
```

Karena ini adalah **global environment** yang kosong, tidak ada bagian yang tersedia. Kita dapat mengonfirmasi hal ini dengan menggunakan kode ini:

```
Expression.Evaluate("#sections")
```

Ini juga mengembalikan record null. Namun, jika kita menambahkan *record* lingkungan seperti berikut, maka record dengan kolom x, y, dan z dengan nilai masing-masing 10, 20, dan 30 akan dikembalikan:

```
Expression.Evaluate("#shared", [x = 10, y = 20, z = 30])
```

Anda bahkan dapat meneruskan fungsi ke dalam lingkungan. Misalnya, perhatikan contoh yang lebih kompleks berikut:

```
Expression.Evaluate(
    "let Source = List.Sum( {x, y, z} ) in Source",
    [x = 10, y = 20, z = 30, List.Sum = List.Sum]
)
```

Nilai ini bernilai 60 (10 + 20 + 30). Anda bahkan dapat menggunakan ini untuk mengganti nama fungsi M standar dalam lingkungan Anda sendiri. Kode berikut juga mengembalikan nilai 60:

```
Expression.Evaluate(
    "let Source = SumThatList( {x, y, z} ) in Source",
    [x = 10, y = 20, z = 30, SumThatList = List.Sum]
)
```

Menjalankan ekspresi dalam lingkungan kita sendiri yang hanya terdiri dari elemen yang diperlukan untuk evaluasi dapat dianggap sebagai tindakan pengamanan. Tentu saja, jika Anda tidak peduli dengan hal-hal seperti itu, meneruskan #shared ke *Expression.Evaluate* karena argumen keduanya mengevaluasi ekspresi dalam lingkungan global yang sama seperti yang dibahas sebelumnya. Misalnya, berikut ini adalah ekspresi yang valid dan juga mengevaluasi ke 60:

```
Expression.Evaluate(
    "let x = 10, y = 20, z = 30
    in List.Sum( { x, y, z } )",
    #shared
)
```

Ini mengakhiri penjelajahan kita terhadap lingkungan global. Selanjutnya, mari kita bahas topik yang penting tetapi agak abstrak, yaitu closure. Memahami closure akan membantu Anda membuat kode yang dinamis dan dapat digunakan kembali dalam bahasa M.

E. Understanding *closures*

Dalam bahasa pemrograman, closure adalah konsep hebat yang memungkinkan suatu fungsi untuk menangkap dan menyimpan referensi ke variabel dari lingkungan leksikalnya (lingkungan tempat fungsi tersebut didefinisikan). Ini berarti bahwa bahkan setelah fungsi luar selesai dijalankan atau telah keluar dari cakupan, fungsi dalam (closure) masih menyimpan akses ke variabel dari cakupan yang melingkupinya.

Closure dibuat ketika suatu fungsi dalam merujuk variabel dari fungsi yang memuatnya atau cakupan lain di sekitarnya. Fungsi dalam menutup variabel tersebut, oleh karena itu disebut closure.

Kemampuan closure untuk menyimpan akses ke variabel dari lingkungan leksikalnya sangat berguna dalam skenario saat Anda perlu membuat fungsi dengan perilaku yang bergantung pada nilai variabel tertentu pada saat fungsi tersebut didefinisikan. Berikut adalah contoh sederhana closure di Power Query M:

```
let
  x = 10,
  closureFunction = () => x * 2
in
  closureFunction()
```

Kueri ini menghasilkan nilai 20. Dalam contoh ini, variabel x didefinisikan dalam blok let dan memiliki nilai 10 (cakupan kueri global). *closureFunction* adalah closure karena menangkap referensi ke variabel x dari lingkungan leksikal (saat ini). Ketika *closureFunction* dipanggil kemudian dalam kueri, ia

masih memiliki akses ke nilai x (10), dan ia akan mengembalikan hasil x * 2, yaitu 20.

Sekarang mari kita pertimbangkan contoh closure yang lebih kompleks di Power Query M. Pertimbangkan kueri berikut:

```
let
  x = 10,
  closureFunction = () => x *
  2, Evaluation =
    Expression.Evaluate(
      "closureFunction()",
      [
        closureFunction = closureFunction,
        x = 20
      ]
    )
in
  Evaluation
```

Mengingat penjelasan kita tentang cakupan dan prioritas cakupan lokal, Anda mungkin berharap bahwa kueri ini menghasilkan 40 (20 * 2). Namun, ini tidak terjadi. Ekspresi ini sebenarnya menghasilkan 20 (10 * 2). Apakah kita baru saja membuang semua pembahasan tentang cakupan dan prioritas cakupan lokal? Sama sekali tidak, dan penjelasannya adalah penutup.

Perhatikan bahwa *closureFunction* awalnya didefinisikan dalam cakupan di mana x sama dengan 10. Sama seperti contoh aslinya, *closureFunction* masih menangkap referensi ke variabel x dari lingkungan leksikal (saat ini) aslinya. Jadi, penutupan penangkapan nilai saat ini dalam konteks tempat fungsi awalnya didefinisikan lebih diutamakan daripada meneruskan nilai yang berbeda untuk x (20) ke lingkungan ekspresi *Expression.Evaluate* kita, tempat kita memanggil *closureFunction*. Dengan kata lain, nilai untuk x dari cakupan tempat fungsi awalnya didefinisikan "tertutup" dan tidak dapat diubah.

Mari kita bawa konsep konteks asli (yang mendefinisikan) ini satu langkah lebih jauh. Pertimbangkan kueri berikut:

```
let

    multiplyFunction = ( x ) as function =>
        ( multiplier ) => x * multiplier,

    closureFunction = multiplyFunction( 10 )

in

    closureFunction( 2 )
```

Kueri ini juga mengembalikan 20, tetapi bagaimana cara kerjanya? Sekilas, kode tersebut mungkin tampak akan menghasilkan kesalahan karena tampaknya tidak pernah ada waktu di mana ekspresi x dan multiplier berada dalam cakupan yang sama. Rahasianya terletak pada pembuatan fungsi yang mengembalikan fungsi dan penerapan penutupan.

Fungsi, *multiplyFunction*, menerima satu parameter, x, dan mengembalikan fungsi yang menerima satu parameter, multiplier. Kita mendefinisikan *closureFunction* sebagai fungsi yang dikembalikan dari pemanggilan *multiplyFunction* dengan nilai 10.

Penutupan berperan di sini karena *closureFunction* mengingat konteks definisi aslinya di mana x ditetapkan ke 10 melalui pemanggilan *multiplyFunction* dengan argumen 10. Dengan demikian, definisi fungsi yang dikembalikan oleh *multiplyFunction* ditutup dan didefinisikan sebagai fungsi di mana x sama dengan 10. Oleh karena itu, ketika kita kemudian memanggil closureFunction dengan nilai 2, parameter yang diteruskan adalah parameter pengali dari fungsi yang dikembalikan dari pemanggilan *multiplyFunction* dengan nilai 10, dan karenanya 10 * 2 = 20.

Penutupan bermanfaat untuk membuat fungsi yang dapat digunakan kembali dengan perilaku dinamis. Penutupan memungkinkan Anda untuk mendefinisikan fungsi yang bergantung pada nilai di luar isi fungsi, memberi Anda lebih banyak fleksibilitas dan kemampuan beradaptasi dalam kode Power

Query M Anda. Penutupan sering digunakan dalam skenario tingkat lanjut di mana Anda perlu membangun fungsi dengan perilaku khusus berdasarkan konteks atau kondisi tertentu.

Dalam hal penggunaan praktis, penutupan sangat berguna untuk dipahami saat menangani fungsi transformasi, lihat Bab 6, Nilai Terstruktur. Selain itu, penutupan pada dasarnya diperlukan untuk mengimplementasikan pelipatan kueri.

1. Lipatan kueri

Pelipatan kueri merupakan fitur Power Query secara keseluruhan dan memerlukan logika implementasi khusus dalam konektor data melalui fungsi *Table.View*. Pelipatan kueri terjadi saat kode M ditulis ulang menjadi kueri sumber data asli.

Secara umum, pelipatan kueri meningkatkan kinerja Power Query karena sumber daya sisi server dan kueri asli digunakan untuk melakukan transformasi data. Kita akan membahas pelipatan kueri secara mendalam di Bab 15, Mengoptimalkan Kinerja, tetapi perlu disebutkan di sini bahwa penutupan berperan saat mengimplementasikan pelipatan kueri.

Pertama, penting untuk dipahami bahwa bahasa M sendiri tidak mengetahui apa pun tentang pelipatan kueri. Artinya, pelipatan kueri bukanlah bagian dari mesin inti M atau pustaka standarnya.

Pada dasarnya, saat kode M dijalankan dalam Power Query Editor/Power BI Desktop, proses tersebut dicegat dan kueri ditulis ulang menggunakan pengendali peristiwa yang ditentukan dalam konektor sumber data, jika pengendali peristiwa tersebut ada. Dengan kata lain, alih-alih mesin M yang melakukan transformasi kueri, kode tersebut ditulis ulang sebagai kueri sumber asli dan dijalankan oleh sistem sumber, bukan Power Query. Pengendali peristiwa dipicu saat peristiwa tertentu terjadi, seperti mengkueri tabel basis data SQL Server. Peristiwa tersebut memicu pengendali peristiwa, yang kemudian melakukan tugas tersebut. Dalam kasus pelipatan kueri,

pengendali peristiwa bertanggung jawab untuk menulis ulang kueri M sebagai kueri yang asli dari sistem sumber. Karena pelipatan kueri bukan bagian dari bahasa inti M, kita hanya akan membahas konsep tersebut secara singkat di sini dari perspektif teknis.

Agar pelipatan kueri berfungsi, konektor sumber data harus mengimplementasikan fungsi yang mengembalikan tipe *Table.View*. Definisi *Table.View* ini harus menyertakan pengendali peristiwa tertentu yang diteruskan sebagai record. Ada dua tipe pengendali peristiwa, Get dan On. Kelas pengendali peristiwa *On* menangani perecord saat peristiwa tertentu diminta, seperti *OnSort*, *OnTake*, *OnSkip*, atau *OnSelectColumns*, sementara kelas pengendali peristiwa *Get* mengembalikan data atau informasi tentang data seperti *GetType*, *GetRows*, dan *GetRowCount*.

Pertimbangkan kueri M berikut, yang mengambil tabel *DimCurrency* dari database SQL Server dan mengurutkan tabel berdasarkan kolom *CurrencyAlternateKey*:

```
let
    Source = Sql.Database("localhost", "AdventureWorksDW2022"),
    dbo_DimCurrency = Source{[Schema="dbo",Item="DimCurrency"]}[Data],
    #"Sorted Rows" = Table.Sort(dbo_DimCurrency,{{"CurrencyAlternateKey",
Order.Ascending}})
in
    #"Sorted Rows"
```

Karena konektor untuk SQL Server mendukung pelipatan kueri dan juga mendukung pengendali peristiwa *GetRows* dan *OnSort*, kueri ini ditulis ulang sebagai kueri SQL berikut:

```
select [_].[CurrencyKey],
    [_].[CurrencyAlternateKey],
    [_].[CurrencyName]

from [dbo].[DimCurrency] as [_]
order by [_].[CurrencyAlternateKey]
```

Kueri asli ini dapat dilihat dalam Editor Power Query dengan mengklik kanan langkah terakhir di area **APPLIED STEPS** untuk kueri tersebut dan memilih **View Native Query**.

Meskipun jelas apa yang dikembalikan oleh kelas pengendali peristiwa Get, apa yang dikembalikan oleh kelas pengendali peristiwa On? Sederhana: kelas pengendali peristiwa On melakukan penutupan untuk menangkap informasi yang berkaitan dengan permintaan pelipatan dan mengembalikan objek *Table.View* yang mengingat informasi ini. Jadi, ketika pengendali peristiwa kelas Get mengakses *Table.View* yang dikembalikan dari kelas pengendali peristiwa On, *Table.View* telah mengodekan detail permintaan pelipatan.

Mekanisme cara kerjanya pada dasarnya adalah mekanisme yang sama seperti contoh terakhir kita, di mana *closureFunction* secara efektif mengingat bahwa nilai x adalah 10. Anda dapat mengetahui lebih lanjut tentang penerapan pelipatan kueri di sini: https://learn.microsoft.com/en-us/power-query/samples/trippin/10-tableview1/readme#using-tableview.

Dengan pemahaman dasar tentang penutupan, pelipatan kueri, dan bagaimana penutupan digunakan untuk memfasilitasi pelipatan kueri, selanjutnya mari kita beralih ke konsep metadata.

F. Mengelola Metadata

Sederhananya, metadata adalah data tentang data. Dalam Power Query M, metadata dapat dikaitkan dengan nilai apa pun menggunakan kata kunci meta untuk menentukan record metadata. Metadata itu sendiri tidak melakukan apa pun, juga tidak mengubah perilaku suatu nilai dengan cara apa pun. Pertimbangkan kueri berikut:

```
let
    x = 10 meta [Type = "Whole number", OoM = 1],
    y = 20 meta [Type = "Whole number", OoM = 1],
    z = 30 meta [Type = "Whole number", OoM = 1]
in
    x * y * z
```

Kueri ini masih mengembalikan 6000, sama seperti sebelumnya. Namun, metadata yang terkait dengan setiap nilai dapat diakses dan dikembalikan dengan menggunakan fungsi *Value.Metadata*, sebagai berikut:

```
let
    x = 10 meta [Type = "Whole Number", OoM = 1],
    y = 20 meta [Type = "Whole Number", OoM = 1],
    z = 30 meta [Type = "Whole Number", OoM = 1]
in
    Value.Metadata(x)[Type]
```

Kueri ini mengembalikan Bilangan Bulat. Ada dua fungsi metadata tambahan, *Value.RemoveMetadata* dan *Value.ReplaceMetadata*. *Value.RemoveMetadata* menghapus semua metadata dari suatu nilai. Misalnya, kode berikut mengembalikan record *null*:

```
let
    x = 10 meta [Type = "Whole Number", OrderOfMagnitude = 1],
    x1 = Value.RemoveMetadata(x)
in
```

Fungsi *Value.ReplaceMetadata* mengganti semua metadata dengan record metadata baru. Misalnya:

```
let
    x = 10 meta [Type = "Whole Number", OrderOfMagnitude = 1],
    x1 = Value.ReplaceMetadata(x, [Divisors = 4])
in
    Value.Metadata(x1)
```

Kueri ini mengembalikan record dengan satu kolom, *Divisors*, dengan nilai 4. Sebaliknya, kode berikut menambahkan metadata tambahan ke metadata asli:

```
let
    x = 10 meta [Type = "Whole Number", OrderOfMagnitude = 1],
    x1 = Value.ReplaceMetadata(x, Value.Metadata(x) & [Divisors = 4])
in
    Value.Metadata(x1)
```

Kueri ini mengembalikan record dengan tiga bidang, Jenis, *OoM*, dan *Divisors* dengan nilai Bilangan Bulat, 1, dan 4. Terakhir, pertimbangkan contoh ini:

```
let
    x = 10 meta [Type = "Whole Number", OrderOfMagnitude = 1],
    x1 = Value.ReplaceMetadata(x, Value.Metadata(x) & [Divisors = 4]),
    multiply = x * x1
in
    Value.Metadata(multiply)
```

Apakah outputnya adalah metadata untuk x, metadata untuk x1, atau keduanya? Jawabannya bukan salah satu dari kedua jawaban tersebut, karena outputnya sebenarnya adalah record kosong. Ini karena metadata tidak dibawa maju selama operasi.

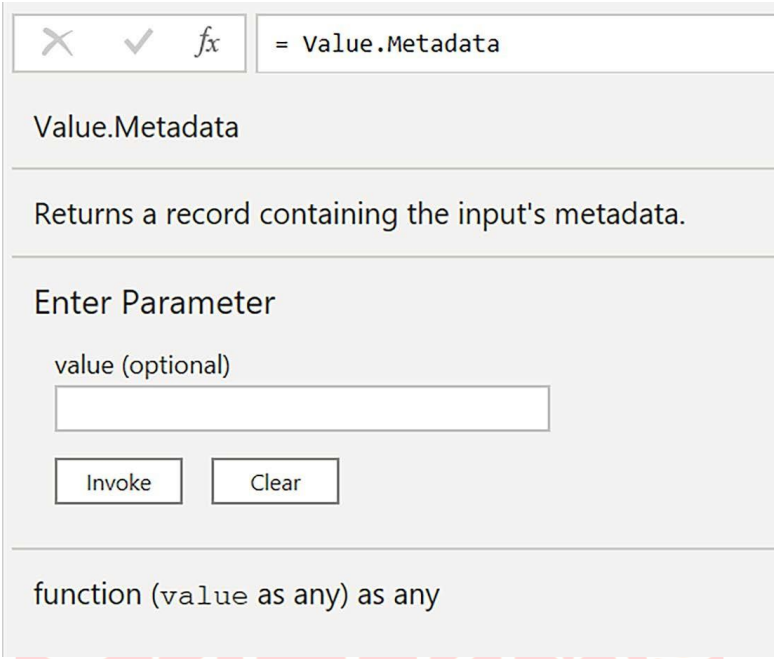
Seperti yang dinyatakan, metadata itu sendiri tidak melakukan apa pun dan tidak ada nilai metadata khusus yang dikenali oleh mesin M. Namun, aplikasi perangkat lunak seperti Power Query Editor menggunakan metadata untuk mengubah perilakunya.

Salah satu contohnya dapat dilihat saat menggunakan Get Data dan khususnya dialog Navigator. Meta-data mengontrol tampilan Navigator, seperti ikon yang ditampilkan untuk item navigasi serta apakah item tersebut adalah folder atau simpul daun yang mengembalikan data pratinjau. Informasi lebih lanjut tentang topik ini dapat ditemukan di Bab 16, Mengaktifkan Ekstensi.

Contoh lain adalah jika Anda membuat kueri yang menggunakan fungsi *Value.Metadata* terhadap kueri lain. Ini mengembalikan record dengan bidang *QueryFolding* yang memiliki record sebagai nilai. Memperluas record ini mengembalikan bidang dan nilai untuk yang berikut:

- *IsFolded*
- *HasNativeQuery*
- *Kind*
- *Path*

Aplikasi metadata yang lebih praktis dalam Editor Power Query adalah dokumentasi fungsi. Buat kueri kosong dan gunakan Editor Lanjutan untuk mengganti seluruh konten kueri dengan Value.Metadata. Informasi berikut ditampilkan:

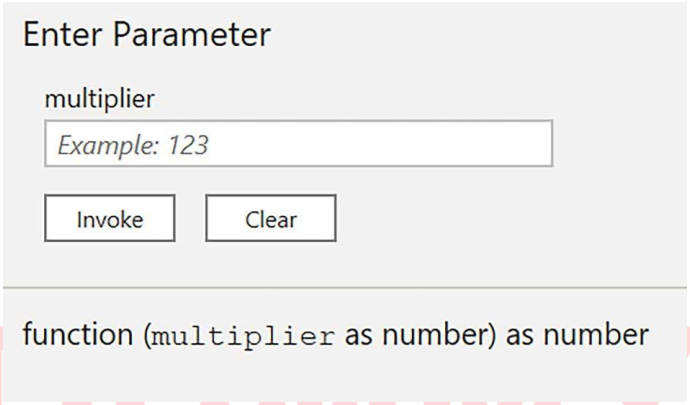


Gambar 3. 1 Dokumentasi internal untuk fungsi Value.Metadata

Bandingkan ini dengan output yang dikembalikan dari fungsi kustom, seperti kueri bernama *multiplyFunction*, dengan definisi berikut:

```
let
    Source = ( multiplier as number ) as number =>
    let
        x = 10
    in
        x * multiplier
in
    Source
```

Dalam kasus ini, informasi yang ditampilkan oleh kueri yang hanya merujuk ke *multiplyFunction* jauh lebih sedikit:

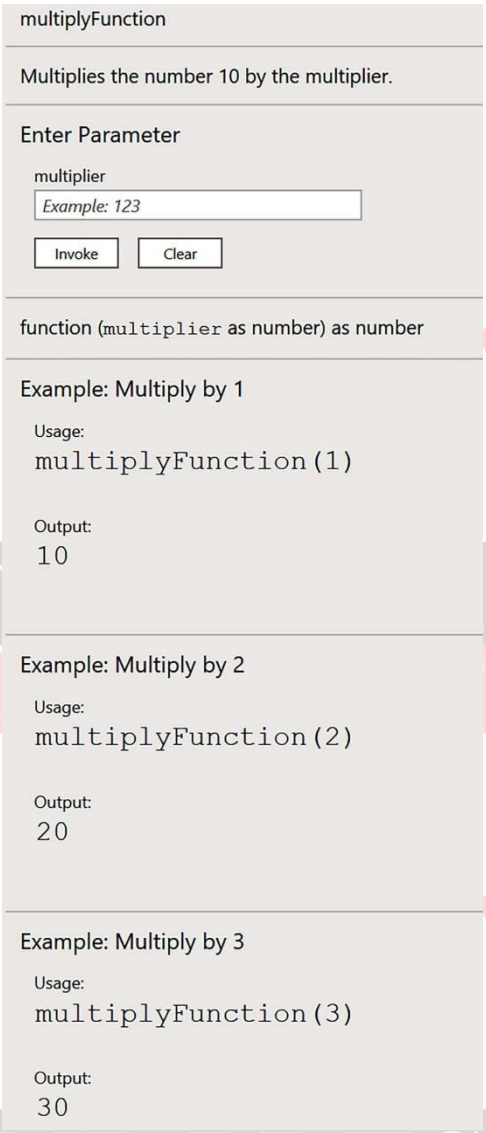


Gambar 3. 2 Dokumentasi fungsi kustom default

Namun, kita dapat menggunakan metadata untuk merapikan dokumentasi fungsi kita sebagai berikut:

```
let
    Source = ( multiplier as number ) as number =>
    let
        x = 10
    in
        x * multiplier,
    Type = type function (value as number) as number
        meta[
            Documentation.Name = "multiplyFunction",
            Documentation.LongDescription = "Multiplies the number 10 by the
multiplier.",
            Documentation.Examples =
            {
                [Description = "Multiply by 1", Code = "multiplyFunction(1)",
Result = "10"],
                [Description = "Multiply by 2", Code = "multiplyFunction(2)",
Result = "20"],
                [Description = "Multiply by 3", Code = "multiplyFunction(3)",
Result = "30"]
    }
        ]
in
    Value.ReplaceType(Source, Type)
```

Ini menghasilkan keluaran berikut:



Gambar 3. 3 Dokumentasi fungsi kustom yang ditingkatkan

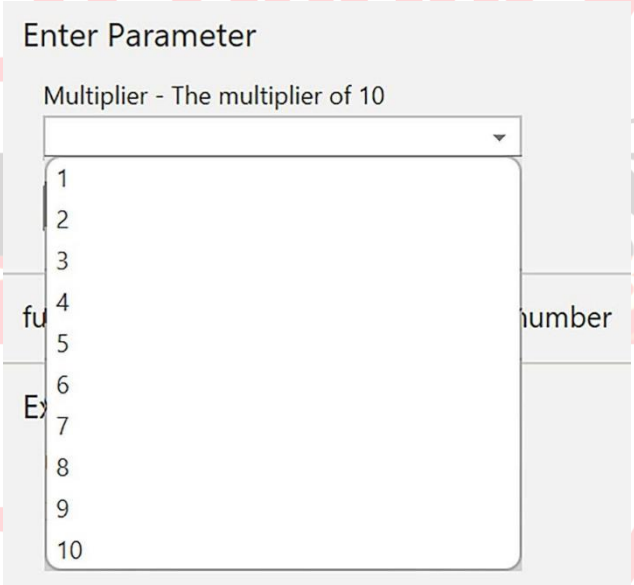
Bidang record metadata yang digunakan dalam contoh terakhir ini adalah bidang khusus yang dikenali oleh Editor Power Query. Bidang record metadata khusus ini harus dilampirkan ke jenis fungsi, bukan ke fungsi itu sendiri. Inilah sebabnya mengapa perlu membuat jenis fungsi baru dengan record metadata yang diinginkan lalu mengaitkan jenis tersebut dengan fungsi kita melalui fungsi *Value.ReplaceType*.

Ada bidang record metadata lain selain yang digunakan sejauh ini. *Documentation.Description* juga mengubah deskripsi fungsi. Record metadata ini ditimpa oleh *Documentation*. *LongDescription* jika keduanya ada. *Documentation.Syntax* dan *Documentation.Result* keduanya tidak berpengaruh pada antarmuka pengguna Power Query.

Ada juga bidang record metadata khusus yang dapat digunakan untuk parameter fungsi. Berikut adalah contoh penggunaan record metadata untuk parameter *multiplier* kita:


```
let
    Source = ( multiplier as number ) as number =>
    let
        x = 10
    in
        x * multiplier,
    Type =
        type function
        (
            multiplier as
            (
                type number
                    meta
                    [
                        Documentation.FieldCaption = "Multiplier - The
multiplier of 10",
                        Documentation.AllowedValues = { 1, 2, 3, 4, 5, 6, 7, 8,
9, 10}
                    ]
            )
        ) as number
        meta
        [
            Documentation.Name = "multiplyFunction",
            Documentation.LongDescription = "Multiplies the number 10 by the
multiplier.",
            Documentation.Examples =
            {
                [Description = "Multiply by 1", Code = "multiplyFunction(1)",
Result = "10"],
                [Description = "Multiply by 2", Code = "multiplyFunction(2)",
Result = "20"],
                [Description = "Multiply by 3", Code = "multiplyFunction(3)",
Result = "30"]
            }
        ]
in
    Value.ReplaceType(Source, Type)
```

Di sini, kita mengaitkan record metadata dalam definisi tipe untuk parameter pengali. *Documentation.FieldCaption* menggantikan keterangan parameter default sementara *Documentation.AllowedValues* mengubah bidang input ke list dropdown dari nilai yang ditentukan. Ini sebenarnya tidak mengubah nilai yang dapat diteruskan ke fungsi, hanya antarmuka dalam Editor Power Query yang terpengaruh. Metadata ini memiliki efek berikut pada antarmuka pengguna:



Gambar 3. 4 Dokumentasi parameter fungsi kustom yang disempurnakan

Bidang record metadata terkait dokumentasi lainnya tersedia untuk parameter seperti *Documentation.FieldDescription*, *Documentation.IsMultiLine*, dan *Documentation.IsCode*. Akan tetapi, bidang-bidang ini tampaknya tidak memengaruhi antarmuka Editor Power Query. Bidang lain, *Documentation.SampleValues*, mengambil list sebagai nilai, tetapi hanya nilai pertama dalam list yang ditampilkan sebagai contoh. Informasi tambahan mengenai fungsi dokumentasi dapat ditemukan di sini: https://learn.microsoft.com/en-us/power-query/handling-documentation.

Penggunaan praktis metadata pada dasarnya tidak terbatas; akan tetapi, metadata jarang digunakan oleh sebagian besar pengguna bahasa M. Hal ini

sangat disayangkan karena metadata merupakan area potensi yang belum dimanfaatkan yang memungkinkan Anda melakukan berbagai hal dalam kode M yang tidak mungkin dilakukan dengan cara lain. Misalnya, pertimbangkan kueri *PreviousSteps* berikut:

```
let
    Step1 = 10,
    Step2 = Step1 * 2,
    Step2a = Step2 meta [PreviousStep = Step1],
    Step3 = Step2 * 2,
    Step3a = Step3 meta [PreviousStep = Step2a]
in
    Step3a
```

Kueri ini mengembalikan 40. Jika kita membuat kueri lain yang merujuk ke kueri ini, maka kueri ini juga akan mengembalikan 40. Biasanya, hanya output dari langkah terakhir kueri yang dapat diakses oleh kueri lain. Namun, karena kita telah menambahkan metadata, kita sebenarnya dapat mengakses nilai dari langkah sebelumnya dalam kueri.

```
let
    Source = Value.Metadata(PreviousSteps)[PreviousStep]
in
    Source
```

Misalnya, kueri berikut mengembalikan 20, nilai dari *Step2*: Sebaliknya, kueri berikut mengembalikan 10, nilai dari *Step1*:

```
let
    Source =
        Value.Metadata(
            Value.Metadata(PreviousSteps)[PreviousStep]
        )[PreviousStep]
in
    Source
```

Ini melengkapi eksplorasi kita tentang metadata dalam bahasa Power Query M. Seperti yang disebutkan, meta-data mungkin merupakan sumber potensi terbesar yang belum dimanfaatkan dalam keseluruhan bahasa M, jadi ingatlah metadata saat Anda terus memperluas pengetahuan praktis Anda tentang M.

G. Ringkasan

Dalam bab ini, kita membahas beberapa aspek penting tetapi mungkin agak kurang jelas dan abstrak dari bahasa M, termasuk cakupan, lingkungan global, penutupan, dan metadata. Kita juga secara singkat menjelajahi bagian-bagian serta membuat lingkungan global kita sendiri menggunakan fungsi *Expression.Evaluate*. Bagian-bagian dibahas secara mendalam di Bab 16, Mengaktifkan Ekstensi.

Meskipun informasi yang dibahas dalam bab ini mungkin tampak kurang praktis dibandingkan dengan bab-bab lain, konsep yang disampaikan dalam bab ini sangat penting untuk benar-benar memahami bahasa M dan sepenuhnya memanfaatkan potensinya.

Dalam bab berikutnya, kita mulai menerapkan konsep-konsep abstrak dari bab ini ke penggunaan praktis dengan bekerja dengan struktur bersarang, di mana konsep cakupan sangat penting.

H. Penutup

1. Tes Formatif

No	Soal	Bobot
1.	Jelaskan konsep "cakupan" (scope) dalam bahasa M! Bagaimana perbedaan antara cakupan lokal dan global memengaruhi visibilitas dan aksesibilitas variabel atau fungsi dalam Power Query?	10
2.	Apa yang dimaksud dengan "lingkungan global" (global environment) dalam M? Jelaskan bagaimana fungsi *Expression.Evaluate* berperan dalam membuat lingkungan global kustom!	10
3.	Uraikan pengertian dan fungsi dari "penutupan" (closures) dalam bahasa M!. Mengapa pemahaman terhadap closures	10

	penting dalam pengembangan fungsi-fungsi yang kompleks?	
4.	Jelaskan cara kerja metadata dalam Power Query M!. Bagaimana metadata mempengaruhi interpretasi dan pemrosesan data?	10
5.	Diskusikan hubungan antara cakupan, lingkungan global, dan closures!. Bagaimana ketiga konsep tersebut saling mempengaruhi dalam pengelolaan logika kueri?	10
6.	Berikan contoh kasus penggunaan Expression.Evaluate dalam membuat kueri dinamis! Apa manfaat utama penggunaan ekspresi ini dalam pengolahan data?	10
7.	Bagaimana pengelolaan metadata dapat membantu dalam validasi dan transformasi data secara otomatis? Sertakan contoh penggunaannya dalam praktik.	10
8	Apa tantangan umum yang mungkin dihadapi ketika bekerja dengan penutupan dalam Power Query? Bagaimana cara mengatasi tantangan tersebut?	10
9.	Jelaskan bagaimana pemahaman tentang cakupan dan lingkungan global dapat meningkatkan efisiensi penulisan kode M! Berikan ilustrasi penggunaannya.	10
10.	Menurut Anda, mengapa konsep-konsep abstrak seperti cakupan, lingkungan global, penutupan, dan metadata sangat penting dalam bahasa M? Apa dampaknya jika konsep-konsep ini diabaikan dalam pengembangan kueri kompleks?	10

BAB 4

Bekerja dengan Struktur Bersarang (*Nested Structures*)

Topik Bab

Pada bab ini, topik yang akan dibahas adalah:

- Transisi ke pengkodean
- Mengubah nilai dalam tabel
- Bekerja dengan list
- Bekerja dengan catatan
- Bekerja dengan tabel
- Bekerja dengan struktur campuran

A. Pendahuluan

Struktur bersarang sangat umum dan merujuk pada organisasi hierarkis data, di mana tabel, record, atau list terdapat di dalam yang lain. Ini menyediakan cara yang efisien untuk mengatur hubungan dan hierarki serta menyimpan array nilai. Sumber data umum yang mendukung struktur bersarang meliputi basis data relasional, JSON, dan XML. Selain itu, ada berbagai fungsi M yang menghasilkan nilai terstruktur. Memahami dengan baik cara menangani struktur ini sangat penting; ini membuka banyak kemungkinan.

Bab ini membahas fungsi umum untuk bekerja dengan struktur bersarang dalam bahasa M Power Query. Fungsi ini memungkinkan Anda mengekstrak elemen tertentu dari tabel, list, dan record bersarang, memfilter dan mengubah data bersarang, dan membuat struktur baru berdasarkan yang sudah ada. Ini bertujuan untuk membekali Anda dengan pengetahuan yang diperlukan untuk menerapkan transformasi ini dalam alur kerja Anda sendiri. Topik utama yang dibahas dalam bab ini adalah:

- Transisi ke pengodean
- Transformasi nilai dalam tabel
- Bekerja dengan list
- Bekerja dengan record
- Bekerja dengan tabel
- Bekerja dengan struktur campuran

Untuk mendapatkan hasil maksimal dari bab ini, penting untuk memahami konsep yang dibahas dalam Gbapter 6, Nilai Terstruktur, yang memperkenalkan nilai terstruktur dalam bahasa M. Lebih jauh, kita menganjurkan Anda untuk membuka editor Power Query favorit Anda dan mencoba contoh yang disediakan. Dengan menjalankan dan menjelajahi skrip ini, Anda akan memperoleh pemahaman yang lebih mendalam.

1. Kasus Pemantik Berfikir Kritis: Struktur Data Bersarang

Seorang analis data mendapatkan file hasil ekspor dari sistem aplikasi penjualan online dalam format JSON. Di dalam file tersebut, setiap data pelanggan memiliki informasi bersarang seperti:

- Data pribadi (nama, alamat, email)
- Riwayat pembelian (dalam bentuk list transaksi)
- Setiap transaksi menyimpan list produk yang dibeli, termasuk jumlah dan harga.

Analis tersebut diminta oleh manajer untuk menyajikan data berupa:

- Total pembelian per pelanggan
- Produk terlaris secara keseluruhan
- Pelanggan dengan jumlah transaksi terbanyak

Namun, data ini tidak bisa langsung digunakan karena masih dalam bentuk bersarang dan tidak terstruktur seperti tabel.

Pertanyaan Pemantik:

1) Menurut kamu, apa langkah pertama yang harus dilakukan untuk mengolah data bersarang ini di Power Query?
2) Bagaimana cara mengekstrak informasi pembelian dari list bersarang menjadi tabel biasa yang bisa dianalisis?
3) Apa tantangan yang mungkin dihadapi saat mengubah struktur bersarang menjadi tabel datar (flat table)?
4) Jika kamu menjadi analis data tersebut, fitur atau fungsi Power Query apa saja yang akan kamu manfaatkan untuk menyelesaikan permintaan manajer?
5) Bagaimana cara memastikan bahwa data yang di-flatten (diratakan) tetap mempertahankan hubungan antar elemen seperti antara pelanggan dan transaksi?

6) Apakah kamu akan menggunakan antarmuka Power Query atau langsung menulis kode M untuk menangani struktur bersarang ini? Jelaskan alasan pilihanmu.

B. Transisi ke pengkodean

Sepanjang buku ini, kita telah secara bertahap mengembangkan pemahaman tentang bahasa Power Query M. Pengetahuan itu sangat penting untuk mencapai tugas manipulasi dan transformasi data yang lebih canggih. Banyak operasi umum dapat dijalankan melalui ***user interface (UI)*** Power Query, yang memungkinkan pengguna untuk mengubah nilai dengan berinteraksi dengan menu dan tombol.

Akan tetapi, ada banyak kasus di mana transformasi struktur bersarang tidak dapat dilakukan melalui UI dan diperlukan pengodean manual. Di bagian ini, kita akan membahas beberapa hal mendasar, berbagi trik untuk mendapatkan hasil maksimal dari UI, dan menyiapkan Anda dengan keterampilan dasar yang Anda perlukan untuk memulai pengodean M.

1. Memulai

Tabel merupakan struktur data utama dalam Power Query. Tabel menawarkan cara intuitif untuk merepresentasikan data, sehingga memudahkan pengguna untuk memahami dan mengolah data tersebut. Selain itu, editor Power Query secara khusus dirancang untuk mengolah tabel, menyediakan berbagai operasi transformasi tabel langsung dari UI. Baik Anda perlu memfilter, mengurutkan, menggabungkan, atau membentuk ulang data, yang Anda perlukan biasanya hanya berjarak beberapa klik saja.

Interaksi dengan kueri melalui UI menghasilkan kode M, yang dapat dilihat di bilah rumus dan **Advanced Editor**. Setiap tindakan yang dijalankan pengguna ditetapkan ke pengenal atau variabel dalam ekspresi let. Namun, UI memiliki keterbatasan. Pertama, UI tidak menyediakan akses ke semua fungsi transformasi yang tersedia dalam bahasa M, dan kedua, UI tidak dapat

membantu mengubah struktur bertingkat. Apakah ini berarti Anda harus mulai membuat kode secara ekstensif? Tidak harus: kita akan membagikan teknik yang dapat mengatasi peringatan ini dalam beberapa kasus. Namun, pertama-tama, Anda perlu memahami Drill Down dalam M.

Memahami Drill Down

Drill Down adalah fitur yang memungkinkan pengguna mengakses atau mengekstrak elemen tertentu dari struktur data kompleks lainnya. Operasi ini khususnya berguna saat menangani data bersarang, seperti kolom dalam record, elemen dalam list, atau sel dalam tabel. Pertimbangkan kueri berikut:

```
let
    t = Table.FromRows(
        {
            {"Question", "Response1", "Response2", "Response3"},
            {"Overall quality?", "High", "Medium", "Low"},
            {"Ease of use?", "Good", "Average", "Poor"},
            {"Would you recommend?", "Yes", "No", "No"}
        }
    ),
    Source = Table.FromRecords(
        {
            [Survey = "a", Results = t],
            [Survey = "b", Results = t],
            [Survey = "c", Results = t]
        }, type table [ Survey = text, Results = table]
    )
in
    Source
```

Sama seperti interaksi kueri lainnya, **Drill Down** akan secara otomatis menghasilkan kode yang diperlukan untuk mengakses nilai spesifik tersebut.

Saat bekerja dengan tabel, editor Power Query menyediakan dua cara sederhana untuk menyelami data Anda lebih dalam:

1) Dapatkan semua nilai dari kolom:

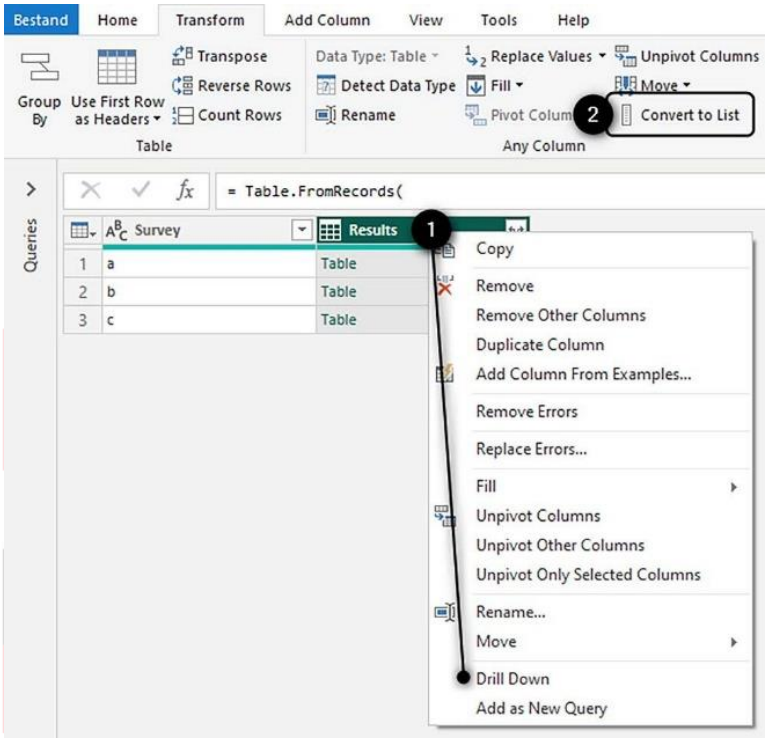
a) Metode klik kanan:

Pertama, temukan kolom yang Anda minati. Misalnya, kolom tersebut adalah kolom Hasil; lihat Gambar 8.1. Sekarang, klik kanan pada tajuk kolom tersebut **(1)**. Menu konteks akan muncul dan Anda akan melihat opsi yang bertuliskan **Drill Down**; klik saja opsi tersebut. Ini akan menambahkan langkah baru ke kueri Anda.

Anda dapat menghapus langkah ini dengan mengklik x di depan nama langkah di bagian **Applied Steps** pada panel **Query Settings**. Jika tidak terlihat, buka tab **View** untuk mengaktifkannya.

b) Menggunakan pita:

Klik tajuk kolom yang Anda minati **(1)** untuk memilihnya. Sekarang navigasikan ke tab **Transform**. Di sini, Anda akan melihat opsi yang bertuliskan **Convert to List (2)**. Saat Anda mengkliknya, langkah baru akan ditambahkan ke kueri Anda, dan kolom Anda akan diubah menjadi list. Proses ini ditunjukkan pada gambar berikut. Bila diterapkan, Anda dapat menghapus langkah ini:



Gambar 4. 1 Menelusuri lebih dalam ke kolom tabel

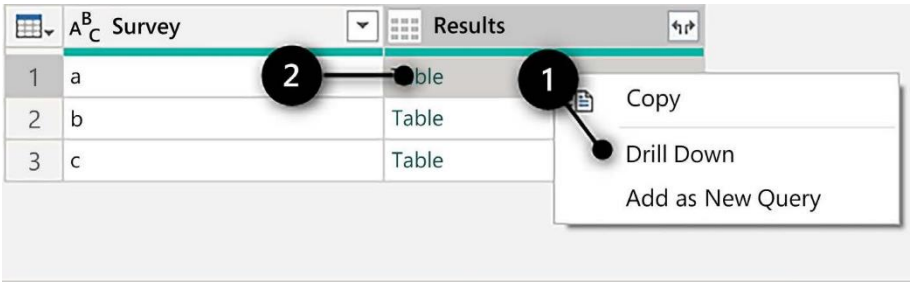
2) Dapatkan nilai dari sel:

a) Metode klik kanan:

Pertama, temukan sel yang Anda minati. Misalnya kolom Hasil untuk Survei a; lihat Gambar 8.2. Klik kanan pada spasi kosong kolom tersebut **(1)**. Menu konteks akan muncul dan Anda akan melihat opsi yang bertuliskan **Drill Down**. Saat diterapkan, Anda dapat menghapus langkah ini.

b) Klik nilai terstruktur:

Untuk melakukannya, Anda perlu mengklik nilai terstruktur (2), jika sel berisi nilai tersebut. Ini akan menghasilkan nilai sel tersebut sebagai hasilnya. Kode yang dihasilkan oleh tindakan drill-down tidak berbeda dengan pemilihan item dan/atau bidang, yang dijelaskan dalam Bab 2, Nilai Terstruktur. Ini ditunjukkan pada gambar berikut. Saat diterapkan, Anda dapat menghapus langkah ini.



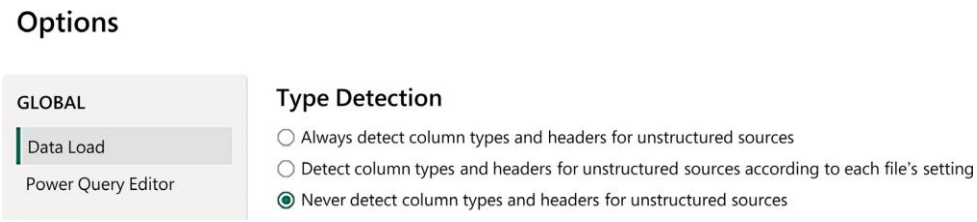
Column1	Column2	Column3	Column4
Question	Response1	Response2	Response3
Overall quality?	High	Medium	Low
Ease of use?	Good	Average	Poor
Would you recommend?	Yes	No	No

Gambar 4. 2 Menelusuri lebih dalam ke sel tabel

Trik Untuk Mendapatkan Hasil Maksimal Dari UI

UI menyediakan cara yang sangat efisien untuk menghasilkan sejumlah besar kode M. Bahkan pengode M yang paling ahli pun memanfaatkan fitur ini. Namun, mengubah tabel bersarang di dalam tabel memerlukan peralihan dari metode tunjuk dan klik yang intuitif ke teknik pengodean manual. Pendekatan manual itu penting karena UI Power Query, meskipun tangguh, memiliki keterbatasan. Secara khusus, saat menangani struktur bersarang, UI hanya dapat menampilkan nama tipe literalnya, seperti Tabel, Record, atau List. Representasi generik ini dapat membuat proses penerapan dan verifikasi transformasi langkah demi langkah menjadi rumit dan tidak efisien. Namun demikian, jika Anda memahami alur bahasa M, ekspresi let, dan **Drill Down**, Anda dapat memanfaatkan potensi penuh UI sekali lagi.

Saat mengikuti, harap perhatikan bahwa kita telah menonaktifkan deteksi tipe otomatis di Opsi dan Pengaturan Power Query:



Gambar 4. 3 Opsi dan Pengaturan Power Query

Contoh yang telah kita jelajahi memiliki dua kolom. Kolom Hasil berisi tabel bertingkat dengan respons survei, seperti yang ditunjukkan pada Gambar 4.2. Mengklik spasi (1) akan menampilkan pratinjau sekunder di bagian bawah layar Anda di panel Pratinjau, yang menawarkan tampilan terbatas dari konten dalam struktur bertingkat tersebut. Misalkan kita ingin mengubah semua tabel bertingkat di kolom ini. Anda dapat mengikuti petunjuk berikut; petunjuk ini menunjukkan pendekatan kode rendah yang akan memungkinkan Anda memanfaatkan UI lagi:

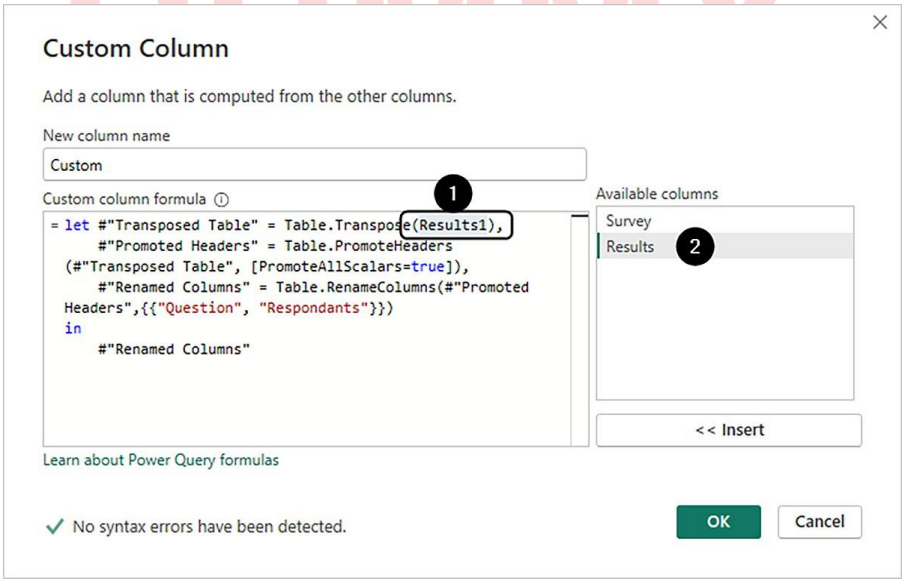
1) Klik kanan spasi **(1)**, seperti yang ditunjukkan pada Gambar 8.2, dan pilih **Add as new query** dari menu konteks. Tindakan ini menambahkan kueri baru yang disebut Hasil ke panel **Queries** di sisi kiri, yang akan menampilkan nilai tabel dari sel tersebut.
2) Klik kanan nama kueri baru ini di panel **Queries**, dan di menu konteks, nonaktifkan opsi **Enable load**; kueri ini hanya akan digunakan sebagai area pementasan untuk mengembangkan dan menghasilkan kode M guna mengubah semua tabel bertingkat.
3) Navigasi ke tab **Transform** dan pilih **Transpose**.
4) Pilih **Use First Row as Headers**.
5) Klik dua kali header kolom pertama dan ubah namanya menjadi Respondents.
6) Langkah 3 hingga 5 menambahkan langkah baru ke kueri, yang terlihat di panel **Applied Steps**. Yang pertama diberi label Transposed Table. Saatnya memeriksa kode M dan membuka **Advanced Editor**. Karena variabel ini

berisi spasi, pengidentifikasi yang dikutip telah digunakan: *#"Transposed Table"* (baris 18 dalam tangkapan layar berikut):

```
17      Results1 = Source{0}[Results],
18      #"Transposed Table" = Table.Transpose(Results1),
19      #"Promoted Headers" = Table.PromoteHeaders(#"Transposed Table", [PromoteAllScalars=true]),
20      #"Renamed Columns" = Table.RenameColumns(#"Promoted Headers",{{"Question", "Respondants"}})
21  in
22      #"Renamed Columns"
```

Gambar 4. 4 Bagian dari skrip kode M diambil dari Advanced Editor

7) Kita perlu menyalin pengenal ini dan semua yang mengikutinya, hingga akhir. Setelah selesai, tutup Advanced Editor dan kembali ke kueri awal kita. Navigasi ke tab **Add Column** dan pilih **Custom Column**. Di area **Custom Column Formula** pada kotak dialog, masukkan klausa let karena kita tidak menyalinnya sebelumnya, lalu tempel kode yang disalin.
8) Langkah terakhir adalah yang paling penting: mengganti tabel input atau nilai yang akan diubah. Perlu diketahui bahwa sebagian besar fungsi tabel menggunakan tabel sebagai argumen pertama. Setelah memeriksa kode pada Gambar 4.5, kita melihat bahwa Results1 perlu diganti. Sorot Results1 atau hapus dan masukkan **Results** dari **Available Columns** dengan mengklik dua kali:



Gambar 4. 5 Mengganti referensi tabel pada langkah transformasi awal

Saat meninjau pratinjau untuk setiap tabel bersarang, kita melihat semuanya telah berhasil diubah, seperti yang digambarkan di sini:

	Survey	Results	Custom
1	a	Table	Table
2	b	Table	Table
3	c	Table	Table

Respondants	Overall quality?	Ease of use?	Would you recommend?
Response1	High	Good	Yes
Response2	Medium	Average	No
Response3	Low	Poor	No

Gambar 4. 6 Pratinjau salah satu tabel bersarang yang telah diubah

Menggunakan trik ini akan menghemat banyak kode yang rumit, setidaknya untuk sementara, saat menangani transformasi tabel bersarang yang rumit. Namun, penting untuk diingat bahwa meskipun fleksibilitas baru ini tersedia, opsi transformasi Anda masih tunduk pada batasan UI.

Atau, Anda dapat membuat fungsi transformasi kustom Anda sendiri dari kueri helper. Proses tersebut dibahas dalam Bab 1 Jilid 3, Parameter dan Fungsi *Custom*.

Metode Untuk Transformasi Nilai Multi Langkah

Saat mentransformasi data, Anda sering kali perlu menerapkan beberapa transformasi untuk mendapatkan nilai yang tepat. Opsi apa yang tersedia untuk mengelolanya dengan baik? Sekarang saatnya untuk menyoroti empat metode berbeda yang menangani transformasi nilai multi langkah; masing-masing memiliki kelebihan dan kekurangannya.

Fungsi Nasting (bersarang)

Mirip dengan fungsi lembar kerja Excel, fungsi M dapat disarangkan. Penyarangan mengacu pada proses penggunaan fungsi untuk menghasilkan nilai argumen sebagai input untuk fungsi lain. Ini memerlukan pengetahuan

tentang fungsi dan argumennya. Saat membuat jenis logika ini, Anda biasanya memulai dari dalam dan mengerjakannya secara menyeluruh, menambahkan lebih banyak fungsi di sekitar sintaksis yang ada. Perlu diingat bahwa ini dapat dengan cepat menghasilkan kode yang rumit, sulit dibaca, dan sulit di-debug.

Menggunakan Ekspresi Let

Metode ini diilustrasikan dalam contoh sebelumnya. Metode ini memungkinkan Anda untuk memecah transformasi kompleks menjadi bagian-bagian yang lebih kecil dan lebih mudah dikelola atau membuat variabel untuk menyimpan nilai yang dibutuhkan beberapa kali. Modularitas ini membuat kode lebih terorganisasi dan sering kali lebih mudah dipahami. Ingatlah bahwa setelah klausa in, Anda dapat mengembalikan salah satu variabel atau ekspresi lainnya.

Menggunakan Ekspresi Record

Sangat mirip dengan ekspresi let, perbedaan utama antara let dan record adalah record lebih fleksibel. Hal ini membuat ekspresi record sangat berharga untuk adopsi, validasi, pemecahan masalah, dan debugging, karena Anda dapat dengan cepat mengubah nilai pengembaliannya ke seluruh record, pilihan bidang, atau nilai bidang tunggal. Semua ini dibahas secara mendalam di Bab 6, Nilai Terstruktur.

Menggunakan Fungsi Khusus (Custom)

Pengodean bentuk bebas, serta pembuatan fungsi kustom dari awal, memerlukan pemahaman bahasa M; namun, saat Anda membuat fungsi kustom dari kueri lain, kebutuhan akan pengetahuan M yang mendalam berkurang secara substansial. Yang lebih penting, opsi **Create function** yang tersedia dalam menu konteks kueri akan menyediakan kueri pembantu untuk mengembangkan dan memecahkan masalah fungsi Anda jika diperlukan. Ini, dan lebih banyak lagi, dibahas dalam Bab 1 jilid 3, Parameter dan Fungsi

Kustom. Keuntungan utama fungsi kustom adalah dapat digunakan kembali, standarisasi, dan kode yang lebih bersih melalui pendelegasian.

2. Mengubah Nilai Dalam Tabel

Mengubah nilai merupakan keterampilan mendasar dan sering kali menjadi bagian penting dari proses saat membersihkan, menyiapkan, dan membentuk ulang data untuk analisis. Bahasa M mencakup fungsi untuk mengubah kolom, baris, bidang record, dan list tabel, antara lain. Saat mengubah nilai terstruktur dalam tabel, kemungkinan besar Anda akan menemukan *Table.AddColumn*, *Table.TransformColumns*, dan *Table.ReplaceValue*. Masing-masing memiliki serangkaian kelebihan dan kasus penggunaan sendiri.

Banyak fungsi pustaka standar yang menggunakan fungsi sebagai argumen. Sering kali fungsi ini bersifat unary – artinya fungsi tersebut menerima satu argumen. Ekspresi each termasuk dalam kategori ini; ini adalah singkatan untuk mendeklarasikan fungsi tanpa tipe yang menggunakan satu parameter, garis bawah (_). Saat Anda menemukan each, penting untuk mengetahui bahwa rumus yang mengikutinya akan dieksekusi pada setiap item dalam list atau baris dalam tabel, misalnya, dan garis bawah menyediakan akses ke item saat ini. Untuk pemahaman yang lebih komprehensif, silakan lihat Gbapter 9, Parameter dan Fungsi Gustom.

Menggunakan Fungsi Khusus (Custom)

Memilih **Add Column | Custom Column** memanggil fungsi *Table.AddColumn* M. Fungsi ini menawarkan cara praktis untuk membangun logika yang lebih kompleks secara bertahap, dengan membuat nilai baru berdasarkan nilai yang sudah ada. Ini berarti bahwa saat Anda membuat kolom tambahan, nilai input apa pun dari kolom lain dalam tabel tetap utuh. Ini memastikan Anda selalu dapat melacak kembali dan membandingkan hasil Anda dengan data asli, aspek penting untuk validasi dan audit. Namun, terkadang, Anda dapat membuat kolom kustom untuk kalkulasi antara yang

tidak diperlukan dalam hasil akhir. Sebagai praktik terbaik, sebaiknya hapus semua kolom yang tidak diperlukan sebelum memuat kueri ke model data.

Pertimbangkan contoh sederhana ini: tabel kita berisi dua baris dengan nilai teks. Untuk membiasakan diri dan mempraktikkan penggunaan fungsi M, kita akan menggabungkan nilai-nilai ini dalam dua langkah. *Step1* sudah ada dan menghasilkan list untuk setiap baris, seperti yang ditunjukkan dalam cuplikan kode berikut, yang berisi semua nilai dari baris tersebut:

```
let
    Source = Table.FromRows(
        {
            {"Power", "BI"},
            {"Power", "Query"}
        }
    ),
    getAllRowValues = Table.AddColumn(
        Source,
        "Step1",
        each Record.ToList(_)
    )
in
    getAllRowValues
```

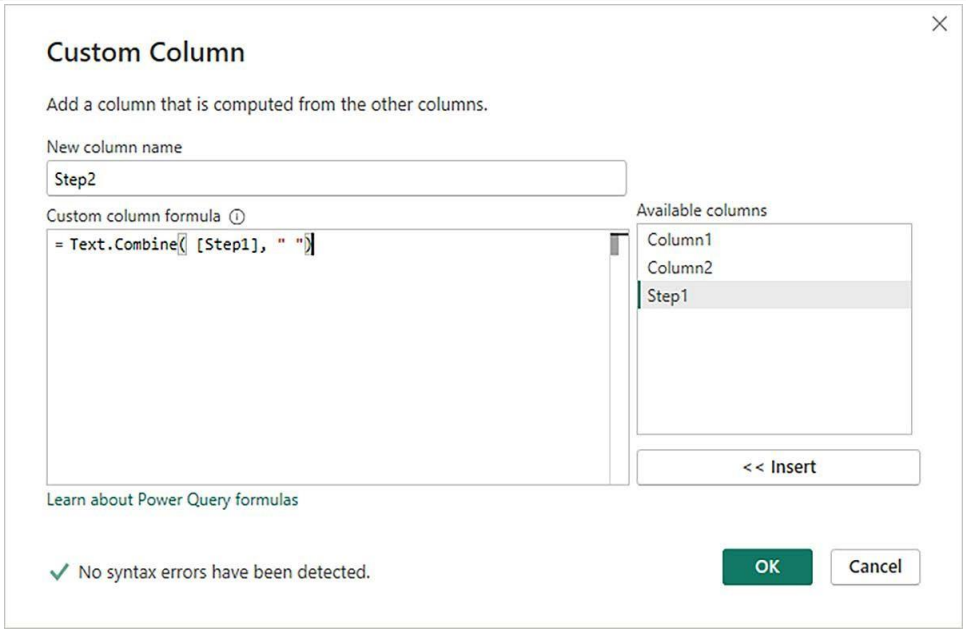
Teknik ini dan teknik lainnya dibahas di bagian Bekerja dengan catatan di bab ini.

Navigasi ke tab *Add Column* dan pilih ***Custom Column***. Kotak dialog ***Custom Column*** (Gambar 4.7) ditampilkan dan ini memungkinkan Anda melakukan hal berikut:

1) Tentukan nama kolom baru.
2) Manfaatkan bagian ***Available columns*** di sebelah kanan, untuk memasukkan nilai dari kolom lain sebagai argumen dengan mudah hanya dengan mengklik dua kali kolom tersebut.

3) Gunakan ***Custom column formula*** untuk menulis ekspresi, seperti:

```
Text.Combine( [Step1], " ")
```



Gambar 4. 7 Kotak dialog Kolom Kustom dengan area rumus dan Kolom yang tersedia

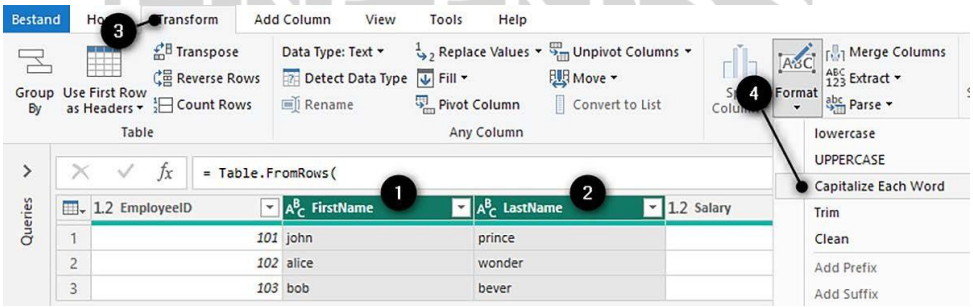
Setelah dikonfirmasi, ekspresi tersebut diterapkan ke setiap baris tabel. Pendekatan langkah demi langkah ini berguna saat merancang transformasi yang rumit. Namun, penting untuk memperhatikan implikasi kinerja yang mungkin terjadi, terutama saat menangani kumpulan data yang besar. Menambahkan banyak kolom ke kumpulan data yang cukup besar dapat memengaruhi pemuatan pratinjau, waktu eksekusi kueri, dan kinerja keseluruhan. Oleh karena itu, menghapus Langkah1, yang sekarang berlebihan, dianggap sebagai praktik terbaik. Perlu diingat bahwa meskipun kolom tersebut bukan bagian dari keluaran akhir, Anda tetap dapat meninjaunya dengan memilih langkah sebelumnya di panel ***Applied Steps***.

Table.TransformColumns

Fungsi *Table.TransformColumns* yang serbaguna dapat dipanggil dari UI, misalnya, saat menerapkan fungsi format teks. Pertimbangkan *EmployeeData* berikut:

```
let
    Source = Table.FromRows(
        {
            {101, "john", "prince", 50000},
            {102, "alice", "wonder", 60000},
            {103, "bob", "bever", 55000}
        },
        type table [
            EmployeeID=number,
            FirstName=text,
            LastName=text,
            Salary=number
        ]
    )
in
    Source
```

Pilih kolom *FirstName* (**1**) dan *LastName* (**2**) di dalam tabel *EmployeeData* sambil menahan tombol GTRL (atau tombol *Sbiß*, karena keduanya berdekatan). Selanjutnya, navigasikan ke tab **Transform** (*3*) dan pilih opsi **Format** (**4**), lalu pilih **Capitalize Each Word** (Buat Setiap Kata Bermodalkan Huruf Kapital), seperti berikut:



Gambar 4. 8 Tindakan untuk memulai transformasi yang memanfaatkan *Table.TransformColumns*

Tindakan ini memulai transformasi yang menggunakan fungsi *Table.TransformColumns*. Anda dapat menemukan kode M yang sesuai di dalam bilah rumus, meskipun kode tersebut diformat di sini agar lebih jelas. Sumber adalah tabel yang akan diubah, diikuti oleh parameter kedua dari fungsi ini, *transformOperations*, sebagai list:

```
Table.TransformColumns(Source,
    {
        {"FirstName", Text.Proper, type text},
        {"LastName", Text.Proper, type text}
    }
)
```

List *transformOperations* berisi list untuk setiap kolom yang akan ditransformasikan dalam format berikut: { *column name, transformation* } atau { *column name, transformation, new column type* }.

Namun, ketika Anda berhadapan dengan tabel yang sangat luas dan perlu menerapkan satu transformasi ke semua kolom, mengelola list *transformOperations* yang panjang dapat menjadi rumit. Sebagai gantinya, Anda dapat menentukan *defaultTransformation* untuk diterapkan ke semua kolom tabel.

Kosongkan saja list *transformOperations*, masukkan koma tepat setelahnya, dan tentukan fungsi transformasi default untuk mengonversi semua kolom menjadi teks dengan format yang tepat, seperti yang ditunjukkan di sini:

```
Table.TransformColumns(Source,
    {},
    each Text.Proper( Text.From(_))
)
```

Namun, bagaimana jika Anda perlu menerapkan satu transformasi ke semua kolom kecuali beberapa kolom tertentu? Dalam kasus seperti itu, Anda dapat menyertakan list *transformOperations* untuk setiap kolom yang ingin

Anda kecualikan dari transformasi default. Di sini, kita mengecualikan *EmployeeID* dan Salary dari transformasi *Text.Proper* default, dengan memastikan kita mempertahankan nilai saat ini:

```
Table.TransformColumns(Source,
    {
        {"EmployeeID", each _, type number},
        {"Salary", each _, type number}
    },
    Text.Proper
)
```

Selain itu, ada dua metode lanjutan yang akan kita sebutkan secara singkat di sini untuk melengkapinya. Yang pertama memungkinkan Anda mengubah pilihan kolom dengan *List.Accumulate*; fungsi ini dibahas secara terperinci dalam Bab 13, Iterasi dan Rekursi, dalam buku ini. Untuk setiap kolom yang namanya diakhiri dengan Nama, isinya akan diubah menjadi teks yang tepat:

```
List.Accumulate(
    List.Select(
        Table.ColumnNames(Source),
        each Text.EndsWith(_, "Name")
    ),
    Source,
    (s, a) => Table.TransformColumns( s,
        {a, each Text.Proper(_), type text}
    )
)
```

Metode lanjutan terakhir memungkinkan Anda mengubah satu atau beberapa pilihan kolom menggunakan *List.Transform*. Ini menyediakan cara ringkas untuk membuat beberapa list *transformOperations* untuk satu atau beberapa pilihan kolom, baik statis, seperti dalam contoh ini, atau dinamis, seperti yang ditunjukkan dalam contoh sebelumnya. Ini akan mengubah konten

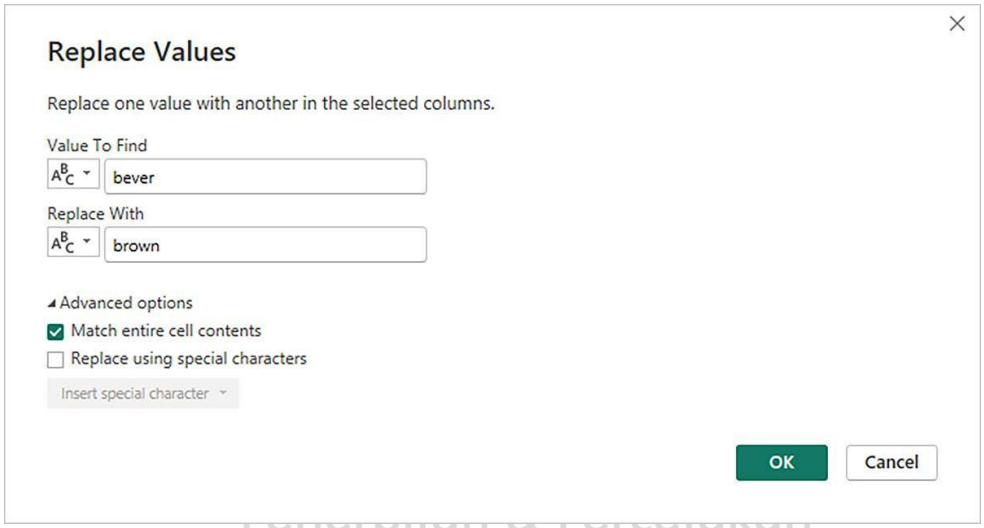
kolom *FirstName* dan *LastName* menjadi teks yang tepat dan konten kolom *Salary* menjadi teks:

```
Table.TransformColumns(Source,
    List.Combine({
        List.Transform({"FirstName", "LastName"},
            each {_, each Text.Proper(_), type text}
        ),
        List.Transform({"Salary"},
            each {_, each Text.From(_), type text}
        )
    })
)
```

Penting untuk dicatat bahwa ketika Anda menentukan lebih dari satu list *transformOperations* untuk kolom mana pun, kesalahan akan muncul. Selain itu, Anda tidak dapat mengakses kolom lain selain kolom yang sedang mengalami transformasi.

Table.ReplaceValue

Fungsi *Table.ReplaceValue* sangat dapat disesuaikan dan memungkinkan Anda memanipulasi data dengan presisi. Fungsi ini dapat dipanggil melalui UI. Kita akan menggunakan kembali data sampel yang sama, *EmployeeData*, dengan memastikannya dalam keadaan aslinya dengan hanya mempertahankan langkah *Source*. Misalnya kita ingin mengganti *LastName* bever dengan brown. Pilih kolom *LastName*, navigasikan ke tab **Transform**, dan pilih **Replace Values**. Kemudian pilih **Replace Values**; kotak dialog akan ditampilkan di mana Anda dapat memasukkan nilai untuk mencari dan nilai untuk menggantinya, dan secara opsional mengatur **Advanced options**, seperti **Match entire cell content**:



Gambar 4. 9 Kotak dialog Ganti Nilai

Saat transformasi dikonfirmasi, kode M akan dihasilkan. Anda dapat meninjau kode tersebut di dalam bilah rumus:

```
Table.ReplaceValue(
    Source,
    "bever",

    "brown",
    Replacer.ReplaceValue,

    {"LastName"}

)
```

Fungsi *Table.ReplaceValue* dipanggil dengan argumen berikut:

- *Source* mewakili tabel yang akan diubah
- *bever* adalah *oldValue* yang akan dicari dan diganti
- *brown* adalah *newValue* yang akan menggantikannya
- *Replacer.ReplaceValue* adalah fungsi replacer dari pustaka standar yang mengganti seluruh konten sel dengan nilai baru, tetapi hanya jika nilai sel saat ini sama persis dengan *oldValue* yang ditentukan
- Terakhir, *columnsToSearch* adalah list nama kolom tempat mencari oldValue dan tempat transformasi akan diterapkan jika kecocokan ditemukan

Namun, alih-alih memberikan konstanta sebagai argumen, Anda juga dapat menggunakan ekspresi. Contoh berikut mengilustrasikan metode untuk menjalankan penggantian dengan cara yang lebih terarah dan terkendali dengan menggunakan kondisi tertentu:

```
Table.ReplaceValue(
    Source,

    each [LastName],

    each if [EmployeeID]=103 then "brown" else [LastName],
    Replacer.ReplaceValue,

    {"LastName"}

)
```

Saat ingin mengganti nilai di beberapa kolom, Anda dapat memperluas list *columnsToSearch* dengan nama kolom tambahan. Namun, penting untuk dipahami bahwa nilai di kolom tersebut harus sesuai dengan *oldValue* atau kondisi yang ditentukan agar penggantian dapat dilakukan. Pertimbangkan modifikasi ini, di mana kolom *FirstName* telah disertakan dalam list kolom:

```
Table.ReplaceValue(
    Source,

    each [LastName],

    each if [EmployeeID]=103 then "brown" else [LastName],
    Replacer.ReplaceValue,

    {"FirstName", "LastName"}

)
```

Kolom FirstName dan *LastName* berisi nilai yang berbeda; oleh karena itu, nilai di kolom *FirstName* tidak akan diganti. Namun, jika kolom *FirstName* berisi bever, identik dengan kolom *LastName*, kolom tersebut akan diubah menjadi *brown*.

Meskipun ini mungkin bukan contoh yang paling praktis, mari kita lanjutkan. Misalkan kita perlu mengganti *FirstName* dan *LastName* dengan

brown untuk *EmployeeID 103*. Kemudian, Anda dapat membuat pengganti kustom seperti yang ditunjukkan di sini:

```
Table.ReplaceValue(
    Source,
    each [EmployeeID]=103,
    "brown",
    (x, y, z)=> if y then z else x,
    {"FirstName", "LastName"}
)
```

Dalam kode yang diberikan, argumen keempat dari fungsi *Table.ReplaceValue* adalah fungsi kustom yang didefinisikan dengan tiga parameter: parameter ini mewakili *currentValue*, *oldValue*, dan *newValue*. Parameter ini, meskipun diberi nama secara acak, sangat penting dalam menentukan perilaku operasi penggantian.

Fungsi ini beroperasi dengan mengevaluasi apakah kondisi yang ditetapkan oleh oldValue (y) berlaku. Jika kondisi ini terpenuhi, yang menunjukkan kriteria terpenuhi, fungsi mengembalikan newValue (z). Jika tidak, fungsi ini mempertahankan currentValue (x) dari sel. Pendekatan kondisional ini memungkinkan kontrol yang tepat atas proses penggantian, memastikan bahwa perubahan hanya dilakukan ketika kriteria tertentu terpenuhi. Untuk eksplorasi yang lebih rinci tentang replacer dan cara kerjanya, lihat *Bab 11, Gomparer, Replacer, Gombiner, dan Splitter*.

Memahami kekuatan dan keterbatasan fungsi yang disorot di sini membantu dalam memilih secara efektif salah satu yang sesuai dengan persyaratan khusus tugas transformasi data Anda. Bahasa M menawarkan berbagai fungsi. Meskipun ada lebih banyak fungsi yang tersedia, fungsi-fungsi yang dibahas di sini adalah yang paling umum digunakan dan menyediakan landasan yang kuat untuk menjelajahi sendiri fungsi-fungsi lainnya.

Seiring Anda semakin terbiasa dengan bahasa M, Anda akan segera menyadari bahwa list sering digunakan sebagai argumen dalam fungsi atau dikembalikan sebagai output. Hal ini menggarisbawahi pentingnya list; oleh karena itu, kita akan menjelajahi cara bekerja dengan list selanjutnya.

C. Bekerja dengan List (List)

Mengingat banyaknya fungsi yang dirancang untuk bekerja dengan list, signifikansinya jelas terlihat. Anda dapat menganggap list yang berisi nilai primitif mirip dengan array satu dimensi: struktur fleksibel yang memungkinkan penyimpanan dan manipulasi data dengan mudah, termasuk menambahkan, menghapus, dan memodifikasi item. Bagian ini hanya berfokus pada bekerja dengan list; list yang berisi tipe nilai lain, seperti record atau tabel, dibahas nanti dalam bab ini.

1. Mengubah list

Setiap elemen dalam list dapat dirujuk dengan indeks posisinya – angka yang mencerminkan posisi nol dalam list. Elemen pertama memiliki indeks posisi 0, diikuti oleh elemen kedua dengan indeks 1, dan seterusnya. Metode yang disebut akses item menyediakan akses ke setiap elemen list dengan menggunakan posisi indeks nol ini dalam serangkaian tanda kurung kurawal, { }.

Table.ReplaceValue

List.Transform menyediakan banyak kemungkinan. Seperti yang tersirat dari namanya, fungsi ini memungkinkan penerapan fungsi transformasi ke setiap elemen list yang terdapat dalam list yang diberikan sebagai argumen pertama. Hasil atau outputnya adalah list dengan nilai yang baru ditransformasikan. Misalnya, ini akan mengembalikan akar kuadrat dari setiap angka yang ada dalam list input:

```
List.Transform( {1, 4, 9}, Number.Sqrt )
// result: {1, 2, 3}
```

Ini menggambarkan penerapan fundamentalnya dengan sempurna. Kasus penggunaan umum lainnya untuk menggunakan *List.Transform* adalah iterasi, khususnya dalam operasi pencarian. Ini biasanya melibatkan pengoperan list indeks untuk mengekstrak nilai terkait dari list lain berdasarkan posisi indeks berbasis nol masing-masing.

Memperoleh nama kolom berdasarkan posisinya menjadi mudah. Dalam skenario seperti itu, input atau list yang akan diubah terdiri dari indeks yang mewakili item yang akan diambil. Fungsi transformasi kemudian menerapkan akses item ke list nama kolom dan mengganti setiap nomor indeks dengan nama kolom yang sesuai. Mari kita coba ini dengan tabel **EmployeeData** yang ditampilkan di bagian *Table.TransformColumns* dari bab ini. Perhatikan bahwa kueri tersebut juga harus diberi nama *EmployeeData*, sehingga dapat direferensikan seperti yang ditunjukkan di sini:

```
let
    colNames = Table.ColumnNames( EmployeeData ),
    requestPositions = { 1, 2 },
    getColNames = List.Transform(
        requestPositions,
        each colNames{_}
    )
in
    getColNames // result: FirstName, LastName
```

Ini mengembalikan nama kolom untuk kolom kedua dan ketiga tabel, *FirstName* dan *LastName*. Bab 6 menawarkan eksplorasi yang lebih mendalam tentang proses akses item, sementara Bab 13 menyediakan penelaahan mendalam tentang iterasi dan rekursi.

List.Zip

Fungsi lain yang berguna saat bekerja dengan list adalah *List.Zip*. Fungsi ini tidak dapat dipanggil dari sisi UI. Fungsi ini mengambil list list sebagai

argumen tunggal dan mengembalikan list baru dengan list yang elemen-elemennya dari semua list input telah digabungkan berdasarkan posisi indeks bersama:

```
let
    list1 = {"A", "B", "C"},
    list2 = {1, 2, 3, 4, 5},

    zippedList = List.Zip({list1, list2})
in
    zippedList
```

Lihat Gambar 8.10 untuk ilustrasi cara kerja *List.Zip*. Dalam contoh ini, elemen pertama dari list yang diberikan adalah "A" dari *list1* dan 1 dari *list2*, keduanya pada posisi indeks 0. *List.Zip* menggabungkannya dengan memilih elemen secara berurutan dari *list1* dan kemudian dari *list2*. Proses ini mengikuti urutan penyediaan list pada argumennya, sehingga menciptakan item yang baru dipasangkan: {"*A*", *1*}:

Gambar 4. 10 Memeriksa list input List.Zip dan list hasil

Penting untuk dicatat bahwa list masukan dalam contoh ini memiliki panjang yang tidak sama. Untuk mengatasi hal ini, mesin M secara otomatis mengisi *null* untuk setiap posisi terkait yang hilang. Akibatnya, dua item list bersarang terakhir akan berisi *null*. Variabel *zippedList* mengembalikan list dengan lima list bersarang, yang isinya digambarkan dalam tabel berikut:

Tabel 4. 1 Isi dari list nasting

POSISI INDEKS	NILAI
0	{"A", 1}
1	{"B", 2}
2	{"C", 3}
3	{null, 4}
4	{null, 5}

Kasus penggunaan praktis untuk *List.Zip* melibatkan pembuatan list pengganti yang berfungsi sebagai argumen untuk fungsi M lainnya. Contoh ilustrasinya adalah saat Anda perlu mengganti nama beberapa kolom dalam tabel, seperti yang ditunjukkan di sini:

```
let
    myTable = Table.FromRecords({
        [ID=1, Name="Alice"], [ID=2, Name="Bob"]
    }),
    oldNames = Table.ColumnNames(myTable),
    newNames = {"Identifier", "FullName"},
    zippedNames = List.Zip({oldNames, newNames}),
    renamedCols = Table.RenameColumns(myTable, zippedNames)
in
    renamedCols
```

Langkah *zippedNames* mengambil item dari *oldNames* dan item dari list input *newNames* dan membentuk list baru, yang masing-masing berisi nama kolom lama diikuti oleh nama kolom baru.

Proses penggantian nama kolom dalam tabel bersarang serupa. Kita akan memanfaatkan *Table.TransformColumns* untuk mengubah nilai dalam kolom Value. Transformasi ini, yang terjadi baris demi baris, diarahkan oleh ekspresi *each*, yang memungkinkan kita untuk merujuk tabel bersarang dari baris saat ini dengan memberikan garis bawah:

```
let
    myTable = Table.FromRecords({
        [Type = "nested table", Value=
            Table.FromRecords({
                [ID=1, Name="Alice"], [ID=2, Name="Bob"]
            })
        ]}),
    NestedRename = Table.TransformColumns( myTable, {{"Value", each
    let
        oldNames = Table.ColumnNames(_),
        newNames = {"Identifier", "FullName"},
        zippedNames = List.Zip({oldNames, newNames}),
        renamedCols = Table.RenameColumns(_, zippedNames)
    in
        renamedCols, type table}}
    )
in
    NestedRename
```

Dalam ekspresi let bersarang, variabel dari contoh sebelumnya digunakan kembali untuk mengubah semua tabel bersarang di kolom Nilai.

2. Mengekstraksi item

Selain akses item, hanya ada beberapa fungsi M yang tersedia yang dapat mengekstrak elemen dari list, dan jangkauannya terbatas. Fungsi-fungsi tersebut meliputi *List.First, List.Last, List.Single*, dan *List.SingleOrDefault*. Yang membuat semuanya, kecuali satu, mudah digunakan adalah bahwa dengan pengecualian *List.Single*, fungsi-fungsi tersebut tidak menimbulkan kesalahan saat list inputnya kosong; sebaliknya, fungsi-fungsi tersebut hanya mengembalikan null atau, dalam kasus *List.SingleOrDefault*, nilai default jika ada yang ditentukan. Periksa ekspresi record ini:

```
[
    myEmptyList = {},
    listFirst = List.First(myEmptyList),
    listLast = List.Last(myEmptyList),
    listSingle = List.Single( myEmptyList ),
    singleOrDefault = List.SingleOrDefault(myEmptyList),
    singleOrDefault2 = List.SingleOrDefault(myEmptyList, 99),
    myNonEmptyList = {1, 2, 3},
    listFirst2 = List.First(myNonEmptyList),
    listLast2 = List.Last(myNonEmptyList),
    listSingle2 = List.Single(myNonEmptyList),
    singleOrDefault3 = List.SingleOrDefault(myNonEmptyList, 99)
]
```

Nilai pengembalian untuk semua ekspresi dalam record ini ditunjukkan pada gambar berikut:

myEmptyList	List
listFirst	null
listLast	null
listSingle	Error
singleOrDefault	null
singleOrDefault2	99
myNonEmptyList	List
listFirst2	1
listLast2	3
listSingle2	Error
singleOrDefault3	Error

Gambar 4. 11 Mengakses item list dengan fungsi M

Harap perhatikan bahwa menyediakan list dengan lebih dari satu elemen akan menimbulkan kesalahan di *List.Single* dan *List.SingleOrDefault*. Lebih jauh, penting untuk diketahui bahwa efektivitas fungsi ekstraksi nilai ini akan meningkat secara signifikan jika dikombinasikan dengan fungsi M lain yang mengubah ukuran atau memfilter list.

3. Mengubah ukuran list

Berbagai fungsi dapat menyesuaikan ukuran list tanpa mengubah elemen yang ada di dalamnya; sebagai gantinya, fungsi tersebut membuat list baru dengan lebih sedikit atau lebih banyak elemen berdasarkan offset atau hitungan. Fungsi-fungsi tersebut meliputi *List.FirstN*, *List.RemoveLastN*, *List.Skip*, dan *List.Repeat*. Di bagian ini, kita akan menyoroti dua di antaranya: *List.Range* dan *List.Alternate*.

List.Range

Fungsi *List.Range* berguna untuk mendapatkan subset elemen list. Seperti kebanyakan fungsi list, fungsi ini tidak dapat dipanggil dari UI. Fungsi ini memerlukan dua parameter: list dan offset numerik. Secara opsional, Anda dapat menentukan jumlah. Penting untuk dicatat bahwa argumen kedua, yang dikenal sebagai offset, harus diteruskan sebagai posisi indeks awal berbasis nol, sedangkan argumen ketiga opsional, yang dikenal sebagai count, harus diteruskan sebagai jumlah item berbasis satu. Kita akan membuat serangkaian contoh yang memanfaatkan kueri ini:

```
let
    Source = Table.FromColumns(
        {
           {1..5},
           {987, 645, 843, 754, 398}
        }, type table
        [i = number, Amount = number]
    ),
    listAmount = Source[Amount]
in
    listAmount
```

Anda mungkin menyadari bahwa kita telah menelusuri kolom Jumlah, seperti yang ditunjukkan pada Gambar 4.12. Meskipun pengenal ini bernama listAmount, pengenal ini mungkin ditampilkan sebagai **Navigation** di panel **Applied Steps**. Pengenal ini telah disiapkan untuk membuat total berjalan sederhana:

List
987
645
843
754
398

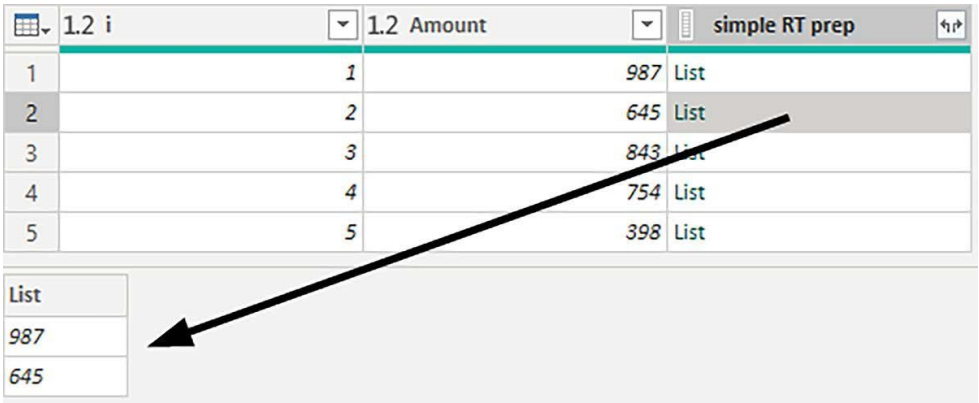
Gambar 4. 12 *listAmount*

Meskipun Bab 1 Jilid 4, *Iteration and Recursion*, memperkenalkan teknik yang lebih efisien untuk menghitung total yang berjalan, *List.Range* dapat menjadi pilihan yang layak dalam skenario tertentu. Ia menyediakan metode yang mudah untuk menghasilkan list untuk penjumlahan. Mari kita bahas:

1) Klik **fx** di sebelah bilah rumus untuk menyisipkan langkah manual.
2) Ganti ekspresi di dalam bilah rumus dengan Source.
3) Navigasi ke tab **Add column** dan pilih **Custom column**.
4) Masukkan *List.Range()* ke dalam area rumus dan masukkan variabel *listAmount* sebagai argumen pertama. Selalu mulai dari item pertama, yang berada pada posisi indeks 0, dan sertakan jumlah elemen hingga baris tabel saat ini [i], atau salin kode berikut ke dalam bilah rumus tanpa *simpleRunningTotal =*

```
simpleRunningTotal = Table.AddColumn(
    Source,
    "simple RT prep",
    each List.Range( listAmount, 0, [i] ), type list
)
```

Seperti yang dapat Anda lihat pada Gambar 4.13, nilai pada kolom i sesuai dengan jumlah elemen yang terdapat dalam setiap list. Baris teratas memiliki list dengan satu elemen, sedangkan baris terbawah memiliki list yang berisi semua elemen:



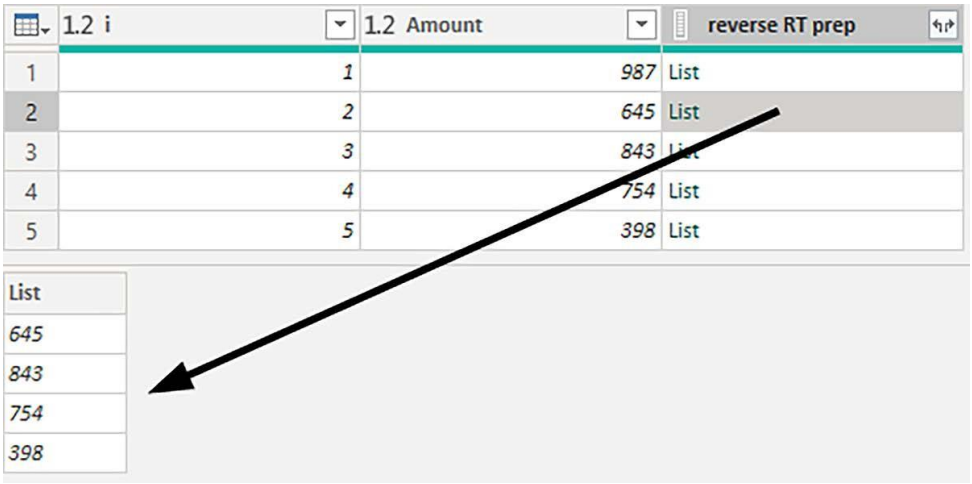
Gambar 4. 13 List baris kedua berisi dua nilai pertama dari listAmount

Mari kita balikkan dan buat total berjalan terbalik, jika Anda mau – berikut caranya:

1) Klik **fx** di sebelah bilah rumus untuk memasukkan langkah manual.
2) Ganti ekspresi di dalam bilah rumus dengan **Source**.
3) Navigasi ke tab **Add column** dan pilih **Custom column**.
4) Masukkan *List.Range()* ke dalam area rumus dan masukkan variabel *listAmount* sebagai argumen pertama. Mulai dari item, yang berada pada posisi indeks [i]-1, atau salin kode berikut ke dalam bilah rumus tanpa *reverseRunningTotal =*

```
reverseRunningTotal = Table.AddColumn(
    Source,
    "reverse RT prep",
    each List.Range( listAmount, [i]-1 ), type list
)
```

Bagian awal tetap sama; namun, kita perlu membuat argumen kedua menjadi dinamis dengan merujuk ke kolom pertama, [i], yang merupakan indeks yang dimulai dari 1. Untuk mendapatkan nilai berbasis nol, kurangi 1. Tidak perlu menentukan argumen ketiga karena kita ingin mengambil semua item list yang tersisa dari titik tersebut, seperti yang ditunjukkan pada Gambar 4.14:



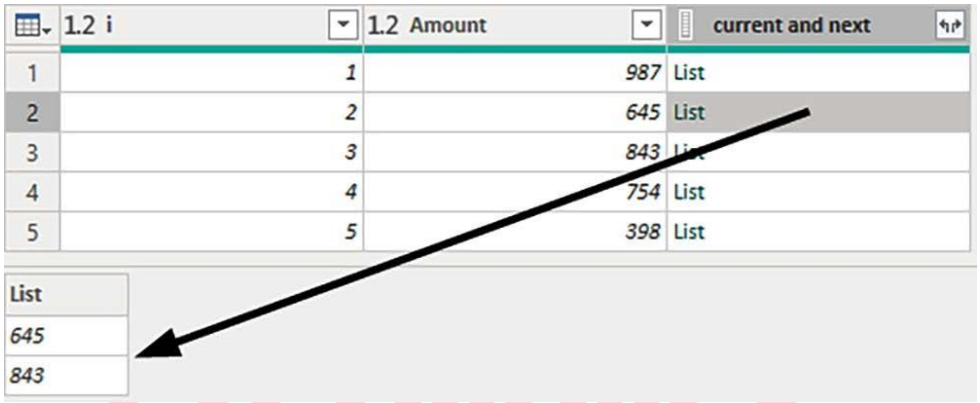
Gambar 4. 14 List baris kedua berisi semuanya kecuali nilai pertama dari *listAmount*

Baiklah, mari kita lihat betapa mudahnya mengembalikan nilai baris saat ini dan berikutnya. Pada dasarnya, yang akan berubah hanyalah menambahkan hitungan 2 sebagai argumen ketiga – berikut caranya:

1) Klik **fx** di samping bilah rumus untuk memasukkan langkah manual.
2) Ganti ekspresi di dalam bilah rumus dengan **Source**.
3) Navigasi ke tab **Add column** dan pilih **Custom column**.
4) Masukkan *List.Range()* ke dalam area rumus sebagai argumen pertama dan masukkan variabel *listAmount*. Mulai dari item, yang berada pada posisi indeks [i]-1, dan hitungan 2, atau salin kode berikut ke dalam bilah rumus tanpa *getCurrentAndNext* =:

```
getCurrentAndNext = Table.AddColumn(
    Source,
    "current and next",
    each List.Range( listAmount, [i]-1, 2 ), type list
)
```

Gambar 4.15 menunjukkan hasilnya:



	1.2 i	1.2 Amount	current and next
1	1	987	List
2	2	645	List
3	3	843	List
4	4	754	List
5	5	398	List

List
645
843

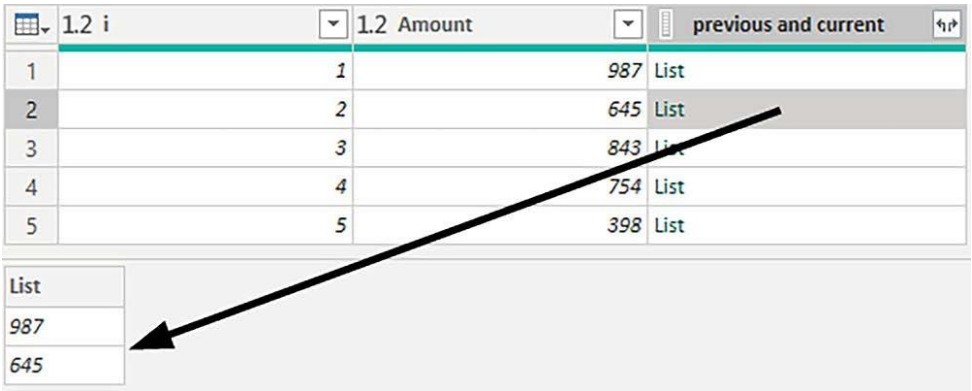
Gambar 4. 15 Dapatkan jumlah saat ini dan berikutnya

Mendapatkan nilai baris sebelumnya dan saat ini juga dapat dilakukan. Ini memerlukan penanganan kesalahan; Anda dapat mempelajari semua tentang itu di Bab 4 Jilid 3 buku ini:

1) Klik **fx** di sebelah bilah rumus untuk memasukkan langkah manual.
2) Ganti ekspresi di dalam bilah rumus dengan **Source**.
3) Navigasi ke tab **Add column** dan pilih **Custom column**.
4) Masukkan *List.Range()* ke dalam area rumus dan masukkan variabel *listAmount* sebagai argumen pertama. Mulai dari item, yang berada pada posisi indeks [i]-2, mengingat indeks dimulai dari 1 dengan hitungan 2.
5) Masukkan klausa *try* antara setiap ekspresi dan fungsi *List.Range*; setelah fungsi, masukkan klausa otherwise. Ini memungkinkan kita untuk menentukan nilai default jika terjadi kesalahan; kita akan memberikan list: {*null, [Amount]*}. Atau, Anda dapat menyalin kode berikut ke dalam bilah rumus tanpa *getPrevAndCurrent* =:

```
getPrevAndCurrent = Table.AddColumn(
    Source,

    "previous and current",

    each try List.Range( listAmount, [i]-2, 2 )
        otherwise {null, [Amount]}, type list

)
```

Hasilnya dapat dilihat pada gambar tangkapan layar berikut:



	1.2 i	1.2 Amount	previous and current
1	1	987	List
2	2	645	List
3	3	843	List
4	4	754	List
5	5	398	List

List
987
645

Gambar 4. 16 Dapatkan jumlah sebelumnya dan saat ini

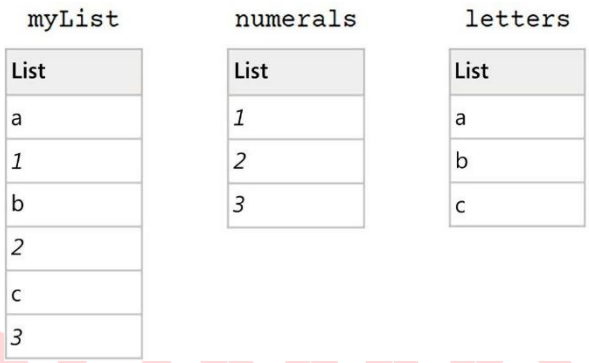
Saat menggunakan *List.Range* dalam ruang tabel, Anda memerlukan kolom indeks untuk membuat parameter offset dan count menjadi dinamis. Dua fungsi yang terkait erat dengan *List.Range* adalah *List.RemoveRange* dan *List.InsertRange*. Parameter *List.RemoveRange* berfungsi sama dengan *List.Range*. Namun, *List.InsertRange* memerlukan list nilai sebagai argumen ketiga, bukan count opsional.

List.Alternate

Di sisi lain, *List.Alternate* mengambil list, jumlah lompatan, jumlah ekstraksi opsional, dan offset opsional yang menentukan posisi indeks tempat nilai akan mulai dilewati. Kedua hitungan tersebut berbasis satu dan offset berbasis nol. Lihat catatan ini:

```
[
    myList = {"a", 1, "b", 2, "c", 3},
    numerals = List.Alternate( myList, 1, 1 ),
    letters = List.Alternate( myList, 1, 1, 1 )
]
```

Lihat Gambar 4.17 untuk mengekstrak angka dari *myList*. Kita bergantian antara melewati dan mengekstrak satu item dari atas list:



Gambar 4. 17 Mengembalikan nilai untuk setiap variabel dalam record

Akan tetapi, untuk mengekstrak huruf-huruf tersebut, kita bergantian antara melompati dan mengekstraksi satu item dengan memulai dari item list kedua, yang memiliki posisi indeks 1, dengan tetap mempertahankan item list awal sebelum lompatan dimulai.

4. Memfilter list

Mirip dengan mengubah ukuran list, operasi select dan filter mengurangi jumlah elemen dalam list – bukan berdasarkan posisi, tetapi berdasarkan nilai item list. Beberapa contohnya adalah *List.RemoveNulls*, *List.FindText*, dan *List.Select*.

List.FindText

Fungsi *List.FindText* menawarkan metode yang mudah untuk memilih semua item list yang berisi substring tertentu, yang diberikan sebagai argumen kedua. Harap perhatikan bahwa argumen kedua ini hanya menerima nilai bertipe teks, dan fungsi tersebut melakukan pencocokan peka huruf besar-kecil:

```
let

    myList = {"a", 1, "b", 2, "c", 3, "ba"},
    findText = List.FindText( myList, "a")

in

    findText  //output: {"a", "ba"}
```

List.Select

Di sisi lain, *List.Select* menerima list dan fungsi pemilihan sebagai parameternya, yang menawarkan kontrol lebih besar atas proses pemilihan. Misalnya, perhatikan record ini, yang menunjukkan cara memilih item berdasarkan *type*, *value*, atau panjang teks:

```
[
    myList = {"a", 1, "b", 2, "c", 3, "ba"},
    selectByType = List.Select( myList, each _ is number),
    selectIsOdd = List.Select( myList, each
        try Number.IsOdd(_) otherwise false),
    selectByLen = List.Select( myList, each
        try Text.Length(_) >=2 otherwise false)
]
```

Gambar 4.18 menunjukkan bahwa item hanya disimpan ketika fungsi pemilihan mengembalikan nilai *true*:

myList
List
a
1
b
2
c
3
ba

ByType
List
1
2
3

IsOdd
List
1
3

ByLen
List
ba

Gambar 4. 18 Mengembalikan nilai untuk setiap variabel dalam *record*

Penting untuk diingat bahwa list dapat berisi nilai jenis apa pun. Oleh karena itu, jika Anda tidak dapat menjamin jenis nilai setiap item list, sebaiknya sertakan penanganan kesalahan. Dalam kasus ini, menyertakan pernyataan try- otherwise akan efektif. Informasi selengkapnya tentang kesalahan dan penanganan kesalahan dapat ditemukan di Bab 4 Jilid 3, Penanganan Kesalahan dan Debugging.

Skenario *List.Select* yang lebih canggih dibahas dalam contoh praktis Bab 3 Jilid 3, *Gomparer*, *Replacer*, *Gombiner*, dan *Splitter*, seperti memilih item berdasarkan list nilai lain.

5. Konversi *To-list*

Sampai titik ini, kita telah membangun fondasi untuk meningkatkan tingkat kenyamanan Anda dalam bekerja dengan list. Sementara bahasa M menawarkan banyak fungsi list, fokus kita adalah pada fungsi yang mengubah, mengekstrak, mengubah ukuran, dan memilih item. Untuk memahami alasannya, penting untuk mengetahui konversi nilai-ke-list terstruktur apa yang tersedia untuk Anda, dengan mempertimbangkan anatomi tabel dan record. Kita akan menggunakan kueri **Employ-eeData** untuk tujuan ilustrasi. Berikut kodenya sekali lagi, demi kenyamanan Anda. Pastikan untuk memberi nama kueri ini **EmployeeData**, karena kita akan menyebutnya dengan nama ini mulai sekarang:

```
let
    Source = Table.FromRows(
        {
            {101, "john", "prince", 50000},
            {102, "alice", "wonder", 60000},
            {103, "bob", "bever", 55000}
        },
```

```
        type table [
            EmployeeID=number
            , FirstName=text,
            LastName=text,
            Salary=number
        ]
    )
in
    Source
```

Jika melihat tabel, blok penyusun utamanya mudah dikenali:

- Header, seperti nama kolom atau bidang
- Bagian data, yang merupakan semua yang ada di bawah header, dibagi menjadi kolom atau baris

Setiap kumpulan item dapat diubah menjadi list. Kita akan membahasnya di bagian berikutnya.

Nama Kolom Atau Bidang

Ekspresi record ini menunjukkan tiga fungsi umum yang akan memperoleh nama kolom atau bidang; masing-masing menghasilkan list:

```
[
    tblColNames =  Table.ColumnNames( EmployeeData ),
    recFieldNames = Record.FieldNames( EmployeeData{0} ),

    tblColumnsOfType = Table.ColumnsOfType( EmployeeData, {Text.Type})
]
```

Gambar 4.19 menunjukkan output dari ekspresi berikut:

tblColNames

List
EmployeeID
FirstName
LastName
Salary

recFieldNames

List
EmployeeID
FirstName
LastName
Salary

tblColumnsOfType

List
FirstName
LastName

Gambar 4. 19 Mendapatkan nama kolom atau bidang

Satu Kolom

Ekspresi record ini menunjukkan lima metode dan fungsi umum untuk mendapatkan semua nilai dari kolom sebagai list:

```
[
    drilldown = EmployeeData[FirstName],

    tblColumn1 = Table.Column( EmployeeData, "FirstName" ),
    tblColumn2 = Table.Column( EmployeeData,

        Table.ColumnNames( EmployeeData ){1} ),
    tblToColumns1 = Table.ToColumns( EmployeeData){1},
    tblToList1 = Table.ToList( EmployeeData[[FirstName]] )

]
```

Gambar 4.20 menunjukkan output dari ekspresi ini; semuanya menghasilkan nilai yang sama. Penting untuk dicatat bahwa urutan nilai selaras dengan urutan baris dalam tabel:

List
john
alice
bob

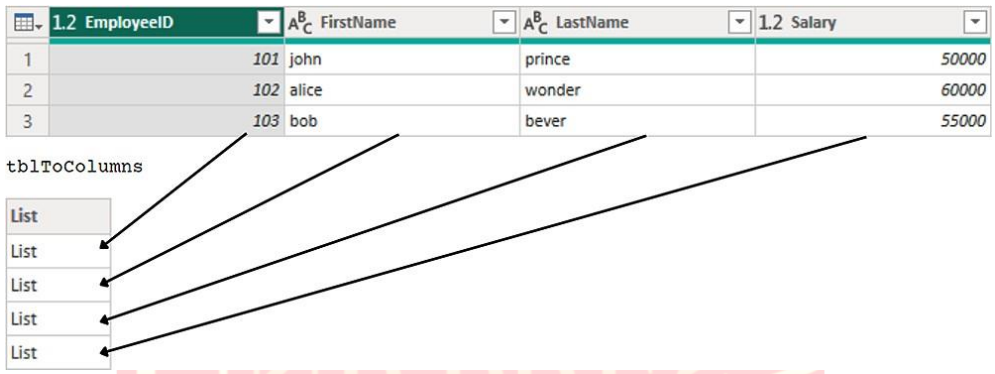
Gambar 4. 20 Mendapatkan nilai kolom tunggal

Semua Kolom

Mendapatkan semua kolom tabel akan mengembalikan list list, seperti yang diilustrasikan dalam catatan ini:

```
[
    tblToColumns = Table.ToColumns( EmployeeData )
]
```

Gambar 4.21 menunjukkan output dari ekspresi ini. Penting untuk dicatat bahwa urutan list bertingkat selaras dengan urutan kolom tabel:



Gambar 4. 21 Mendapatkan nilai kolom

Semua Baris

Metode dan fungsi untuk mendapatkan nilai dari baris diilustrasikan di sini:

```
[
    tblToRows = Table.ToRows( EmployeeData ),
    tblToList = Table.ToList(
        Table.TransformColumnTypes( EmployeeData,
            List.Transform(
                Table.ColumnNames(EmployeeData),
                each {_, type text }
            )
        )
    ),
    tblToRecords = Table.ToRecords( EmployeeData ),
    recFieldValues = Record.FieldValues( EmployeeData{0}
    ), recToList = Record.ToList( EmployeeData{0} )
]
```

Gambar 4.22 menunjukkan output dari ekspresi ini. Penting untuk dicatat bahwa urutan nilai (bertingkat) selaras dengan urutan baris tabel. Ekspresi record dapat diterapkan baris demi baris dan mengembalikan nilai dalam urutan kolom atau bidang.

tblToRows

List
List
List
List

tblToList

List
101,john,prince,50000
102,alice,wonder,60000
103,bob,bever,55000

tblToRecords

List
Record
Record
Record

recFieldValues, recToList

List
101
john
prince
50000

Gambar 4. 22 Mendapatkan nilai baris

6. Operasi lainnya

Ada operasi lain yang relevan dan umum, seperti *split* dan *combine*, seperti yang ditunjukkan dalam kode berikut:

```
[
    tblSplit = Table.Split( EmployeeData, 1),
    lst = Table.ToRows( EmployeeData),
    lstCombine = List.Combine( lst ),

    lstSplit = List.Split( lstCombine, Table.ColumnCount(EmployeeData))
]
```

Gambar 4.23 menunjukkan output dari ekspresi berikut:

tblSplit

List
Table
Table
Table

lst

List
List
List
List

lstCombine

List
101
john
prince
50000
102
alice
wonder
60000
103
bob
bever
55000

lstSplit

List
List
List
List

Gambar 4. 23 Mendapatkan nilai baris

Semua operasi ini menghasilkan nilai list yang berisi nilai primitif atau terstruktur, seperti list, record, atau tabel, seperti yang ditunjukkan pada Gambar 4.23. Memahami berbagai metode konversi yang tersedia memberikan fleksibilitas yang cukup besar, membuka jalan bagi berbagai transformasi data tingkat lanjut. Dari bagian ini dan seterusnya, banyak contoh yang memanfaatkan satu atau beberapa teknik ini akan disajikan.

Mari kita akhiri bagian ini tentang bekerja dengan list dengan dua contoh praktis.

Memperluas Beberapa Kolom List Secara Bersamaan

Bila Anda menemukan data yang disimpan dalam beberapa kolom list, seperti ditunjukkan pada Gambar 4.24, memperluas masing-masing kolom secara individual sering kali menyebabkan duplikasi nilai yang tidak diinginkan.

	Type	ID	Name
1	set 1	List	List
2	set 2	List	List

Gambar 4. 24 Tabel yang berisi beberapa kolom list

Untuk menghindari hal ini, kolom-kolom list ini dapat diubah menjadi satu struktur tunggal, dalam bentuk tabel. Dengan memperluas tabel tersebut kemudian, semua kolom list sebelumnya akan diperluas sekaligus. Pertimbangkan kueri ini:

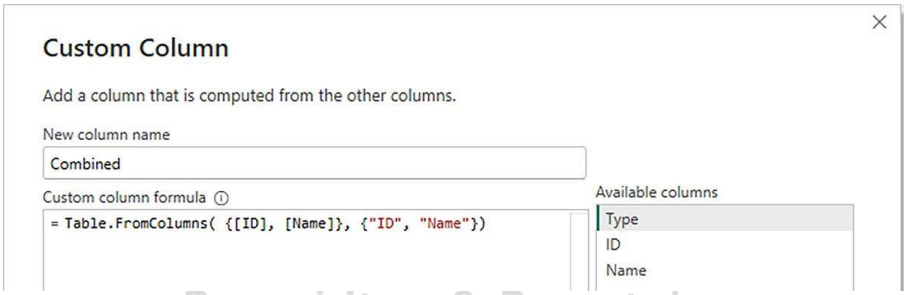

```
let
    myTable = Table.FromRecords({
        [   Type = "set 1",
            ID = {1, 2},
            Name = {"Alice", "Bob"}
        ],
        [   Type = "set 2",
            ID = {3, 4},
            Name = {"Sam", "Kate"}
        ]}),
    combinedLists = Table.AddColumn(myTable, "Combined", each
        Table.FromColumns( {[ID], [Name]}, {"ID", "Name"}),
        type table [ID=Int64.Type, Name=text]
    ),
    cleanUpColumns = Table.SelectColumns( combinedLists,
        {"Type",
        "Combined"},
        MissingField.UseNull
    ),
    ExpandAllLists =
        Table.ExpandTableColumn(cleanUpColumns,
        "Combined",
        {"ID", "Name"}, {"ID", "Name"}
    )
in
    ExpandAllLists
```

Mari kita uraikan kode dan prosesnya bersama-sama:

1) *myTable* menghasilkan tabel selebar tiga kolom. Kolom "*Type*" berisi nilai teks, dan kolom "ID" dan "*Name*" berisi list, seperti yang ditunjukkan pada Gambar 4.24. Untuk menghindari duplikasi yang tidak diinginkan saat

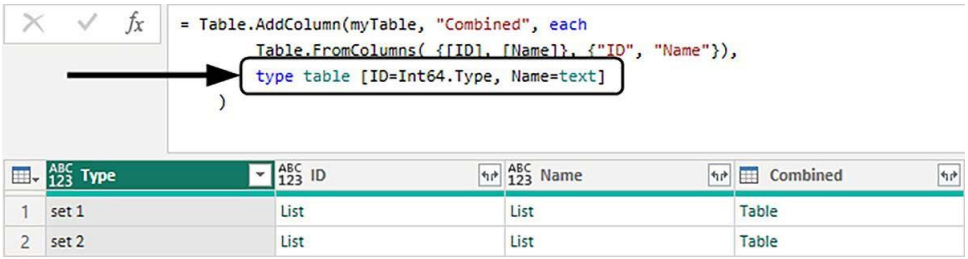
memperluas kedua kolom list, kita akan memperlakukan list tersebut sebagai nilai kolom dan menggabungkannya untuk membentuk tabel.

2) *combinedList* menambahkan kolom bernama "*Combined*" ke *myTable* menggunakan opsi **Custom Column** pada tab **Add Column**. Ini memanggil fungsi *Table.AddColumn* dan akan menampilkan kotak dialog **Custom Column**.
3) Ini memungkinkan kita untuk menentukan nama kolom baru, "Combined", dan menyisipkan rumus. Kita akan menggunakan fungsi *Table.FromColumns* dan meneruskan nilai list dari kolom "ID" dan "*Name*" sebagai list dengan membungkus serangkaian tanda kurung kurawal di sekelilingnya dan memisahkannya dengan koma, seperti ini: {*[ID]*, *[Name]*}, untuk menghasilkan tabel selebar dua kolom; sebagai argumen kedua, kita akan memberikan list dengan nama kolom baru: {*"ID"*, *"Nama"*}, seperti yang ditunjukkan pada Gambar 4.25:



Gambar 4. 25 Kotak dialog Kolom Kustom

4) Setelah mengklik tombol **OK**, kita dapat meninjau kode yang telah dibuat di dalam bilah rumus. Kita dapat memperbarui kode ini dengan menetapkan tipe pada nilai yang baru saja kita buat. Pertama, masukkan koma di antara dua tanda kurung terakhir, lalu tipe, seperti ini: type table [*ID=Int64.Type*, *Name=text*], seperti yang ditunjukkan pada Gambar 4.26:



Gambar 4. 26 Menetapkan tipe ke tabel di dalam kolom Gabungan

5) Sekarang untuk *cleanUpColumns*. Semua kolom list telah menjadi redundan: untuk menghapusnya, kita pilih **Choose Columns** pada tab **Home**. Ini memanggil fungsi *Table.SelectColumns* untuk mempertahankan kolom *Type* dan *Combined*. Memasukkan tipe *MissingField* opsional secara manual, *MissingField.UseNull*, akan melindungi struktur tabel luar dengan memasukkan nilai *null* untuk setiap kolom yang hilang.
6) *ExpandAllLists* selanjutnya mengekstrak semua nilai melalui opsi expand column yang tersedia di header kolom, yang dapat Anda temukan di sebelah kanan nama kolom, yang ditunjukkan oleh dua panah menyamping.

Alternatifnya, ada metode yang lebih canggih yang dapat mengoptimalkan proses ini dengan menghilangkan kebutuhan untuk menambahkan kolom khusus ke tabel. Metode ini melibatkan pembentukan ulang tabel sebagai bagian dari alur kerja. Untuk memeriksa pendekatan tersebut, buka **Advanced Editor**, sisipkan koma di akhir ekspresi *ExpandAllLists*, buka baris baru, lalu salin dan tempel kode berikut:

```
fxToTable = ( twoCols as list ) as table =>
    Table.FromColumns( twoCols, {"ID", "Name"}),
NoAddCols = Table.FromList(
```

```
    Table.ToList( myTable, each {_{0}} & {fxToTable({_{1},
    _{2}})}), each _,
    type table [Type=text, Combined=table [ID=Int64.Type, Name=text]]
)
```

Perbarui variabel setelah klausa in dengan mengganti *ExpandAllLists* dengan *NoAddCols*, dan klik Selesai.

Meskipun saat ini Anda tidak diharapkan untuk memahami kode ini sepenuhnya, kita akan menguraikannya untuk referensi di masa mendatang:

1) Fungsi kustom yang dikenal sebagai *fxToTable* dibuat yang disesuaikan untuk dipanggil pada nilai dari kolom 2 dan 3, yang sesuai dengan kolom *ID* dan *Name* dari *myTable*. Sintaksnya sesuai dengan ekspresi **Custom Column** yang didefinisikan (3) dalam perincian sebelumnya.
2) *NoAddCols* lebih kompleks dan melakukan beberapa operasi. Kita akan mulai dengan ekspresi internal ini:
   a) *Table.ToList(myTable, each {_{0}} & {fxToTable({_{1}, _{2}})})*
      Ini mengubah *myTable* menjadi list list nilai baris dengan menerapkan fungsi ke setiap nilai baris. Dari tabel yang ditunjukkan pada Gambar 4.24, Anda dapat melihat bahwa kita mengambil nilai dari kolom pertama dan menempatkannya di dalam list: {_{0}}. Tanda &, memungkinkan kita untuk menggabungkan list ini dengan list lain, list yang berisi tabel. Tabel tersebut dibuat dengan memanggil fungsi *fxToTable* kustom pada nilai kolom kedua dan ketiga: *{fxToTable({_{1}, _{2}})}*.
   b) *Table.FromList(..., each _)*
      Ini mengubah list list (a) menjadi tabel dengan menerapkan fungsi yang, secara default, dibagi dengan koma. Karena setiap item list adalah list yang berisi dua item, ini menghasilkan tabel selebar dua kolom.
   c) *type table [Type=text, Combined=table [ID=Int64.Type, Name=text]]*
      Table.FromList juga memungkinkan kita untuk menentukan kolom untuk tabel baru; kita menyediakan *type table* untuk menentukan nama kolom dan menetapkan tipe kolom.

Proses ini tidak hanya menghilangkan kebutuhan untuk menambahkan kolom khusus ke tabel tetapi juga menghilangkan kebutuhan untuk membersihkan kolom yang berlebihan sesudahnya.

Meratakan List Bertingkat yang Tidak Konsisten

List bertingkat dapat berisi data berharga yang memerlukan pembongkaran atau perataan untuk analisis lebih lanjut. Sering kali, Anda akan melihat rekursi digunakan untuk meratakan list bertingkat multilevel yang tidak konsisten. Namun, kita akan menyajikan alternatif untuk mencapai hasil yang sama. Ini melibatkan penggunaan fungsi seperti *Json.FromValue* dan *Text.FromBinary* untuk menghasilkan string untuk manipulasi, seperti yang ditunjukkan dalam fungsi kustom flattenList dalam kode berikut. Jangan khawatir jika saat ini Anda belum terbiasa dengan fungsi kustom, karena fungsi tersebut akan dibahas secara mendalam di bab berikutnya dalam buku ini.


```
  let
    inputList =
      {
        {"Product A", {"Product A1", "Product A2"}},
        {{"Product A1", "Product A2"}, {{"Product B"}}},
        {"Product C", {"Product C1", "Product C2",
          {"Product C2a", {{"Product C2b"}}}}, "Product D"},
        {"Product B", {{{"Product A", {
          {"Product A1", "Product A2"}}}}}},
        {"Product C", {"Product C1", "Product C2"}, {}}
      },
    myTable =
      Table.FromColumns({inputList},
      type table[to flatten = list]
    ),
    flattenList = (myList as list) as list =>
    let
      string = Text.FromBinary( Json.FromValue( myList )),
      cleanString = Text.Trim(
        Text.Remove( string, {"[", "]"}), "," ),
      convertToList = Splitter.SplitTextByDelimiter(",")(cleanString)
    in
      convertToList,
    InvokedOnList = flattenList( inputList ),
    InvokedOnTable = Table.TransformColumns( myTable,
      {{ "to flatten", flattenList, type list }}
)
  in
    InvokedOnTable
```

Kita akan menguraikan kode ini untuk referensi di masa mendatang:

1) *flattenList* adalah fungsi kustom yang mengambil input tipe list dan menghasilkan nilai output tipe list. Badan fungsi berisi ekspresi let yang dapat kita jelajahi:
   a) string menggunakan fungsi *Json.FromValue* untuk menghasilkan representasi JSON dari nilai input list, yang menerjemahkan setiap nilai list ke dalam array JSON, mengelilinginya dengan serangkaian tanda kurung siku, [ ]. Fungsi *Text.FromBinary* mengonversi JSON tersebut menjadi nilai tipe teks.
   b) *cleanString* menggunakan ekspresi *Text.Remove(string, {"[", "]"})* untuk menghapus semua tanda kurung siku, pada dasarnya menghapus semua level list. String output dapat diakhiri dengan koma jika list bersarang terakhir kosong. Untuk melindungi fungsi kita, kemungkinan tanda koma trailing ini harus dihapus: *Text.Trim(..., ",")*.
   c) *convertToList* memanggil *Splitter.SplitTextByDelimiter(",")* untuk menghasilkan fungsi yang dipanggil pada variabel cleanString untuk menghasilkan nilai tipe list.
2) *InvokedOnList* mengilustrasikan bagaimana fungsi kustom *flattenList* dapat dipanggil pada variabel *inputList*.
3) *InvokedOnTable* mengilustrasikan bagaimana fungsi kustom flattenList dapat digunakan dalam *Table.TransformColumns* untuk mengubah setiap nilai baris dalam kolom "*to flatten*".

Setelah fungsi *flattenList* dipanggil, hanya satu list yang berisi nilai teks yang tersisa. Namun, jika ada nilai teks dalam list bertingkat ini yang berisi tanda kurung siku, operasi *cleanString* akan menghapusnya; itu adalah batasan yang harus Anda ketahui.

Bahasa M mencakup beragam fungsi list. Bagian ini bertujuan untuk menyoroti beberapa fungsi pilihan untuk dijelaskan dan memberikan gambaran sekilas tentang kemampuannya. Fokus utama kita adalah

mendemonstrasikan teknik untuk mengubah, mengekstrak, mengubah ukuran, dan memilih item list, yang semuanya telah diilustrasikan melalui contoh kode yang dapat dieksekusi. Selanjutnya, kita akan mengeksplorasi cara bekerja dengan catatan.

D. Bekerja dengan Record

Catatan adalah struktur yang memungkinkan pengorganisasian data dalam bidang; setiap bidang adalah pasangan nama-nilai. Mirip dengan list, catatan dapat menyimpan berbagai jenis data dan menawarkan kemampuan untuk manipulasi, ekstraksi, dan modifikasi data.

Meskipun hanya ada sedikit koleksi fungsi catatan yang tersedia dalam bahasa M dibandingkan dengan list atau tabel, ada fungsi yang menerima catatan sebagai argumen atau mengembalikan catatan sebagai output. Bagian ini akan membahas aspek umum dalam bekerja dengan catatan. Sekali lagi, penting untuk disebutkan bahwa fokus kita adalah bekerja dengan catatan, dan kita akan membahas bekerja dengan struktur data campuran nanti di bab ini.

1. Mengubah Record

Setiap nilai bidang dalam record dapat dirujuk dengan nama bidangnya – pengenal unik dalam record. Metode yang dikenal sebagai akses bidang menyediakan akses ke setiap nilai bidang dengan menggunakan nama tersebut dalam serangkaian tanda kurung siku, [ ].

Bahasa M menyertakan fungsi yang disebut *Record.TransformFields*, yang dapat digunakan untuk mengubah nilai bidang record. Akan tetapi, fungsi ini hanya memainkan peran yang sederhana jika dibandingkan dengan *Table.TransformColumns*, yang dibahas secara luas di bagian Mengubah nilai dalam tabel. Mengubah nilai menurut kolom lebih umum daripada mengubah bidang menurut baris. Akibatnya, penggunaan fungsi *Record.TransformFields* lebih eksklusif dan sering kali disediakan untuk kasus-kasus khusus.

Selain itu, hasil yang sama dapat dicapai melalui operasi penggabungan record. Selama penggabungan record, bidang di sebelah kiri ditimpa oleh bidang-bidang dengan nama yang sama dari sebelah kanan. Pertimbangkan contoh berikut yang mengilustrasikan kedua teknik tersebut:

```
let
    rec = [
        EmployeeID = "102",
        FirstName = "Alice",
        LastName = "Wonder",
        Salary = "6000.0"
    ],
    recTransformFields = Record.TransformFields( rec,
        {
            {"EmployeeID", Number.FromText},
            {"Salary", each Number.FromText(_, "en-US")}
        }
    ),
    recMerge = rec &
        [
            EmployeeID = Number.FromText(rec[EmployeeID]),
            Salary = Number.FromText(rec[Salary], "en-US")
        ],
    recCombine = Record.Combine(
        {
            rec, [
                EmployeeID = Number.FromText(rec[EmployeeID]),
                Salary = Number.FromText(rec[Salary], "en-US")
            ]
        })
in
    recCombine
```

Gambar 4.27 menunjukkan nilai rek:

EmployeeID	102
FirstName	Alice
LastName	Wonder
Salary	6000.0

Gambar 4. 27 nilai rec sebelum modifikasi

Variabel *recTransformFields*, *recMerge*, dan *recCombine* menghasilkan hasil yang sama (terlihat pada Gambar 4.28), meskipun menggunakan metode

yang berbeda. Langkah *recTransformFields* menggunakan fungsi pustaka standar *Record.TransformFields*, yang kurang serbaguna dibandingkan *Table.TransformColumns*. Misalnya, list *transformOperations* terdiri dari list dalam format { *field name, transformation* }, satu untuk setiap bidang yang akan diubah, tanpa opsi untuk menentukan jenis.

Dapat dimengerti, ia tidak memiliki *defaultTransformation*; sebaliknya, ia dilengkapi dengan parameter *missingField* opsional.

EmployeeID	102
FirstName	Alice
LastName	Wonder
Salary	6000

Gambar 4. 28 Nilai rec modifikasi aker

Pada langkah *recMerge* dan *recCombine*, record asli direferensikan terlebih dahulu, menjadikannya operan kiri dan, diikuti oleh record yang baru dibuat sebagai operan kanan. Record baru memiliki dua bidang dengan nama bidang yang sesuai dalam record asli, yang secara efektif menimpa nilai bidang tersebut selama operasi kombinasi atau penggabungan untuk menghasilkan hasil yang sama.

2. Mengekstraksi Record

Selain akses bidang, ada dua fungsi, *Record.Field* dan *Record.FieldOrDefault*, yang dapat mengekstrak nilai bidang dari record. Tak satu pun dari fungsi ini menyertakan parameter *missingField*. Fungsi *Record.Field* akan memunculkan kesalahan jika string teks dilewatkan dan argumen kedua tidak cocok dengan salah satu nama bidang dalam record. Di sisi lain, *Record.FieldOrDefault* akan mengembalikan null atau nilai default jika itu ditentukan:

```
[
    rec = [
        EmployeeID = "102",
        FirstName = "Alice",
        LastName = "Wonder",
        Salary = 6000
    ],
    recField = Record.Field( rec, "FirstName"),
    recField2 = Record.Field( rec, "FirstNames"),
    recFieldOrDefault = Record.FieldOrDefault( rec, "FirstName"),
    recFieldOrDefault2 = Record.FieldOrDefault( rec, "FirstNames"),
    recFieldOrDefault3 = Record.FieldOrDefault( rec, "FirstNames",
    "Unknown")
]
```

Gambar 4.29 menunjukkan output dari ekspresi berikut:

rec	Record
recField	Alice
recField2	Error
recFieldOrDefault	Alice
recFieldOrDefault2	null
recFieldOrDefault3	Unknown

An error occurred in the '' query. Expression.Error: The field 'FirstNames' of the record wasn't found.

Gambar 4. 29 Output untuk ekspresi nilai Record.Field

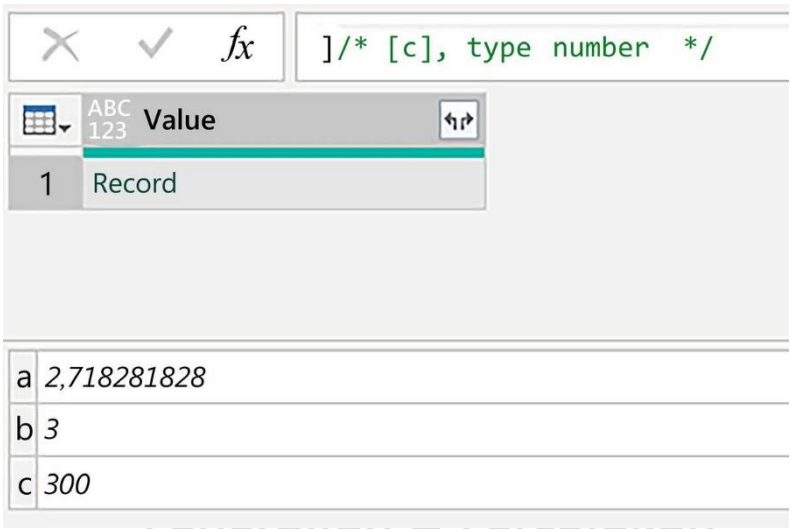
Di bagian awal bab ini, kita menjelaskan bahwa ekspresi record mirip dengan ekspresi *let*. Proses tersebut ditunjukkan di sini. Ini adalah contoh sederhana dari proses transformasi multi langkah untuk konten dalam kolom Value:

```
let
    tbl = #table(type table[Value=number], {{100}}),
    transform = Table.TransformColumns( tbl,
        {
            { "Value", each [
                    a = Number.E,
                    b = Number.RoundAwayFromZero(a, 0),
                    c = _ * b
                ][c], type number
            }
        }
    )
in
    transform
```

Perhitungan atau transformasi spesifik tidaklah penting; yang penting adalah proses pembuatan nilai dan meneruskan nilai yang baru dibuat tersebut untuk digunakan dalam ekspresi lain. Akses bidang akhirnya diterapkan pada ekspresi record yang menghasilkan hasil akhir dengan menambahkan nama bidang dalam serangkaian tanda kurung siku; di sini, nilai untuk bidang c dikembalikan: [c].

Memecahkan masalah ekspresi record bersarang itu mudah. Cukup kembalikan seluruh record dengan mengomentari akses bidang dan, jika ada, tipe yang ditetapkan; berikut langkah-langkahnya:

1) Mulailah dengan menyorot bagian kode tersebut: [c], ketik angka.
2) Tekan Alt + Shift + A pada papan ketik Anda untuk mengubahnya menjadi komentar.



Gambar 4. 30 Hilangkan akses bidang dan tipe yang ditetapkan untuk mengembalikan seluruh record

Itu akan mengembalikan tampilan record yang lengkap, yang sangat berguna saat Anda perlu memecahkan masalah atau meninjau nilai antara, seperti yang ditunjukkan pada Gambar 4.30.

3. Mengubah ukuran *Record*

Selain operator kombinasi dan proyeksi, ada empat fungsi yang dapat mengubah ukuran record: *Record.RemoveFields*, *Record.AddField*, *Record.SelectFields*, dan *Record.Combine*. Kita akan menggunakan ini untuk membuat record baru dan mengubah jumlah kolom:

```
[
    rec = [
        EmployeeID = "102",
        FirstName = "Alice",
        LastName = "Wonder",
        Salary = "6000.0"
    ],
    recRemoveFields = Record.RemoveFields( rec, {"LastName"}),
    recRemoveFields2 = Record.RemoveFields( rec, {"LastNames"},
        MissingField.Ignore ),
    recAddField = Record.AddField( rec, "fxGreeting", ()=> "Hi!" ),
    recAddField2 = Record.AddField( rec, "fxGreeting", ()=> "Hi!",
        true ),
```

```
    recCombine = Record.Combine({rec, [fxGreeting = ()=> "Hi!"]}),
    recSelectFields = Record.SelectFields( recAddField2,
        {"fxGreeting"} ),
    recSelectFields2 = Record.SelectFields( recAddField2,
        {"fxGreetings"}, MissingField.UseNull )
]
```

Record.RemoveFields dan *Record.SelectFields* hadir dengan parameter *missingField* opsional. *Record.AddField* hadir dengan parameter *delayed* opsional. Saat meneruskan fungsi parameter nol sebagai nilai argumen ketiga, definisi fungsi tersebut dievaluasi dan mengembalikan nilai fungsi. Namun, saat Anda menyetel argumen keempat opsional ke true, nilai fungsi tersebut dipanggil untuk menghasilkan nilai pengembaliannya, seperti yang ditunjukkan pada gambar berikut:

rec	Record
recRemoveFields	Record
recRemoveFields2	Record
recAddField	Record
recAddField2	Record
recCombine	Record
recSelectFields	Record
recSelectFields2	Record

EmployeeID	102
FirstName	Alice
LastName	Wonder
Salary	6000.0
fxGreeting	Hi!

Gambar 4. 31 Nilai record baru yang berisi lebih sedikit atau lebih banyak bidang

4. Memfilter Record

Tidak ada fungsi pustaka standar yang memfilter record, tetapi Anda dapat membuatnya sendiri. Kita membagikan dua contoh fungsi kustom dalam kode berikut untuk memberikan gambaran tentang apa yang mungkin; jangan

khawatir jika Anda tidak mengerti cara kerjanya saat ini. Fungsi kustom dibahas secara mendalam di Bab 9, Parameter dan Fungsi Gustom.

```
[
    rec = [
        EmployeeID = "102",
        FirstName = "Alice",
        LastName = "Wonder",
        Salary = 6000
    ],
    fxFieldsOfType =
        (
            r as record,
            t as type
        ) as record => [
            findFields = List.Select(
                Record.FieldNames(r),
                each Value.Is(Record.Field(r, _), t)
            ),
            getRecord = Record.SelectFields(
                r,
                findFields
            )
        ][getRecord],
    invokedCF1 = fxFieldsOfType(rec, Number.Type ),
    fxFieldNameStartsOrEndsWith =
        (
            r as record,
            t as text,
            optional startsWith as logical,
            optional ignoreCase as logical
        ) as record => [
            lookAtFunction = if startsWith ?? true
                then Text.StartsWith
                else Text.EndsWith,
            ignoreCase = if ignoreCase ?? true
                then Comparer.OrdinalIgnoreCase
                else Comparer.Ordinal,
            findFields = List.Select(
                Record.FieldNames(r),
                each lookAtFunction(_, t, ignoreCase)
            ),
            getRecord = Record.SelectFields(
                r,
                findFields
            )
        ][getRecord],
    invokedCF2 = fxFieldNameStartsOrEndsWith(rec, "name", false )
]
```

Fungsi kustom *fxFieldsOfType* memungkinkan Anda memilih nilai bidang dari jenis yang ditentukan:

1) *fxFieldsOfType* adalah fungsi kustom yang mengambil nilai jenis record dan nilai jenis-jenis sebagai input untuk menghasilkan nilai jenis record. Badan fungsi berisi ekspresi record yang dapat kita jelajahi lebih lanjut:
   a) *findFields* menggunakan *Record.Field* untuk memperoleh nilai dari setiap bidang dalam record dan memasoknya ke fungsi *Value.Is*, satu per satu. Ini mengevaluasi apakah jenis nilai kompatibel dengan jenis input (t). Jika benar, *List.Select* mempertahankan nama bidang yang disediakan oleh fungsi *Record.FieldNames(r)*, jika tidak, ia menghilangkan nama bidang.
   b) *getRecord* menggunakan fungsi *Record.SelectFields* untuk memilih semua bidang dari record (r) yang selamat dari pemeriksaan kompatibilitas dan yang namanya tercantum dalam list *findFields*.
   c) *[getRecord]* menerapkan akses bidang ke ekspresi record untuk mengembalikan nilai dari bidang *getRecord*.
2) *invokedCF1* mengilustrasikan bagaimana fungsi kustom *fxFieldsOfType* dapat dipanggil pada suatu record dengan memberikan nilai record dan nilai tipe record sebagai argumen.

Di sisi lain, *fxFieldNameStartsOrEndsWith* memungkinkan Anda memilih nama kolom yang dimulai atau diakhiri dengan substring tertentu, mengabaikan huruf besar/kecil secara default:

1) *fxFieldNameStartsOrEndsWith* adalah fungsi kustom yang memiliki empat parameter dan akan menghasilkan nilai tipe record. Dua dari parameter ini wajib diisi dan dua opsional:
   a) *r* sebagai *record* wajib diisi dan menerima nilai *record-type*
   b) *t* sebagai *text* wajib diisi dan menerima nilai *text-type*
   c) *startsWith* opsional sebagai logika wajib diisi dan default ke *true*
   d) *ignoreCase* opsional sebagai logika wajib diisi dan default ke *true*

Isi fungsi berisi ekspresi record yang dapat kita jelajahi lebih lanjut.

a) *lookAtFunction* adalah pernyataan kondisional yang mengambil input *startsWith*, yang menerapkan *coalesce: ??*, default ke *true*, dan *Text.StartsWith* saat tidak ada nilai yang ditentukan. Saat salah, disetel ke *Text.EndsWith*.

b) *ignoreCase* bekerja dengan cara yang sama, default ke *true* dan *Comparer.OrdinalIgnoreCase*. Saat salah, disetel ke *Comparer.Ordinal*.

c) *findFields* menggunakan fungsi *List.Select* untuk menyimpan semua nama bidang dari list yang disediakan oleh fungsi *Record.FieldNames* yang mana *lookAtFunction(_, t, ignoreCase)* yang diterapkan mengembalikan nilai *true*.

2) *invokedCF2* mengilustrasikan bagaimana fungsi kustom *fxFieldNameStartsOrEndsWith* dipanggil pada sebuah record dengan menyediakan record (*rec*) dan string (“*name*") dan menyetel *startsWith* (*false*) opsional.

5. Konversi *To-record*

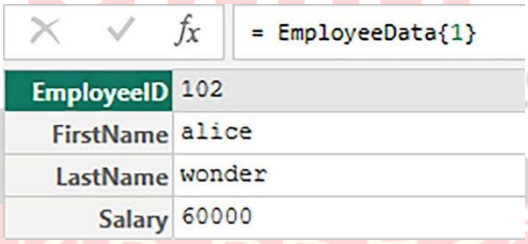
Kita telah mengeksplorasi berbagai cara untuk mengubah, mengekstrak, mengubah ukuran, dan memilih record. Dari perspektif tabel, setiap tabel dapat diubah menjadi list record. Setiap baris dalam tabel mewakili record atau entri unik, dengan propertinya yang disusun di seluruh kolom. Kolom-kolom ini sesuai dengan bidang yang menyimpan nilai untuk setiap properti record. Sekarang, mari kita jelajahi konversi nilai-ke-record yang terstruktur.

Baris Tabel Untuk *Record*

Mengakses satu baris dalam tabel akan menghasilkan record. Hal ini sebenarnya dapat ditunjukkan tanpa menggunakan kode M. Buat kueri kosong baru dan rujuk tabel *EmployeeData* dengan memasukkan “= *EmployeeData*" langsung ke bilah rumus. Tabel memperlihatkan nomor baris di awal setiap baris dalam tabel; nomor ini bukan bagian dari data dan tidak dapat diakses

atau diminta. Namun, saat Anda mengklik nomor baris, seperti 2, pratinjau baris tersebut akan ditampilkan sebagai record di bagian bawah panel pratinjau.

Untuk memperoleh baris tersebut, penting untuk menyatakan kembali bahwa indeks posisional dalam M berbasis nol. Oleh karena itu, untuk memperoleh record kedua, Anda perlu menerapkan operator indeks posisional dan menyertakan 1 (ditunjukkan pada Gambar 4.32), seperti ini: *EmployeeData{1}*:



Gambar 4. 32 Mengakses satu baris dari tabel mengembalikan sebuah record

Untuk membuat catatan untuk setiap baris tabel, Anda dapat menambahkan **Custom Column** dan merujuk ke baris (atau catatan) saat ini dengan memasukkan garis bawah di bagian rumus kotak dialog. Ini akan menghasilkan ekspresi yang mirip dengan berikut ini:

```
Table.AddColumn( EmployeeData, "rec", each _ )
```

Kolom baru telah ditambahkan ke tabel, yang berisi record bertingkat untuk setiap baris. Hal ini dan lainnya dibahas dalam Bab 2, Nilai Terstruktur.

Baris Tabel Untuk *Record*

Kita telah membahas proses pengambilan satu record dari tabel dan pembuatan record untuk setiap baris dalam tabel, yang keduanya merupakan transformasi umum. Namun, mengonversi seluruh tabel menjadi satu record cukup jarang. Hal ini dikarenakan persyaratan bahwa nama kolom dalam record harus unik dan berjenis teks. Oleh karena itu, konversi tabel ke record

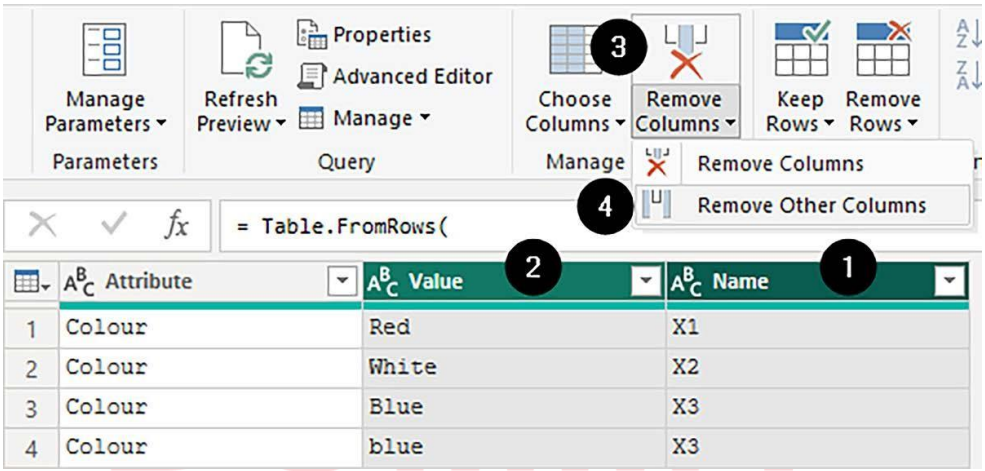
tidak hanya terbatas pada tabel selebar dua kolom, tetapi kolom pertama juga harus berisi kunci tekstual yang unik.

Aplikasi praktis untuk proses ini adalah membuat record pencarian yang dapat digunakan dalam penggantian pencocokan teks persis, di mana konten sel diganti secara bersyarat oleh nilai lain. Pertimbangkan kueri ini:

```
let
    Attributes = Table.FromRows(
        {
            {"Colour", "Red", "X1"},
            {"Colour", "White", "X2"},
            {"Colour", "Blue", "X3"},
            {"Colour", "blue", "X3"}
        }, type table[Attribute=text, Value=text, Name=text]
    )
in
    Attributes
```

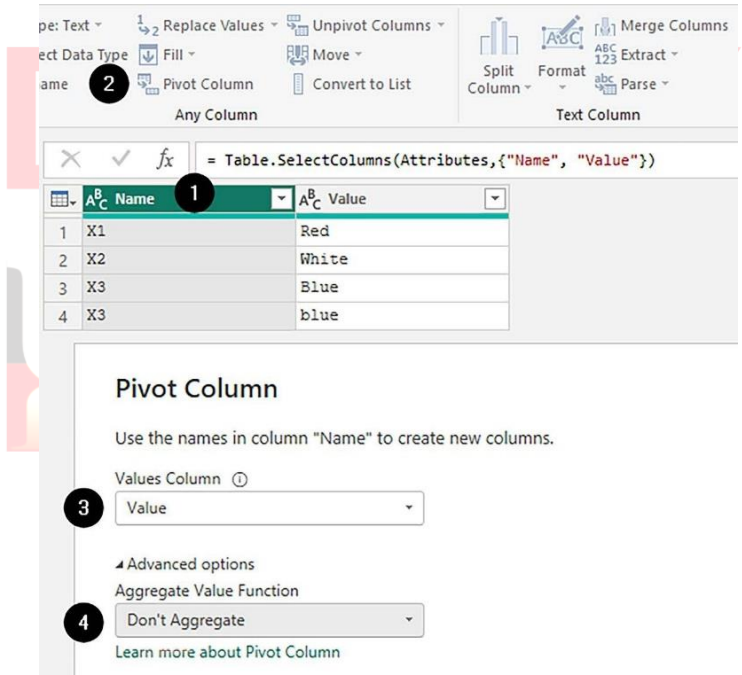
Untuk membuat record pencarian dari tabel, ikuti langkah-langkah berikut:

1) Kita perlu memastikan bahwa tabel ini memiliki dua kolom. Kolom pertama harus berisi nilai teks unik, yang akan digunakan sebagai nama bidang record: ini akan menjadi kolom Nama. Kolom kedua harus berisi nilai bidang; nilai tersebut dapat berupa jenis apa pun – ini akan menjadi kolom Nilai. Dalam record pencarian, nama bidang bertindak sebagai teks yang akan dicocokkan dan nilai bidang sebagai nilai pengganti yang sesuai.
2) Tekan terus tombol GTRL dan pilih kolom dalam urutan ini: kolom Nama terlebih dahulu (**1**), kemudian kolom Nilai (**2**). Sekarang pilih **Remove Columns** (**3**), kemudian klik **Remove Other Columns** (**4**), seperti yang ditunjukkan pada Gambar 4.33:



Gambar 4. 33 Urutan pemilihan kolom dan menghapus kolom lainnya

3) Pastikan kolom pertama (Nama) dipilih (**1**), navigasikan ke tab **Transform**, dan pilih **Pivot Column** (**2**). Di kotak dialog, kolom kedua akan secara otomatis ditetapkan sebagai *Value* (**3**). Buka bagian **Advanced options** dan tetapkan agregasi ke **Don't Aggregate** (**4**), seperti yang ditunjukkan pada Gambar 4.34.



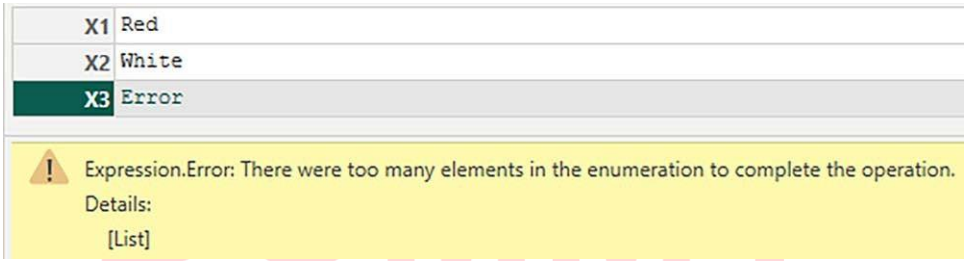
Gambar 4. 34 Langkah kolom pivot

Di bilah rumus, bungkus fungsi *Table.ToRecords* di sekitar ekspresi; jangan lupa untuk menyertakan tanda kurung tutup di akhir, seperti yang ditunjukkan pada Gambar 4.35:

Gambar 4. 35 Membungkus fungsi di sekitar ekspresi terluar di bilah rumus

Ini mengembalikan list yang berisi satu record. Untuk mengakses nilai tersebut, Anda dapat menelusurinya dengan mengeklik Record dan menggunakan *access* item dengan menambahkan {0} setelah tanda kurung tutup fungsi terluar atau membungkusnya dengan fungsi lain, seperti *List.First*.

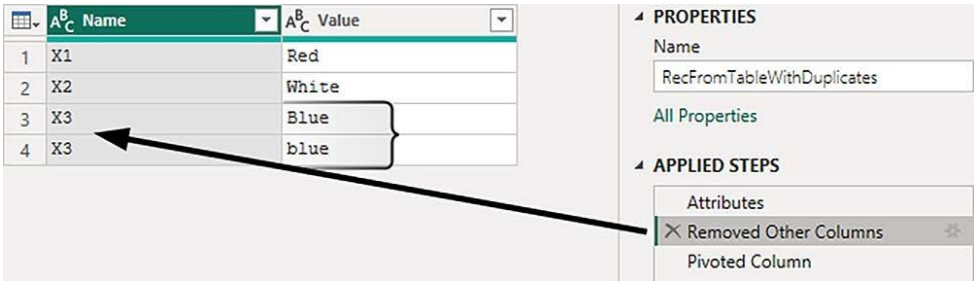
Jika menemui pesan kesalahan *Expression.Error*: Ada terlalu banyak elemen dalam enumerasi untuk menyelesaikan operasi. di salah satu bidang, ini menunjukkan bahwa kolom pertama tabel Anda berisi nilai duplikat dan tidak berbeda; lihat Gambar 4.36:



Gambar 4. 36 Menampilkan kesalahan tingkat bidang atau sel

Untuk mengatasi masalah tersebut, ikuti langkah-langkah berikut:

1) Kembali ke kueri Anda dan pilih variabel yang terkait dengan langkah 1 di panel **Applied Steps** di sisi kanan. Di sini, variabel tersebut adalah **Removed Other Columns**.
2) Periksa data Anda untuk mengetahui adanya duplikat dan tentukan tindakan yang tepat untuk menghapusnya; lihat Gambar 8.37:



Gambar 4. 37 Urutan pemilihan kolom dan menghapus kolom lainnya

3) ID di kolom Nama identik; Anda dapat memfilter warna biru di kolom Nilai atau menavigasi ke tab **Home** dan memilih **Remove Rows**, diikuti oleh **Remove Duplicates**. Konfirmasikan saat diminta untuk memasukkan langkah. Ini akan menghapus nilai duplikat dengan menghapus seluruh baris, menyelesaikan jenis kesalahan tingkat sel khusus ini.

Bahasa M menyertakan fungsi *Record.FromTable*, yang memerlukan tabel dengan dua kolom, satu berisi kunci tekstual yang unik. Kolom harus diberi nama *Name* dan *Value*. Jika nama kolom tabel berbeda atau kolom Nama tidak berisi nilai yang berbeda, kesalahan akan muncul. Sebagai ilustrasi, saat Anda menghapus langkah Kolom Berputar, masukkan langkah manual dan bungkus fungsi ini di sekitar variabel, seperti ini:

```
Record.FromTable( #"Removed Duplicates" )
```

Menghasilkan hasil yang sama persis dengan output.

Record dari *List*

Metode lain untuk membuat record adalah dengan menggunakan fungsi *Record.FromList*. Fungsi ini mengambil list nilai dan list nama kolom atau jenis record, seperti yang diilustrasikan di sini:

```
let
    Rec1 = Record.FromList(
        {102, "alice", "wonder", 60000},
        { "EmployeeID", "FirstName", "LastName", "Salary" }
    ),
    Rec2 = Record.FromList(
        {102, "alice", "wonder", 60000},
        type [
            EmployeeID=number,
            FirstName=text,
            LastName=text,
            Salary=number
        ]
    )
in
    Rec2
```

Fungsi ini berguna saat Anda memiliki list nilai dan ingin mengaitkan setiap item list dengan nama tertentu.

Mari kita simpulkan bagian tentang bekerja dengan catatan ini dengan memeriksa contoh praktis yang menggunakan pencarian untuk menjalankan penggantian bersyarat atau mengumpulkan informasi tambahan.

6. Pencarian bersyarat atau penggantian nilai

Kemampuan untuk memanfaatkan catatan pencarian sangatlah hebat. Teknik ini dapat menggantikan gabungan kompleks, Excel vlookup, dan xlookup, dan dapat digunakan untuk menghasilkan nilai antara dalam kalkulasi atau transformasi yang lebih kompleks.

Misalnya, mengganti ID dengan deskripsi aktual akan mempermudah analisis data ini di Excel, memperkaya dan membuat data lebih mudah dipahami oleh pengguna akhir:

```
let
    Attributes = Table.FromRows(
        {
            {"Colour", "X1", "Red"},
            {"Colour", "X2", "White"},
            {"Colour", "X3", "Blue"},
            {"Size", "X1", "S"},
            {"Size", "X2", "M"},
```

```
            {"Size", "X3", "L"},
            {"Size", "X4", "XL"},
            {"Size", "X5", "XXL"}
        }, type table[Attribute=text, ID=text, Value=text]
    ),
    toRecord = Table.Group(Attributes, {"Attribute"},
        {{"lookup", each Record.FromList( [Value], [ID] )}}),
    lookup = Record.FromList( toRecord[lookup], toRecord[Attribute] ),
    Source = Table.FromRows(
        {
            {"345-6245", "X5", "X1"},
            {"645-1366", "X3", "X2"},
            {"824-9206", "X4", "X3"},
            {"627-2078", "X3", "X3"},
            {"273-7134", "X4", "X2"}
        }, type table[Item=text, Size ID=text, Colour ID=text]
    ),
    AddDescription = Table.AddColumn( Source, "Description", each
        Record.FieldOrDefault( lookup[Size], [Size ID], "Unknown" )
        & ", " &
        Record.FieldOrDefault( lookup[Colour], [Colour ID], "Unknown" ),
        type
text
    ),
    ReplaceWithDescription = Table.TransformColumns( Source,
        {{"Size ID", each Record.FieldOrDefault( lookup[Size], _,
"Unknown" ), type text},
        {"Colour ID", each Record.FieldOrDefault( lookup[Colour], _,
        "Unknown"
), type text}}
    )
in
    ReplaceWithDescription
```

Mari kita periksa apa yang direpresentasikan oleh setiap variabel dalam kode ini:

- *Attributes* adalah tabel yang menyimpan atribut item, seperti warna dan ukurannya.
- *toRecord* mengelompokkan tabel *Attributes* menurut kolom Atribut untuk memastikan ID unik untuk setiap kelompok. Sebuah record dibuat di mana *ID* adalah nama bidang dan *Values* adalah nilai bidang. Penjelasan lebih rinci tentang operasi *Group By* dibahas di bagian berikutnya, Bekerja dengan tabel.

- Record yang dikelompokkan kemudian digunakan untuk membuat satu record, bernama *lookup*, yang berfungsi sebagai kamus. Nilai *Attributes* adalah kunci, dan record terkait (dibuat pada langkah sebelumnya) adalah nilai.
- *Source* adalah tabel yang berisi kode item dengan ID ukuran dan warnanya.
- *AddDescription* mengilustrasikan cara menambahkan kolom baru bernama *Description* ke tabel *Source*. Ini mencari ukuran dan warna dalam record pencarian. Jika ID tidak ditemukan, maka akan mengembalikan nilai yang ditetapkan sebagai default: "Unknown".
- *ReplaceWithDescription* mengilustrasikan cara mengganti nilai ID ukuran dan ID warna di tabel *Source* dengan deskripsi ukuran dan warna yang cocok menggunakan record pencarian.

Catatan dalam bahasa M merupakan struktur dasar. Eksplorasi kita difokuskan pada demonstrasi teknik untuk memanipulasi catatan. Teknik-teknik ini meliputi transformasi, ekstraksi, pengubahan ukuran, dan pemilihan bidang dari catatan; semuanya telah diilustrasikan melalui contoh kode yang dapat dieksekusi.

Setelah memeriksa list dan catatan, sekarang kita mengalihkan perhatian kita ke nilai terstruktur utama yang tersisa: tabel. Bagian ini akan membahas hal-hal penting dalam bekerja dengan tabel bersarang.

E. Bekerja dengan Tabel

Tabel adalah kumpulan baris dan kolom terstruktur, yang setiap selnya dapat menampung data jenis apa pun. Tabel adalah struktur dominan dalam Power Query, sebagaimana dibuktikan oleh cara UI dirancang untuk mengoperasikannya. Sekarang saatnya untuk menjelajahi dasar-dasar tabel bersarang dan melihat contoh cara membuat, mengakses, dan memanipulasinya.

Fokus utama kita adalah memahami nuansa bekerja dengan struktur tabel bersarang.

Berikut adalah kumpulan data contoh kita. Sebut kueri ini *SurveysData*; kita akan menyebutnya menggunakan nama itu mulai sekarang:

```
let
    Source = Table.FromRows( List.Zip( { List.Transform( {"1".."5"}, each "Wave
"& _ ),
        Table.Group( Table.FromRows(
        {
            {456, 30, "Female", "Intermediate", "High", "Good", "Yes", "Well
organized content", #date(2023,3,1), #date(2023,3,10), 5, 8},
            {457, 52, "Male", "Expert", "Medium", "Good", "Yes",
"Requires frequent updates", #date(2023,3,1), #date(2023,3,10), 3, 6},
            {458, 24, "Female", "Beginner", "High", "Excellent", "Yes", "Very
engaging and easy to follow", #date(2023,3,1), #date(2023,3,10), 7, 9},
            {459, 43, "Male", "Intermediate", "Low", "Average", "No", "The
interface is not user-friendly", #date(2023,3,1), #date(2023,3,10), 2, 4},
            {460, 28, "Non-binary", "Expert", "High", "Good", "Yes", "Feature-
rich and versatile", #date(2023,3,1), #date(2023,3,10), 6, 8},
            {461, 37, "Female", "Beginner", "Medium", "Poor", "No",
"Overwhelming for beginners", #date(2023,3,11), #date(2023,3,20), 1, 5},
            {462, 46, "Male", "Intermediate", "High", "Excellent", "Yes", "High
performance and reliability", #date(2023,3,11), #date(2023,3,20), 4, 8},
            {463, 21, "Female", "Beginner", "Low", "Poor", "No", "Too complex
for novices", #date(2023,3,11), #date(2023,3,20), 1, 3},
```

```
            {464, 55, "Male", "Expert", "Medium", "Average", "Yes", "Solid but
could be more intuitive", #date(2023,3,11), #date(2023,3,20), 3, 6},
            {465, 33, "Non-binary", "Intermediate", "High", "Good", "Yes",
"Adaptable to various tasks", #date(2023,3,11), #date(2023,3,20), 5, 7},
            {466, 40, "Female", "Beginner", "Medium", "Average", "No", "Lacks
in-depth tutorials", #date(2023,3,21), #date(2023,3,30), 2, 5},
            {467, 26, "Male", "Expert", "High", "Excellent", "Yes",
"Efficient in processing large datasets", #date(2023,3,21), #date(2023,3,30),
6, 9},
            {468, 48, "Female", "Intermediate", "Low", "Poor", "No",
"Unreliable with frequent downtime", #date(2023,3,21), #date(2023,3,30), 1, 3},
            {469, 23, "Non-binary", "Beginner", "Medium", "Good", "Yes",
"Good for beginners, but lacks advanced features", #date(2023,3,21),
#date(2023,3,30), 3, 6},
            {470, 39, "Male", "Expert", "High", "Excellent", "Yes",
"Supports a wide range of functions", #date(2023,3,21), #date(2023,3,30), 5,
8},
```

```
            {471, 34, "Female", "Intermediate", "High", "Good", "Yes", "User
community is very helpful", #date(2023,3,21), #date(2023,3,30), 4, 7},
            {472, 50, "Male", "Beginner", "Low", "Average", "No",
"Difficult to get started without assistance", #date(2023,4,1),
#date(2023,4,10), 2, 4},
            {473, 22, "Female", "Expert", "Medium", "Good", "Yes",
"Impressive analytics capabilities", #date(2023,4,1), #date(2023,4,10), 6, 7},
            {474, 41, "Non-binary", "Intermediate", "High", "Excellent", "Yes",
"Meets professional standards", #date(2023,4,1), #date(2023,4,10), 4, 8},
            {475, 29, "Male", "Beginner", "Medium", "Average", "No", "Features
are not well explained", #date(2023,4,1), #date(2023,4,10), 2, 5}
        },  {"RespondentID", "Age", "Gender", "ExperienceLevel",
"OverallQuality", "EaseOfUse", "WouldRecommend", "SpecificFeedback",
"SurveyStartDate", "SurveyEndDate", "UsageFrequency",
"SatisfactionScore"}
    ), {"SurveyStartDate"}, {{"Results", each _ , type table }})[Results] &
{#table({"RespondentID", "Age", "Gender", "ExperienceLevel",
"OverallQuality", "EaseOfUse", "WouldRecommend", "SpecificFeedback",
"SurveyStartDate", "SurveyEndDate", "UsageFrequency",
"SatisfactionScore"}, {})} }),
    type table [Survey=text, Results= table [RespondentID=number, Age=number,
Gender=text, ExperienceLevel=text, OverallQuality=text,
EaseOfUse=text, WouldRecommend=text, SpecificFeedback=text,
SurveyStartDate=date, SurveyEndDate=date, UsageFrequency=number,
SatisfactionScore=number]]
    )
in
    Source
```

Kueri *SurveysData* digambarkan pada Gambar 4.38, beserta pratinjau hasil bersarang Gelombang 1. Perhatikan bahwa hanya lima kolom pertama yang terlihat. Untuk melihat pratinjau struktur bersarang, pilih sel dengan mengeklik di mana saja pada spasi; Anda dapat mengubah pilihan dengan menggunakan tombol panah atas dan bawah pada papan ketik Anda. Tinjau tabel bersarang ini dengan cepat untuk memahami data yang sedang kita tangani:

	Survey	Results
1	Wave 1	Table
2	Wave 2	Table
3	Wave 3	Table
4	Wave 4	Table
5	Wave 5	Table

RespondentID	Age	Gender	ExperienceLevel	OverallQ
456	30	Female	Intermediate	High
457	52	Male	Expert	Medium
458	24	Female	Beginner	High
459	43	Male	Intermediate	Low
460	28	Non-binary	Expert	High

Gambar 4. 38 *SurveysData* dan lima kolom pertama terlihat di pratinjau sekunder

1. Mengubah Tabel

Tabel bersarang adalah tabel yang terdapat dalam tabel lain yang memungkinkan penyimpanan, manipulasi, penambahan, penghapusan, dan pemilihan data. Seperti yang dijelaskan di awal bab ini, meskipun banyak transformasi tabel tersedia dari dalam UI, transformasi tersebut tidak dapat dipanggil pada struktur tabel bersarang. Untuk mengatasi keterbatasan ini, kita telah menyediakan strategi untuk transisi ke kode, metode untuk transformasi nilai multilangkah yang kompleks, dan teknik untuk mengubah nilai dalam tabel; bagian ini merupakan prasyarat untuk apa yang terjadi selanjutnya. Mari kita tekankan bahwa meskipun metode add-custom-column sering diterapkan untuk demonstrasi, validasi, dan peningkatan pengalaman belajar Anda, metode ini belum tentu merupakan pendekatan terbaik.

2. Mengekstrak nilai sel

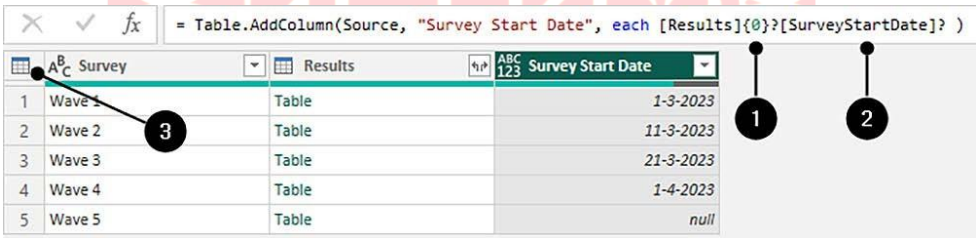
Tindakan drill-down ke dalam satu sel tabel akan menghasilkan satu dari dua varian sintaksis, tergantung pada apakah tabel Anda memiliki kunci atau tidak. Apa pun itu, untuk memperoleh nilai dari persimpangan tertentu dalam tabel, metode berikut akan selalu berfungsi. Pertama, pilih baris (1) dengan

menerapkan akses item. Kemudian, cari kolom (2), dengan menerapkan pemilihan bidang. Metode ini ditunjukkan pada Gambar 4.39. Untuk melindungi ekspresi, Anda dapat menjadikan pilihan ini opsional dengan menambahkan tanda tanya, ?.

Misalnya, untuk mengekstrak tanggal mulai survei untuk setiap gelombang tanpa memperluas tabel bersarang penuh, ikuti langkah-langkah berikut:

1) Buat kueri kosong baru atau rujuk kueri *SurveysData: = SurveysData*
2) Pilih **Add custom column** dari menu yang muncul saat Anda mengklik ikon tabel mini di sudut kiri atas tabel (3)
3) Masukkan kode ini ke bagian rumus: *[Results]{0}?[SurveyStartDate]?*

Hasilnya ditunjukkan pada gambar tangkapan layar berikut:



Gambar 4. 39 Menerapkan akses item dan bidang ke tabel

Apa yang terjadi di sini? *[Results]{0}* adalah bagian awal kode. Kode ini mencoba mengambil baris pertama (indeks 0) dari tabel di kolom Hasil. Namun, jika tabel kosong, seperti pada kasus baris yang terkait dengan Wave5, operasi ini menghasilkan kesalahan karena tidak ada baris yang dapat diakses. Dengan menambahkan tanda tanya (?) seperti ini: *[Results]{0}?*, pilihan menjadi opsional. Jika baris yang diminta hilang, alih-alih kesalahan, nilai null akan dikembalikan. Hal ini penting untuk memastikan kode tidak gagal saat menemukan nilai null atau tabel kosong.

Bagian selanjutnya, *[Results]{0}?[SurveyStartDate]*, dibangun berdasarkan operasi sebelumnya. Operasi ini mencoba mengakses kolom bernama *SurveyStartDate* dari baris yang diambil oleh *[Results]{0}*?. Namun,

jika *[Results]{0}*? mengembalikan null (yang menunjukkan tabel kosong atau baris yang hilang), upaya mengakses TanggalMulaiSurvei akan menyebabkan kesalahan. Hal ini karena operasi mencoba mengekstrak kolom dari nilai null, bukan dari record. Oleh karena itu, penyertaan tanda tanya kedua (?) di akhir membuat akses ke kolom *SurveyStartDate* juga bersifat opsional, sehingga mencegah kesalahan jika bagian awal menghasilkan null atau kolom tidak ada dalam record. Pendekatan metodis ini memastikan kueri tetap bebas kesalahan bahkan saat menangani data yang hilang, tidak lengkap, atau tidak cocok.

Teknik untuk mengekstrak nilai sel ini dapat berguna dalam skenario transformasi data tertentu. Bayangkan data mentah Anda dilengkapi dengan tajuk halaman yang berisi elemen yang perlu disertakan dalam keluaran akhir. Hal ini memungkinkan Anda untuk melakukannya. Contoh praktis dari persyaratan tersebut disertakan dalam Bab 14, Pola Data yang Bermasalah.

Ada satu fungsi M yang dapat mengekstrak nilai sel, *Table.FirstValue*; namun, fungsi tersebut hanya dapat memperoleh nilai di kolom dan baris pertama tabel atau mengembalikan nilai default yang ditentukan. Tentu saja, ada fungsi yang dapat mengembalikan satu record dari tabel, seperti *Table.First* atau *Table.Last*. Ketika dikombinasikan dengan akses bidang, fungsi tersebut juga memungkinkan Anda untuk memperoleh nilai sel tertentu. Selain itu, penting untuk diingat bahwa efektivitas fungsi ini meningkat ketika dikombinasikan dengan fungsi lain yang mengubah ukuran atau memfilter tabel.

3. Mengubah ukuran tabel sepanjang panjangnya

Fungsi list dan tabel memiliki banyak kesamaan, sehingga ada juga fungsi tabel yang memiliki kemampuan untuk mengubah ukuran tabel tanpa mengakses atau mengubah nilai di setiap baris. Fungsi ini dapat dilakukan dengan mengembalikan tabel baru dengan lebih sedikit atau lebih banyak baris berdasarkan offset atau hitungan. Fungsi-fungsi tersebut antara lain *Table.Skip, Table.FirstN, Table.Range, Table.RemoveLastN,* dan

Table.Repeat. Di bagian ini, kita akan fokus pada *Table.RemoveFirstN* dan *Table.AlternateRows*. Fungsi-fungsi ini membantu memanipulasi ukuran tabel.

Fungsi *Table.RemoveFirstN* dan *Table.Skip* pada dasarnya dapat dipertukarkan. Keduanya berguna untuk menghilangkan baris yang tidak diinginkan, atau "*junk*," dari bagian atas tabel. Fungsi-fungsi ini menawarkan fleksibilitas; Anda dapat menentukan baris yang akan dihapus. Baris ini bisa berupa baris pertama, sejumlah baris yang telah ditentukan, atau baris yang memenuhi kriteria tertentu. Mari kita ilustrasikan hal ini dengan contoh kueri:

```
let
    Source = SurveysData,
    Step1 = Table.AddColumn(Source, "NoCountOrCondition",
        each Table.RemoveFirstN([Results])),
    Step2 = Table.AddColumn(Step1, "Count",
        each Table.RemoveFirstN([Results], 2)),
    Step3 = Table.AddColumn(Step2, "Condition",
        each Table.RemoveFirstN([Results], each [Age] < 45 ))
in
    Step3
```

Gambar 4.40 menyajikan dua kolom pertama dari hasil Gelombang 1, yang ditampilkan di paling kiri gambar. Untuk menghapus hanya baris teratas, Anda dapat menghilangkan parameter *countOrCondition*, yang merupakan argumen kedua dalam fungsi ini. Pendekatan ini diilustrasikan dalam contoh kedua dari kiri. Atau, ketika Anda perlu menghapus sejumlah baris teratas tertentu, Anda dapat meneruskan hitungan berbasis satu sebagai argumen kedua. Misalnya, untuk menghapus dua baris pertama, Anda akan meneruskan angka 2. Skenario ini digambarkan dalam contoh ketiga dari kiri. Pilihan lain adalah menggunakan pendekatan berbasis kondisi. Di sini, Anda dapat menentukan kondisi, seperti usia < 45, untuk menghapus baris yang memenuhi kriteria ini hingga baris tidak memenuhi kondisi tersebut. Baris pertama yang gagal memenuhi kondisi tersebut menjadi baris teratas baru dari tabel. Metode ini ditunjukkan dalam contoh di paling kanan:

Wave 1, first 2 cols

RespondentID	Age
456	30
457	52
458	24
459	43
460	28

NoCountOrCondition

RespondentID	Age
457	52
458	24
459	43
460	28

Count

RespondentID	Age
458	24
459	43
460	28

Condition

RespondentID	Age
457	52
458	24
459	43
460	28

Gambar 4. 40 Nilai pengembalian gelombang 1 untuk kolom RespondentID

Table.AlternateRows adalah fungsi lain yang berguna, mirip dengan *List.Alternate* dalam hal fungsionalitas tetapi berbeda dalam jenis nilai yang diprosesnya. Satu aspek yang perlu diperhatikan adalah fakta bahwa urutan parameternya tidak sama, suatu ketidakkonsistenan yang perlu diperhatikan.

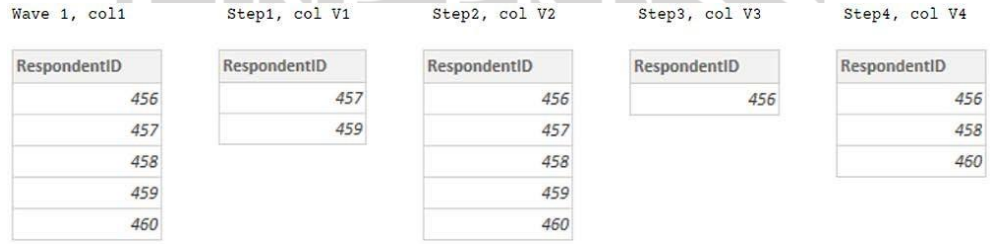
Seperti padanannya dalam list, *Table.AlternateRows* memfilter data menurut pola baris yang berulang. Kasus penggunaan praktis meliputi:

- **Pengambilan sampel atau reduksi data**: Dalam kumpulan data besar, fungsi ini dapat digunakan untuk memperoleh sampel representatif untuk pengembangan kueri awal atau analisis awal.
- **Menghapus tajuk yang berlebihan setelah menggabungkan data**: Terbatas pada kasus di mana tajuk diulang pada interval tetap.
- **Memfilter gangguan**: Saat mengalami gangguan berkala dalam data, seperti saat setiap baris ke-n merupakan pemeriksaan pemeliharaan atau kalibrasi.

Table.AlternateRows mengambil tabel, offset, jumlah lompatan, dan jumlah ekstraksi, yang menunjukkan berapa banyak baris yang harus diambil atau disimpan setiap saat. Offset berbasis nol dan kedua hitungan berbasis satu; semua argumen diperlukan. Mari terapkan fungsi ini pada data survei kita untuk memperoleh pemahaman yang lebih baik tentang apa artinya. Pertimbangkan kueri ini:

```
let
    Source = SurveysData,
    Step1 = Table.AddColumn(Source, "V1",
        each Table.AlternateRows([Results], 0, 1, 1)),
    Step2 = Table.AddColumn(Step1, "V2",
        each Table.AlternateRows([Results], 1, 0, 1)),
    Step3 = Table.AddColumn(Step2, "V3",
        each Table.AlternateRows([Results], 1, 1, 0 )),
    Step4 = Table.AddColumn(Step3, "V4",
        each Table.AlternateRows([Results], 1, 1, 1 ))
in
    Step4
```

Ia menghasilkan output yang digambarkan dalam gambar berikut untuk record pertama tabel, yaitu Gelombang 1. Kali ini, tampilan dibatasi untuk hanya menampilkan kolom awal:



Wave 1, col1

RespondentID
456
457
458
459
460

Step1, col V1

RespondentID
457
459

Step2, col V2

RespondentID
456
457
458
459
460

Step3, col V3

RespondentID
456

Step4, col V4

RespondentID
456
458
460

Gambar 4. 41 Gelombang 1 mengembalikan nilai untuk langkah dan kolom dengan nama yang sama. Tampilan terbatas pada kolom pertama

Untuk memahami setiap output yang digambarkan pada Gambar 4.41, mari kita periksa Langkah 1: *Table.AlternateRows([Results], 0, 1, 1)*. Terapkan pola ini untuk setiap nilai argumen dalam tabel input ([*Results*]): mulai dari baris teratas (posisi offset 0), lewati 1 baris, pertahankan 1 baris, dan ulangi pola ini untuk semua baris yang tersisa dalam tabel. Sekarang terapkan pola ini dan isi nilai argumen untuk memahami sendiri nilai yang dikembalikan untuk kolom lainnya.

4. Mengubah ukuran tabel menjadi lebar

Mengubah ukuran tabel dapat melibatkan penyesuaian baris dan kolomnya. Misalnya, jika Anda memerlukan jumlah kolom tertentu – katakanlah tiga kolom pertama – proyeksi merupakan teknik yang berguna.

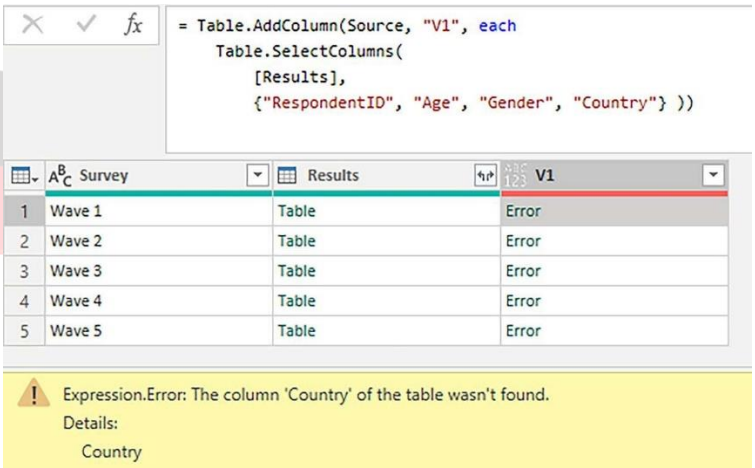
Teknik ini memungkinkan Anda membuat tabel dengan kolom yang lebih sedikit. Metode ini efisien dan dilengkapi dengan kemampuan untuk menjadikan pemilihan kolom atau bidang bersifat opsional, melindungi ekspresi seperti yang ditunjukkan dalam Bab 2, Nilai Terstruktur.

Operasi **Remove Columns** yang dipanggil UI memicu fungsi *Table.RemoveColumns*. Sementara itu, **Choose Columns** dan **Remove Other Columns** keduanya memanggil fungsi *Table.SelectColumns*. Semua fungsi ini dilengkapi dengan *MissingField.Type* sebagai parameter ketiga opsional. Enumerasi ini menentukan bagaimana fungsi tersebut berperilaku saat kolom yang tercantum dalam parameter keduanya, columns, hilang atau tidak ada.

Memilih di antara fungsi-fungsi ini bergantung pada persyaratan khusus tugas transformasi data Anda dan sifat sumber data Anda. Misalnya, kemungkinan mengubah kolom jauh lebih besar dalam sumber data Excel daripada dalam database. Oleh karena itu, penting untuk mempertimbangkan apakah perubahan kolom diharapkan terjadi seiring waktu dan bagaimana perubahan tersebut harus ditangani. Apakah kolom baru harus disertakan secara otomatis? Apa dampaknya terhadap transformasi berikutnya atau bahkan lebih jauh ke hilir? Mari kita bahas kedua metode tersebut; pertimbangkan kueri ini:

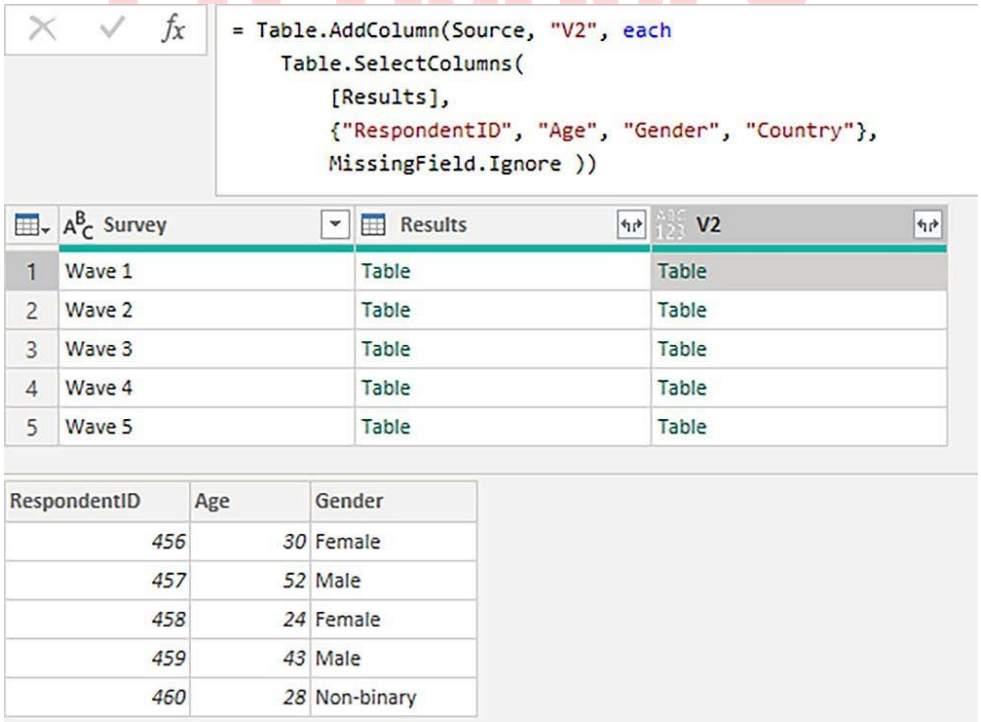
```
let
    Source = SurveysData,
    Step1 = Table.AddColumn(Source, "V1", each
    Table.SelectColumns(
        [Results],
        {"RespondentID", "Age", "Gender", "Country"} )),
    Step2 = Table.AddColumn(Source, "V2", each
    Table.SelectColumns(
        [Results],
        {"RespondentID", "Age", "Gender", "Country"},
        MissingField.Ignore )),
    Step3 = Table.AddColumn(Source, "V3", each
    Table.SelectColumns(
        [Results],
        {"RespondentID", "Age", "Gender", "Country"},
        MissingField.UseNull ))
in
    Step3
```

Untuk membatasi tabel ke subset kolom tertentu, *Table.SelectColumns* dapat dimanfaatkan. *Table.SelectColumns* secara eksplisit mencantumkan kolom yang akan dipertahankan; perhatikan bahwa kolom yang baru ditambahkan dalam sumber data tidak akan disertakan tanpa memperbarui kueri secara manual (lihat Langkah 1). Lebih jauh, secara default, kesalahan akan muncul jika salah satu nama kolom yang tercantum tidak sesuai dengan nama kolom sebenarnya dalam tabel:



Gambar 4. 42 Memilih kolom yang tidak ada menimbulkan kesalahan

Bila Anda menggunakan *Table.SelectColumns* bersama dengan tipe *MissingField* yang ditetapkan ke *MissingField.Ignore* (lihat Langkah 2 dan Gambar 4.43), fungsi tersebut akan berperilaku berbeda. Jika nama kolom yang ditentukan dalam list tidak cocok dengan kolom dalam tabel, kolom tersebut akan dikecualikan tanpa menimbulkan kesalahan. Namun, pendekatan ini dapat menyebabkan masalah pada langkah kueri berikutnya, terutama jika mengandalkan kolom yang tidak ditemukan secara tiba-tiba.



```
= Table.AddColumn(Source, "V2", each
    Table.SelectColumns(
        [Results],
        {"RespondentID", "Age", "Gender", "Country"},
        MissingField.Ignore ))
```

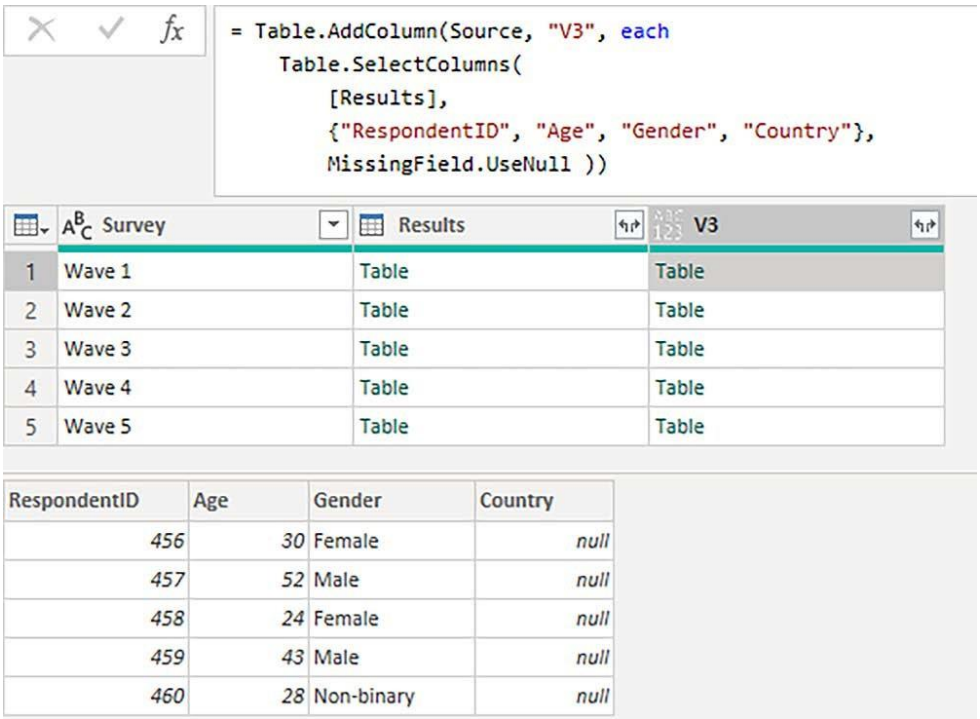
	Survey	Results	V2
1	Wave 1	Table	Table
2	Wave 2	Table	Table
3	Wave 3	Table	Table
4	Wave 4	Table	Table
5	Wave 5	Table	Table

RespondentID	Age	Gender
456	30	Female
457	52	Male
458	24	Female
459	43	Male
460	28	Non-binary

Gambar 4. 43 *MissingField.Ignore* mencegah kesalahan dan mengembalikan semua kolom yang valid

Bila tipe *MissingField* ditetapkan ke *MissingField.UseNull* dalam *Table.SelectColumns* (lihat Langkah 3 dan Gambar 4.44), fungsi tersebut menyertakan semua kolom yang ditentukan dalam output. Ini mencakup kolom yang tidak sesuai dengan kolom aktual dalam tabel. Dalam kasus seperti itu, kolom yang tidak ada disertakan dan akan berisi nilai *null*, tanpa menyebabkan kesalahan. Namun, perlu diketahui bahwa hal ini dapat menyebabkan

komplikasi pada tahap kueri selanjutnya, terutama dalam kalkulasi atau transformasi yang bergantung pada nilai kolom ini.



```
= Table.AddColumn(Source, "V3", each
    Table.SelectColumns(
        [Results],
        {"RespondentID", "Age", "Gender", "Country"},
        MissingField.UseNull ))
```

	Survey	Results	V3
1	Wave 1	Table	Table
2	Wave 2	Table	Table
3	Wave 3	Table	Table
4	Wave 4	Table	Table
5	Wave 5	Table	Table

RespondentID	Age	Gender	Country
456	30	Female	null
457	52	Male	null
458	24	Female	null
459	43	Male	null
460	28	Non-binary	null

Gambar 4. 44 *MissingField.UseNull* mencegah kesalahan dan mengembalikan semua kolom yang dipilih

Terakhir, Anda dapat memilih untuk menghapus subset kolom tertentu dengan *Table.RemoveColumns*, seperti yang ditunjukkan dalam kode berikut. Fungsi ini secara eksplisit mencantumkan kolom yang akan dihapus; kolom baru yang berasal dari sumber data secara otomatis disertakan dalam output. Tak perlu dikatakan lagi, hal ini dapat mengakibatkan kolom yang tidak diperlukan tetap ada kecuali kueri diperbarui secara manual. Menggunakan fungsi ini dengan tipe *MissingField* seperti *MissingField.UseNull* atau *MissingField.Ignore* mencegah kesalahan saat kolom yang ditentukan dalam parameter kedua tidak ada.

Tentu saja, bergantung pada transformasi lebih jauh ke hilir, penambahan kolom baru secara otomatis dapat menimbulkan masalah.

```
let
    Source = SurveysData,
    Step1 = Table.AddColumn(Source, "V1", each
    Table.RemoveColumns(
        [Results],
        {"Age", "Gender", "Country"} )),
    Step2 = Table.AddColumn(Source, "V2", each
    Table.RemoveColumns(
        [Results],
        {"Age", "Gender", "Country"},
        MissingField.Ignore )),
    Step3 = Table.AddColumn(Source, "V3", each
    Table.RemoveColumns(
        [Results],
        {"Age", "Gender", "Country"},
        MissingField.UseNull ))
in
    Step3
```

Tentu saja, ada pendekatan dinamis untuk memilih kolom. Dalam beberapa skenario, teknik semacam itu dapat meningkatkan ketahanan kueri Anda secara signifikan dan meminimalkan kebutuhan pemeliharaan.

5. Memfilter tabel

Operasi pemilihan atau penyaringan dapat mengurangi jumlah baris dalam tabel berdasarkan nilai bidangnya, seperti *Table.SelectRowsWithErrors*, *Table.FindText*, dan *Table.SelectRows*. Yang terakhir adalah salah satu fungsi yang paling banyak digunakan dalam bahasa M dan umumnya dipanggil melalui UI. Parameter kedua adalah kondisi sebagai fungsi; namun, sebelum kita membahasnya lebih jauh, sekarang saatnya untuk membahas dasar-dasarnya.

Memfilter tabel menggunakan UI umumnya mudah, tetapi penting untuk diingat bahwa Anda mungkin tidak melihat gambaran lengkapnya. Itu karena UI menampilkan 1.000 baris teratas di panel pratinjau. Hasilnya, variasi dalam kolom bisa lebih banyak daripada yang digambarkan di dalam area pemilihan filter. Gambar 8.45 mengilustrasikan hal ini; Anda dapat memasukkan kode

berikut langsung ke bilah rumus kueri kosong baru untuk menghasilkan data sampel:

```
= Table.FromColumns(
        {List.Repeat({"High"}, 999) & {"Medium", "Low"}}
    )
```

Saat melihat menu drop-down tajuk kolom dari Kolom1, akan ditampilkan dua nilai berbeda (**High** dan **Medium**), meskipun ada tiga (**High**, **Medium**, dan **Low**), seperti yang ditunjukkan pada Gambar 8.45. Saat Anda membatalkan pilihan **High** dan menekan **OK** untuk meninjau semua nilai yang tersisa, hanya **Medium** yang muncul dan bukan **Low**. Itu karena kode M yang dihasilkan oleh tindakan ini diterjemahkan menjadi: *([Column1] = "Medium")* yang bertentangan dengan apa yang mungkin Anda harapkan, seperti *([Column1] <> "High")*.

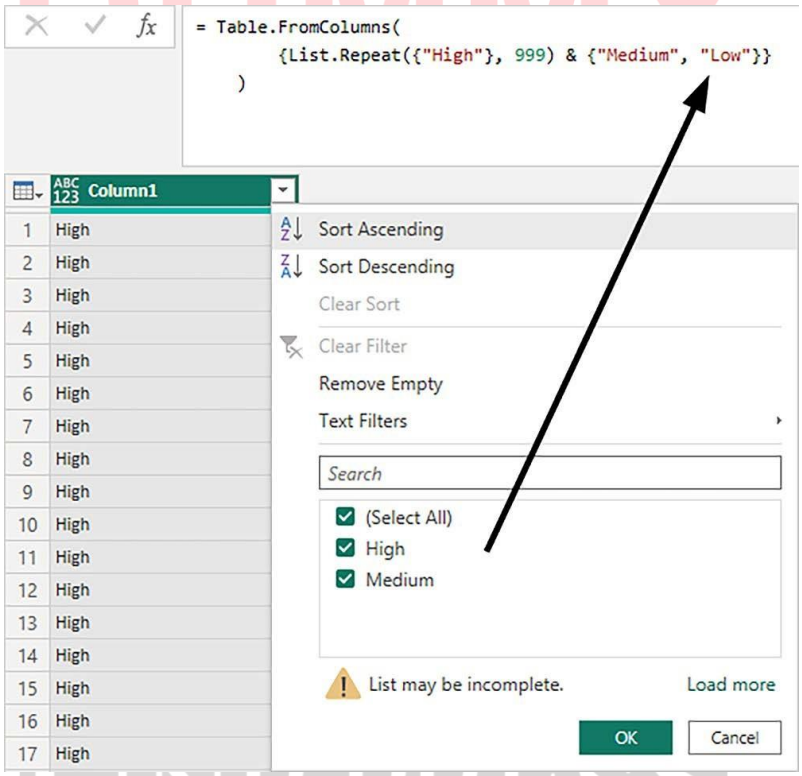
Sejujurnya, kita telah diperingatkan: pesan yang mengatakan List mungkin tidak lengkap dan opsi Muat lainnya ditampilkan. Perlu diingat bahwa tergantung pada jumlah data, Anda mungkin melihat pesan itu lebih dari sekali.

Namun, UI juga menyediakan filter khusus untuk jenis data dalam kolom tersebut. Untuk kolom teks, menempatkan kursor Anda pada **Text Filters** akan menampilkan opsi seperti **Equals, Begins With, Ends With, Contains**, dan negasinya. Opsi ini bervariasi untuk kolom jenis angka dan tanggal, yang menawarkan serangkaian pilihan filter yang sesuai untuk jenis data tersebut.

Untuk lebih mengeksplorasi pemfilteran, pertama-tama hapus langkah **Filtered Rows** dengan mengeklik x di depan namanya di panel Langkah yang Diterapkan. Sekarang mari terapkan **Text Filter** sebagai gantinya, dengan memilih **Does Not Equal** dari opsi. Ini akan membuka kotak dialog **Filter Rows**. Di sini, Anda dapat memasukkan **High** dan mengeklik **OK**; sekarang hasilnya sesuai dengan harapan kita, dan terlebih lagi, kode M secara akurat mencerminkan hal itu: *([Column1] <> "High")*.

Kotak dialog **Filter Rows** memiliki pengaturan **Advanced** yang memungkinkan Anda menentukan kondisi filter gabungan yang kompleks. Namun, pengaturan ini tidak menyediakan kontrol atas prioritas operator, yang menentukan urutan evaluasi operator individual.

Dalam bahasa M, logika dan dievaluasi sebelum logika atau, tetapi ekspresi dalam tanda kurung dapat digunakan untuk mengubah prioritas default ini:



Gambar 4. 45 List mungkin tidak lengkap. pesan, yang menunjukkan bahwa tidak semua nilai berbeda dalam kolom ditampilkan

Setelah kita mengetahui hal ini, mari kita periksa parameter kedua dari fungsi *Table.SelectRows*. Parameter tersebut mengharapkan suatu kondisi, yang pada dasarnya merupakan pengujian logis, sebagai suatu fungsi. Ketika ekspresi tersebut bernilai benar, baris tersebut akan memenuhi kriteria dan disimpan; jika bernilai salah, baris tersebut akan dikecualikan dari output.

Lihatlah kriteria pada Gambar 4.46. Kita akan menerapkan perbandingan langsung ke nilai kolom tertentu, yang berarti kolom *ExperienceLevel* hanya boleh menyertakan baris dengan nilai Intermediate atau Expert. Demikian pula, kolom *OverallQuality*, *EaseOfUse*, dan *WouldRecommend* harus sesuai dengan nilai *High*, *Good*, dan *Yes*, secara berurutan. Untuk memulai, buat referensi ke kueri *SurveysData*:

ExperienceLevel	OverallQuality	EaseOfUse	WouldRecommend
Intermediate	High	Good	Yes
Expert	High	Good	Yes

Gambar 4. 46 Beberapa kriteria filter

Berikut adalah kode M yang digunakan untuk mengubah setiap tabel bersarang. Perhatikan bagaimana ekspresi dalam tanda kurung digunakan untuk mengubah evaluasi prioritas operator default:

```
let
    Source = SurveysData,
    SelectRows = Table.TransformColumns( Source, {{"Results", each
        Table.SelectRows( _, each
            (
                [ExperienceLevel] = "Intermediate"
                or [ExperienceLevel] = "Expert"
            )
            and [OverallQuality] = "High"
            and [EaseOfUse] = "Good"
            and [WouldRecommend] = "Yes"
        ) }} )
in
    SelectRows
```

Untuk menghilangkan kemungkinan bias dan memastikan keakuratan filter yang kompleks, penyertaan langkah validasi dapat dipertimbangkan. Meskipun menambahkan kolom mungkin tampak bertentangan dengan praktik terbaik, hal itu menawarkan manfaat yang signifikan sebagai alat bantu pengembangan, yang memungkinkan penyempurnaan kriteria filter dan verifikasi hasil sebelum mentransfer logika filter ke langkah *SelectRows*. Lebih jauh, Anda dapat mencegah langkah *ValidateFilter* ini dijalankan saat runtime

dengan memastikan bahwa langkah tersebut tidak dirujuk oleh langkah lain dan bukan merupakan langkah terakhir dalam kueri Anda. Jika Anda mematuhi hal ini, tidak ada dampak pada kinerja. Ikuti langkah-langkah berikut:

1) Buka jendela **Advanced Editor**.
2) Salin langkah *SelectRows*, termasuk pengenalnya.
3) Buka langkah *Source* dan pindahkan kursor ke akhir baris.
4) Tekan Enter.
5) Tempel kode.
6) Tambahkan koma setelah tanda kurung tutup.
7) Ubah pengenal langkah menjadi *ValidateFilter*.
8) Ganti *Table.SelectRows* dengan *Table.AddColumn*.
9) Setelah input tabel, yaitu garis bawah, masukkan koma dan berikan nama kolom baru: "*Validation* ". Klik **Done**.

```
let
  Source = SurveysData,
  ValidateFilter = Table.TransformColumns( Source, {{"Results", each
    Table.AddColumn( _, "Validation", each
        (
            [ExperienceLevel] = "Intermediate"
            or [ExperienceLevel] = "Expert"
        )
        and [OverallQuality] = "High"
        and [EaseOfUse] = "Good"
        and [WouldRecommend] = "Yes"
    ) }} ),
  SelectRows = Table.TransformColumns( Source, {{"Results", each
    Table.SelectRows( _, each
        (
            [ExperienceLevel] = "Intermediate"
            or [ExperienceLevel] = "Expert"
        )
        and [OverallQuality] = "High"
        and [EaseOfUse] = "Good"
        and [WouldRecommend] = "Yes"
    ) }} )
in
  SelectRows
```

10) Pilih langkah *ValidateFilter* dan periksa hasil untuk setiap tabel bersarang; Gambar 4.47 menunjukkan kriteria dan kolom *Validation* untuk Gelombang 1.

ExperienceLevel	OverallQuality	EaseOfUse	WouldRecommend	Validation
Intermediate	High	Good	Yes	*TRUE*
Expert	Medium	Good	Yes	*FALSE*
Beginner	High	Excellent	Yes	*FALSE*
Intermediate	Low	Average	No	*FALSE*
Expert	High	Good	Yes	*TRUE*

Gambar 4. 47 Langkah ValidateFilter menunjukkan kolom Validasi

Selain itu, tabel bersarang tidak disediakan secara eksklusif oleh sumber eksternal; tabel bersarang juga dapat merupakan hasil transformasi lain untuk memecahkan skenario transformasi data yang lebih kompleks. Misalnya, tabel bersarang dapat digunakan saat mencari kecocokan perkiraan, persyaratan umum saat menangani diskon volume atau harga. Yang membuat jenis transformasi ini lebih rumit adalah kebutuhan untuk mengakses bidang dari tabel luar dan dalam, yang memperkenalkan cakupan, yang dibahas dalam Bab 3, Gonceptualizing M, dalam buku ini.

Perkiraan Kecocokan

Mari kita periksa data pada Gambar 4.48. Gambar tersebut memperlihatkan dua tabel: *SalesData*, yang berisi pesanan penjualan, dan *DiscountRates*. Bergantung pada nilai total pesanan, diskon mungkin berlaku. Tabel-tabel ini tidak dapat digabungkan karena tidak ada kunci bersama di antara keduanya. Pertimbangkan kueri ini:

```
let
    SalesData = Table.FromRows(
        {
            {"ORD-123", 149},
            {"ORD-124", 650},
            {"ORD-125", 749},
            {"ORD-126", 543},
            {"ORD-127", 324},
            {"ORD-128", 1685},
            {"ORD-129", 750},
            {"ORD-130", 999}
        },
        type table [OrderID=text, Total Value=number]
    ),
    DiscountRates = Table.Buffer( Table.FromColumns(
        {
            {750, 1500},
            {0.02, 0.05}
        },
        type table [Value=number, Discount=number]
    )),
    AddNetValue = Table.AddColumn( SalesData,
        "Net Value",
        each [Total Value] * ((1-List.Last(Table.SelectRows(DiscountRates,
(row)=> row[Value] <=[Total Value])[Discount])) ?? 1), type number
    )
in
    AddNetValue
```

Di sini, kita menambahkan kolom baru bernama *Net Value* ke tabel *SalesData*. Untuk alasan audit, kita akan membiarkan kolom Total Value tidak berubah. Nilai kolom baru tersebut akan dihitung dengan mencari tingkat diskonto yang relevan di tabel *DiscountRates* untuk setiap baris di SalesData dan menerapkannya ke Total *Value*. Perhitungan ini akan menghasilkan *Net Value* setelah diskon. Jika tidak ada diskon yang berlaku, *Net Value* akan sama dengan *Total Value* awal.

SalesData

	OrderID	Total Value
1	ORD-123	149
2	ORD-124	650
3	ORD-125	749
4	ORD-126	543
5	ORD-127	324
6	ORD-128	1685
7	ORD-129	750
8	ORD-130	999

DiscountRates

	Value	Discount
1	750	0,02
2	1500	0,05

Gambar 4. 48 Contoh SalesData dan DiscountRate

Hal ini ditunjukkan pada langkah AddNetValue; berikut ini uraiannya:

- *each [Total Value]* merujuk pada nilai di kolom Nilai *Total* dari tabel *SalesData* luar, yang berlaku untuk baris saat ini.
- *Table.SelectRows(DiscountRates, (row)=> row[Value] <= [Total Value])* memfilter tabel *DiscountRates* dalam untuk menyimpan baris yang nilainya kurang dari atau sama dengan *Total Value* untuk baris saat ini di *SalesData* luar.
- [*Discount*] mengekstrak kolom Diskon dari tabel yang difilter, sebagai list.
- *List.Last()* mengambil nilai terakhir dari list *Discount* ini.
- *(1 -) ??* 1 mengurangi nilai Diskon dari 1; jika tidak ada diskon (misalnya, jika nilai Diskon adalah *null*), nilai defaultnya adalah 1 (artinya tidak ada diskon).

6. Konversi ke tabel

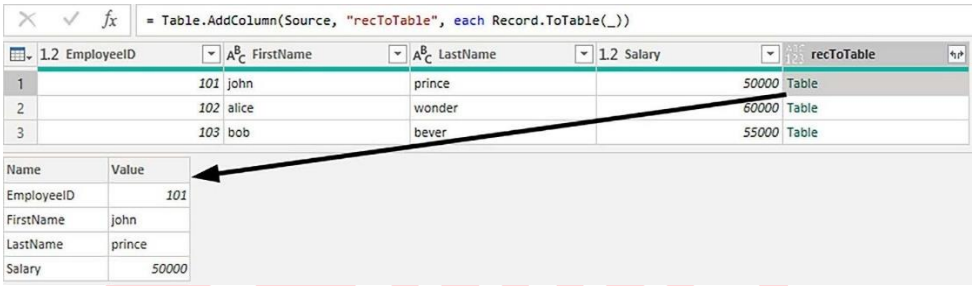
Tujuannya adalah untuk mendorong transisi dari mengandalkan UI semata menjadi mampu membaca, memahami, memodifikasi, dan menulis ekspresi secara mandiri. Banyak fungsi transformasi tabel dapat diakses melalui UI dan siap untuk Anda jelajahi. Secara khusus, kita telah berfokus pada fungsi yang mengubah, mengekstrak, mengubah ukuran, dan memilih data dalam tabel bersarang, menggunakan berbagai metode.

Untuk mendapatkan manfaat lebih dari teknik ini, penting untuk memahami konversi yang tersedia dari nilai terstruktur ke tabel. Konversi ini memungkinkan transformasi yang lebih canggih dan kompleks, memberi Anda kemampuan untuk membuat tabel dari berbagai jenis nilai. Di bagian berikut, kita akan menyoroti beberapa konversi yang paling umum digunakan.

Konversi *Record-to-table*

Fokus pertama kita adalah pada konversi baris tabel. Setiap baris dalam tabel pada dasarnya adalah sebuah record, dan record ini dapat diubah menjadi tabel dua kolom. Dalam tabel baru ini, kolom pertama disebut Nama dan berisi

semua nama bidang dari record tersebut. Kolom kedua, bernama Nilai, berisi nilai bidang yang sesuai dari record tersebut. Cuplikan layar berikut mengilustrasikannya:



Gambar 4. 49 Pratinjau output Record.ToTable saat diterapkan ke setiap baris dalam tabel

Transformasi ini khususnya berguna saat Anda memerlukan fungsionalitas standar yang disediakan fungsi tabel, tetapi tidak ada fungsi serupa untuk list atau record. Ambil contoh, fungsi tabel yang mengisi nilai secara vertikal, ke atas atau ke bawah kolom. Tidak ada padanan langsung untuk menyebarkan nilai secara horizontal, melintasi baris, dari kiri ke kanan atau sebaliknya. Namun, dengan mengonversi setiap baris (record) menjadi tabel dengan *Record.ToTable* (seperti yang ditunjukkan pada Gambar 4.49), Anda dapat mengatasi keterbatasan itu. Lihat kode berikut:

```
let
  Source = Table.FromRows({
    {null, 2023, null, null, null, 2024},
    {"Branch", "Q1", "Q2", "Q3", "Q4", "Q1"},
    {"BU1", 463, 582, 325, 487, 582},
    {"BU2", 264, 384, 201, 298, 275}
  }),
  FillRight = Table.AddColumn(Source, "Rec to tbl", each
    if [Column1] is null
    then Record.FromTable( Table.FillDown( Record.ToTable(_), {"Value"}))
    else _),
  ToTable = Table.FromRecords( FillRight[Rec to tbl] )
in
  ToTable
```

Kueri ini menunjukkan metode kreatif untuk mengisi nilai secara horizontal, menyebarkannya ke seluruh baris di dalam tabel, sebuah fungsi yang tidak tersedia secara langsung dalam fungsi tabel standar.

Proses ini dimulai dengan membuat tabel, Source, dengan berbagai titik data, termasuk nilai *null* yang perlu diisi. Bagian utama dari kueri adalah langkah *FillRight*, tempat kita menambahkan kolom baru, *Rec to tbl*. Di kolom ini, untuk setiap baris, kita memeriksa apakah kolom pertama (*Column1*) null. Hanya untuk baris ini kita perlu menggunakan operasi fill-right. Jika null, kita ubah baris menjadi tabel menggunakan *Record.ToTable*, lalu terapkan fungsi *Table.FillDown* ke kolom *Value*. Tindakan ini mengisi nilai *null* secara horizontal di seluruh baris.

Di baris tempat *Column1* bukan *null*, kita biarkan baris apa adanya. Langkah terakhir, *ToTable*, mengubah objek tabel ini kembali menjadi satu tabel. Hasilnya adalah tabel tempat celah horizontal (nilai null) di setiap baris diisi dengan nilai dari kiri, mereplikasi fungsi fill-down secara horizontal.

Membuat Tabel dari Kolom, Baris, atau Record

Kategori fungsi lain yang berguna adalah fungsi yang mengubah tabel menjadi kolom, baris, atau record, beserta fungsi "to" yang merupakan padanannya. Fungsi-fungsi ini termasuk yang paling banyak digunakan dalam skenario transformasi lanjutan khusus. Pola umum yang akan Anda temui melibatkan pengubahan tabel menjadi list kolom, baris, atau record, penerapan beberapa teknik transformatif, lalu penyusunan ulang elemen-elemen ini ke dalam format tabel.

Ada banyak contoh dalam buku ini yang menggunakan fungsi-fungsi ini dan mengikuti pola ini. Untuk menghindari pengulangan, demonstrasi terperinci tidak disertakan dalam bagian ini. Namun, bagian berikutnya, berjudul Bekerja dengan struktur campuran, akan secara komprehensif membahas aplikasi untuk setiap fungsi *Table.From*. Berikut ini deskripsi singkat fungsi-fungsi dalam grup *Table.From*:

- *Table.FromColumns*: Ini membuat tabel dari list list nilai kolom. Parameter kolom opsional dapat disertakan. Urutan item list menentukan urutan kolom dari awal hingga akhir tabel. Ketika list nilai kolom memiliki panjang yang bervariasi, nilai null digunakan untuk menjembatani kesenjangan tersebut. Padanannya adalah fungsi *Table.ToColumns*.
- *Table.FromRows*: Ini menggunakan list list nilai baris tunggal dan parameter kolom opsional. Urutan nilai dalam setiap list bersarang menentukan urutan nilai di seluruh kolom tabel. Semua list nilai baris harus berisi jumlah elemen yang sama, jika list memiliki panjang yang tidak sama atau terjadi kesalahan. Padanannya adalah fungsi *Table.ToRows*.
- *Table.FromRecords*: Ini mengambil list record, dan dapat menyertakan kolom opsional dan parameter *missingField*. Urutan record dalam list menentukan urutan baris dalam tabel. Padanannya adalah fungsi *Table.ToRecords*.

7. Table information

Bahasa M mencakup serangkaian fungsi yang dirancang khusus untuk memberikan informasi tentang struktur dan isi tabel. Fungsi-fungsi ini meliputi *Table.ColumnCount*, *Table.RowCount*, *Table.IsEmpty*, *Table.Keys*, *Table.Profile*, dan *Table.Schema*. Fungsi-fungsi ini dapat membantu tugas-tugas penilaian dan manipulasi data tertentu.

Misalnya, untuk mencegah kesalahan dan menghindari pemrosesan yang tidak perlu, Anda mungkin perlu mengonfirmasi keberadaan data dalam tabel sebelum melakukan transformasi yang rumit. Atau, saat mengintegrasikan data dari beberapa sumber, sering kali penting untuk membandingkan struktur tabel yang berbeda. Ada situasi di mana operasi penambahan hanya boleh dilanjutkan jika skema tabel memenuhi kriteria tertentu, seperti memiliki nama

kolom yang cocok, jumlah kolom yang identik, atau kolom dengan tipe tertentu, sambil mengecualikan atau menandai yang lain.

Di bagian ini, fokus utama kita adalah pada fungsi *Table.Profile*. Namun sebelum membahasnya, mari kita bahas *Table.Schema* secara singkat. Fungsi ini mengembalikan skema tabel input, yang menghasilkan baris untuk setiap kolom di tabel input. Setiap baris menyediakan properti kolom yang terperinci, seperti nama kolom, posisi, tipe, dan metadata lainnya. Gambar berikut mengilustrasikan lima kolom pertama dari output dari *Table.Schema*, yang menyediakan informasi utama tentang data:

```
= Table.Schema( Source )
```

	Name	Position	TypeName	Kind	IsNullable
1	RespondentID	0	Int64.Type	number	TRUE
2	Age	1	Int64.Type	number	TRUE
3	Gender	2	Text.Type	text	TRUE
4	ExperienceLevel	3	Text.Type	text	TRUE
5	OverallQuality	4	Text.Type	text	TRUE
6	EaseOfUse	5	Text.Type	text	TRUE
7	WouldRecommend	6	Text.Type	text	TRUE
8	SpecificFeedback	7	Text.Type	text	TRUE
9	SurveyStartDate	8	Date.Type	date	TRUE
10	SurveyEndDate	9	Date.Type	date	TRUE
11	UsageFrequency	10	Int64.Type	number	TRUE
12	SatisfactionScore	11	Int64.Type	number	TRUE

Gambar 4. 50 Tampilan sebagian dari nilai pengembalian Table.Schema, memberikan wawasan tentang properti kolom

Table.Profile, di sisi lain, menghasilkan profil tabel input. Untuk setiap kolom dalam tabel input, baris dihasilkan dalam tabel output yang menyediakan informasi untuk menilai data dalam kolom tersebut, seperti namanya, nilai minimum dan maksimum, jumlah nilai unik, dan sebagainya (lihat Gambar 4.51), yang berguna untuk statistik ringkasan.

```
= Table.Profile( Source )
```

	Column	Min	Max	Average	StandardDeviation	Count	NullCount	DistinctCount
1	Age	24	52	35,4	11,69615321	5	0	5
2	EaseOfUse	Average	Good	null	null	5	0	3
3	ExperienceLevel	Beginner	Intermediate	null	null	5	0	3
4	Gender	Female	Non-binary	null	null	5	0	3
5	OverallQuality	High	Medium	null	null	5	0	3
6	RespondentID	456	460	458	1,58113883	5	0	5
7	SatisfactionScore	4	9	7	2	5	0	4
8	SpecificFeedback	Feature-rich and versatile	Well organized content	null	null	5	0	5
9	SurveyEndDate	10-3-2023	10-3-2023	10-3-2023	null	5	0	1
10	SurveyStartDate	1-3-2023	1-3-2023	1-3-2023	null	5	0	1
11	UsageFrequency	2	7	4,6	2,073644135	5	0	5
12	WouldRecommend	No	Yes	null	null	5	0	2

Gambar 4. 51 Tampilan nilai pengembalian *Table.Profile*, yang membantu menilai data kolom

Yang lebih penting, validasi data dapat mengakomodasi kebutuhan spesifik yang tidak termasuk dalam output default. Hal ini dimungkinkan dengan memperluas *Table.Profile* dengan agregat tambahan melalui parameter kedua yang opsional.

Parameter ini adalah list yang berisi list bertingkat, satu untuk setiap kolom yang akan ditambahkan ke output-nya. Setiap list bertingkat ini harus secara berurutan berisi tiga elemen berikut:

1) **Nama kolom baru**: Ini adalah nama kolom tambahan yang akan ditambahkan ke output default. Sebaiknya berikan nama yang deskriptif dan membantu pengguna memahami metrik yang diwakilinya.
2) **Fungsi pemeriksaan tipe**: Ekspresi sederhana seperti *each true* memastikan bahwa fungsi agregasi akan diterapkan ke semua kolom, apa pun tipenya. Namun, jika Anda ingin membatasi fungsi agregasi ke kolom dengan tipe tertentu, Anda perlu menyediakan fungsi yang menguji setiap nilai. Misalnya, *Type.Is(_, type number)* akan mengidentifikasi kolom dengan tipe number.
3) **Fungsi agregasi**: Ketika kolom gagal dalam *type check (2)*, ini akan mengembalikan null, dan evaluasi fungsi agregasi akan dilewati. Untuk kolom yang lolos pemeriksaan tipe, fungsi agregasi ini akan dipanggil.

Kueri berikut menggabungkan tiga kolom agregat tersebut:

```
let
  Source = Table.FromColumns(
    {List.Numbers(1, 200, 1) & {null, "text", null, null}},
    type table [myNumber = number] // ascribing a type
  ),
  getProfile = Table.Profile( Source,
    {
      {
        "count of Elements",
        each Type.Is(_, type number),
        each List.Count(_)
      },
      {
        "count No Nulls",
        each Type.Is(_, type number),
        each List.NonNullCount(_)
      },
      {
        "raise Type Error",
        each Type.Is(_, type number),
        each List.NonNullCount(List.Transform(_, Number.From))
      },
      {
        "count of Numbers",
        each Type.Is(_, type number),
        each List.Count(List.Select(_, (x) =>
          Type.Is(Value.Type(x), type number))
        )
      }
    }
     ),
  demoteHeaders = Table.DemoteHeaders(getProfile),
  transposeTable = Table.Transpose(demoteHeaders)
in
  transposeTable
```

Kueri ini menghasilkan tabel selebar satu kolom; kita mentransposisikan outputnya, seperti yang dapat Anda lihat pada Gambar 4.52. Agregasi yang ditambahkan berisi kondisi yang memeriksa apakah suatu kolom bertipe angka atau tidak. Karena kolom *myNumber* telah ditetapkan bertipe angka, kolom tersebut akan melewati fungsi pemeriksaan tipe dan fungsi agregasi akan dijalankan.

Memahami nilai yang dikembalikan oleh agregasi adalah penting, karena hal ini memengaruhi cara fungsi menangani data. Untuk memudahkan validasi

hasil, kita hanya akan menghitung item. Ambil *List.Count*, misalnya: fungsi ini tidak harus mengakses item list mana pun untuk menentukan jumlah elemen dalam list. Fungsi ini mengembalikan angka yang sama dengan jumlah baris tabel, 204, seperti yang terlihat pada baris 9 dalam gambar berikut:

	Column1	Column2
1	Column	myNumber
2	Min	1
3	Max	text
4	Average	Error
5	StandardDeviation	Error
6	Count	204
7	NullCount	3
8	DistinctCount	202
9	count of Elements	204
10	count No Nulls	201
11	raise Type Error	Error
12	count of Numbers	200

Gambar 4. 52 Agregat tambahan untuk *Table.Profile*

Di sisi lain, *List.NonNullCount* mengakses semua item list karena perlu menentukan apakah item list adalah sesuatu selain null. Ini mengembalikan hitungan 201, seperti yang terlihat pada baris 10 Gambar 4.52. Tetapi apakah angka itu diharapkan?

Itu tergantung pada apakah kita harus atau tidak mempertimbangkan jenis nilai yang dihitung dalam list. Bagaimanapun, list tersebut berisi nilai teks, meskipun kolom tersebut telah ditetapkan sebagai jenis angka. Mengetahui bahwa kueri dasar (dalam langkah *Source*) akan menimbulkan kesalahan konversi jenis pada waktu eksekusi kueri jika dimuat secara terpisah ke dalam model data, Anda dapat memilih untuk menimbulkan kesalahan dalam profil juga, seperti yang terlihat pada baris 11, atau hanya menghitung item list yang dikonversi ke angka, seperti yang ditunjukkan pada baris 12 Gambar 4.52.

Ini adalah pertimbangan dan perbedaan penting saat menambahkan agregat tambahan ke fungsi *Table.Profile*.

Secara umum, ada tingkat kesamaan yang signifikan antara fungsi list dan tabel, dan sebagian besar diberi nama dengan cara yang mudah dipahami dan konsisten. Ini berarti Anda dapat dengan cepat memperluas pengetahuan Anda tentang fungsi M dengan membiasakan diri dengan konvensi penamaan dan sintaksis. Tumpang tindih yang substansial juga berarti tidak perlu mempelajari semua fungsi yang tersedia dengan hati.

F. Bekerja dengan struktur campuran

Bagian ini berfokus pada nilai terstruktur yang berisi struktur dengan tipe berbeda. Meskipun berbagai skenario mungkin terjadi, kita tidak mungkin menggambarkan semuanya; namun, bagian ini akan memberikan wawasan yang cukup tentang cara mengatasi tantangan paling umum yang mungkin Anda hadapi.

1. List Tabel, *List*, atau *Record*

Seperti yang ditunjukkan sebelumnya, penanganan list item yang memiliki struktur konsisten dan dikelompokkan bersama dapat dengan mudah dikelola dan diubah menjadi tabel. Pendekatan ini sangat hebat dan berlaku sama untuk bekerja dengan satu kolom dalam tabel atau mengonversi list di setiap baris tabel dalam tabel untuk membentuk ulang data.

Mari kita jelajahi skenario pertama, yang menangani kolom yang berisi nilai terstruktur. Bayangkan tabel yang disusun dengan cara berikut:

- Nilai tabel
- List *list*, di mana setiap list bagian dalam mewakili nilai kolom
- List *list*, di mana setiap list bagian dalam mewakili nilai baris
- List *record*, di mana setiap record mewakili baris tabel

Pertimbangkan kueri ini, yang akan menghasilkan tabel seperti itu:

```
let
  Source = #table(
    {"Tables", "ColumnLists", "RowLists", "Records"},
    {
      {
        #table(2, {{1, 2}, {3, 4}}),
        {{1, 3}, {2, 4}},
        {{1, 2}, {3, 4}},
        {[C1 = 1, C2 = 2], [C1 = 3, C2 = 4]}
      },
      {
        #table(3, {{5, 6, 7}, {8, 9, 0}}),
        {{5, 8}, {6, 9}, {7, 0}},
        {{5, 6, 7}, {8, 9, 0}},
        {[C1 = 5, C2 = 6, C3 = 7], [C1 = 8, C2 = 9, C3 = 0]}
      }
    }
  )
  in
Source
```

Dalam sebagian besar skenario, pola generik dapat diterapkan untuk mengonversi kolom yang berisi nilai terstruktur menjadi satu tabel. Berikut ini adalah apa yang dimaksud dengan pola dasar tersebut:

- Telusuri kolom yang berisi nilai terstruktur, untuk mengekstraknya sebagai list.
- Ubah setiap item list menjadi tabel menggunakan fungsi *Table.From* yang sesuai.
- Gunakan *Table.Combine* untuk menggabungkan semua tabel dalam list menjadi satu tabel.

Pengecualian untuk pola ini adalah saat Anda berurusan dengan kolom tabel. Dalam kasus seperti itu, operasi gabungan sudah cukup, melewati kebutuhan untuk langkah konversi. Mari kita praktikkan teori dan terapkan langkah-langkah ini ke data sampel:

1) Telusuri kolom *Tables*. Ini akan menghasilkan list tabel; di dalam bilah rumus, Anda akan menemukan ekspresi ini: *Source[Tables]*

2) List tabel ini dapat digabungkan, menumpuk semua tabel di atas satu sama lain untuk membentuk satu tabel dengan membungkus *Table.Combine* di sekitarnya, seperti yang diilustrasikan di sini:

```
listOfTables = Table.Combine(Source[Tables])
```

Untuk terus berlatih, Anda dapat menghapus langkah untuk kembali ke Sumber, atau menyisipkan langkah manual dengan mengeklik **fx** di samping bilah rumus dan mengganti variabel yang dikembalikan oleh variabel *Source*, seperti ini: = *Source*

3) Menelusuri kolom lainnya akan menghasilkan list list. List ini dapat diubah menjadi tabel. Untuk kolom *ColumnLists*, artinya mengubah setiap item list dengan fungsi *Table.FromColumns* terlebih dahulu untuk menempatkan list kolom berdampingan, membentuk tabel, sebelum menggabungkannya, seperti yang ditunjukkan di sini:

```
listOfColumnLists = Table.Combine(
    List.Transform(Source[ColumnLists], each Table.FromColumns(_)))
```

4) Demikian pula, variabel *listOfRowLists* dan *listsOfRecords* mengubah item list dengan menumpuknya satu di atas yang lain untuk membentuk baris dalam tabel, sebelum menggabungkan tabel:

```
listOfRowLists = Table.Combine(
    List.Transform(Source[RowLists], each Table.FromRows(_))),
listsOfRecords = Table.Combine(
    List.Transform(Source[Records], each Table.FromRecords(_)))
```

Dalam contoh ini, kita telah mengekstrak satu kolom; pada dasarnya, itu berarti bahwa semua data lain dari tabel utama hilang. Tentu saja, ada beberapa kasus di mana ini baik-baik saja dan hanya apa yang dibutuhkan, tetapi Anda juga mungkin menghadapi skenario di mana data dari tabel utama harus dipertahankan. Dalam kasus seperti itu, Anda dapat menerapkan operasi transformasi ke setiap baris tabel dan mengikutinya dengan operasi perluasan kolom. Itu akan memperlebar dan memperluas tabel utama untuk menyertakan

kolom dan baris baru untuk bidang yang dipilih dari tabel bersarang. Ini dapat menyebabkan duplikasi bidang tabel utama.

Kita akan menggunakan kembali *ColumnLists* dari contoh sebelumnya untuk mengilustrasikan bagaimana pola dasar berubah dalam jenis skenario ini:

- Memilih metode untuk mengubah nilai dalam tabel. Kita akan mendemonstrasikan penggunaan fungsi *Table.TransformColumns* untuk mengubah kolom *ColumnLists*.
- Melewati fungsi ke tabel Source sebagai argumen pertama dan transformOperations sebagai list. {{"*ColumnLists*", setiap *Table.FromColumns(_)* }} menentukan nama kolom serta transformasi yang akan diterapkan ke setiap baris dalam tabel.
- Menggunakan *expand column option*, yang digambarkan dengan panah menyamping, di tajuk kolom *ColumnLists* dan pilih semua kolom; centang atau hapus centang **Use original column name as prefix**.

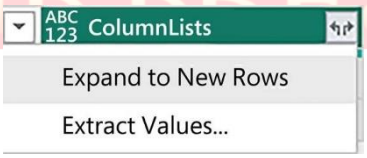
Kueri berikut mencerminkan langkah-langkah yang dijelaskan sebelumnya:

```
let
    Source = Table.FromColumns(
        {
            {"A", "B"},
            {"a", "b"},
            {{{1, 3}, {2, 4}}, {{5, 8}, {6, 9}, {7, 0}}}
        },  {"Field1",  "Field2", "ColumnLists"}
    ),
    Transform = Table.TransformColumns( Source,
        {{"ColumnLists", each Table.FromColumns(_) }}
    ),
    ExpandColumnLists = Table.ExpandTableColumn( Transform,
        "ColumnLists",
        {"Column1", "Column2", "Column3"},
        {"Column1", "Column2", "Column3"}
    )
in
    ExpandColumnLists
```

Secara default, operasi *Expand Column* akan mengkodekan secara permanen nama kolom yang dipilih (argumen ketiga) dan nama kolom baru opsional (argumen keempat). Ketahui bahwa ada metode untuk membuat ini dinamis.

2. Tabel dengan List, Record, atau Tabel

Kolom tabel yang berisi nilai tabel, list, atau record biasanya menampilkan opsi **Expand Column** di tajuk kolom. Secara default, opsi ini memungkinkan pengguna untuk memilih kolom dari tabel atau record. Untuk kolom list, opsi ini menyediakan opsi untuk memperluas ke baris baru atau mengekstrak dan menggabungkan semua item list menjadi string (dibatasi), seperti yang ditunjukkan pada Gambar 4.53. Pembatas opsional tersebut dapat digunakan pada langkah berikutnya untuk membagi kolom tersebut jika diperlukan.

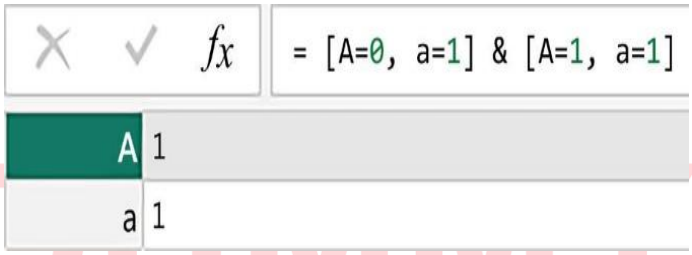


Gambar 4. 53 Menampilkan list Perluas opsi Kolom

Semua tindakan ini dapat dilakukan melalui UI dan tidak akan dibahas secara terperinci di sini. Untuk skenario yang mengharuskan satu item diambil, akses bidang dan item opsional dapat diterapkan sebagaimana dijelaskan secara terperinci dalam Bab 2, Nilai Terstruktur. Sebagai alternatif, fungsi dapat digunakan untuk memperoleh nilai berdasarkan posisinya atau kriteria, sebagaimana ditunjukkan dalam bagian sebelumnya dari bab ini.

Memperluas beberapa kolom list secara bersamaan telah ditunjukkan dalam bagian Bekerja dengan list. Ini melibatkan penggabungan kolom-kolom tersebut untuk membentuk tabel. Beberapa kolom record dapat diperluas satu per satu; ini hanya akan memperluas tabel utama dengan menambahkan kolom tambahan, tetapi tidak akan memperluas tabel dengan menambah jumlah baris. Sebagai alternatif, record dapat digabungkan menjadi satu record. Ingatlah bahwa jika Anda memilih penggabungan, nama bidang harus unik; jika tidak,

operasi penimpaan akan terjadi. Ini berarti nilai bidang dari operan kiri akan ditimpa oleh bidang dengan nama yang sama dari operan kanan, sebagaimana diilustrasikan dalam Gambar 4.54:



Gambar 4. 54 Penggabungan record ini menggambarkan operasi penimpaan

Memperluas beberapa kolom tabel secara bersamaan jarang terjadi dan mungkin berlaku untuk kasus-kasus di mana tabel dapat digabungkan pada indeks atau kunci yang sama. Yang lebih unik lagi adalah kasus-kasus di mana data disejajarkan dengan sempurna pada basis baris demi baris dan tabel-tabel perlu dijahit atau disatukan, berdampingan. Pola untuk mencapai hal ini diilustrasikan di sini:

```
let
    allTables = {
        #table({"A", "B"}, {{1, 2}}),
        #table({"a", "b"}, {{3, 4}, {5, 6}}),
        #table({"x", "y"}, {{null, null}, {9, 0}})
    },
    allCols = List.Combine(
        List.Transform( allTables, Table.ToColumns )),
    allNames = List.Combine(
        List.Transform( allTables, Table.ColumnNames )),
    result = Table.FromColumns( allCols, allNames )
in
    result
```

Fungsi ini membuat satu list, *allCols*, yang berisi semua list kolom, dan list lain, *allNames*, dengan semua nama kolom. Semua ini dimasukkan ke dalam fungsi *Table.FromColumns* untuk menghasilkan tabel. Perhatikan bahwa nama kolom harus unik atau kesalahan akan muncul.

3. Struktur campuran

Ada berbagai macam kebutuhan transformasi data; bagian selanjutnya ini memperkenalkan pola yang dapat membantu dalam meratakan dan mengekstrak data dari tipe data terstruktur.

Ratakan Semua

Bayangkan Anda menemukan sebuah tabel, atau mungkin satu atau beberapa kolom dalam tabel, di mana Anda menemukan tumpukan berbagai tipe data, seperti yang Anda lihat pada Gambar 4.55. Ini bukan sesuatu yang Anda lihat setiap hari. Jadi, bagaimana cara Anda mengekstrak semua nilai yang berbeda dari struktur bertingkat ini?

	Column1	Column2	Column3	Column4
1	List		List	List
2	List	List	Record	Record
3	*21-1-2024*		List	List
4	*7-3-2024*		List	List
5	List	List	Record	Record

Gambar 4. 55 Meja dengan tipe campuran

Memahami metode untuk menentukan jenis dan mengubah serta mengonversi nilai akan sangat membantu. Pada dasarnya, kueri ini mengubah semua sel dalam tabel, tergantung pada apakah itu list, record, atau jenis nilai lainnya:

```
let
    Source = #table(4, {
        {{"Let's", "go"}, "", {1, 2}, {"3".."9"}},
        {{null}, {true, true, false}, [a=1, b=2], [a=1, b=2]},
        {#date(2024, 1, 21), "", {1, 2}, {"3".."9"}},
        {#date(2024, 3, 7), "", {1, 2}, {"3".."9"}},
        {{null}, {true, true, false}, [a=1, b=2], [a=1, b=2]}
    }),
    getValues = Table.TransformColumns( Source,
        {}, each if Value.Is(_, type list)
        then Text.Combine( List.Transform(_, Text.From), ", ")
```

```
        else if Value.Is(_, type record)
        then Text.Combine( List.Transform(

            List.Zip({ Record.FieldNames(_), Record.ToList(_) }),
                (x)=> Text.Combine({ x{0}, " = ", Text.From(x{1})
                })

            ), ", ")
        else _ )
in
    getValues
```

Dengan memanfaatkan transformasi default *Table.TransformColumns*, logika diterapkan ke semua kolom tabel Source. Kita menyediakan tiga metode untuk menangani nilai:

- Jika nilainya berupa list, ia mengubah setiap item menjadi teks dan menggabungkannya menjadi satu string teks yang dipisahkan dengan koma.
- Jika nilainya berupa record, ia mengubah setiap kolom menjadi string teks dalam format *"field name = field value"* dan menggabungkan string ini, dipisahkan dengan koma.
- Namun, jika nilainya bukan list atau record, ia membiarkan nilainya apa adanya.

Output ditunjukkan pada gambar berikut:

	Column1	Column2	Column3	Column4
1	Let's, go		1, 2	3, 4, 5, 6, 7, 8, 9
2		true, true, false	a = 1, b = 2	a = 1, b = 2
3	21-1-2024		1, 2	3, 4, 5, 6, 7, 8, 9
4	7-3-2024		1, 2	3, 4, 5, 6, 7, 8, 9
5		true, true, false	a = 1, b = 2	a = 1, b = 2

Gambar 4. 56 Nilai pengembalian untuk langkah getValues menunjukkan semua data

Tentu saja, ada banyak varian yang dapat diperoleh dari pola ini, seperti mengekstraksi setiap item awal atau akhir dari list atau record, atau memilih sejumlah bidang record atau item list berdasarkan suatu kondisi. Semua teknik tersebut telah diilustrasikan dalam bab ini dan dapat dimasukkan di sini untuk

memenuhi persyaratan khusus Anda, betapapun eksotisnya persyaratan tersebut.

Membongkar Semua Bidang Record dari *List*

Sering kali, data hadir dalam format bertingkat, seperti record dalam list. Kita telah melihat bagaimana list record dapat diubah menjadi tabel, tetapi bagaimana jika Anda memerlukan pendekatan terstruktur untuk mengekstrak dan meratakan data ini, dengan menambahkan kolom baru ke tabel luar untuk setiap bidang record bertingkat? Pertimbangkan pola ini:


```
  let
    Source = Table.FromColumns(
      {
        {1, 2},
        {
          {
            [Tag = 2, Other = "text1"],
            [Tag = 3, Other = "text2"],
            [Tag = 4, Other = "text3"],
            [Tag = 5, Other = "text4"]
          },
          {
            [Tag = 9, Other = "textA"],
            [Tag = 8, Other = "textB"],
            [Tag = 7, Other = "textC"],
            [Tag = 6, Other = "textD"]
          }
}
    },
      type table [Key = number, Changes = list]
    ),
    PrefixNewColumnNamesWith = "Change ", ExtractRecords =
    List.Accumulate(Record.FieldNames(Source[Changes]{0}{0}), Source,
      (s, a) =>
        Table.AddColumn( s,PrefixNewColumnNamesWith & a,
          (x) => Text.Combine(List.Transform(x[Changes],
              (y) => Text.From(Record.Field( y, a))), ", "),
          type text
        )
)
     in
    ExtractRecords
```

Ini adalah skenario tingkat lanjut; oleh karena itu, jangan khawatir jika Anda tidak sepenuhnya memahami cara kerjanya saat ini. Anda selalu dapat kembali ke sana di tahap pembelajaran Anda selanjutnya. Mari kita lihat lebih dekat apa yang sedang dilakukan di sini:

- Variabel *PrefixNewColumnNamesWith* diatur ke Change, string yang akan digunakan untuk mengawali nama bidang yang ada, yang diekstrak dari record, untuk membantu menghindari potensi konflik nama dengan kolom yang ada di tabel Source luar.
- Langkah *ExtractRecords* menggunakan fungsi *List.Accumulate*, yang penting untuk memproses setiap nama bidang dalam record ini. Anda dapat mempelajari lebih lanjut tentang fungsi ini di Gbapter 13, Iteration and Recursion, tetapi kita akan memberikan ikhtisar singkat di sini.

Fungsi *List.Accumulate* mengiterasi nama bidang record (diperoleh oleh *Record.FieldNames(Source[Changes]{0}{0})*). Nama bidang merujuk ke baris pertama kolom Changes di tabel Source. Fungsi ini mengakses item list pertama, yang merupakan record untuk mengambil semua nama bidang:

- Untuk setiap iterasi, fungsi diterapkan untuk menambahkan kolom baru ke tabel (s), yang merujuk ke argumen kedua di *List.Accumulate*, tabel Sumber.
- Nama kolom baru dibuat dengan menggabungkan PrefixFieldNamesWith dengan nama bidang iterasi saat ini (a), yang merujuk ke argumen pertama di *List.Accumulate*.
- Kemudian fungsi diterapkan ke setiap list di kolom Perubahan untuk mengubah setiap record dalam list dengan mengekstrak nilai bidang saat ini (a) dan mengubahnya menjadi teks. Terakhir, nilai teks ini digabungkan menjadi satu string, dipisahkan dengan koma.

Kode ini lebih tangguh dalam menangani potensi konflik nama bidang. Awalan nama kolom baru dengan Perubahan memastikan bahwa kolom baru yang ditambahkan ke tabel cenderung tidak bentrok dengan nama kolom yang

ada. Ini adalah pertimbangan penting untuk skenario di mana nama bidang di dalam record mungkin cocok dengan nama kolom yang ada di tabel luar. Jika ini tidak diperlukan, Anda cukup meneruskan string teks kosong sebagai gantinya.

Mengekstrak Data Melalui Pencarian

Contoh terakhir ini difokuskan pada perataan data, tetapi ada banyak skenario yang memerlukan pendekatan yang lebih terarah untuk mengekstrak nilai. Karena tabel adalah struktur yang dominan, kita akan mengilustrasikan cara mencari baris yang cocok dengan kriteria dalam tabel bersarang, yang merupakan persyaratan umum. Dalam contoh ini (lihat Gambar 4.57), tabel luar menyertakan satu baris per proyek, dan kolom Detail berisi record dengan dua bidang: Versi, list dengan data log perubahan yang item list terakhirnya mewakili versi "aktif" (untuk Proyek 98731 yaitu 1,01); dan Detail, tabel bersarang yang menyediakan catatan untuk setiap versi.

	Project	Status	Details
1	98731	In development	Record
2	98732	Internal test	Record
3	98733	Beta test	Record
4	98734	In development	Record
5	98735	Rework	Record

Version

List
1
1,01
1,1
1,01

Details

Version	Notes
1	Draft version
1,01	Minor updates
1,1	Updated schema

Gambar 4. 57 Data proyek dan tampilan konten dalam record bersarang

Tugasnya adalah mengekstrak versi aktif dan menggunakannya untuk melakukan pencarian di tabel *Details* bersarang yang sesuai untuk mengambil

catatan yang terkait dengan versi tersebut. Ini melibatkan pencocokan nomor versi dan mengekstrak catatan yang relevan.

Dalam semua skenario pencarian, *Table.SelectRows* akan melakukan tugasnya. Namun, karena hanya akan ada satu baris yang cocok di tabel *Details*, kita dapat menggunakan pencarian juga dalam kasus khusus ini. Namun, perlu diingat bahwa ketika *lookup* menghasilkan lebih dari satu baris, kesalahan akan muncul. Pertimbangkan kueri ini:

```
let
  Source = Table.FromRows(
    {
      { 98731, "In development",
        [
          Version = {1.00, 1.01, 1.10, 1.01},
          Details = #table(
            {"Version", "Notes"},
            {
              {1.00, "Draft version"},
              {1.01, "Minor updates"},
              {1.10, "Updated schema"}
            }
          )
        ]
      },
      { 98732, "Internal test",
        [
          Version = {1.00, 2.00, 2.50},
          Details = #table(
            {"Version", "Notes"},
            {
              {1.00, "Beta version"},
              {2.00, "Major overhaul"},
              {2.50, "Performance improvements"}
            }
          )
        ]
      },
      { 98733, "Beta test",
        [
          Version = {2.00, 2.01},
          Details = #table(
            {"Version", "Notes"},
```

```
            {
              {2.00, "New release"},
              {2.01, "Minor Bug fixes"}
            }
          )
        ]
      },
      { 98734, "In development",
        [
          Version = {1.00, 1.20},
          Details = #table(
            {"Version", "Notes"},
            {
              {1.00, "Initiated from template"},
              {1.20, "Feature additions"}
            }
          )
        ]
      },
      { 98735, "Rework",
        [
          Version = {1.00, 2.00, 3.00, 3.50, 4.00, 3.50},
          Details = #table(
            {"Version", "Notes"},
            {
              {1.00, "Draft version"},
              {2.00, "New release"},
              {3.00, "Major overhaul"},
              {3.50, "Beta version"},
              {4.00, "To Production environment"}
            }
          )
        ]
      }
    }, {"Project", "Status", "Details"}
  ),
    ReplaceValue = Table.ReplaceValue( Source,
      each [Details],
      each [
        Version=List.Last([Details][Version]),
        Notes=[Details][Details]{[Version=Version]}[Notes] ],
```

```
        Replacer.ReplaceValue,
        {"Details"}
    ),
    ExpandDetails = Table.ExpandRecordColumn( ReplaceValue,
      "Details",
      {"Version", "Notes"},
      {"Version", "Notes"}
    )
in
    ExpandDetails
```

Langkah *ReplaceValue* melakukan transformasi pada kolom *Details* dari tabel *Source*. *Table.ReplaceValue* mengganti nilai tertentu dalam kolom tabel. Diperlukan lima argumen: tabel yang akan diubah, nilai lama, nilai baru, replacer, dan kolom tempat penggantian harus dilakukan. Berikut cara kerjanya:

- *OldValue: each [Details]* menentukan bahwa nilai lama yang akan diganti adalah nilai saat ini di kolom *Details*.
- *NewValue*: Ini menginisialisasi record dengan dua kolom: Version dan Notes. Kedua kolom diberi ekspresi:
    - *Version=List.Last([Details][Version])*: Ini mengekstrak item terakhir dari list Version dalam record Details saat ini.
    - *Notes=[Details][Details]{[Version=Version]}[Notes]*: Ini melakukan pencarian dalam tabel bersarang dari record Details saat ini dan menggunakan *{[Version=Version]}* untuk menemukan baris tempat kolom Version cocok dengan nomor versi terakhir yang diekstrak sebelumnya (i), mengambil kolom [Notes] dari baris yang cocok.
- Pada dasarnya, untuk setiap baris dalam tabel *Source*, record *Details* diganti dengan record baru yang berisi item list terakhir dari list Version dan catatan terkait dari tabel bersarang.

- *Replacer.ReplaceValue* adalah salah satu replacer bawaan; fungsi ini akan mengganti seluruh konten sel jika nilai saat ini cocok dengan *oldValue* yang ditentukan secara tepat.
- *Columns to search:* {"*Details*"} menentukan list kolom untuk mencari dan mengganti nilai lama; penggantian apa pun hanya akan terjadi di kolom ini.

Seperti yang disebutkan, dalam semua kasus di mana lebih dari satu baris tunggal diharapkan – atau, lebih tepatnya, satu baris tunggal tidak dapat dijamin – *Table.SelectRows* harus digunakan sebagai ganti pencarian. Ini berarti nilai argumen ketiga yang ditentukan dalam *Table.ReplaceValue*, *NewValue*, juga dapat ditentukan seperti ini:

```
each Table.SelectRows( [Details][Details],
    (row)=> row[Version] = List.Last([Details][Version])
)
```

Di sini, *Table.SelectRows* memfilter tabel *Details* yang bertingkat untuk menyimpan hanya baris yang kolom *Version*-nya cocok dengan versi yang diambil dari list *Version* dalam record *Details* saat ini, yang merupakan kriteria untuk memilih baris. Untuk setiap baris dalam tabel *Source*, record Details diganti dengan tabel yang difilter yang hanya berisi baris yang cocok dengan item terakhir dari list *Version*.

Terakhir, operasi perluasan kolom dimulai melalui panah samping di header kolom, yang digunakan untuk mengekstrak data. Anda dapat mengharapkan perilaku berikut:

- Ketika lebih dari satu kolom dipilih, record akan memperlebar tabel utama.
- Jika tabel berisi lebih dari satu baris dan beberapa kolom dipilih, tabel utama akan diperlebar dan diperluas.

Menavigasi struktur campuran adalah tentang mengenali nilai dan mengetahui cara menanganinya. Meskipun setiap skenario berbeda, ada banyak teknik yang dapat Anda gunakan.

G. Ringkasan

Dalam bab ini, kita telah menyiapkan panggung bagi Anda untuk melangkah dengan percaya diri melampaui batasan UI, dengan memperkenalkan pendekatan kode rendah untuk memulai perjalanan itu. Perubahan dalam kemampuan Anda ini akan menyediakan akses ke sejumlah besar fungsi yang akan meningkatkan kemampuan transformasi data Anda lebih jauh.

Kita mengeksplorasi cara bekerja dengan list, record, dan tabel, termasuk saat bersarang di dalam struktur lain atau saat tercampur. Tujuannya adalah untuk memberi Anda teknik praktis untuk mengekstraksi, membentuk ulang, dan mengubah nilai-nilai ini, terkadang memamerkan beberapa metode yang mencapai hasil yang sama dan dengan demikian menyoroti fleksibilitas bahasa M.

Memahami setiap kumpulan data disertai dengan tantangan dan persyaratannya yang unik. Kita menyajikan berbagai contoh yang mencakup berbagai skenario. Baik yang langsung, seperti memilih data, atau yang lebih rumit, tujuannya adalah untuk menginspirasi dan mempersiapkan Anda untuk berbagai teka-teki potensial yang mungkin menemukan jalannya kepada Anda.

Di bab berikutnya, kita akan mulai mempelajari cara menulis fungsi kustom Anda sendiri.

H. Penutup

1. Tes Formatif

No	Soal	Bobot
1.	Jelaskan apa yang dimaksud dengan struktur bersarang dalam konteks Power Query M. Berikan contoh bagaimana struktur ini muncul dalam sumber data seperti JSON atau XML.	10

2.	Uraikan proses transisi dari penggunaan antarmuka pengguna (UI) ke pengkodean dalam Power Query M. Mengapa pemahaman kode penting saat bekerja dengan data bersarang?	10
3.	Jelaskan cara kerja transformasi nilai dalam tabel bersarang. Bagaimana Anda dapat memodifikasi nilai spesifik di dalam struktur bersarang?	10
4.	Apa perbedaan antara list (list) dan catatan (record) dalam Power Query M? Berikan contoh penggunaan masing-masing dalam manipulasi data.	10
5.	Bagaimana cara Anda menavigasi dan memanipulasi data dalam tabel bersarang? Sertakan contoh skenario transformasi yang umum.	10
6.	Jelaskan bagaimana fungsi M dapat digunakan untuk mengekstrak elemen tertentu dari struktur bersarang. Berikan contoh penggunaan fungsi seperti *Record.Field* atau *List.Transform*.	10
7.	Bagaimana pendekatan penanganan data berbeda ketika bekerja dengan struktur campuran yang mengandung tabel, catatan, dan list secara bersamaan?	10
8.	Apa tantangan umum dalam menangani struktur bersarang dalam Power Query M? Bagaimana Anda menyarankan untuk mengatasinya?	10
9.	Jelaskan bagaimana struktur data bersarang dapat meningkatkan fleksibilitas dan representasi relasi dalam kumpulan data besar. Berikan ilustrasi nyatanya.	10
10.	Analisislah bagaimana pemahaman mendalam tentang struktur bersarang dapat meningkatkan kemampuan	10

	pengguna dalam menerapkan transformasi kompleks di Power Query.	

DUMMY
Penerbitan & Percetakan
UNP PRESS

DAFTAR PUSTAKA

Bayazit M. and Önöz B. (2007), To prewhiten or not to prewhiten in trend analysis?, Hydrological Sciences Journal, 52:4, 611-624. https://doi.org/10.1623/hysj.52.4.611.

Box, G. E. P. and Cox, D. R. (1964), An analysis of transformations. Journal of the Royal Statistical Society, Series B, 26, 211-252. http://www.ime.usp.br/~abe/lista/ pdfQWaCMboK68.pdf.

Breiman, L. Random Forests, Machine Learning 45, 5–32 (2001): https://doi.org/10.1023/A:1010933404324.

Chen, Tianqi and Guestrin, Carlos. (2016). XGBoost: A Scalable Tree Boosting System. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16). Association for Computing Machinery, New York, NY, USA, 785–794: https://doi.org/10.1145/2939672.2939785.

David Salinas, Valentin Flunkert, Jan Gasthaus, Tim Januschowski (2020). DeepAR: Probabilistic forecasting with autoregressive recurrent networks. International Journal of Forecasting. 36-3. 1181-1191: https://doi.org/10.1016/j.ijforecast.2019.07.001

David Salinas, Valentin Flunkert, Jan Gasthaus, Tim Januschowski (2020). DeepAR: Probabilistic forecasting with autoregressive recurrent networks. International Journal of Forecasting. 36-3. 1181-1191: https://doi.org/10.1016/j.ijforecast.2019.07.001

Guerrero, Victor M. (1993), Time-series analysis supported by power transformations. Journal of Forecasting, Volume 12, Issue 1, 37-48. https://onlinelibrary.wiley.com/ doi/10.1002/for.3980120104.

Ke, Guolin et.al. (2017), LightGBM: A Highly Efficient Gradient Boosting Decision Tree. Advances in Neural Information Processing Systems, pages 3149-3157: https://dl.acm.org/doi/ pdf/10.5555/3294996.3295074.

Montero-Manso, P., Hyndman, R.J.. (2020), Principles and algorithms for forecasting groups of time series: Locality and globality. arXiv:2008.00444[cs.LG]: https://arxiv.org/ abs/2008.00444

Montero-Manso, P., Hyndman, R.J.. (2020), Principles and algorithms for forecasting groups of time series: Locality and globality. arXiv:2008.00444[cs.LG]: https://arxiv.org/ abs/2008.00444

Prokhorenkova, Liudmila, Gusev, Gleb et al. (2018), CatBoost: unbiased boosting with categorical features. Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18): https://dl.acm.org/doi/abs/10.5555/3327757.3327770.

Slawek Smyl (2020). A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. International Journal of Forecasting. 36-1: 75-85 https://doi. org/10.1016/j.ijforecast.2019.03.017

Slawek Smyl (2020). A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. International Journal of Forecasting. 36-1: 75-85 https://doi. org/10.1016/j.ijforecast.2019.03.017

White, H. (1980), A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity. Econometrica Vol. 48, No. 4 (May 1980), pp. 817-838 (22 pages). https://doi.org/10.2307/1912934.

GLOSARIUM

AAS	**Azure Analysis Services:** Layanan Platform-as-a-Service (PaaS) yang disediakan oleh Microsoft Azure untuk membuat solusi Business Intelligence (BI) tingkat perusahaan berbasis model data analitik
ACE	**Access Database Engine:** Komponen inti Microsoft Access yang bertanggung jawab untuk menyimpan dan mengambil data dalam format basis data Access (.accdb dan .mdb).
AD	**Active Directory:** Layanan direktori yang dikembangkan oleh Microsoft untuk mengelola dan mengatur sumber daya dalam jaringan komputer berbasis Windows Server.
ADO.NET	**ActiveX Data Objects .NET:** Seperangkat kelas dalam .NET Framework yang menyediakan akses terpadu ke berbagai sumber data.
AI	**Artificial Intelligence:** Bidang ilmu komputer yang berfokus pada pengembangan sistem komputer yang dapat melakukan tugas-tugas yang biasanya membutuhkan kecerdasan manusia.
API	**Application Programming Interface:** Sekumpulan aturan, protokol, dan alat yang memungkinkan berbagai aplikasi perangkat lunak untuk saling berkomunikasi dan bertukar data.
BI	**Business Intelligence:** Proses pengumpulan, pengorganisasian, analisis, interpretasi, dan penyajian data bisnis untuk membantu para pengambil keputusan membuat keputusan yang lebih baik dan lebih terinformasi.
COM	**Component Object Model:** Kerangka kerja biner yang dikembangkan oleh Microsoft yang memungkinkan aplikasi untuk berinteraksi dengan objek perangkat lunak tanpa perlu mengetahui detail implementasi internal objek tersebut.
CRM	**Customer Relationship Management:** Strategi bisnis yang berfokus pada membangun dan memelihara hubungan yang kuat dan positif dengan pelanggan.
CSS	**Cascading Style Sheets:** Bahasa stylesheet yang digunakan untuk menggambarkan tampilan (presentasi) dokumen yang ditulis dengan bahasa markup seperti HTML atau XML.
CSV	**Comma-Separated Value:** Format file teks sederhana yang digunakan untuk menyimpan data tabular (seperti spreadsheet atau database) di mana setiap baris mewakili satu catatan dan setiap kolom dalam catatan tersebut dipisahkan oleh tanda koma.

DBMS	**Database Management System:** Perangkat lunak yang memungkinkan pengguna untuk membuat, memelihara, dan berinteraksi dengan database secara efisien dan aman.
DOM	**Document Object Model:** Representasi terstruktur (berbentuk pohon) dari dokumen HTML atau XML yang memungkinkan program (seperti JavaScript) untuk mengakses dan memanipulasi konten, struktur, dan gaya dokumen tersebut.
DSN	**Data Source Name:** String koneksi yang berisi informasi yang dibutuhkan oleh aplikasi untuk terhubung ke sumber data tertentu.
ECMA	**European Computer Manufacturers Association:** Organisasi standar internasional untuk sistem informasi dan komunikasi.
ELT	**Extract-Load-Transform:** Proses integrasi data yang terdiri dari tiga tahap utama: Ekstraksi (Extract), Pemuatan (Load), dan Transformasi (Transform).
ERP	**Enterprise Resource Management:** Sebuah sistem yang mengintegrasikan proses bisnis inti perusahaan
EST	**Eastern Standard Time:** Zona waktu yang digunakan di sebagian Amerika Utara dan beberapa wilayah Karibia.
ETL	**Extract, Transform, And Load:** Proses integrasi data yang umum digunakan untuk memindahkan data dari berbagai sumber ke sistem target, seperti data warehouse, untuk tujuan analisis dan pelaporan.
GUI	**Graphical User Interface:** Jenis antarmuka pengguna yang memungkinkan pengguna berinteraksi dengan perangkat elektronik melalui elemen visual seperti ikon, menu, jendela, tombol, dan pointer (biasanya dikendalikan oleh mouse, trackpad, atau layar sentuh).
HDFS	**Hadoop Distributed File System:** Sistem file terdistribusi yang dirancang untuk menyimpan dan mengelola dataset yang sangat besar yang berjalan di atas perangkat keras komoditas (perangkat keras standar yang tidak terlalu mahal).
HL7	**Health Level 7:** Seperangkat standar internasional yang digunakan untuk pertukaran, integrasi, berbagi, dan pengambilan informasi kesehatan elektronik antara berbagai sistem perangkat lunak yang digunakan oleh penyedia layanan kesehatan.
HTML	**HyperText Markup Language:** Bahasa markup standar untuk membuat halaman web.
IT	**Information Technology:** Penerapan komputer dan sistem telekomunikasi untuk menyimpan, mengambil, mengirim, dan memanipulasi data atau informasi.
JDN	**Julian Day Number:** Jumlah hari yang telah berlalu sejak epoch Julian tengah siang di Greenwich pada tanggal 1 Januari 4713 SM (kalender Julian proleptik).

JSON	**JavaScript Object Notation:** Format data ringan yang digunakan untuk mentransmisikan data antara server dan aplikasi web, serta untuk menyimpan data terstruktur.
ODBC	**Open Database Connectivity:** Antarmuka pemrograman aplikasi (API) standar yang memungkinkan aplikasi untuk mengakses berbagai sistem manajemen basis data (DBMS) secara independen dari DBMS yang mendasarinya.
OLAP	**Online Analytics Processing:** Pendekatan untuk menganalisis data multidimensi dalam jumlah besar dengan cepat dari berbagai perspektif.
OLE DB	**Object Linking and Embedding Database:** Sekumpulan antarmuka yang dikembangkan oleh Microsoft yang menyediakan akses terpadu ke berbagai sumber data.
OLTP	**Online Transactional Processing:** Jenis pemrosesan data yang berfokus pada eksekusi dan pencatatan sejumlah besar transaksi kecil dan bersamaan secara real-time.
OPC	**Open Packaging Conventions:** Sebuah teknologi format file kontainer yang awalnya dibuat oleh Microsoft untuk menyimpan kombinasi file XML dan non-XML yang bersama-sama membentuk satu entitas logis.
PBRS	**Power BI Report Server:** Solusi pelaporan on-premises (di tempat) yang memungkinkan Anda untuk membuat, menerbitkan, dan mengelola laporan serta KPI (Key Performance Indicators) di dalam infrastruktur organisasi Anda sendiri.
PDF	**Portable Document Format:** Format file yang dikembangkan oleh Adobe untuk menyajikan dokumen, termasuk teks, gambar, hyperlink, font yang disematkan, dan lainnya, dengan cara yang independen dari perangkat lunak aplikasi, perangkat keras, dan sistem operasi.
SSAS	**SQL Server Analysis Services:** Komponen dalam Microsoft SQL Server yang menyediakan kemampuan pemrosesan analitik daring (OLAP) dan data mining.
SSIS	**SQL Server Integration Services:** Platform untuk membangun solusi integrasi data dan transformasi data berkinerja tinggi.
UI	**User Interface:** Bagian dari sistem (perangkat lunak atau perangkat keras) yang memungkinkan manusia berinteraksi dengannya.
URI	**Uniform Resource Identifier:** String karakter yang mengidentifikasi sumber daya fisik atau abstrak.
UTC	**Coordinated Universal Time:** Standar waktu utama dunia yang digunakan sebagai dasar untuk semua zona waktu sipil di seluruh dunia.

VoIP	**Voice over Internet Protocol:** Sekelompok teknologi yang memungkinkan Anda melakukan panggilan suara menggunakan koneksi internet broadband, bukan saluran telepon analog biasa.
WMS	**Warehouse Management System:** Sistem perangkat lunak yang dirancang untuk mengelola dan mengoptimalkan operasi gudang.
XML	**eXtensible Markup Language:** Bahasa markup yang dirancang untuk membawa data dan mendeskripsikan struktur data.

DUMMY
Penerbitan & Percetakan
UNP PRESS

INDEKS

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

TENTANG PENULIS

Dr. Phil. Dony Novaliendry, S.Kom., M.Kom., Lahir dan besar di Padang tanggal 4 November 1975. Merupakan anak dari Prof. Dr. H. Aljufri B. Syarif, M.Sc (Alm) dengan Endang Ratna Sulistri. Anak ke-4 dari 4 orang bersaudara. Menamatkan S1 di Universitas Gunadarma Jurusan Sistem Informasi dan di lanjutkan S2 di Universitas Gadjah Mada Jurusan Ilmu Komputer dan melanjutkan S3 di National Kaohsiung University of Science and Technology (NKUST) di Taiwan dengan bidang Bio-Informatics. Saat ini tertarik untuk mengembangkan diri di bidang bio-informatics, bio-medics, Artificial Intelligence, Decision Support System, Multimedia, Big Data dan Data Mining. Ini adalah buku yang ke enam yang diterbitkan. Semoga ditahun mendatang masih bisa terus berkarya menciptakan beberapa buku lagi.

Quote:Life Must Go On.

Saran, kritik dan kerjasama: dony.novaliendry@ft.unp.ac.id

RINGKASAN ISI BUKU

Buku "Panduan Definitif untuk Power Query (M) – Jilid 2" melanjutkan pembahasan dari jilid sebelumnya dengan fokus yang lebih mendalam pada tipe data, nilai terstruktur, dan konsep lanjutan dalam bahasa M. Buku ini dirancang untuk membantu pembaca memahami bagaimana data direpresentasikan, dikelola, dan dimanipulasi secara efisien di dalam Power Query.

Pembahasan dimulai dengan pemahaman tipe data—baik primitif maupun khusus—serta pentingnya konsistensi, validasi, dan konversi tipe dalam proses transformasi data. Selanjutnya, buku menguraikan secara rinci tentang nilai terstruktur seperti list, record, dan tabel, lengkap dengan metode pembuatan, manipulasi, serta penetapan tipe data pada masing-masing struktur.

Pada bagian lanjutan, buku ini membahas konseptualisasi M, termasuk ruang lingkup, lingkungan global, closures, hingga pengelolaan metadata. Topik terakhir memperdalam kemampuan pembaca dalam bekerja dengan struktur bersarang (nested structures), mencakup pengolahan list, record, tabel, hingga kombinasi struktur yang lebih kompleks.

Dengan disertai kasus pemantik berpikir kritis, tes formatif, glosarium, dan lampiran, jilid kedua ini memberikan wawasan praktis sekaligus teoritis, sehingga menjadi panduan berharga bagi mahasiswa, dosen, peneliti, maupun praktisi data yang ingin menguasai Power Query secara komprehensif.

LAMPIRAN

1. Kartu Rencana Studi (KRS)

	KEMENTERIAN PENDIDIKAN, KEBUDAYAAN, RISET DAN TEKNOLOGI UNIVERSITAS NEGERI PADANG – DIREKTORAT AKADEMIK – SUBDIT. INOVASI PEMBELAJARAN DAN MBKM					
	RENCANA PEMBELAJARAN SEMESTER					
MATA KULIAH (MK)	**KODE**	**Rumpun MK**	**BOBOT (sks)**		**SEMESTER**	**Tgl Penyusunan**
Praktikum Basis Data	INF1.62.4008	Mata kuliah Wajib Program Studi	1 SKS	Praktek	4	30 Januari 2024
OTORISASI / PENGESAHAN	**Dosen Pengembang RPS**		**Koordinator RMK**		**Koordinator PRODI**	
Direktorat Akademik, UNP **Subdit. Inovasi Pembelajaran dan MBKM (Dr. Nofrion, M. Pd)**	**1. Dr. Phil. Dony Novaliendry, S. Kom., M. Kom**		**Dr. Phil. Dony Novaliendry, S. Kom., M. Kom** **NIP. 197511042006041002**		**Dr. Yeka Hendriyani, S.Kom., M.Kom** **NIP. 198405202010122003**	

Capaian Pembelajaran	**CPL-PRODI yang dibebankan pada MK**	
	S5	Bertaqwa kepada Tuhan Yang Maha Esa dan mampu menunjukkan sikap religious dan Menunjukkan sikap bertanggungjawab atas pekerjaan di bidang keahliannya secara mandiri
	P2	Memahami konsep dasar matematika, ilmu listrik dan elektronika dalam bidang komputer
	KU5	mampu mengambil keputusan secara tepat dalam konteks penyelesaian masalah di bidang keahliannya, berdasarkan hasil analisis informasi dan data.
	KK 2	Kemampuan menguasai dasar pemograman phyton,metode komputasi Gauss dan komputasi metode LU Decomposition
	Capaian Pembelajaran Mata Kuliah (CPMK)	
	CPMK 1	Menguasai konsep bahasa pemrograman.
	CPMK 2	Mengidentikasi model- model bahasa pemrograman.
	CPMK 3	Membandingkan berbagai solusi.
	CPMK 4	Menguasai konsep-konsep basis data dan mampu membangun basis data untuk pngembangan sistem berbasis komputer
	Kemampuan akhir tiap tahapan belajar (Sub-CPMK)	
	SUB CPMK 1	Mahasiswa mampu memahami dan menganalisa konsep Bahasa pemrograman
	SUB CPMK 2	Mahasiswa mampu merumuskan konsep model-model Bahasa pemrograman
	SUB CPMK 3	Mahasiswa mampu merumuskan latar belakang konsep membandingkan berbagai solusi terkait Bahasa pemrograman.
	SUB CPMK 4	Mahasiswa mampu memahami, menganalisa, dan membangun konsep basis data, perancangan basis data, perancangan tabel-tabel, entry data pada tabel, relasi antar tabel, dan pembuatan report.

Peta CPL – CP MK

	Sub-CPMK1	Sub-CPMK2	Sub-CPMK3	Sub-CPMK4
CPMK 1	√			
CPMK 2		√		
CPMK 3			√	
CPMK 4				√

Deskripsi Singkat Mata Kuliah	Mata kuliah ini mempelajari dan menguasai konsep dan mengimplementasi pembuatan aplikasi basis data menggunakan salah satu DBMS dan bahasa pemrograaman basis data dengan urutan: perancangan tabel-tabel, entry data pada tabel, pembuatan form (untuk entry data dan untuk tampilan menu), pembuatan query, pembuatan report.
Bahan Kajian: Materi pembelajaran	1. Aplikasi DBMS, MariaDB, MySQL, PostgreSQL, phpMyAdmin 2. Bahasa pemograman sql, DDL 3. DML (Data Manipulation Language) 4. Arithmetic Operator, Agregate function, String function, Numeric function, Date/Time Function 5. Group by, group by with order, having 6. Fungsi agregat 7. Database Relation 8. Database Relation Join 9. Union, Intersect, Except 10. View dan control flow function 11. Create Procedure

Pustaka Buku 10 tahun terakhir Artikel 5 tahun terakhir Kecuali buku atau artikel yang memuat grand theory.	**Utama:** 1. Modul Praktikum Sistem Basis Data: Tim Dosen Program Studi Informatika 2. C. J. Date. 2006. An Introduction to Database Systems 8th. Pearson Education
	Pendukung: 1. Churcher, C., 2007, Beginning Database Design: From Novice to Professional (ebook available) 2. Oppel, A. & Sheldon, R., 2009, SQL: A Beginner's Guide- 3. Taylor, A.G., 2011, SQL Essential-All in One for Dummies
Dosen Pengampu	Dr. Phil. Dony Novaliendry, S. Kom., M. Kom NIP. 197511042006041002
Mata kuliah syarat	-

Mg Ke-	SUB-CPMK (Kemampuan Akhir Yang Diharapkan)	Penilaian		Bentuk Pembelajaran, Metode Pembelajaran, Penugasan Mahasiswa [Estimasi Waktu]		Materi Pembelajaran [Rujukan]	Bobot Penilaian (%)
		Indikator	Bentuk & Kriteria	Luring (Tatap Muka)	Daring (Online)		
1	Mahasiswa mengenal dan dapat menggunakan Aplikasi DBMS	1. Ketepatan menjelaskan aplikasi Xampp (MariaDB) 2. Ketepatan menjelaskan aplikasi MySQL 3. Ketepatan menjelaskan aplikasi PostgreSQL 4. Ketepatan menjelaskan aplikasi Oracle	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	Mengenal Aplikasi DBMS - MariaDB - MySQL - PostgreSQL - Oracle	**10%**
2	Mahasiswa mampu menjelaskan perintah-perintah dasar SQL dan kelompok pernyataan SQL untuk pendefinisian basis data	1. Ketepatan menjelaskan DDL (Data Defenition Language)	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	SQL: - Pengenalan SQL - DDL (Data Definition Language)	**5%**
3	Mahasiswa mampu menggunakan perintah-perintah DML	1. Ketepatan menjelaskan Data Manipulation Language	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	DML (Data Manipulation Language)	**5%**
4	Mahasiswa Memahami perintah-perintah SQL untuk mengambil data dan kemudian melakukan perhitungan-perhitungan aritmatika dari datatersebut	1. Ketepatan menjelaskan Arithmetic Operators 2. Ketepatan menjelaskan agregate function 3. Ketepatan menjelaskan aString function 4. Ketepatan menjelaskan	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	Operator, Agregate Function, String Function, Numeric Function, Date/Time Function.	**10%**

		5. Ketepatan menjelaskan Date Function					
5	Mahasiswa mampu menggunakan perintah SQL untuk menyatukan dua atau lebih grup data kedalam suatu fungsi data tunggal	1. Ketepatan menjelaskan Group By 2. Ketepatan menjelaskan group by with order 3. Ketepatan menjelaskan having	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	Group By, Group By with Order By, Having.	**10%**
6	Mahasiswa mampu menjalankan perintah-perintah fungsi Agregat pada tabel	1. Ketepatan menjelaskan fungsi agreagat	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	Fungsi Agregat	**10%**
7	Mahasiswa mampu merelasikan tabel	1. Ketepatan menjelaskan relasi 2 tabel 2. Ketepatan menjelaskan relasi 3 tabel 3. Ketepatan menjelaskan lebih dari 3 tabel	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	Database Relation - Relasi 2 tabel - Relasi 3 tabel - Relasi Lebih dari 3 tabel	**10%**
8	**Ujian Tengah Semester (UTS) = 10%**						
9	Mahasiswa mampu merelasikan tabel dengan menggunakan perintah Join	1. Ketepatan menjelaskan relasi antar tabel menggunakan join	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	Relasi Antar Tabel - Join	**10%**
10	Mahasiswa mampu menjalankan perintah-perintah Union, Intersect, Except.	1. Ketepatan menjelaskan Union 2. Ketepatan menjelaskan Intersect 3. Ketepatan menjelaskan except	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	- Union - Intersect - Except	**10%**
11-12	Mahasiswa Memahami dan mampu menggunakan View dan control flow function	1. Ketepatan menjelaskan view di database 2. Ketepatan menjelaskan control flow function	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	View dan control flow function.	**10%**
13-15	Mahasiswa memahami dan mampu menggunakan Store Procedure dan Function yang merupakan perintah-perintah SQL yang diletakkan di dalam server database.	1. Ketepatan menjelaskan create procedure 2. Ketepatan menjelaskan create function	Menggunakan Rubrik Penilaian	1. Presentasi 2. Praktek **TM : 1x (1 x 100 Menit)** 3. Penugasan Terstruktur **BM+BT : 1x(1x70 Menit)**	Bahan ajar materi dapat diunduh melalui LMS UNP pada tautan : https://elearning.unp.ac.id/my/	Create Procedure dan Create Function.	**15%**
16	**Ujian Akhir Semester (UAS) = 10%**						