ChatGPT

Power Query M: Datentyp List und Record

Power Query M unterscheidet einfache (primitive) Datentypen wie Zahl, Text, Datum, Boolesch etc. von komplexen, **strukturierten** Typen. Zu den wichtigsten strukturierten Typen gehören **Listen**, **Records** und **Tabellen**. Eine **Liste** (`list`) ist eine *geordnete* Folge von Werten (ähnlich einem Array), auf die man per Index zugreifen kann [1]. Ein **Record** (`record`) ist eine *ungeordnete* Sammlung benannter Felder (Schlüssel-Werte-Paare), ähnlich einem Datensatz oder einem JSON-Objekt [2]. Tabellen sind im Wesentlichen Listen von Records mit definierten Spalten (Schema). In Power Query gelten Listen und Records als primitive (nicht abstrakte) Basistypen – sie haben per se keine Feld- oder Elementeinschränkungen und können beliebige Werte enthalten [1] [2].

- **Primitive vs. strukturiert**: Primitive Typen haben einen einzelnen Wert (z.B. `1`, `"Text"`, `true`). **Listen** und **Records** können dagegen mehrere Werte enthalten. Eine Tabelle besteht aus Records (Zeilen) mit identischem Aufbau (Spaltenschema).
- **Struktur**: Eine List ist *geordnet* (Position 0,1,2,...), ein Record *benannt* (jedes Feld hat einen eindeutigen Namen). Beispielsweise ist `{1,2,3}` eine Liste von Zahlen, während `[Name="Eva", Alter=30]` ein Record mit den Feldern *Name* und *Alter* ist.
- **Vergleich List/Record/Tab**: Tabellen definieren feste Spalten (mit Typen) und enthalten nur Records, Listen können heterogene Werte enthalten (z.B. `1`, `"abc"`, `false` gemeinsam) [3]. Listen sind flexibel, aber Operationen darauf werden komplett in M ausgeführt. Für große Listen empfiehlt sich ggf. *Buffering* (siehe unten). Tabellen können dagegen oft „Query Folding“ nutzen (Aggregationen in der Datenquelle).

Vergleich: List vs. Record vs. Table

Merkmal	**List**	**Record**	**Table**
Struktur	Geordnete Sequenz von Werten, adressierbar über Index (z.B. `{10,20,30}`) [1].	Ungeordnete Sammlung benannter Felder (z.B. `[Name="Max", Alter=28]`). Jedes Feld hat einen eindeutigen Namen [2].	Geordnetes Datenraster: besteht aus Spalten (Feldnamen) und Zeilen (Records). Beispiel: `#table({"Name","Alter"}, {{"Max",28},{"Eva",30}})`.
Elementtypen	Können heterogen sein (z.B. Zahlen, Text, Records in einer Liste). Keine Typsicherheit per Element unless mit Typisierung.	Felder können verschiedene Typen haben, aber Feldernamen sind fix. Ein Record kann auch andere Records oder Listen als Feldwerte enthalten.	Jede Spalte hat einen definierten Datentyp. Alle Zeilen haben gleich benannte Spalten.

Merkmal	**List**	**Record**	**Table**
Zugriff	Funktional: `List{Index}` für Element an Position oder Funktionen (z.B. `List.First`, `List.Transform` 等).	Per Feldname: `record[Feldname]` oder `Record.Field(record, "Feldname")`. Im Code oft auch `record.Feldname`, wenn der Name keine Sonderzeichen hat.	Über Tabellenspalten: `Table.SelectColumns`, `Table.AddColumn` etc. Datensatzwerte per Zeilen-Typ, oft in Kombination mit List/ Record.
Typusbedingungen	Lockertyp: Default ohne Einschränkung. Man kann auch mit Typ-Angaben Liste definieren (z.B. `type { number }`).	Lockertyp: Default ohne Feldbeschränkung. Man kann record-Typen definieren, die erlaubte Felder und Typen fixieren (z.B. `type [Name=text, Age=number]`).	Festes Schema: Tabellentyp enthält Spaltennamen und Spaltentypen [4]. Tabellen können Schlüssel (Keys) haben.
Anwendungsfälle	*Iterationen, Sequenzen, temporäre Sammlungen.* Gut für dynamisches Filtern, Aggregieren (z.B. `List.Sum`, `List.Average`) oder Erzeugen von Reihen (z.B. `List.Numbers`) [5].	*Parameter, Metadaten, strukturierte Datensätze.* Eignet sich als Konfigurations- oder Rückgabetyp für Funktionen, zum Speichern von Detailinfos (z.B. `Value.Metadata` gibt ein Record-Objekt zurück) [6].	*Datenergebnisse, Berichtsdaten.* Tabellen sind das Standardformat für Abfrageergebnisse in Power BI/ Power Query, ermöglichen Gruppierungen, Joins, Sortierung, Filterung mit nativen Befehlen.
Performance	Listenoperationen sind in M nicht „gefaltet“ und können bei großen Datenmengen teuer sein. Mit `List.Buffer` kann man die Evaluierung optimieren (z.B. mehrfach benötigte Listen im Speicher halten) [7].	Sehr klein (nur einzelne Zeile) – kaum Overhead. Das Zugreifen auf Felder in Records ist günstig. Bei sehr großen Records (viele Felder) steigt allerdings Aufwand.	Tabellen können (je nach Quelle) in der Datenquelle berechnet werden (Query Folding). Komplexe Operationen in M (z.B. `Table.Group`) können viel Zeit und Speicher beanspruchen, aber M verwendet oft effiziente interne Algorithmen.

Einsatz in ETL-Szenarien

In typischen ETL-Prozessen kommen Listen und Records für verschiedene Aufgaben zum Einsatz:

- **Aggregation und Gruppierung:** Beim Gruppieren von Daten erzeugt `Table.Group` häufig Listen oder Tables als Ergebnis. Beispiel:

```
let
    Quelle = #table({"Kategorie","Wert"}, {{"A",10},{"A",20},{"B",5}}),
```

```
    Gruppiert = Table.Group(Quelle, {"Kategorie"}, {{"Werte", each
Table.Column(_, "Wert"), type list}})
in
    Gruppiert
```

In diesem Beispiel enthält die Spalte *Werte* pro Kategorie eine Liste aller zugehörigen Werte. Solche Listen lassen sich mit Funktionen wie `List.Sum`, `List.Min`, `List.CumSum` usw. weiter aggregieren.

- **Transformation von Spaltenwerten:** Listen werden oft für Spaltentransformationen genutzt. Etwa kann man eine CSV-Zeichenkette in eine Liste aufteilen und anschließend die Liste erweitern oder zusammenführen. Beispiel – Splitten und Erweitern:

```
let
    Tabelle = #table({"CSV"}, {{"A,B,C"}, {"D,E"}}),
    Hinzu = Table.AddColumn(Tabelle, "Geteilt", each Text.Split([CSV],
","")),
    Erweitert = Table.ExpandListColumn(Hinzu, "Geteilt")
in
    Erweitert
```

Hier wandelt `Text.Split` einen Text in eine Liste um, und `Table.ExpandListColumn` verwandelt die Listenwerte in zusätzliche Zeilen.

- **Metadaten-Verwaltung:** Record-Typen werden genutzt, um Metadaten oder Parameter zu speichern. Power Query speichert z.B. Metadaten als Record (`Metadata`-Record) [6]. Ebenso kann man Konfigurationsparameter in einem Record bündeln und im M-Code darauf zugreifen:

```
let
    config = [FilterColumn="Score", Threshold=50],
    result = Table.SelectRows(Quelle, each Record.Field(_,
config[FilterColumn]) > config[Threshold])
in
    result
```

In diesem Beispiel legt der Record `config` fest, in welcher Spalte gefiltert werden soll und welcher Schwellenwert gilt. Das macht den Code flexibel und parametrisiert.

- **Dynamische Spaltenwahl:** Listen und Records erlauben dynamische Auswahl von Spalten. Zum Beispiel:

```
let
    tbl = #table({"x","y","z"}, {{1,2,3},{4,5,6}}),
    spalten = {"x","z"},
    gefiltert = Table.SelectColumns(tbl, spalten)
```

```
in
    gefiltert
```

Hier wird mittels einer Liste (`spalten`) dynamisch festgelegt, welche Spalten aus der Tabelle ausgewählt werden. Ähnlich kann ein Record Feldnamen oder Operationen speichern, um sie dann programmgesteuert anzuwenden.

Syntax und Erstellung von List und Record

Listen und Records lassen sich sowohl **manuell** als auch per **Funktion** erstellen:

- **Manuell (Literal-Syntax):**
- Liste: Verwende geschweifte Klammern `{}` mit Komma-getrennten Werten, z.B. `{1, 2, 3}`, oder literale gemischte Typen wie `{42, "Hallo", null}`. Eine leere Liste ist `{}`. Listen können verschachtelt werden, z.B. `{{1,2},{3,4}}`.
- Record: Verwende eckige Klammern `[]` mit Feldzuweisungen, z.B. `[Name="Anna", Alter=30]`. Die Felder sind `Name` (Text) und `Alter` (Zahl) in diesem Beispiel. Nicht vorhandene Felder in einem Record führen beim Zugriff zu einem Fehler.

- **Per Funktion:**

- **Listen-Erzeugung:** Beispiele sind `List.Numbers(1,5,2)` (erzeugt `{1,3,5}`), `List.Dates`, `List.Generate` (flexibles Listengenerieren), `List.Repeat({0}, 4)` (viermal `[0]`), etc.
- **Record-Erzeugung:** Z.B. `Record.FromList({1,"Bob"}, {"ID","Name"})` erstellt den Record `[ID=1, Name="Bob"]`. Auch `Record.Combine` kann verwendet werden, um mehrere Records zusammenzufügen.

Beispiel-Code:

```
let
    -- Listen erstellen
    lst1 = {1, 2, 3, 4},
    lst2 = List.Numbers(1, 5, 2),   // {1,3,5}
    lst3 = List.Generate( ()=>0, each _<3, each _+1), // {0,1,2}
    -- Record erstellen
    rec1 = [Name="Max", Alter=28],
    rec2 = Record.AddField(rec1, "Ort", "Berlin"),  // [Name="Max", Alter=28,
Ort="Berlin"]
    rec3 = Record.FromList({"Eintrag1", "Eintrag2"}, [FeldA = text, FeldB =
text])
in
    [Listen={lst1, lst2, lst3}, Records={rec1, rec2, rec3}]
```

In diesem Code sieht man beide Ansätze: Literale (`{…}` und `[…]`) sowie Funktionsaufrufe (`List.Numbers`, `Record.AddField`, `Record.FromList`).

Wichtige Built-in-Funktionen (Listen und Records)

List-Funktionen

Power Query M bietet Hunderte von Funktionen für Listen. Wichtige Beispiele (themenbezogen gruppiert) sind:

- **Erzeugung:** `List.Numbers`, `List.Dates`, `List.Generate`, `List.Repeat` – erzeugen Listen von Zahlen, Datumswerten oder nach Regeln.
- **Selektion:** `List.First`, `List.FirstN`, `List.Last`, `List.LastN`, `List.Select`, `List.Skip`, `List.Range` – geben Teilstücke oder gefilterte Teilmengen zurück.
- **Transformation:** `List.Transform`, `List.TransformMany`, `List.Reverse`, `List.Split`, `List.Combine`, `List.Zip` – wenden Funktionen auf Elemente an, zerlegen oder kombinieren Listen (z.B. `List.Reverse` kehrt die Reihenfolge um, `List.Split` teilt eine Liste in Seiten auf [8]).
- **Aggregation/Statistik:** `List.Sum`, `List.Min`, `List.Max`, `List.Average`, `List.Median`, `List.Mode`, `List.StandardDeviation`, `List.Covariance`, u.v.m. – berechnen Summen, Mittelwerte, Min/Max oder Streuungswerte von Listen [9] [10].
- **Mitgliedschaft/Set-Operationen:** `List.Contains`, `List.ContainsAny`, `List.ContainsAll`, `List.Distinct`, `List.Union`, `List.Intersect`, `List.Difference`, etc. – prüfen auf Einhaltung von Bedingungen und führen Mengenoperationen durch.
- **Sonstiges:** `List.Buffer` (speichert eine Liste im Speicher für bessere Performance) [7], `List.IsEmpty`, `List.Count`, `List.Positions` (gibt die Position(en) eines Wertes zurück), `List.FindText`, `List.RemoveNulls`, u.v.m.

Record-Funktionen

Für Records gibt es eigene Funktionen (alle aus den Power Query Referenzen):

- **Erzeugung/Kombination:** `Record.FromList`, `Record.FromTable` (wandeln Liste/Tab in Record um), `Record.Combine` (verschmilzt mehrere Records zu einem).
- **Abfrage:** `Record.Field` (holt den Feldwert zu einem Namen), `Record.FieldOrDefault` (holt Feld oder liefert Default, wenn nicht vorhanden), `Record.FieldNames` (gibt Liste der Feldnamen zurück), `Record.FieldValues` (Liste der Feldwerte in Reihenfolge) [11]. `Record.HasFields` prüft, ob ein oder mehrere Felder existieren. `Record.FieldCount` zählt die Felder.
- **Änderung:** `Record.AddField` (Feld hinzufügen), `Record.RemoveFields` (Felder löschen), `Record.RenameFields` (Felder umbenennen), `Record.ReorderFields` (Reihenfolge ändern), `Record.TransformFields` (Feldwerte mit Funktionen transformieren). Diese Funktionen erleichtern das Modellieren von Records.
- **Spezial:** `Record.ToList` konvertiert alle Feldwerte in eine Liste; `Record.ToTable` liefert eine zweispaltige Tabelle (Feldname, Wert) für einen Record.

Zum Beispiel gibt `Record.FieldNames([x=1,y=2])` die Liste `{"x","y"}` zurück und `Record.Field([x=1,y=2], "y")` den Wert `2` [11].

Zugriff: Punkt-Notation vs. funktionaler Zugriff

- **Listen:** Elemente einer Liste werden über geschweifte Klammern adressiert: `lst{Index}` (0-basiert). Beispiel:

```
let lst = {10,20,30} in lst{1}  // Ergebnis: 20
```

  Alternativ gibt es Funktionen wie `List.First(lst)` oder `List.Last(lst)`. Wählt man `List.FirstN(lst,2)`, erhält man die ersten 2 Elemente als neue Liste.

- **Records:** Felder eines Records erreicht man mit der Punkt-Schreibweise oder Funktionen. Beispiel:

```
let rec = [Name="John", Alter=35, Email="john@example.com"] in rec.Name
```

  Die Variable `rec.Name` liefert `"John"`, ähnlich wie `rec[Name]` oder `Record.Field(rec, "Name")`. Werden Feldnamen dynamisch (z.B. aus einer anderen Spalte oder Variable) festgelegt, benutzt man `rec[Feldname]` bzw. `Record.Field(rec, Feldname)`. Beispiel verschachtelt:

```
let
    data = [Person=[Vorname="Eva", Nachname="Muster"], Werte={100,200}],
    vorname = data[Person][Vorname],       // "Eva"
    zweiterWert = data[Werte]{1}          // 200
in
    [Name=vorname, Wert=zweiterWert]
```

  Hier zeigt `data[Person][Vorname]`, dass man die Punkt-Notation bzw. verschachtelte Record-Felder verwenden kann.

Die funktionale Zugriffsmethode (`Record.Field`, `List.PositionOf` etc.) ist besonders nützlich, wenn Feldnamen oder Indizes erst zur Laufzeit bestimmt werden müssen.

Praxisbeispiele

- **Gruppierung und Aggregation:** Wie oben gezeigt, kann man mit `Table.Group` Listen erzeugen. Anschließend lassen sich Listen mit Aggregationsfunktionen auswerten. Beispiel: Für jede Kategorie die Summe berechnen:

```
let
    Daten = #table({"Kategorie","Wert"}, {{"A",10},{"A",20},{"B",5}}),
    Gruppe = Table.Group(Daten, {"Kategorie"}, {{"Werte", each
Table.Column(_, "Wert"), type list}}),
    Summen = Table.TransformColumns(Gruppe, {"Werte", each List.Sum(_),
Int64.Type})
```

```
in
    Summen
```

Hier enthält *Gruppe[Werte]* pro Zeile eine Liste. Mit `Table.TransformColumns` wird diese Liste durch ihre Summe ersetzt.

- **Transformationen in Spalten:** Listen erlauben auch komplexere Operationen in Zeilen. Beispiel: Aus einer Spalte mit Koordinaten-Strings `“lat,lon”` eine Liste und dann zwei Spalten extrahieren.

```
let
    Ortstabelle = #table({"Koordinate"}, {{"52.5,13.4"},{"48.1,11.6"}}),
    Aufteilen = Table.AddColumn(Ortstabelle, "Koords", each
Text.Split([Koordinate], ",")),  // Liste aus [lat, lon]
    InSpalten = Table.ExpandListColumn(Aufteilen, "Koords")
in
    InSpalten
```

Anschließend könnte man `Table.SplitColumn` oder weitere Schritte nutzen, um *lat* und *lon* in getrennte Spalten zu überführen.

- **Dynamische Spaltenwahl:** Manchmal soll über Parametern (etwa ein Record mit Spaltennamen) flexibel eine Spalte bearbeitet werden. Beispiel:

```
let
    Tabelle = #table({"x","y","z"}, {{1,2,3},{4,5,6}}),
    config = [FilterSpalte = "x", Grenzwert = 3],
    Gefiltert = Table.SelectRows(Tabelle, each Record.Field(_,
config[FilterSpalte]) > config[Grenzwert])
in
    Gefiltert
```

Hier wird dank des Records `config` festgelegt, dass auf Spalte `"x"` gefiltert werden soll, und mit welchem Wert.

- **Parameter-Objekte (Records):** In Power BI können *Parameterabfragen* genutzt werden. Mit benutzerdefinierten Funktionen nimmt man häufig einen Record als Parameter entgegen, um mehrere Werte zu übergeben. Beispiel:

```
let
    MeineFunktion = (pars as record) =>
        let
            Quelle = Sql.Database(pars[Server], pars[DB]),
            Gefiltert = Table.SelectRows(Quelle, each [Category] =
pars[Category])
        in
            Gefiltert,
```

```
    // Aufruf mit Record-Parameter
    Ergebnis = MeineFunktion([Server="srv1", DB="DatenbankX",
Category="A"])
in
    Ergebnis
```

In diesem Beispiel enthält `pars` Verbindungs- und Filterparameter als Record. Durch Zuweisung via `pars[...]` kann man jeden Parameter flexibel nutzen.

Häufige Fehlerquellen und Best Practices

- **Typen-Fehlanpassungen:** Ein klassischer Fehler ist das Verwechseln von Listen, Records und Tabellen. Etwa führt `each _[Feld]` auf einer Listenvariable zu `Expression.Error: We cannot convert a value of type List to type Record` [12]. Prüfe deshalb genau, welche Datenstruktur eine Funktion erwartet und welche du lieferst. Ein Listenwert kann etwa nicht wie ein Record behandelt werden – und umgekehrt.
- **Nicht vorhandene Felder:** Beim Zugriff auf `record[Feld]` wirft Power Query einen Fehler, falls das Feld fehlt. Besser: Zuerst mit `Record.HasFields` prüfen oder `Record.FieldOrDefault`, um einen Standardwert anzugeben.
- **Dot-Notation-Vorsicht:** Feldnamen mit Leerzeichen oder Sonderzeichen können nicht in der Form `record.Feld` genutzt werden (z.B. `[Name-Feld="x"]`). In solchen Fällen muss man `record["Name-Feld"]` oder `Record.Field` verwenden.
- **Große Listen:** Bei sehr großen Listen können mehrfaches Durchlaufen teuer sein. Wenn eine Liste in mehreren Schritten benötigt wird, kann `List.Buffer` helfen – es speichert die Liste im Speicher, sodass sie nicht jedes Mal neu evaluiert werden muss [7]. Allerdings kann `List.Buffer` die Query Folding-Fähigkeit verhindern und ggf. die Leistung mindern, falls die Quelle das Folding unterstützt. Teste den Performance-Effekt (siehe Chris Webb: *List.Buffer* [7]).
- **Performance-Planung:** Generell gilt: Operations auf *Tabellen* (z.B. `Table.Group`, `Table.Join`) können oft in der Datenbank oder Quelle ausgeführt werden (Query Folding). Listen sind rein in Power Query. Wenn möglich, lieber mit Tabellen arbeiten und nur nach Bedarf zu Listen wechseln.
- **Fehlerbehandlung:** Nutze Funktionen wie `try ... otherwise` oder `Value.Is` / `Value.Type`, um Ausnahmen abzufangen und Typen zu prüfen. So vermeidest du unauffindbare NullReference- oder Typ-Konvertierungsfehler.
- **Best Practices:** Halte Namen konsistent (keine vermischten Fälle), dokumentiere komplizierte Verschachtelungen per Kommentar, verwende **Record-Datentypen** für Parameter von Funktionen (bessere Wartbarkeit) und gruppiere verwandte Funktionstypen (s.o.), um den Überblick zu bewahren.

Zusammenfassung: Listen und Records sind in Power Query flexible Datenstrukturen für vielfältige ETL-Aufgaben. Ihr Verständnis ermöglicht leistungsfähige Datenbearbeitung. Während **Listen** geordnete Reihen von Werten sind und sich gut für Aggregationen und sequentielle Operationen eignen, sind **Records** benannte Feldsammlungen, ideal für Zeilenwerte, Metadaten und Parameter. Mit den oben vorgestellten Funktionen und Techniken lassen sich viele alltägliche Power-Query-Herausforderungen elegant lösen [1] [2] [12].

---

[1] [2] [4] M Language types - PowerQuery M | Microsoft Learn
https://learn.microsoft.com/en-us/powerquery-m/m-spec-types

3 excel - Difference between #table and #list of #records in PowerQuery - Stack Overflow
https://stackoverflow.com/questions/59940580/difference-between-table-and-list-of-records-in-powerquery

5 8 9 10 List functions - PowerQuery M | Microsoft Learn
https://learn.microsoft.com/en-us/powerquery-m/list-functions

6 Metadata - PowerQuery M | Microsoft Learn
https://learn.microsoft.com/en-us/powerquery-m/metadata

7 Chris Webb's BI Blog: Improving Power Query Calculation Performance With List.Buffer()
https://blog.crossjoin.co.uk/2015/05/05/improving-power-query-calculation-performance-with-list-buffer/

11 Record functions - PowerQuery M | Microsoft Learn
https://learn.microsoft.com/en-us/powerquery-m/record-functions

12 [Expression.Error] We cannot convert a value of ty... - Microsoft Fabric Community
https://community.fabric.microsoft.com/t5/Power-Query/Expression-Error-We-cannot-convert-a-value-of-type-List-to-type/td-p/1430544