

Dynamische Schema-Migration von Power BI Semantic Models mit einem Low-Code-Ansatz

INNOVATE. INTEGRATE. EXCEED.



inovex



AGENDA

- Motivation
- Architektur
- Implementierung
 - a. Schema-Sync
 - b. Low-Code Schema Evolution
- Recap





Hallo,

ich bin Marvin Klosek

Data Engineer, Karlsruhe

- Data Warehousing & ELT Pipelines
- Infrastructure Automation

 marvin.klosek@inovex.de

 +49 1522 2778 633

 @Marvin Klosek

 @marvin-klosek

Motivation

Databricks & Power BI

- Zentrales DWH für verschiedene Konsumenten
- DWH ist in Databricks implementiert
- Databricks Delta Lake in Medaillon Architektur
- Verschiedene Quellsysteme Angebunden
 - Oracle DB
 - SAP
 - Excel
- DWH das Gold Layer
- Sternschema
- Unity Catalog
- Workflows über Databricks Asset Bundles orchestriert

Databricks & Power BI

- Power BI ist ein Konsument
- Power BI Setup im Service mit zentralem Semantic Model
- Service kopiert die Daten des Gold layer
- Dazu müssen alle Schemas auch in Power BI selbst vorliegen
- Wie halten wir die Schemas synchron?
- Einfachste Lösung: BI Engineers übernehmen das von Hand
- Skaliert, solange nur die DEs am DWH Schema Änderungen vornehmen
- Redundanter Code

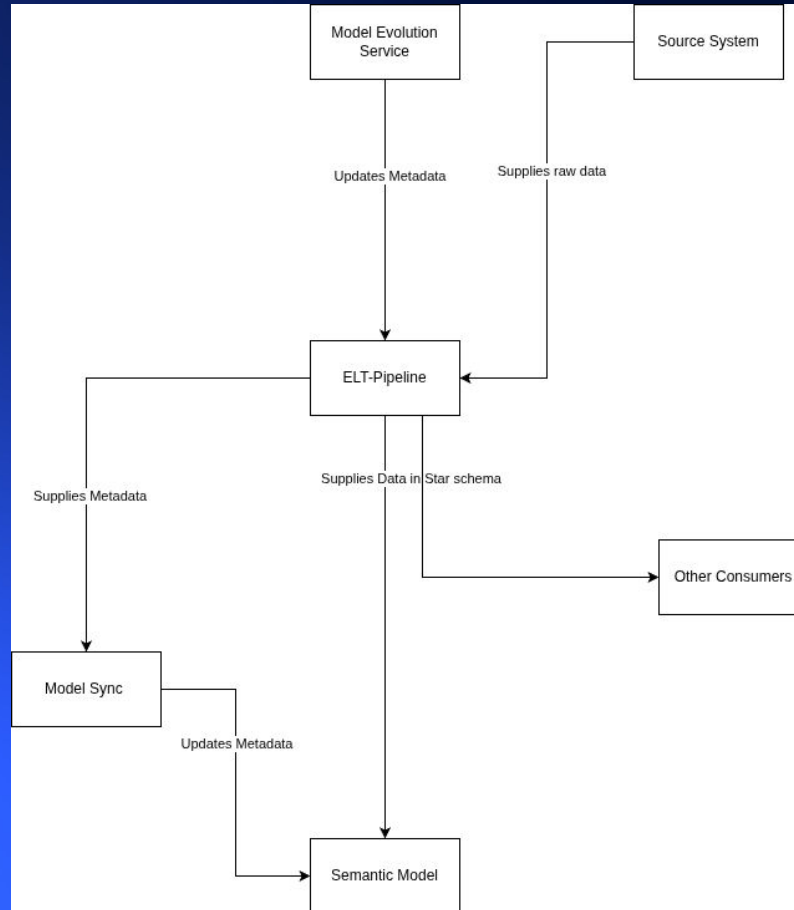
Aber was, wenn wir unsere Nutzer in den Fokus stellen?

- Großteil der neuen Felder, die die User Wünschen, sind simple:
 - Flags
 - Coalesce
- Braucht keinen DE um das zu Implementieren
- Low Code Self Service Lösung um das Schema zu erweitern
- Power BI Measures wären eine Option, wenn da nicht die anderen Konsument
- Und auch zu Komplex für die meisten User
- Muss also direkt über Databricks implementiert werden
- Jetzt skaliert die manuelle Anpassung des Semantic Model nicht mehr
- Fleißige Nutzer könnten Dutzende Felder anlegen
- Wir müssen automatisieren!

Make or buy?

- Gibt keine schlüsselfertige Lösung
- “Publish to Power BI Online from Databricks” keine Option:
 - Erzeugt ein Komplettes Semantic Model
 - Entweder aus einem UC Schema oder einer Menge von Tabellen
 - Keine Versionierung -> Die Arbeit der BI Engineers würde überschrieben!
 - Keine Automatisierung: Reines GUI Feature
- Also müssen wir selber Automatisieren

Architektur



Implementierung

Schema-Sync

Tabellenmodell

- Eigene Implementierung, kapselt Spark/Databricks
- Tabellen sind Instanzen von Klassen
- Durch Vererbung lässt sich Bronze-Silver-Gold abbilden
- Auch Spezielle Klassen pro Quellsystem Möglich
- Tabellen können ihre Metadaten speichern
 - Schema
 - Validierungen
 - Pfade
- Besitzen Funktionen, um ihre unterliegende Implementierung (z. B. Pyspark Dataframe) zu materialisieren
- Dependency Injection versorgt Tasks mit den benötigten Tabellen
- Konkret mit pydantic umgesetzt

Vorteile

- Tabellen sind fest mit ihren Metadaten verbunden
- Wiederkehrende Schemas können über Vererbung oder Factories mit wenig Code abgebildet werden
- Kann mit komplexer Schemavalidierung verbunden werden
- Standard-Tasks können auf Tabellenebenen generisch implementiert werden (z. B. Runden)
- Aus den Schemas können automatisiert Testdaten erzeugt werden
- Vereinfacht das lokale Entwickeln
 - Lokal pyspark
 - In der Cloud Databricks
- Dieses Pattern kann mit jeder beliebiger SQL/DataFrame-Technologie verbunden werden
- Bonus in Databricks: Die Metadaten können direkt in den Unity Catalog überführt werden
- Vorteile gegenüber DBT:
 - Vererbung
 - Testbarkeit
 - Flexibilität python vs sql

Nutzen für den Use Case

- Alle Schemainformationen liegen in einer Single Source of Truth
- Und das als Code
- Muss “nur noch” in das Semantic Model übertragen werden

```
table_silver.example_table = SilverTable(  
    name="example_table",  
    table_schema=Schema(  
        columns=[  
            SchemaColumn.id_column(  
                name="id",  
                reproducible_id="example_key",  
                is_key_column=True,  
            ),  
            SchemaColumn.number_column(  
                name="kpi",  
                fs_type="decimal_number",  
                start=0.0,  
                end=1e12,  
                absence_probability=0.25,  
            ),  
            SchemaColumn.string_column(  
                name="industry",  
                fs_type="occupation",  
                absence_probability=0.85  
            ),  
            SchemaColumn.choice(  
                name="status",  
                spark_type=types.StringType(),  
                items=[  
                    "New",  
                    "Renewed",  
                    "Not active",  
                ],  
                absence_probability=0.1,  
            ),  
            SchemaColumn.currency("currency", absence_probability=0.5),  
        ],  
    ),  
)
```

```
def create_kpi_schema(
    kpi_type: Optional[PRICING_KPI_TYPES] = None, add_gold_layer_metadata: bool = False
) -> list[SchemaColumn]:
    if kpi_type is None:
        kpi_dict = PRICING_KPIS
    else:
        kpi_dict = _create_mapping_for_kpi(kpi_type)
    column_list = []
    for kpi, mapping in kpi_dict.items():
        column = SchemaColumn.number_column(
            name=kpi, fs_type="decimal_number", start=0.0, end=1000.0, absence_probability=0.1
        )
        if add_gold_layer_metadata:
            if kpi_type not in mapping["ratio_types_not_in_source"]:
                column.bms_links = [
                    SourceName(
                        internal_name=f"{mapping['internal_name']}OriginalCurrency",
                        display_name=mapping["display_name"],
                    )
                ]
            column.can_be_compared_to_source = mapping["can_be_compared"]
        else:
            column.can_be_compared_to_source = False
        column_list.append(column)
    return column_list
```



```
logger.info("Checking result table for nullable values in non-nullable columns")
for column in self.table_to_write.table_schema.non_nullable_columns:
    null_data_frame = data_frame.where(data_frame[column.name].isNull())
    assert null_data_frame.isEmpty(), f"Data frame contained null values in column {column.name}"
```

TMDL

- Tabular Model Definition Language
- Neue Sprache, um Semantic Models zu spezifizieren
- An YAML angelehnt
- Lässt sich also mit python parsen und dumpen
- Für C# liefert MS ein SDK zum serialisieren und deserialisieren

```
database Sales
  compatibilityLevel: 1567

model Model
  culture: en-US

table Sales

  partition 'Sales-Partition' = m
    mode: import
    source =
      let
        Source = Sql.Database(Server, Database)
      ...

  measure 'Sales Amount' = SUMX('Sales', 'Sales'[Quantity] * 'Sales'[Net Price])
    formatString: $ #,##0

  column 'Product Key'
    dataType: int64
    isHidden
    sourceColumn: ProductKey
    summarizeBy: None

  column Quantity
    dataType: int64
    isHidden
    sourceColumn: Quantity
    summarizeBy: None

  column 'Net Price'
    dataType: int64
    isHidden
    sourceColumn: "Net Price"
    summarizeBy: none
```

Pydantic <-> TMDL

- Wir spezifizieren ein pydantic Schema für ein Semantic Model in TMDL
- Parsen es
- Jetzt ist das Semantic Model ein python Objekt!
- Wir vergleichen die Tabellen mit unseren Tabellenklassen
- Nehmen die Anpassungen vor
- Dumpen wieder in TMDL

```
class Column(BaseModel):
    dataType: str
    isHidden: Optional[bool] = False
    sourceColumn: str
    summarizeBy: Optional[str] = None

class Measure(BaseModel):
    formula: str
    formatString: str

class Partition(BaseModel):
    name: str
    mode: str
    source: str

class Table(BaseModel):
    name: str
    partitions: Partition
    columns: List[Column]
    measures: Optional[List[Measure]] = []

class Relationship(BaseModel):
    id: str
    fromColumn: str
    toColumn: str

class Role(BaseModel):
    name: str
    modelPermission: str
    tablePermission: str

class Expression(BaseModel):
    name: str
    value: str
    meta: dict

class Database(BaseModel):
    name: str
    compatibilityLevel: int

class Model(BaseModel):
    culture: str

class SemanticModel(BaseModel):
    database: Database
    model: Model
    tables: List[Table]
    relationships: List[Relationship]
    roles: List[Role]
    expressions: List[Expression]
```

Erweiterung der Metadaten

- Einfache Measures können auch in das Tabellenmodel aufgenommen werden
- Wir können dafür Methoden schreiben, die das DAX generieren
- Wir könnten auch komplexes DAX direkt in einem String definieren
- Dadurch sind die Measures auch in der Single Source of Truth für das Datenmodell

Git-Automatisierung via CI/CD

- Wir können eine Pipeline aufsetzen, welche unser Script mit Git commands kombiniert:
 - Die Pipeline checkt unser Data Repo aus auf main
 - Und checkt dann das Power BI repo aus auf dem Working branch
 - Wir parsen das TMDL aus Power BI mit dem Script
 - Comitten dann die Anpassungen
 - Und öffnen einen PR
 - Hier ist die Model Update Pipeline dann zu Ende
 - Human-in-the-Loop Review
 - Deployment nach PR erfolgt dann über die reguläre Power BI CI/CD

Low Code Schema Evolution

Motivation

- Business User möchten das Schema häufig um neue Spalten erweitern
- Häufig coalesce, flags...
- Dafür ist nicht zwingend ein DE erforderlich
- Aber Business User können auch kein SQL oder gar Python
- Mit einer einfachen DSL können die User die Felder selber spezifizieren
- DEs können sich dann auf technische Probleme konzentrieren

DSL & ANTLR

- ANTLR ist ein Framework zur Spezifikation von regulären Grammatiken
- Unterstützt Python, Go, Java, C#, TypeScript...
- Generiert automatisch Parser in den gewählten Sprachen
- Visitors und Listeners können dann definiert werden, um auf basierend auf den Bestandteilen der Grammatik Code auszuführen
- Parser erzeugt einen AST
- Visitor/Listener kann auf einzelne Knoten des AST reagieren
- Z.b. WHERE CLAUSE bei SQL
- Dann kann spezifischer Code ausgeführt werden

Schema Evolution

- Der Parser stellt die grammatikalische Korrektheit der Statements fest und liefert detaillierte Fehler ähnlich einem Editor
- Kann also in ein FE integriert werden
- Web-App mit Monaco Editor
- Dank der Metadaten des Tabellenmodells kann der Editor die User gezielt leiten
 - Autovervollständigung für Tabellen und Spalten
- Geparste Statements werden in DB abgelegt und dann in pyspark gelesen
- Python Listener reagiert dann auf festgelegte Nodes des AST
- Dadurch können wir Tabellen, Felder und Operatoren identifizieren
- Daraus lassen sich dann die notwendigen Spark Operationen konstruieren
- Da die benutzerdefinierten Felder auch Abhängigkeiten untereinander besitzen dürfen, müssen wir für die Auswertung einen DAG konstruieren
- Felder werden dann pro Entität in der Korrekten reihenfolge Evaluert und ins Schema integriert

Next Steps

- Dem User die Möglichkeit geben, neue Entitäten zu definieren
- Den Usern ermöglichen, ihre Felder zu Testen:
 - Aktuell gibt es die Felder doppelt, einmal als “Draft”
 - User können changes von Draft zu Prod propagieren
 - Wäre eleganter, wenn sie direkt beim Editieren auf den generierten Testdaten arbeiten könnten

Recap

- Ein Klassenmodell für unsere Tabellen ermöglicht uns Metadatenverwaltung
- Dieses Klassenmodell kann dann in TMDL geparsed werden
- Mittels CI/CD können wir so unser LakeHouse und unser Semantic Model synchron halten
- Das erlaubt uns rapide Änderungen, ohne die BI-Engineers zu überlasten
- Mittels ANTLR haben wir eine DSL für Schema-Evolution durch unsere User spezifiziert
- Diese können wir in Spark evaluieren
- Durch den Schema-Sync können die benutzerdefinierten Felder automatisch in das Semantic Model aufgenommen werden

WE ARE INOVEX – INNOVATE. INTEGRATE. EXCEED.

Ihr Partner für die digitale Transformation
mit dem Leistungsspektrum

Data & AI

Application Development

Skalierbare IT-Infrastrukturen

Training & Coaching



Ausschließlich festangestellte
Mitarbeitende



Wachstum durch Innovationen
und erfolgreiche Projekte



inovex

inovex.de



SOFTWARE · DATA & AI · INFRASTRUCTURE

Vielen Dank!

Marvin Klossek

+49 1522 2778 633

marvin.klossek@inovex.de



Feedback

