ISR
A CENIT COMPANY

WHITEPAPER

DAS DATA LAKEHOUSE

Ein Anwendungsfall in Databricks und SAP Datasphere

ISR Information Products AG

INHALTSVERZEICHNIS

1 | EINLEITUNG
DAS DATA-LAKEHOUSE-KONZEPT

Hintergrund und Relevanz

Im Zeitalter datengetriebener Geschäftsentscheidungen stehen Unternehmen vor der Herausforderung, große Mengen an Daten effizient zu verwalten und zu analysieren. Das Data-Lakehouse-Konzept verspricht die besten Eigenschaften von Data Lakes und Data Warehouses zu vereinen, um diesen Herausforderungen zu begegnen.

Ein klassisches Data Warehouse bietet hohe Performance und Datenqualität durch die Integration von Speicher- und Rechenressourcen, ist jedoch weniger flexibel bei der Aufnahme semi- und unstrukturierter Daten und verursacht hohe Kosten durch permanente Rechen-ressourcen. Ein Data Lake ermöglicht die kostengünstige, flexible Speicherung großer Datenmengen in Rohform, aber ohne die strukturierte Organisation und hohe Datenqualität eines Data Warehouses. Data Lakes sind daher insbesondere für Anwendungsfälle wie maschinelles Lernen und prädiktive Analysen geeignet.

Das Data-Lakehouse-Konzept vereint die Flexibilität eines Data Lakes mit der strukturierten Organisation und hohen Datenqualität eines Data Warehouses und zielt so darauf ab, datengetriebene Entscheidungsprozesse zu vereinfachen, die Flexibilität zu erhöhen und die Kosten zu senken. Durch die Trennung von Speicher- und Rechenressourcen werden Letztere nur bei Bedarf genutzt. Open-Table-Formate gewährleisten die Datenkonsistenz und -qualität. Funktionen wie ACID-kompatible Transaktionen, Versio-nierung, Schema-Enforcement und -Evolution sowie die Unterstützung für Batch- und Streaming-Verarbeitung bieten eine robuste Datenverwaltung. Die Medaillenstruktur (Bronze, Silver und Gold Layer) sorgt für eine klare Datenorganisation, die den Anforderungen sowohl von Data Scientists als auch von Data-Warehouse-Experten gerecht wird.

Das Data-Lakehouse-Konzept hat in den letzten Jahren erheblich an Popularität gewonnen und wird als zukunftsweisender Ansatz betrachtet, um große und komplexe Datenmengen effizient zu verwalten und zu analysieren. In diesem Whitepaper möchten wir Ihnen anhand eines anschaulichen Anwendungsbeispiels zeigen, wie das Data-Lakehouse-Konzept in Databricks und SAP Datasphere praktisch umgesetzt werden kann.

Zielsetzung und Methodik

Das Ziel dieses Anwendungsfalls ist es, die Daten effizient zu speichern, zu transformieren und zu analysieren. Wir werden die Daten im mehrschichtigen Ansatz basierend auf der Medaillenstruktur (s. Abb. 1) verarbeiten:

- **Bronze Layer:** Speicherung der Rohdaten in ihrer ursprünglichen Form.
- **Staging/Delta Layer:** Konvertierung und erste Transformation der Daten in das Delta-Lake-Format.
- **Silver Layer:** Anwendung der Data Vault 2.0 Modellierung zur weiteren Datenverarbeitung.
- **Gold Layer:** Veredelung der Daten in SAP Datasphere zur Erstellung eines Analytic Models und zur Visualisierung in einem SAP Analytics Cloud Dashboard.

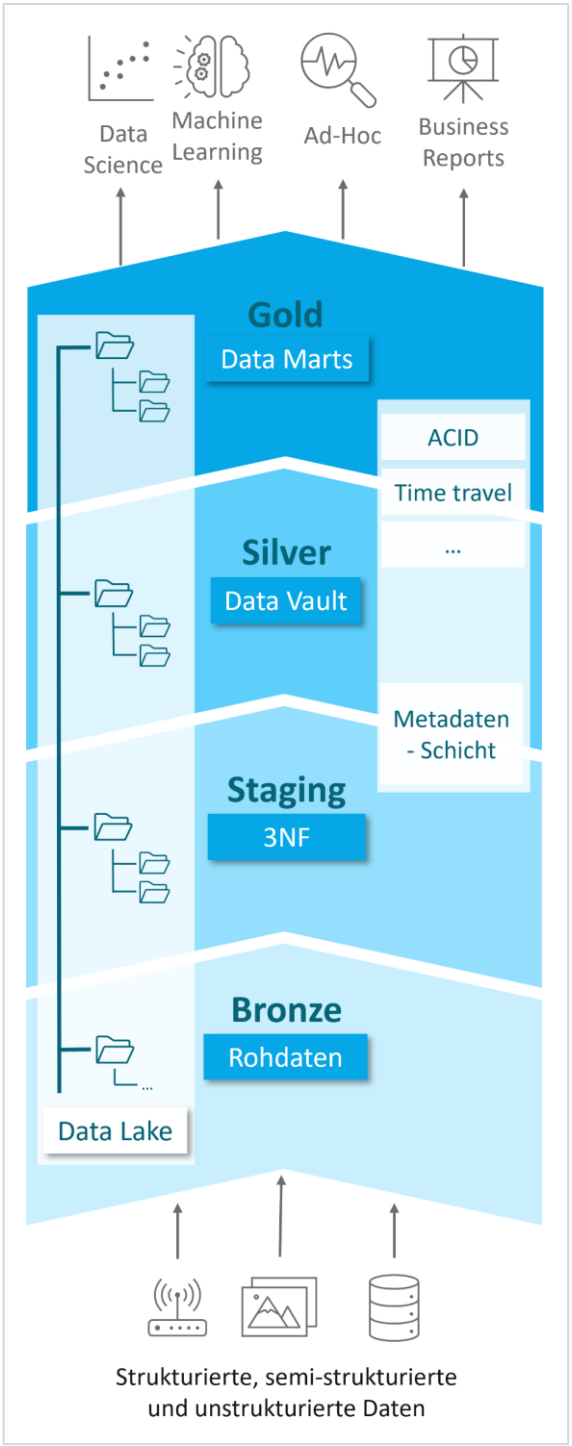


Abbildung 1: Die Medaillenstruktur des Data Lakehouse | ISR

2 | ANWENDUNGSFALL
DAS LAKEHOUSE IN DATABRICKS

Das Data-Lakehouse-Konzept demonstrieren wir anhand eines anschaulichen Anwendungsfalls: Der Verarbeitung und Analyse von US-amerikanischen Flugdaten. Die Daten stammen vom US Bureau of Transportation and Statistics und umfassen sämtliche inländischen Flüge in den USA von 1987 bis zum aktuellen Jahr. Diese umfangreichen Datensätze enthalten Informationen wie Abflug- und Ankunftsflughäfen, Flugzeiten, Verspätungen und deren Gründe. Insofern eigenen sich diese Daten gut zum beispielhaften Aufbau eines Data Lakehouse. Die Daten sind in monatlichen CSV-Dateien organisiert, die jeweils etwa 400.000 bis 600.000 Datensätze und 110 Spalten umfassen. Zusätzlich werden Look-Up-Tabellen (LUT) mit Informationen zu Fluggesell-schaften und Flughäfen verwendet.

Für diesen Anwendungsfall wählen wir das Microsoft-Ökosystem, da es eine abgestimmte Integration mehrerer Komponenten zur Erstellung des Lakehouse bietet. Insbesondere Databricks und das Open-Table-Format Delta Lake, die von den Gründern von Apache Spark entwickelt wurden, bieten eine nahtlose Integration und umfangreiche Erfahrung in der Verarbeitung großer Datenmengen.

2.1 | ZUSAMMENSPIEL ZWISCHEN DATABRICKS UND DATA LAKE

Überblick über Databricks und Delta Lake

Databricks ist ein Tool zur Datenverarbeitung und -analyse, das auf Apache Spark basiert. Als Platform-as-a-Service (PaaS), eingebettet in die Cloud-Strukturen von Hyperscalern wie Microsoft Azure, AWS oder Google Cloud Platform, bietet Databricks leistungsstarke Rechenressourcen für die Transformation, Verwaltung und Analyse von Unternehmensdaten. Durch die strategische Partnerschaft zwischen Databricks und SAP ist es außerdem möglich, die Daten in eine moderne Data-Warehousing-Landschaft aus dem deutschen Geschäftskontext zu integrieren. Delta Lake, ein von Databricks entwickeltes Open-Table-Format, erweitert diese Fähigkeiten um ACID-kompatible Transaktionen, Versionierung und Schema-Management. Diese Eigenschaften machen Delta Lake ideal für die Speicherung und Verarbeitung von Daten im Data Lakehouse.

Spark-Cluster bilden die Grundlage der Rechenressourcen, die Databricks zur Verfügung stellt. Spark-Cluster stellen somit die technischen Möglichkeiten zum Aufbau des Lakehouse zur Verfügung und bieten zahlreiche Vorteile:

- **Performance und Geschwindigkeit**: Im Gegensatz zu anderen Parallel-Processing-Technologien wie Hadoop basiert Spark auf In-Memory-Computing. Die Ergebnisse einzelner Berechnungen werden vollständig im RAM der einzelnen Netzwerkknoten eines Clusters gehalten und können so schneller weiterverarbeitet werden.
- **Flexibilität und Skalierbarkeit**: Spark-Cluster sind in hohem Maß skalierbar und können je nach Bedarf an unterschiedliche Workloads angepasst werden. Es können auf einzelne Beladungsstrecken skalierte Cluster mit unterschiedlich großer Rechenleistung erstellt werden. Die granularen Einstellungsmöglichkeiten ermöglichen somit Kostenvorteile, da nur für in Anspruch genommene Ressourcen gezahlt wird.
- **Unterstützung für verschiedene Datenformate und -quellen:** Spark-Cluster unterstützen eine Vielzahl von Datenformaten und -quellen, was die Integration und Verarbeitung von Daten erleichtert.
- **Erweiterte Funktionalitäten:** Zudem bieten Spark-Cluster Unterstützung für SQL-Abfragen, Machine Learning und den Umgang mit Streaming-Daten. Für Entwickler ist auch die Verfügbarkeit der Versionskontrolle in Databricks interessant. Die Einbindung von externen Git-Repositories oder auch der Azure-internen DevOps-Services-Infrastruktur ermöglicht die bekannte agile Softwareentwicklung in heterogenen Teams, bspw. im Sinne der Scrum- und DevOps-Philosophie.

Integration mit Azure Data Lake

Azure Data Lake ist ein skalierbarer Cloud-Speicher, der speziell für die Speicherung großer Datenmengen entwickelt wurde. Durch die Integration mit Databricks können die Vorteile beider Technologien genutzt werden:

- **Nahtloser Datenzugriff:** Databricks-Cluster können problemlos auf Dateien im Azure Data Lake zugreifen.
- **Speicherung großer Datenmengen:** Azure Data Lake bietet kostengünstige und skalierbare Speichermöglichkeiten für große Datenmengen.
- **Effizientes Datenmanagement:** Die Trennung von Speicher- und Rechenressourcen in Azure Data Lake und Databricks ermöglicht eine effiziente Nutzung der Cloud-Ressourcen.

Unity Catalog für Governance und Verwaltung

Für die Governance und technische Verwaltung der Daten im Data Lake nutzt Databricks das sogenannte Unity-Catalog-Feature (s. Abb.2). Unity Catalog erleichtert die Verwaltung der Daten und sorgt für konsistente Governance und Sicherheitsrichtlinien. Über verschiedene Workspaces (z. B. je einen für Dev, QA und Prod) in derselben Azure-Subscription können in Unity Catalog die Daten und Tabellen im Data Lake in der Medaillenarchitektur verwaltet werden. Unity Catalog bietet eine dreigliedrige Hierarchie zur Organisation und Verwaltung der Daten.

1. **Kataloge**: In der obersten Ebene stehen einzelne Kataloge, die Daten logisch voneinander trennen, etwa für einzelne Fachbereiche, aus einzelnen Quellen etc.

2. **Katalogschemata** (Datenbanken): Sie dienen der weiteren Untergliederung der Tabellen innerhalb der Kataloge.

3. **Tabellen**: Die Tabellen werden im Data Lake an dem Speicherort abgelegt, der dem jeweiligen Katalog und Katalogschema zugeordnet ist.

Für die Verwaltung von Tabellen im Data Lake bietet Unity Catalog zwei Tabellentypen an: externe und managed tables. Letztere werden standardmäßig im Speicherort von Unity Catalog erstellt, wenn man sich innerhalb eines Katalogschemas befindet. Soll die Tabelle gelöscht werden, reicht ein DROP-TABLE-Statement und die Tabelle wird im Databricks-Workspace gelöscht. Gleichzeitig löscht Unity Catalog die dazugehörigen Dateien im Data Lake. Die Tabelle wird also vollständig gelöscht. Externe Tabellen werden zwar im Katalog angezeigt, aber nicht vom Unity Catalog verwaltet, da ihr Speicherort außerhalb eines von Unity Catalog verwalteten Speicherpfads liegt. Wenn ein DROP-TABLE-Statement auf einer externen Tabelle ausgeführt wird, wird zwar die Tabelle aus dem Databricks-Katalog-Explorer gelöscht, die Dateien verbleiben jedoch weiterhin im Data Lake und müssen manuell gelöscht werden, können aber bei Bedarf auch weiterverarbeitet werden. Dies gibt Entwicklern zwar mehr Spielraum bei der Verwaltung von Tabellen, bedeutet aber auch mehr Aufwand. Der Einsatz von externen und managed tables hängt also immer vom Anwendungsfall ab. Auf die Konsumierung durch einen anderen Service außerhalb von Databricks oder des Data Lake hat die Wahl des Tabellentyps keinen Einfluss.

Das folgende Schaubild veranschaulicht das Zusammenspiel zwischen Databricks und Data Lake:

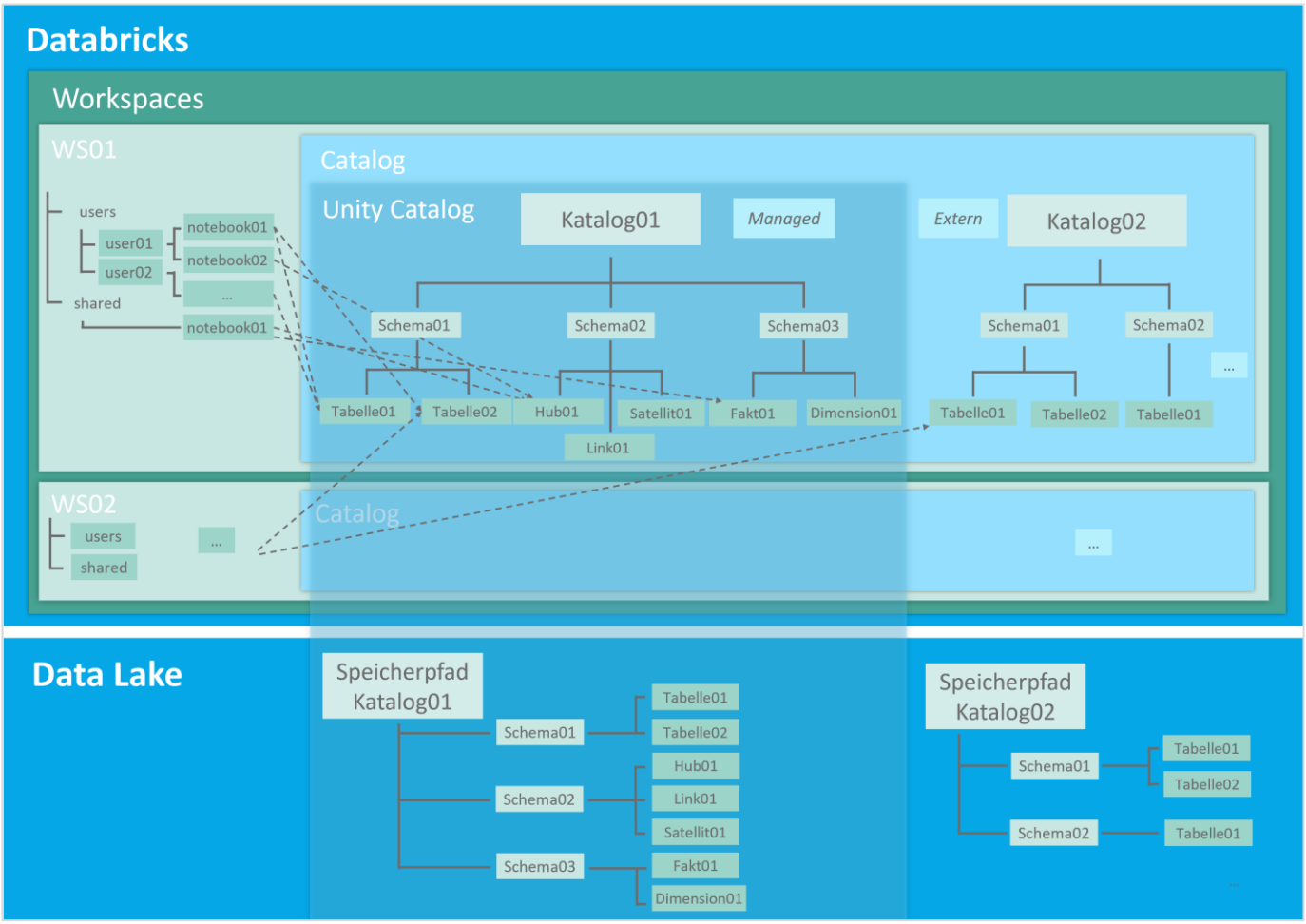


Abbildung 2: Das Management von Daten im Data Lake aus Databricks | ISR

2.2 | Erstellung des Lakehouse

Die Erstellung eines Data Lakehouse für den Anwendungsfall der Flugdaten in den USA erfolgt durch einen mehrschichtigen Ansatz, der die Daten von ihrer Rohform bis hin zu abfrageoptimierten, analytischen Modellen transformiert. Im Folgenden erläutern wir Ihnen die einzelnen Schritte zur Implementierung des Lakehouse mit Databricks und Azure Data Lake anhand der Beispieldaten. Dafür beschreiben wir die Schritte zur Erstellung der Tabellen, die die Schichten des Lakehouse füllen. Um zudem den Unity Catalog in den Anwendungsfall einzubeziehen, erstellen wir pro Schicht in der Medaillenarchitektur ein Katalogschema, dem wir die jeweiligen Tabellen zuordnen (s. Abb. 3). Zum Aufbau und zur Darstellung des Lakehouse wählen wir als Tabellentyp im Folgenden managed tables aus.

Databricks bietet neben external tables auch sog. Delta Live Tables (DLT) an. DLT sind eine Möglichkeit, um Beladungsstrecken in Databricks zu definieren und automatisieren. Sie sind in einer Pipeline miteinander verknüpft und orchestrieren die Transformation von Daten nach vorher bestimmten Regeln. DLT sind sowohl für Batch- als auch für Streaming-Daten geeignet und bieten für letztere insbesondere die Autoloader-Funktion an, mit der ein Speicherort im Data Lake automatisch nach neuen Daten geprüft wird, welche sodann in die DLT-Pipeline geladen werden. Darüber hinaus vereinfacht die Change-Data-Capture-Funktionalität (CDC) von DLT den Umgang mit Spalten, in denen Slowly-Changing Dimensions Typ 1 oder 2 angewendet werden.

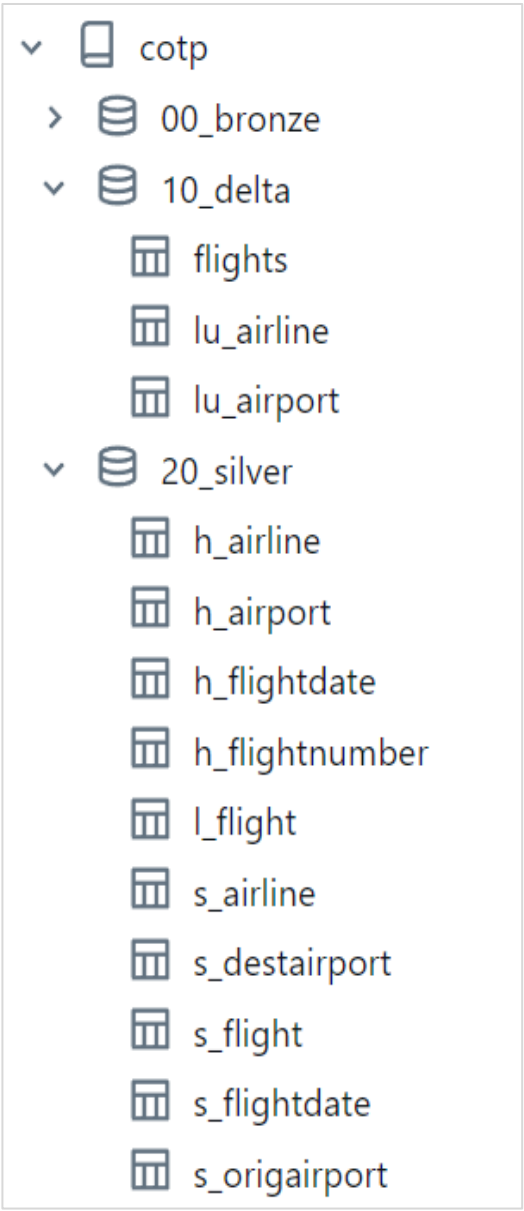


Abbildung 3: Die Struktur des Lakehouse in Databricks | ISR

2.2.1 | Rohdaten im Bronze Layer

ⓘ Der Bronze Layer dient als Landezone für die Rohdaten, die in ihrer ursprünglichen Form gespeichert werden. Diese Schicht ermöglicht eine kostengünstige Speicherung und einfache Replikation der Daten zur Sicherstellung der Datenintegrität und zur Erfüllung von Auditing-Anforderungen.

- **Aufbau der Ordnerstruktur:** Um die Daten für unseren Anwendungsfall in den Azure Data Lake importieren zu können, erstellen wir zunächst die Schichtenstruktur des Lakehouse innerhalb eines neu geschaffenen Containers als Ordnerstruktur. Diese kann über das Azure-Portal, den Azure Storage Explorer oder durch Programmierung im Databricks Notebook und im Azure Command Line Interface (CLI) angelegt werden. Dabei fügen wir neben den drei Schichten der Medaillenarchitektur (Bronze, Silver und Gold) auch ein Staging Layer ein, der in unserem Beispiel als erste Transformationsschicht dient.

- **Import der Rohdaten:** Die Rohdaten laden wir manuell in ihrer Ursprungsform als CSV-Dateien in den Ordner *00_bronze* hoch, welcher die Landezone für sämtliche Daten, die aus Quellsystemen in den Data Lake importiert werden, simuliert. Zusätzlich werden Look-Up-Tabellen (LUT) für Fluggesellschaften und Flughäfen importiert. Die Replikation der Quelldaten bietet sich aufgrund geringer Speicherkosten und zum Zweck des Abgleichs mit transformierten Daten für eine gewisse Zeit an, um das Auditing bei fehlerhaften Beladungsstrecken zu vereinfachen.

2.2.2 | Staging im Delta Layer

ⓘ Im Staging Layer, auch als Delta Layer bezeichnet, werden die Rohdaten erstmals transformiert und in das Delta-Lake-Format überführt. Dieser Schritt erleichtert die nachfolgende Datenverarbeitung und gewährleistet die Datenkonsistenz.

- **Konvertierung der Rohdaten in Delta-Format:** Die Rohdaten werden in das Delta-Lake-Format konvertiert, was effiziente Lese- und Schreiboperationen sowie ACID-kompatible Transaktionen ermöglicht.

- **Erstellung und Beladung von Delta-Lake-Tabellen**: Für die Flugdaten und LUT werden entsprechende Delta-Lake-Tabellen erstellt und beladen. Dabei werden Transformationen durchgeführt, wie das Hinzufügen von Metadaten (z. B. Beladungszeitpunkt) und die Anpassung der Datentypen.

Den Staging Layer nennen wir *10_delta*, da die Daten hier zum ersten Mal in das Delta-Lake-Format überführt werden. Nach der Verbindung↗ des Databricks Workspace mit dem Data Lake erstellen wir für diese Überführung zwei Notebooks.
Das eine (*GENERIERUNG_Delta*) dient der Erstellung der Delta-Lake-Tabellen für die Flugdaten (Tabelle *FLIGHTS*) sowie der LUT für die Fluggesellschaften (Tabelle *LU_AIRLINE*) und Flughäfen (Tabelle *LU_AIRPORT*).

Zunächst definieren wir, in welchem Katalog und in welchem Katalogschema die Tabelle erstellt werden soll:

```
spark.sql("USE CATALOG cotp")
spark.sql("USE 10_delta")
```

Alle infolgedessen erstellten Tabellen werden nun dem Katalogschema *10_delta* zugeordnet, bis in einem erneuten Aufruf des Statements ein anderer Katalog oder Schema referenziert werden. Aufgrund der Tatsache, dass die beiden LUT eine geringe Datenmenge aufweisen, lesen wir die Rohdaten direkt als Spark-Dataframe ein und speichern sie als Tabelle (hier exemplarisch für *LU_AIRLINE*).

```
# Erstellt Referenztabelle LU_AIRLINE von gegebenen Pfad --> beinhaltet alle
korrekten Airline IDs mit Carrier-Kürzel, Beschreibung der Fluggesellschaft und
weiteren Infos.
df_airline = spark.read.format("csv").options(header=True, sep=",",
inferSchema=True).load(csv_airline_path)
df_airline.write.saveAsTable("LU_AIRLINE")
```

Aufgrund der großen Anzahl von CSV-Dateien als Rohdaten für die Tabelle *FLIGHTS* erstellen wir zunächst eine leere Delta-Lake-Tabelle mit dem Schema der Rohdaten und beladen diese erst im Anschluss. Die Tabelle erstellen wir mit einem regulären SQL-CREATE-TABLE-Statement. Dies ist einerseits, wie in diesem Fall, durch einen Wechsel der Programmiersprache innerhalb des Notebooks möglich. Andererseits steht Spark SQL zur Verfügung, wodurch Delta-Lake-Tabellen innerhalb eines Python-Statements mittels SQL erstellt oder abgefragt werden können.

Die Beladung definieren wir anschließend im zweiten Notebook *BELADUNG_Delta*. Normalerweise bietet PySpark die Möglichkeit, mehrere CSV-Dateien, die dasselbe Schema haben, gleichzeitig einzulesen. Allerdings weichen die Schemata der einzelnen CSV-Dateien sowie einige Datentypen leicht voneinander ab, sodass jede Datei in einer Schleife über den Ordnerpfad einzeln eingelesen und an das Zielschema angepasst wird. Beispielsweise ist das Datum eines Flugs als *String* definiert, soll jedoch in diesem Schema den Datentyp *Date* annehmen. Darüber hinaus fügen wir fehlende Spalten hinzu und belassen sie entweder als Null-Werte oder füllen sie, sofern aus anderen Spalten oder den Metadaten der Rohdaten ersichtlich, mit werthaltigen Daten. Außerdem nehmen wir weitere Transformationen vor, indem wir eine Spalte mit dem Beladungszeitpunkt und eine weitere Spalte für die Bestimmung des Quellsystems hinzufügen. Letzteres ist hier zwar für alle Datensätze gleich, aber unser Ziel ist es, die Erstellung der Tabellen für das Data-Vault-2.0-Modell im Silver Layer bereits fachlich korrekt vorzubereiten. Abschließend hängen wir die jeweiligen Rohdaten an die bestehende Delta-Lake-Tabelle an, ähnlich dem vorherigen write-Befehl.

```
# Schreibt den Dataframe als managed Delta-Lake-Tabelle.
df_flights.write.mode("append").options(mergeSchema="True").partitionBy(partiti
on_column).saveAsTable("FLIGHTS")
```

Als Partitionsspalte wählen wir „YEAR“. Mit dem write-mode *append* geben wir an, dass (im Gegensatz zu *overwrite*) die neuen Daten an die bestehende Tabelle unten angefügt werden sollen. Auch ohne Transformationen oder Angleichung der Schemata ist ein Anfügen an die Tabelle durch die Option *mergeSchema* (überwiegend) möglich. Ist dies auf True gesetzt, werden die Schemata der Quell- und Zieltabelle verglichen: Identische Spaltenbezeichnungen werden erkannt und die entsprechenden Daten übernommen, sofern die Datentypen dieselben sind. Andernfalls wird zur Runtime ein Fehler geworfen und die Transaktion vollständig abgebrochen. Existieren Spalten in der Quelltabelle, aber nicht in der Zieltabelle, so werden diese in die Zieltabelle eingefügt und alle bereits bestehenden Daten erhalten dort per default einen Null-Wert. Im umgekehrten Fall, wenn Spalten in der Zieltabelle, aber nicht in der Quelltabelle existieren, erhalten die neu eingefügten Daten ebenfalls Null-Werte. Da in diesem Beispiel einige Datentypen nicht als gleich erkannt werden, sind u.a. die oben beschriebenen Casting-Operationen notwendig.

Diese Vorgehensweise ist zwar initial aufwendiger, bietet uns allerdings auch mehr Möglichkeiten zur individuellen Beladung der Delta-Lake-Tabelle und mehr Kontrolle über die Datenströme. Am Ende des Statements fällt auf, dass der konkrete Name der Delta-Lake-Tabelle angegeben wird (dies ist mit *saveAsTable* möglich) und nicht dessen Speicherort. Stattdessen erstellt Unity Catalog beim ersten Aufruf automatisch einen Pfad unter dem zuvor für den Katalog definierten Speicherort und legt die Tabelle dort ab. Bei den weiteren Iterationen wird dann geprüft und erkannt, dass unter dem angegebenen Pfad eine Delta-Lake-Tabelle existiert, an die die Daten mit den entsprechenden Transformationen angefügt werden können. Anschließend können wir das Ergebnis im Databricks Notebook, im Databricks-Katalog-Explorer sowie im Data Lake betrachten. In Abbildung 4 sehen wir die Beispieldaten der *FLIGHTS*-Tabelle. Neben den ursprünglich vorhandenen Spalten erkennen wir auch die neu eingefügten Spalten *Load_date* und *Source_system.*

Insgesamt verfügt die Tabelle über ca. 210 Mio. Datensätze. Entsprechend der Partitionsspalte *YEAR* wurde die Tabelle aufgeteilt, indem für jedes Jahr ein Ordner angelegt wurde. Innerhalb eines Ordners werden die Daten in komprimierten Parquet-Dateien gespeichert. Im Vergleich zu den Rohdaten wird dadurch der Speicherbedarf erheblich reduziert. Während die gesamten Rohdaten im CSV-Format ca. 71 GB Speicherplatz benötigen, sind es im Delta-Lake-Format nur ca. 7,5 GB. Die Metadaten-Schicht, mit der das Open-Table-Format die Tabelle anreichert, ist anhand des zusätzlichen Ordners *_delta_log* erkennbar, in dem Informationen über die Transaktionen in mehreren JSON-Dateien gespeichert werden.

	Load_date	Source_system	YEAR	QUARTER	MONTH	DAY_OF_MONTH
1	2024-04-10T08:13:17.178+00:...	transtats.bts.gov	1987	4	12	23
2	2024-04-10T08:13:17.178+00:...	transtats.bts.gov	1987	4	12	24
3	2024-04-10T08:13:17.178+00:...	transtats.bts.gov	1987	4	12	25
4	2024-04-10T08:13:17.178+00:...	transtats.bts.gov	1987	4	12	26
5	2024-04-10T08:13:17.178+00:...	transtats.bts.gov	1987	4	12	27

Abbildung 4: Ausschnitt der FLIGHTS-Tabelle in Databricks | ISR

2.2.3 | Data-Vault im Silver Layer

ⓘ Der Silver Layer nutzt die Data-Vault-2.0-Modellierung zur weiteren Strukturierung und Aufbereitung der Daten. Diese Methode ermöglicht eine flexible und skalierbare Datenverarbeitung, die sich an verändernde Geschäftsanforderungen anpassen kann.

Die Data-Vault-2.0-Modellierung wird angewendet, um die Daten in Hubs, Links und Satelliten zu organisieren. Diese Struktur gewährleistet eine hohe Flexibilität und Datenintegrität.

Erstellung von Hubs, Links und Satelliten

Hubs repräsentieren zentrale Entitäten, wie Fluggesellschaften, Flughäfen, Flugdatum und Flugnummer.

Links verbinden die Hubs miteinander und stellen die Beziehungen zwischen den Entitäten dar, wie z. B. die Verknüpfung eines Flugs mit der Fluggesellschaft und den Flughäfen.

Satelliten enthalten zusätzliche Attribute und historische Daten, die den Hubs und Links zugeordnet sind, wie z. B. detaillierte Informationen zu den Fluggesellschaften, Flughäfen und Flugdaten.

Die Erstellung der Data-Vault-Strukturen erfolgt mittels SQL-Statements in Databricks, wobei die Daten aus den Delta-Lake-Tabellen extrahiert und entsprechend den Modellierungskonventionen transformiert werden. Dies ermöglicht eine robuste und skalierbare Datenverarbeitung, die für verschiedene Analysezwecke genutzt werden kann.

Für die Beladung des Silver Layers setzen wir ein weiteres Notebook auf. Die Erstellung der Data-Vault-Tabellen (DV-Tabellen) nehmen wir mittels Extraktion der Daten aus der Delta-Lake-Tabelle *FLIGHTS* vor, d. h. durch Aufteilung und Auswahl der für jeweils benötigten Spalten. Als Namenskonvention wählen wir jeweils den Anfangsbuchstaben der Tabellenentität sowie einen aussagekräftigen Namen, welche mit einem Unterstrich getrennt sind. Folgendes DV-Modell bestimmen wir für den Silver Layer (siehe Tab. 1)

Hubs:	**Link:**
Fluggesellschaft *(H_AIRLINE)* Flughafen *(H_AIRPORT)* Flugdatum *(H_FLIGHTDATE)* Flugnummer *(H_FLIGHTNUMBER)*	Flug *(L_FLIGHT)*
Satelliten: Fluggesellschaft *(S_AIRLINE),* welcher *H_AIRLINE anreichert.* Abflugflughafen *(S_ORIGAIRPORT),* welcher *H_AIRPORT* anreichert. Zielflughafen *(S_DESTAIRPORT),* welcher ebenso *H_AIRPORT* anreichert. Zeitinformationen *(S_FLIGHTDATE),* welcher *H_FLIGHTDATE* anreichert. Fluginformationen *(S_FLIGHT),* welcher *L_FLIGHT* anreichert.	

Tabelle 1: Hubs, Links und Satelliten im Silver Layer | ISR

Daraus ergibt sich folgende Struktur aus Hubs, Links und Satelliten:

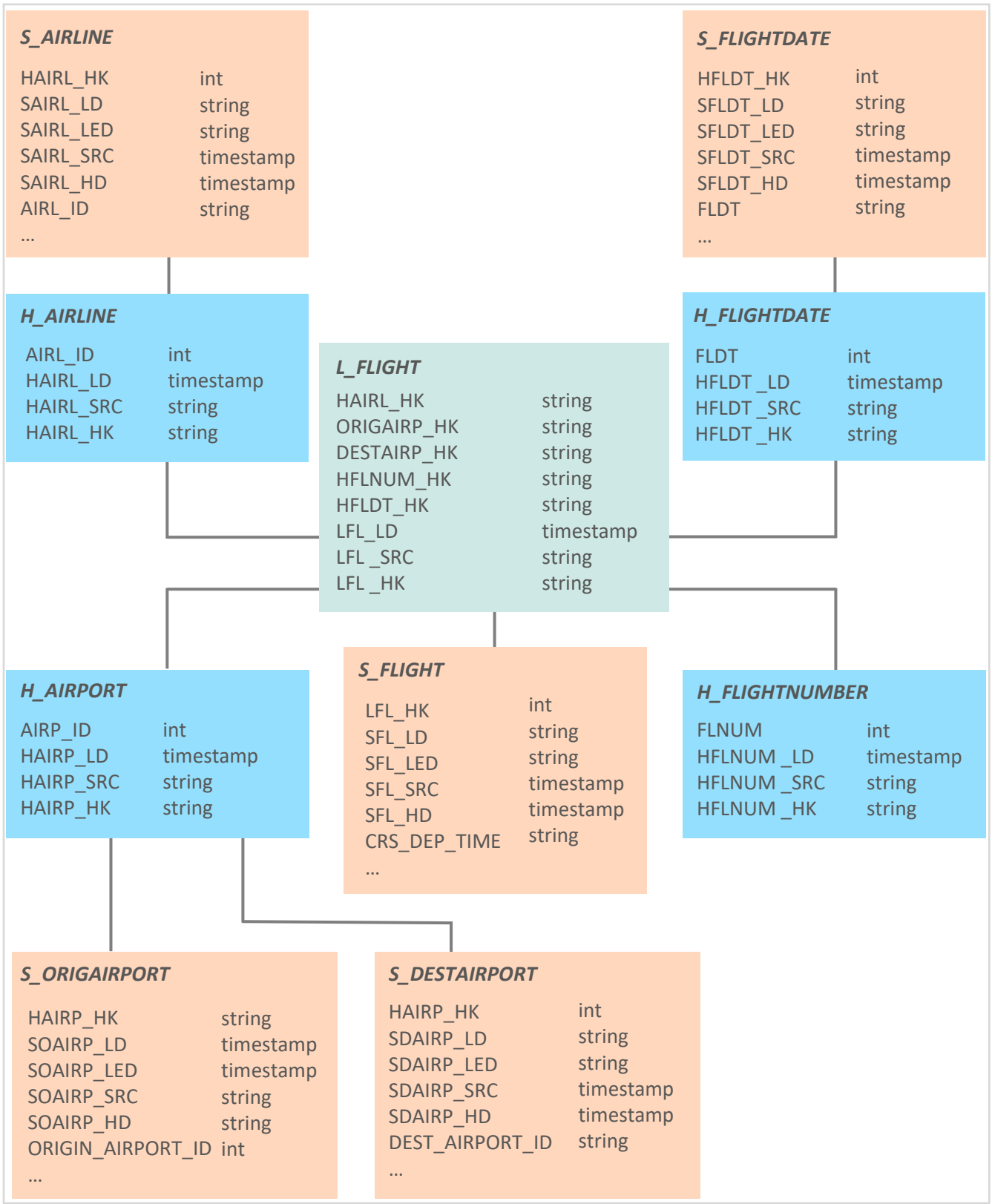


Abbildung 5: Das Data Vault Modell im Silver Layer | ISR

Im Folgenden stellen wir beispielhaft die Erstellung des Hub *H_AIRLINE* mit einem CREATE-TABLE-AS-SELECT-Statement vor. Zunächst legen wir fest, dass die Tabelle in das Katalogschema des Silver Layers geschrieben werden soll. Dafür geben wir vor dem Namen das Katalogschema an, das aufgrund der Namenskonvention von Unity Catalog auch als solches erkannt wird. Anschließend identifizieren wir in der Tabelle *FLIGHTS* die ID-Spalte *OP_CARRIER_AIRLINE_ID* sowie *Source_system* und erweitern den Hub um das Ladedatum. Diese werden selektiert und umbenannt. Außerdem bilden wir aus der ID-Spalte die Hash-Key-Spalte. Die Hashfunktion SHA1 wählen wir als Kompromiss zwischen angemessenem Speicherverbrauch und Hashkollisionswiderstand. Auch für die Spalten wählen wir eine einheitliche Namenskonvention. Die Schlüsselspalte kürzen wir entsprechend dem Tabellennamen (*AIRL_ID*) ab. Jede weitere Spalte setzen wir aus dem Anfangsbuchstaben der Tabellenentität (hier ein „H“), der Abkürzung des Tabellennamens („AIRL“), einem Unterstrich sowie einer sinnvollen Abkürzung der Spaltenbezeichnung zusammen („SRC“ für *Source_system*, „LD“ *für* das Beladungsdatum sowie „HK“ für den Hash-Key). Zusammengesetzt führen wir folgendes Statement aus:

```
%sql
-- Erstellung des Hub H_AIRLINE
USE 10_delta;
CREATE TABLE 20_silver.H_AIRLINE PARTITIONED BY (HAIRL_HK) AS
SELECT
  DISTINCT
    OP_CARRIER_AIRLINE_ID AS AIRL_ID,
    CURRENT_TIMESTAMP AS HAIRL_LD,
    Source_system AS HAIRL_SRC,
    sha(UPPER (TRIM (IFNULL (CAST(OP_CARRIER_AIRLINE_ID AS string), "")))) AS
HAIRL_HK
FROM
  flights
;
```

Obwohl wir uns für das SQL-Statement entschieden haben, ist die Erstellung auch mit Python-Statements, insbesondere mit der PySpark-Bibliothek, unter Berücksichtigung einiger Modifikationen möglich. Da für die Erstellung das Statement in mehrere zerlegt werden muss, wird der Schreibmodus *overwrite* und die Option *overwriteSchema* benötigt, um die Spalten in das Schema des Hubs einzuführen. Mit dem *overwrite*-Modus werden die Daten in der Tabelle neu geschrieben. Diese Operation ist recht teuer und sollte daher so selten wie möglich oder auf möglichst kleinen Tabellen ausgeführt werden. Dies bedeutet jedoch nicht, dass die gesamte Tabelle gelöscht und neu geschrieben wird, sondern lediglich die darin enthaltenen Datensätze. Dies ist u. a. daran erkennbar, dass die Historie der Tabelle erhalten bleibt. Mit dem Befehl

```
%sql
DESCRIBE HISTORY H_AIRLINE
```

können wir Informationen über frühere Tabellen-Snapshots, deren Zeitstempel, die Operationen, die zu dieser Version geführt haben, sowie weitere Informationen abrufen.

Um auch die Funktion der Schemaevolution nutzen zu können, die das Open-Table-Format mit sich bringt, führen wir eine Änderung der Tabelleneigenschaften durch.

```
%sql
ALTER TABLE H_AIRLINE SET TBLPROPERTIES (
  'delta.minReaderVersion' = '2',
  'delta.minWriterVersion' = '5',
  'delta.columnMapping.mode' = 'name'
)
```

Dadurch ist es möglich, nachträglich Änderungen am Tabellenschema vorzunehmen, z.B. eine Spalte umzubenennen, ohne die gesamte Tabelle neu schreiben zu müssen. Dank der Metadaten-Schicht, die das Delta-Lake-Format einführt, kann diese Änderung virtuell gespeichert und abgerufen werden. Für die Erstellung der weiteren Hubs *H_AIRPORT, H_FLIGHTDATE* und *H_FLIGHTNUMBER* wird analog vorgegangen.

Der Link definiert sich als die einzigartige Kombination aus den jeweiligen ID-Spalten der Hubs, wobei zusätzlich Abflug- und Zielflughafen unterschieden werden müssen. Um diese Tabelle zu erstellen, fügen wir die Link-eigene Hash-Key-Spalte sowie Beladungszeitpunkt und Source_system hinzu. Dabei kehren wir zunächst zum Katalogschema *10_delta* zurück, da die benötigten Daten aus der Tabelle *FLIGHTS* stammen.

```
%sql
USE 10_delta;
CREATE TABLE 20_silver.L_FLIGHT PARTITIONED BY (HAIRL_HK) AS
SELECT
  DISTINCT
    sha(UPPER (TRIM (IFNULL (CAST (OP_CARRIER_AIRLINE_ID AS string), "")))) AS
HAIRL_HK,
    sha(UPPER (TRIM (IFNULL (CAST (ORIGIN_AIRPORT_ID AS string), "")))) AS
ORIGAIRP_HK, -- HAIRP_HK
    sha(UPPER (TRIM (IFNULL (CAST (DEST_AIRPORT_ID AS string), "")))) AS
DESTAIRP_HK, -- HAIRP_HK
    sha(UPPER (TRIM (IFNULL (CAST (OP_CARRIER_FL_NUM AS string), "")))) AS
HFLNUM_HK,
    sha(UPPER (TRIM (IFNULL (CAST (FLIGHT_DATE AS string), "")))) AS HFLDT_HK,
    sha(UPPER (TRIM (IFNULL (
      CONCAT(
        CAST(OP_CARRIER_AIRLINE_ID AS string),
        CAST(ORIGIN_AIRPORT_ID AS string),
        CAST(DEST_AIRPORT_ID AS string),
        CAST(OP_CARRIER_FL_NUM AS string),
        CAST(FLIGHT_DATE AS string)
      ), ""
    )))) AS LFL_HK,
    CURRENT_TIMESTAMP AS LFL_LD,
    Source_system AS LFL_SRC
FROM
  FLIGHTS
```

Auch hier ändern wir mit einem ALTER-TABLE-Statement die Tabelleneigenschaften so, dass Schemaevolution möglich ist.

Anschließend erzeugen wir mit entsprechenden SQL-CREATE-AS-SELECT-Befehlen die Satelliten *S_AIRLINE, S_ORIGAIRPORT, S_DESTAIRPORT, S_FLIGHTDATE* und *S_FLIGHT*. Dabei erweitern wir die Satelliten im Sinne der üblichen Data-Vault-Praxis um zusätzliche Spalten für das Ende des Beladungsdatums (Spalte *Sxxx_LED*) sowie um eine Hash-Difference-Spalte. Dadurch müssen lediglich die Hash-Werte verglichen werden, um neue Versionen eines Datensatzes zu identifizieren. Exemplarisch für den Satelliten *S_ORIGAIRPORT* lautet der Befehl daher wie folgt:

```
%sql
USE 10_delta;
CREATE TABLE 20_silver.S_ORIGAIRPORT PARTITIONED BY (HAIRP_HK) AS
SELECT
  DISTINCT
    sha(UPPER (TRIM (IFNULL (
      CAST(ORIGIN_AIRPORT_ID AS string), ""
    )))) AS HAIRP_HK,
    CURRENT_TIMESTAMP AS SOAIRP_LD,
    TO_TIMESTAMP("9999-12-31 23:59:59.999", "yyyy-MM-dd HH:mm:ss.SSS") AS
SOAIRP_LED,
    Source_system AS SOAIRP_SRC,
    sha(UPPER (TRIM (IFNULL (
      CONCAT(
        CAST(ORIGIN_CITY_MARKET_ID AS string),
        ORIGIN,
        ORIGIN_CITY_NAME,
        ORIGIN_STATE_ABR,
        CAST(ORIGIN_STATE_FIPS AS string),
        ORIGIN_STATE_NM,
        CAST(ORIGIN_WAC AS string)
      ), ""
    )))) AS SOAIRP_HD,
    ORIGIN_AIRPORT_ID,
    ORIGIN_CITY_MARKET_ID,
    ORIGIN AS ORIGIN_AIRPORT_CODE,
    ORIGIN_CITY_NAME,
    ORIGIN_STATE_ABR,
    ORIGIN_STATE_FIPS,
    ORIGIN_STATE_NM,
    ORIGIN_WAC
 FROM
  FLIGHTS
```

Für die weiteren Satelliten wird analog vorgegangen. Auf diese Weise füllen wir den Silver Layer, indem wir die Daten aus dem Staging Layer extrahieren. Die Modellierung nach dem Data-Vault-Ansatz ist eine empfohlene Methodik, um den Anforderungen an eine moderne Datenlandschaft gerecht zu werden. Die Flexibilität und die Vorteile des Delta-Lake-Formats können natürlich auch mit weiteren Modellierungskonventionen ausgenutzt werden.

3 | VEREDELUNG IN SAP DATASPHERE
FAKTEN UND DIMENSIONEN IM GOLD LAYER

Der ursprünglichen Idee des Data Lakehouse folgend müsste der Gold Layer schließlich als dritte Schicht in Databricks erstellt und Data Marts für verschiedene Fachbereiche und ihre Reporting- und Analysetools generiert werden. Doch nicht nur im deutschen Unternehmenskontext tritt immer häufiger die Entwicklung auf, dass die Anwender in den Fachbereichen selbst befähigt werden, Datenmodelle zu erstellen und eigene Reports zu generieren, um Entscheidungsprozesse zu beschleunigen. Genau hier setzt SAP Datasphere an. Mit dem Space-Konzept können Fachbereiche individuell die Daten konsumieren, die sie benötigen. Aus diesem Grund empfehlen wir, den Gold Layer in SAP Datasphere zu modellieren, sodass Daten direkt von den Anwendern der Fachbereiche abgerufen werden können.

3.1 | Erstellung der Dimensionen und Fakten

Für die Verbindung von Databricks nach Datasphere bieten dedizierte Cluster einen JDBC-Endpunkt an. Diese Cluster werden in Databricks für SQL-Abfragen, Analysen und Dashboards verwendet. Den JDBC-Endpunkt nutzen wir, um über den JDBC-Adapter des SAP Data Provisioning Agent (DP-Agent) eine Verbindung in Datasphere anzulegen. Das genaue Vorgehen ist hier↗ erklärt. Ist die Verbindung hergestellt, importieren wir die Tabellen aus Databricks als Remote Tables in einen Datasphere-Space. In diesem Anwendungsfall wählen wir für den Import alle Tabellen des Silver Layers sowie die beiden Look-Up-Tabellen aus, die im Delta Layer liegen. Data Marts lassen sich in Datasphere technisch gesehen, als Analytic Models gestalten, welche anschließend auch in SAP Analytics Cloud (SAC) konsumiert werden können. Dafür erstellen wir in Datasphere zunächst Views, welche wir in ihrem semantischen Typ als Dimensionen und Fakten definieren. Für die Erstellung der Views wird jeweils eine grafische View ausgewählt.

Die Kennzahl, die wir in diesem Data Mart berechnen wollen, ist die durchschnittliche Verspätung der US-Inlandsflüge im Jahr 2023 (dem letzten Jahr, für das zu diesem Zeitpunkt vollständige Rohdaten vorliegen). Dazu erstellen wir insgesamt vier Dimensionen und einen Fakt:

Dimensionen:
Fluggesellschaft (V_DIM_AIRLINE)
Abflugsflughafen (V_DIM_ORIGAIRPORT)
Zielflughafen (V_DIM_DESTAIRPORT)
Flugdatum (V_DIM_FLIGHTDATE)

Fakt:
Flüge (V_FACT_FLIGHTS)

Um den Data Mart möglichst modular aufzubauen, modellieren wir die Dimensionen und den Fakt zunächst sehr breit. Dies erklärt einerseits die Namensgebung für den Fakt, der noch nicht direkt auf die Kennzahl bezogen wird. Andererseits wählen wir deshalb noch keine Spalten aus, die spezifisch für die Kennzahl benötigt werden. Die Einschränkung der Spalten erfolgt stattdessen im Analytic Model. Dadurch können wir die Views für verschiedene Kennzahlen in unterschiedlichen Analytic Models wiederverwenden. Zur Erstellung der Dimensionen stehen außer für *V_DIM_FLIGHTDATE* Look-Up-Tabellen zur Verfügung, die wir dafür nutzen, um die bereits existierenden Informationen anzureichern. Da die für den Fakt relevanten Spalten in den Satelliten enthalten sind, joinen wir diese mit der jeweiligen LUT. Für den Join verwenden wir die Spalte, die wir zur Berechnung des Hash-Keys hinzugezogen haben, da die LUT keinen Hash-Key enthält.

Für die Dimension *V_DIM_AIRLINE* wird dieses Vorgehen nun exemplarisch vorgestellt. Nach Auswahl der grafischen View im Data Builder ziehen wir den Satelliten *S_AIRLINE* per Drag & Drop auf die Bearbeitungsfläche. Auf dieselbe Weise ziehen die *LU_AIRLINE* „auf“ den Satelliten, wodurch automatisch ein Join entsteht. Anschließend geben wir an, über welche Spalten die beiden Tabellen gejoined werden sollen. Auch hier genügt ein Drag & Drop der Spalte aus der einen Tabelle „auf“ die entsprechende Spalte aus der anderen Tabelle. Da die resultierende View sowohl doppelte als auch nicht benötigte technische Spalten enthält, schließen wir diese in einer Projektion nach dem Join aus. Darunter fallen u. a. die Spalten zum Beladungsdatum (*SAIRL_LD* und *SAIRL_LED*) oder zur Quelle (*SAIRL_SRC*). Nicht dazu zählt die Hash-Key-Spalte, die für die Assoziation zum Fakt benötigt wird. Die daraus resultierende Dimension *V_DIM_AIRLINE* umfasst schließlich diejenigen Spalten, die semantische Informationen über die Flug-gesellschaften enthalten, z.B. ihren Sitz und ihre Betriebsregion (s. Abb. 6).

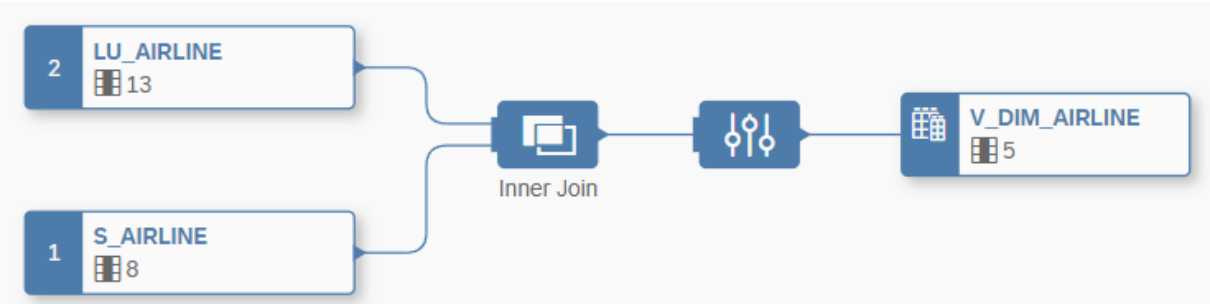


Abbildung 6: Erstellung der Dimension V_DIM_AIRLINE in Datasphere | ISR

Zum Schluss nehmen wir noch einige Einstellungen in den View Properties vor. Zunächst wählen wir den Business Name und Technical Name aus. Diesen setzen wir hier auf *V_DIM_AIRLINE*. Zudem bestimmen wir den Semantic Usage der View. Im Dropdown-Menü wählen wir hier „Dimension“ aus. Nun ist in Datasphere hinterlegt, für welchen semantischen Zweck diese View vorgesehen ist. Wenn zusätzlich erwünscht ist, dass die View im Business Builder der Datasphere oder in SAC konsumiert werden kann, müssen die Schieberegler „Expose for Consumption“ und „Run in Analytical Mode“ eingeschaltet werden (s. Abb. 7). Daraus resultiert schließlich die vollendete *V_DIM_AIRLINE*. Für die Dimensionen *V_DIM_ORIGAIRPORT* und *V_DIM_DESTAIRPORT* gehen wir inkl. Join und Projektion analog vor. *V_DIM_FLIGHTDATE* bringt die Zeitdimension in den Fakt ein. Hier sind bereits alle nötigen Daten im zugrundeliegenden Satelliten enthalten, sodass kein Join notwendig ist. Insofern schließen wir erneut technische Spalten in einer Projektion aus, sodass die resultierende View die semantischen Informationen beinhaltet.

Der Fakt *V_FACT_FLIGHTS* soll schließlich die Spalten zusammenstellen, aus denen die Kennzahlen errechnet werden. Die Daten dafür stellt der Satellit *S_FLIGHT* bereit. Zunächst öffnen wir wieder eine grafische View, in der vergleichbar zur Vorgehensweise bei den Dimensionen, hier *S_FLIGHT* mit den Link *L_FLIGHT* über die Hash-Key-Spalte gejoined werden. Technische Spalten schließen wir anschließend aus.

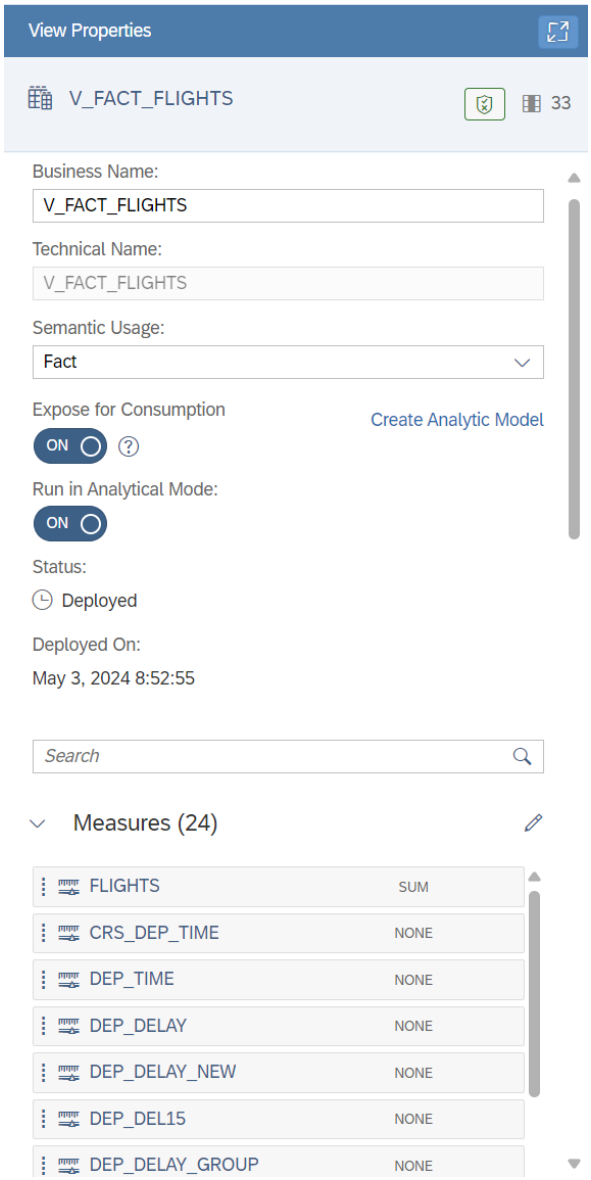


Abbildung 7: View Properties des Fakts V_FACT_FLIGHTS in Datasphere.| ISR

Außerdem berechnen wir eine neue Spalte „ARR_DEL_YN“ (Calculated Column), die uns anzeigt, ob eine Verspätung vorlag oder nicht. Dafür nutzen wir eine Fallunterscheidung auf Basis des Measures „ARRIVAL_DELAY“. Ist dort ein Wert <= 0, so erhält die neue Spalte ebenso den Wert 0. Ist eine Verspätung vorhanden, so soll eine 1 eingetragen werden. In den View Properties wählen wir schließlich nach der Namensgebung die Option „Fact“ als Semantic Usage, um klarzustellen, dass diese View im Zentrum des Data Marts stehen soll. Dies hat auch zur Folge, dass Spalten, die für die Kennzahlenberechnung genutzt werden, nun als Measures bestimmt werden können. Dafür ziehen wir diejenigen Spalten per Drag & Drop aus der Attributsliste nach oben unter „Measures“, die sich für Berechnungen eignen. Aufgrund des modularen Aufbaus des Fakts werden alle Spalten, die einen numerischen Datentyp aufweisen, als Measures definiert. Dadurch ist es möglich, aus der Auswahl der Measures in verschiedenen Analytic Models diejenigen auszuwählen, die für die Berechnung der jeweiligen Kennzahlen benötigt werden. Dafür ändern wir den Aggregationstyp der benötigten Measures. Die neu berechnete Spalte erhält u. a. den Aggregationstyp „SUM“. Außerdem müssen erneut die beiden vorher genannten Schieberegler aktiviert werden. Erst dann kann ein Analytic Model erstellt werden.

3.2 | Aufbau der Analytic Models und Dashboards

Zuletzt verknüpfen wir den Fakt mit den Dimensionen, um das Star-Schema zu vervollständigen. Dazu legen wir in den Properties für jede der vier Dimensionen eine Assoziation an. Darüber hinaus wird die Assoziation automatisch auch visuell im Analytic Model dargestellt, sodass das Star-Schema sichtbar wird. In den angelegten Assoziationen werden die identischen Spalten der beiden Tabellen automatisch verbunden. Dies überprüfen wir und verbinden die entsprechenden Hash-Keys ggf. manuell miteinander. Der resultierende Fakt hat schließlich 24 Measures, neun weitere Spalten (Attribute) und vier Assoziationen.

Zum Schluss erstellen wir das Analytic Model über die entsprechende Auswahlfläche im Data Builder. Zunächst wollen wir sämtliche Assoziationen sowie Measures und Attribute in das Modell einbeziehen. Anschließend werden bereits die verknüpften Dimensionen visualisiert. Von den Measures wählen wir dann diejenigen ab, die für die Berechnung nicht benötigt werden. Wir belassen die Measures im Modell, die sich auf die Ankunftsverspätung (ARRIVAL DELAY) beziehen. Soll ein Measure doch wieder einbezogen werden, kann nach Auswahl des Fakts ein Haken bei der entsprechenden Spalte gesetzt werden. Darüber hinaus ändern wir in diesem Zuge weitere Aggregationstypen. Um die durchschnittliche Verspätung zu berechnen, erhalten die Spalten zur Ankunftsverspätung in der Standardaggregation den Typ „AVG“ und in der Ausnahmeaggregation den Typ „AVERAGENULLZERO“. Letzterer berechnet den Durchschnitt ohne Einbeziehung von NULL-Werten oder Nullen. Das neu berechnete Measure erhält so zusätzlich „SUM“ als Ausnahmeaggregation.

Da wir die Kennzahlen für die verschiedenen Fluggesellschaften berechnen wollen, bestimmen wir die Spalte *UNIQUE_CARRIER* als Aggregationsdimension für die zuvor angepassten Measures. Unter „Dimensions“ sind anschließend alle aufgrund der Assoziationen verknüpften Spalten aufgeführt. Auch diese können beliebig abgewählt werden.

Zuletzt filtern wir unter „Filter“ nach dem Jahr 2023, sodass ein Großteil der Datensätze ausgeschlossen wird. Diese Spalten und Zeilen abzuwählen, kommt zudem der Performance zugute, da sie das Modell verkleinern und einen geringeren Datendurchsatz erfordern. Das vollendete Modell (s. Abb. 8) kann anschließend in SAC weiterverwendet werden.

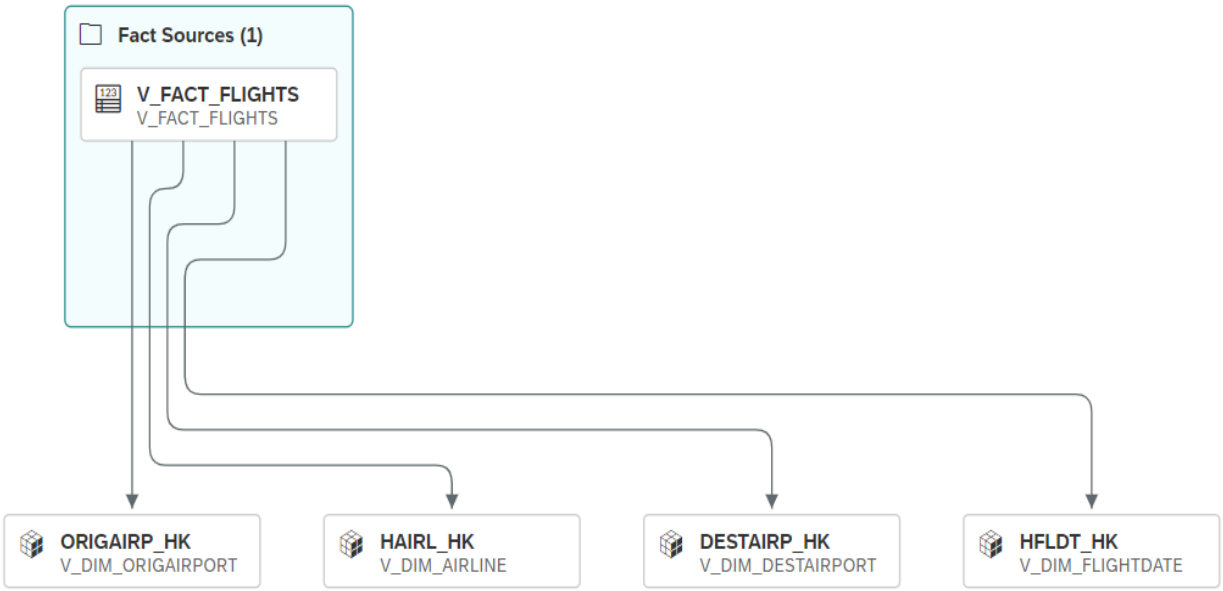
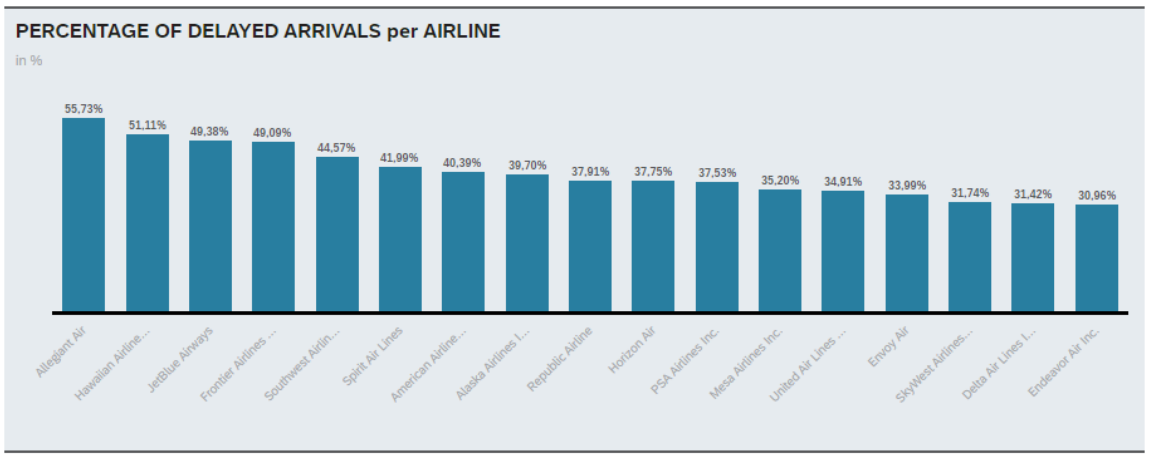


Abbildung 8: Analytic Model im SAP Datasphere - Data Builder | ISR

In SAC legen wir eine Verbindung zu Datasphere an und erstellen eine Story, in die wir das bestehende Modell direkt importieren. Zum einen fügen wir dort eine Tabelle zum Vergleich der durchschnittlichen Verspätung der Fluggesellschaften ein. Zum anderen lassen wir uns die Anzahl und den Prozentsatz der verspäteten Flüge pro Fluggesellschaft anzeigen. Nach abschließender Bearbeitung der Story in SAP Analytics Cloud erhalten wir somit ein kleines Dashboard für unseren Anwendungsfall (s. Abb. 9).

AVERAGE ARRIVAL DELAY per AIRLINE

	AVG DELAY
Horizon Air	8,66
Alaska Airlines Inc.	10,14
Hawaiian Airlines Inc.	10,39
Envoy Air	10,43
Delta Air Lines Inc.	11,72
Endeavor Air Inc.	13,17
Southwest Airlines Co.	13,31
SkyWest Airlines Inc.	13,72
United Air Lines Inc.	13,87
Republic Airline	14,20
PSA Airlines Inc.	15,19
Spirit Air Lines	16,68
American Airlines Inc.	16,96
Mesa Airlines Inc.	17,84
Frontier Airlines Inc.	22,57
Allegiant Air	23,12
JetBlue Airways	25,10



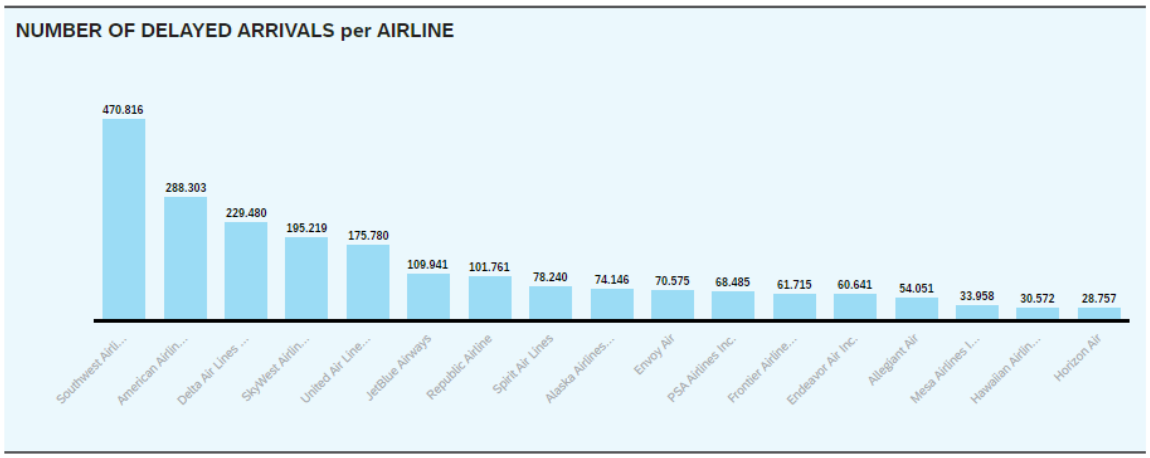


Abbildung 9: Story in SAP Analytics Cloud über verspätete Flüge im Jahr 2023 | ISR

4 | FAZIT & AUSBLICK

In dem vorliegenden Whitepaper haben wir Ihnen das Data-Lakehouse-Konzept anhand eines praktischen Anwendungsfalls detailliert erläutert und dessen Implementierung mithilfe des Azure Data Lake, Databricks und SAP Datasphere aufgezeigt.

Ein Data Lakehouse als strukturgebende Architektur für Unternehmensdaten ist ein wichtiger Bestandteil vieler Datenlandschaften. Open-Table-Formate erlauben dabei sowohl die Flexibilität in der Datenverwaltung und -verarbeitung als auch eine hohe Datenqualität und Governance. Doch es ist wichtig, diese Daten nicht nur „nebenbei“ mitzuziehen, sondern aktiv in moderne Prozesse der Datenkonsumierung einzubinden. Diese zeichnen sich insbesondere durch erhöhte Data Literacy und Self-Service in Fachbereichen aus, welche häufiger eigenhändig auf relevante Daten zugreifen, und Modelle und Analysen erstellen.

Der zweigeteilte Aufbau, den wir Ihnen im Rahmen dieses Whitepapers vorgestellt haben, bietet sich an, um von den Chancen des Data-Lakehouse-Ansatzes zu profitieren und gleichzeitig den Anforderungen an Data Literacy und Self-Service gerecht zu werden: Während die Datenaufbereitung in Databricks erfolgt, ermöglicht die Vollendung in SAP Datasphere, die Daten in eine moderne Cloud-basierte Data-Warehousing-Landschaft einzubinden. Auch wenn Databricks Analyse- und Reporting-Funktionalitäten mit sich bringt, liegt die finale Einbindung der Daten in eine historisch auf SAP-Data-Warehousing-Lösungen gewachsene Systemlandschaft näher am Geschäftskontext vieler deutscher Unternehmen. Die Nutzung von SAP Datasphere für diesen Zweck fördert den Self-Service-Ansatz und trägt so zu schnelleren und fundierten Entscheidungsprozessen bei. Durch die Verknüpfung mit weiteren Unternehmensdaten in den Spaces können die Fachbereiche mit den Daten arbeiten, für die sie berechtigt sind. Somit wird auch dem Prinzip der granularen Zugriffsbeschränkung zur Erhöhung der Datensicherheit Rechnung getragen. Der Aufbau des Gold Layers in SAP Datasphere ist daher eine geeignete Möglichkeit, den Lakehouse-Ansatz zu vollenden.

Die genaue Struktur des Lakehouse in Databricks und Datasphere ist jedoch unabhängig von der Medaillenarchitektur flexibel und kann an Ihre individuellen Bedürfnisse angepasst werden. Damit bietet das Lakehouse die Möglichkeit der strategischen Weiterentwicklung und der Integration modernster Technologien in Ihre Datenlandschaft.

Für dieses Whitepaper wurde auf die folgenden Quellen zurückgegriffen:

Armbrust, M., Ghodsi, A., Xin, R., & Zaharia, M. (2021). *Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics.* Verfügbar 30.04.2024 unter https://www.databricks.com/research/lakehouse-a-new-generation-of-open-platforms-that-unify-data-warehousing-and-advanced-analytics

Haelen, B., & Davis, D. (2023). *Delta Lake Up & Running. Modern Data Lakehouse Architectures with Delta Lake.* O’Reilly Media. Verfügbar 02.05.2024 unter https://www.databricks.com/resources/ebook/delta-lake-running-oreilly

ÜBER ISR

Die ISR Information Products AG agiert seit 1993 als Experte für Data Analytics und Dokumentenlogistik und fokussiert sich somit auf das Datenmanagement und die Automatisierung von Prozessen.

Ganzheitlich begleitet ISR Konzerne und Mittelständler von der strategischen IT-Beratung über konkrete Implementierungen und Lösungen bis hin zum IT-Betrieb.

Unser Ziel ist die Unterstützung unserer Kund:innen bei der Bereitstellung und der wirtschaftlichen Nutzung ihrer Daten im Rahmen eines umfassenden Enterprise Information Managements (EIM).

Als Teil der CENIT EIM-Gruppe und verbunden mit starken Technologiepartnern wie IBM, Microsoft und SAP machen wir unsere Geschäftspartner fit für morgen. Daneben bietet ISR auch eigene Softwarelösungen für Dokumentenlogistik, so z.B. die digitale Aktenlösungen oder eine SaaS-Lösung für Intelligent Document Processing.

Über 260 fokussierte IT-Spezialist:innen im Analytics- und Dokumentenlogistik-Umfeld arbeiten in sechs ISR-Geschäftsstellen in Deutschland.

ISR
A CENIT COMPANY

KONTAKTIEREN SIE UNS!

Ihr Ansprechpartner

Stefan Kahle

Senior Executive Manager | SAP Information Management
stefan.kahle@isr.de

ISR INFORMATION PRODUCTS AG

Hintern Brüdern 23
38100 Braunschweig
Tel. +49 (0) 531 12 08-0
hello@isr.de

Vorstand: André Vogt (Sprecher) | Bernd Rosemeyer | Manfred Merßmann
Aufsichtsratsvorsitzender: Peter Schneck
Sitz der Gesellschaft: Braunschweig

Braunschweig | Münster | Hamburg | Köln | München | Frankfurt a. M.