Think BIGGER

DEEP DIVE FÜR FACHLEUTE

Spark SQL

Verteilte SQL-Verarbeitung, Lakehouse-Praxis, Performance und Betrieb

Autor	Erhard Rainer
Veröffentlicht	28.04.2026
Aktualisiert	28.04.2026
Umfang	ca. 37 Min. Lesezeit
Online	www.thinkbigger.at/wissen/spark-sql/

thinkbigger.at

Business Intelligence · Machine Learning · KI-Agenten · Schulungen

Erhard Rainer · Schönaugasse 106 · 8020 Graz · info@think-bigger.at

Inhalt

0. Überblick (für Eilige) 2
1. Einordnung: Wann Spark SQL die richtige Wahl ist 5
2. Architektur und Kernkonzepte 13
3. Datenmodellierung und Lakehouse-Design 20
4. Spark-SQL-Praxis: Syntax, Patterns und Grenzen 23
5. Performance, Skalierung und Kostenkontrolle 31
6. Betrieb, Monitoring und Troubleshooting 36
7. Sicherheit, Governance und Compliance 38
8. Weiterführende Links und Uebungsaufgaben 40

0. Überblick (für Eilige)

Spark SQL ist das Modul von Apache Spark für strukturierte Datenverarbeitung mit SQL, DataFrames und Datasets. Es verbindet die vertraute Ausdruckskraft von SQL mit der verteilten Ausführung von Spark: Daten werden nicht in einer einzelnen Datenbankinstanz verarbeitet, sondern als Jobs über Driver, Executor, Tasks, Speicher und Cluster-Ressourcen verteilt.

Diese Seite ist eine praxisorientierte Referenz für Spark SQL in Data-Engineering- und Lakehouse-Szenarien. Im Mittelpunkt stehen nicht nur Syntax und einzelne Funktionen, sondern die Frage, wie SQL-Abfragen in Spark geplant, verteilt, ausgeführt, optimiert, beobachtet und sicher betrieben werden.

Wichtig für die Einordnung: Spark SQL ist **kein einzelnes Produkt** wie Microsoft Fabric, Oracle Database oder SQL Server. Es ist eine Komponente von Apache Spark und taucht deshalb in vielen Produkten und Plattformen auf, etwa in Microsoft Fabric Lakehouses, Azure Synapse Spark Pools, Databricks, AWS Glue, Amazon EMR, Google Dataproc, Cloudera-Umgebungen oder selbst betriebenen Spark-Clustern. Spark SQL kann Daten aus SQL Server, Oracle, PostgreSQL und anderen Datenbanken über JDBC lesen oder dorthin schreiben, ersetzt diese Systeme aber nicht als OLTP-Datenbank.

> Hinweis zum Versionsstand: Diese Referenz ist fachlich auf Stand **28. April 2026**. Die offizielle Apache-Spark-Dokumentationsseite führte zu diesem Zeitpunkt **Spark 4.1.1** als stabile Dokumentationsversion und zusätzlich Preview-Dokumentation für **Spark 4.2.0**. Versionssensitive Aussagen sind deshalb bewusst allgemein gehalten oder als Spark-4.x-Kontext markiert.

0.1. Inhaltsverzeichnis (kurz)

- 0. Überblick (für Eilige)
- 1. Einordnung: Wann Spark SQL die richtige Wahl ist
- 2. Architektur und Kernkonzepte
- 3. Datenmodellierung und Lakehouse-Design
- 4. Spark-SQL-Praxis: Syntax, Patterns und Grenzen
- 5. Performance, Skalierung und Kostenkontrolle
- 6. Betrieb, Monitoring und Troubleshooting
- 7. Sicherheit, Governance und Compliance
- 8. Weiterführende Links und Uebungsaufgaben

0.2. Quick-Glossar

- **SparkSession**: Einstiegspunkt für Spark SQL, DataFrames, Catalog-Zugriffe und Konfiguration.
- **DataFrame**: Verteilte, tabellarische Datenstruktur mit Schema; kann per API oder SQL verarbeitet werden.
- **Dataset**: Typisierte Variante in Scala/Java; in PySpark spielt praktisch der DataFrame die Hauptrolle.
- **Catalyst**: Optimizer von Spark SQL, der logische und physische Plaene erzeugt und transformiert.
- **Tungsten**: Ausfuehrungs- und Speicheroptimierungen für effiziente Verarbeitung auf JVM- und CPU-Ebene.
- **Driver**: Prozess, der Anwendung, SparkContext, Planer und Job-Koordination trägt.
- **Executor**: Worker-Prozess, der Tasks ausführt und Daten zwischenspeichern kann.
- **Task**: Kleinste Ausfuehrungseinheit innerhalb einer Stage.

- **Stage**: Gruppe von Tasks ohne Shuffle-Grenze.
- **Shuffle**: Umverteilung von Daten über das Netzwerk, typisch bei Joins, Aggregationen und Repartitionierung.
- **Partition**: Logische Dateneinheit für parallele Verarbeitung; nicht identisch mit Tabellenpartitionen im Storage.
- **AQE**: Adaptive Query Execution; Spark kann Plaene während der Laufzeit anhand echter Statistikdaten anpassen.
- **Broadcast Join**: Join-Strategie, bei der eine kleine Tabelle an alle Executor verteilt wird.
- **Metastore/Catalog**: Verwaltung von Tabellen, Schemas, Speicherorten und Metadaten.
- **Parquet/ORC/Avro/JSON/CSV**: Datenformate, die Spark SQL lesen und schreiben kann.

0.3. Lernpfad: Vom Einstieg zur Produktion

1. Verstehe zuerst, dass Spark SQL kein klassischer Datenbankserver ist, sondern ein verteiltes Rechenmodell.
2. Lerne danach DataFrames, Temp Views, permanente Tabellen, Catalog und Dateiformate sauber zu unterscheiden.
3. Analysiere Abfragen über `EXPLAIN`, Spark UI, SQL UI, Stages, Tasks und Shuffle.
4. Optimiere erst Datenlayout, Joins, Partitionierung und Statistiken, bevor du wahllos Cluster-Groessen erhoehst.
5. Verankere Betrieb, Security, Kostenkontrolle und Governance früh, weil Spark-Jobs sonst schnell teuer und unuebersichtlich werden.

0.4. Welche Kapitel brauche ich?

Der Artikel ist als Referenz aufgebaut. Je nach Rolle lohnt sich ein anderer Pfad:

Rolle oder Frage	Sinnvolle Kapitel
Ich kenne Spark SQL aus Microsoft Fabric und will es einordnen	0, 1.8 bis 1.12, 2.1, 3.2, 4.10
Ich will wissen, ob Spark SQL Oracle oder SQL Server ersetzt	1.1 bis 1.6, 1.9 bis 1.11
Ich baue Lakehouse-Transformationen	2, 3, 4, 5
Ich muss Performanceprobleme analysieren	2.3 bis 2.5, 5, 6
Ich verantwortete Plattform, Betrieb oder Governance	1, 6, 7, 8.4
Ich brauche nur eine schnelle technische Orientierung	0.1, 0.2, 0.5, 1.6, 1.11

0.5. Mini-Projekt (durchgehend): Lakehouse für Verkaufs- und Nutzungsdaten

Dieses Mini-Projekt begleitet alle Kapitel.

Szenario:

- Ein Unternehmen sammelt Webshop-Bestellungen, Produktstammdaten, Kampagnendaten und Nutzungsereignisse.
- Rohdaten landen als Dateien in einer Bronze-Zone.
- Spark SQL transformiert sie in bereinigte Silver-Tabellen und fachliche Gold-Modelle.
- Fachbereiche greifen über BI, Notebooks oder SQL-Endpunkte auf aggregierte Daten zu.
- Die Plattform muss skalierbar, nachvollziehbar, kosteneffizient und auditierbar bleiben.

Wir verwenden dieses Szenario für:

- Tabellen- und Zonenmodellierung.
- Partitionierung und Dateigroessen.
- Joins zwischen Events, Orders und Stammdaten.
- Inkrementelle Verarbeitung.
- Performance- und Betriebsentscheidungen.

0.6. Profi-Heuristiken

- Spark SQL ist stark für grosse, verteilte Datenmengen, aber oft ueberdimensioniert für kleine OLTP-nahe Abfragen.
- Der teuerste Teil einer Spark-SQL-Abfrage ist häufig nicht die SQL-Syntax, sondern Shuffle, Datenlayout oder zu viele kleine Dateien.
- DataFrame-API und SQL sind zwei Oberflächen auf demselben Optimizer-Modell.
- Ein `SELECT *` auf riesigen Rohdaten ist kein harmloser Blick, sondern ein Cluster-Job.
- Gute Spark-Performance beginnt beim Storage: Spaltenformat, Kompression, Partitionierung, Dateigroesse und Statistiken.
- AQE hilft, ersetzt aber keine saubere Modellierung.
- Broadcast Joins sind sehr stark, solange die "kleine" Tabelle wirklich klein bleibt.
- Caching ist ein Werkzeug für wiederholte Zugriffe, kein generelles Performance-Allheilmittel.
- Spark SQL braucht Betrieb: Event Logs, History Server, Kostenmetriken, Runbooks und klare Owner.

0.7. So setzt du es um - Checkliste

- Definiere pro Tabelle Zone, Zweck, Owner, Aktualisierungsrhythmus und Aufbewahrung.
- Halte fest, welche Queries kritisch sind und welche nur explorativ laufen.
- Prüfe für grosse Tabellen Dateiformat, Partitionierung, Dateigroessen und Statistiken.
- Nutze `EXPLAIN`, Spark UI und SQL UI regelmäßig, nicht erst im Incident.
- Vermeide breite Shuffles, unkontrollierte Cross Joins und unnötige Repartitionierung.
- Dokumentiere Security, Zugriffspfade und Datenklassifikation pro Datenzone.

0.8. Mentales Modell für den Alltag

Spark SQL sollte man nicht als "Datenbank mit anderem Logo" verstehen. Eine klassische Datenbank hält Daten, Optimizer, Transaktionslogik und Ausführung eng in einem System zusammen. Spark SQL ist dagegen ein verteilter Ausfuehrungsrahmen, der Daten aus Dateien, Tabellen, Katalogen und externen Quellen liest, sie in Jobs zerlegt und über Cluster-Ressourcen verarbeitet.

Diese Trennung ist machtvoll, aber sie verschiebt Verantwortung. In Spark SQL entscheidet nicht nur die Abfrage über Geschwindigkeit, sondern auch das Layout der Dateien, die Größe der Partitionen, die Guete der Statistiken, die Join-Strategie, die Cluster-Konfiguration und die Frage, ob Daten lokal, remote, komprimiert, sortiert oder fragmentiert vorliegen. Wer Spark SQL produktiv betreibt, muss deshalb SQL, Storage und Laufzeitverhalten gemeinsam lesen.

---

1. Einordnung: Wann Spark SQL die richtige Wahl ist

1.1. Problemklassen, in denen Spark SQL typischerweise stark ist

Spark SQL passt besonders gut, wenn Datenmengen, Datenformate oder Verarbeitungspfade über die Grenzen einer einzelnen Datenbankinstanz hinauswachsen.

Typische Einsatzfelder:

- Batch-Transformationen im Data Lake oder Lakehouse.
- ELT-Strecken mit Bronze-, Silver- und Gold-Zonen.
- Verarbeitung grosser Parquet-, ORC-, Avro-, JSON- oder CSV-Bestaende.
- Joins und Aggregationen über viele Milliarden Zeilen.
- Datenaufbereitung für BI, Machine Learning oder Reporting.
- Notebook-nahe Analyse mit SQL und PySpark.
- Plattformen wie Apache Spark auf Kubernetes, YARN, Standalone-Clustern oder verwalteten Cloud-Umgebungen.

Spark SQL ist besonders stark, wenn:

- Daten bereits verteilt im Object Storage oder HDFS liegen,
- die Verarbeitung parallelisierbar ist,
- Latenz im Sekunden- bis Minutenbereich akzeptabel ist,
- Transformationen reproduzierbar und versionierbar laufen sollen,
- SQL-Kenntnisse im Team vorhanden sind.

1.2. Wann ein klassisches SQL-System besser passt

Ein relationales Datenbanksystem passt oft besser, wenn du brauchst:

- sehr kurze Transaktionen mit Millisekundenlatenz,
- viele kleine Einzelabfragen gegen wenige Zeilen,
- strikte OLTP-Konsistenzlogik,
- fein granulare Sperr- und Transaktionsmodelle,
- dauerhaft geoeffnete Low-Latency-Clients,
- einfache Verwaltungslogik ohne Clusterbetrieb.

Spark SQL kann SQL ausführen, aber es ist kein Ersatz für eine OLTP-Datenbank. Eine Abfrage startet Arbeit im Cluster, erzeugt Tasks, liest Dateien, plant Shuffles und schreibt Ergebnisse. Für kleine interaktive Punktabfragen ist dieser Aufwand häufig zu hoch.

1.3. Spark SQL im Lakehouse-Kontext

In modernen Datenplattformen arbeitet Spark SQL häufig im Lakehouse:

- Rohdaten werden in offenen Formaten gespeichert.
- Tabellenmetadaten liegen in einem Catalog.
- Transformationen werden als SQL, PySpark oder Scala-Jobs ausgeführt.
- BI- und Analytics-Workloads greifen auf kuratierte Tabellen zu.

Wichtig ist die Trennung:

- **Storage** speichert Dateien und Tabellenformate.
- **Catalog** beschreibt Datenbanken, Tabellen, Schemas und Orte.
- **Spark SQL** plant und verarbeitet Abfragen.
- **Orchestrierung** startet Jobs, verwaltet Abhängigkeiten und Zeitplaene.
- **Governance** regelt Zugriff, Klassifikation und Nachvollziehbarkeit.

1.4. Spark SQL ist nicht nur SQL-Syntax

Spark SQL umfasst mehrere Zugangswege:

- SQL-Strings über `spark.sql(...)`.
- Notebook-Zellen mit SQL-Magic, etwa `%%sql`.
- Temp Views auf DataFrames.
- permanente Tabellen im Catalog.
- JDBC/ODBC-Zugriff über den Spark Thrift Server.
- DataFrame-Transformationen, die denselben Optimizer nutzen.

Das ist ein zentraler Punkt: Ein DataFrame-Join und ein SQL-Join können im Kern denselben logischen Plan erzeugen. Entscheidend ist nicht, ob die Abfrage als SQL-Text oder API-Kette geschrieben wurde, sondern ob der entstehende Plan sinnvoll ist.

1.5. Harte No-Go-Faelle für Spark SQL

Spark SQL ist meist die falsche Wahl, wenn:

- ein Webservice pro Request einzelne Zeilen lesen oder schreiben muss,
- Transaktionen über viele kleine Updates zentral sind,
- Datenmengen klein und lokal gut beherrschbar sind,
- kein Team für Cluster-, Storage- und Jobbetrieb vorhanden ist,
- Query-Kosten nicht gemessen oder begrenzt werden,
- Governance erst "später" kommen soll.

> **Anti-Pattern:** "Wir schreiben alles in Spark SQL, weil SQL alle verstehen." Ergebnis in der Praxis: lange Laufzeiten, teure Cluster, unklare Verantwortungen und Jobs, die bei Datenwachstum kippen.

1.6. Entscheidungsmatrix (kompakt)

Kriterium	Spark SQL passt	Spark SQL passt eher nicht
Datenmenge	gross, verteilt, dateibasiert	klein, transaktional, punktuell
Latenz	Sekunden bis Minuten akzeptabel	Millisekunden erforderlich
Workload	Batch, ELT, Analytics, Lakehouse	OLTP, Request/Response, Einzelupdates
Team	Data Engineering und Plattformbetrieb vorhanden	kein Cluster- oder Storage-Know-how
Optimierung	Datenlayout und Jobs werden aktiv gesteuert	"SQL schreiben und vergessen"

1.7. Mini-Projekt: Technologieentscheidung

Für das Verkaufs- und Nutzungsdaten-Lakehouse ist Spark SQL sinnvoll:

- Rohdaten liegen als Dateien in grossen Tages- oder Stundenbestaenden vor.

- Silver-Transformationen können parallel laufen.
- Joins zwischen Events, Orders und Produktdaten sind datenintensiv.
- Gold-Tabellen können für BI voraggregiert werden.

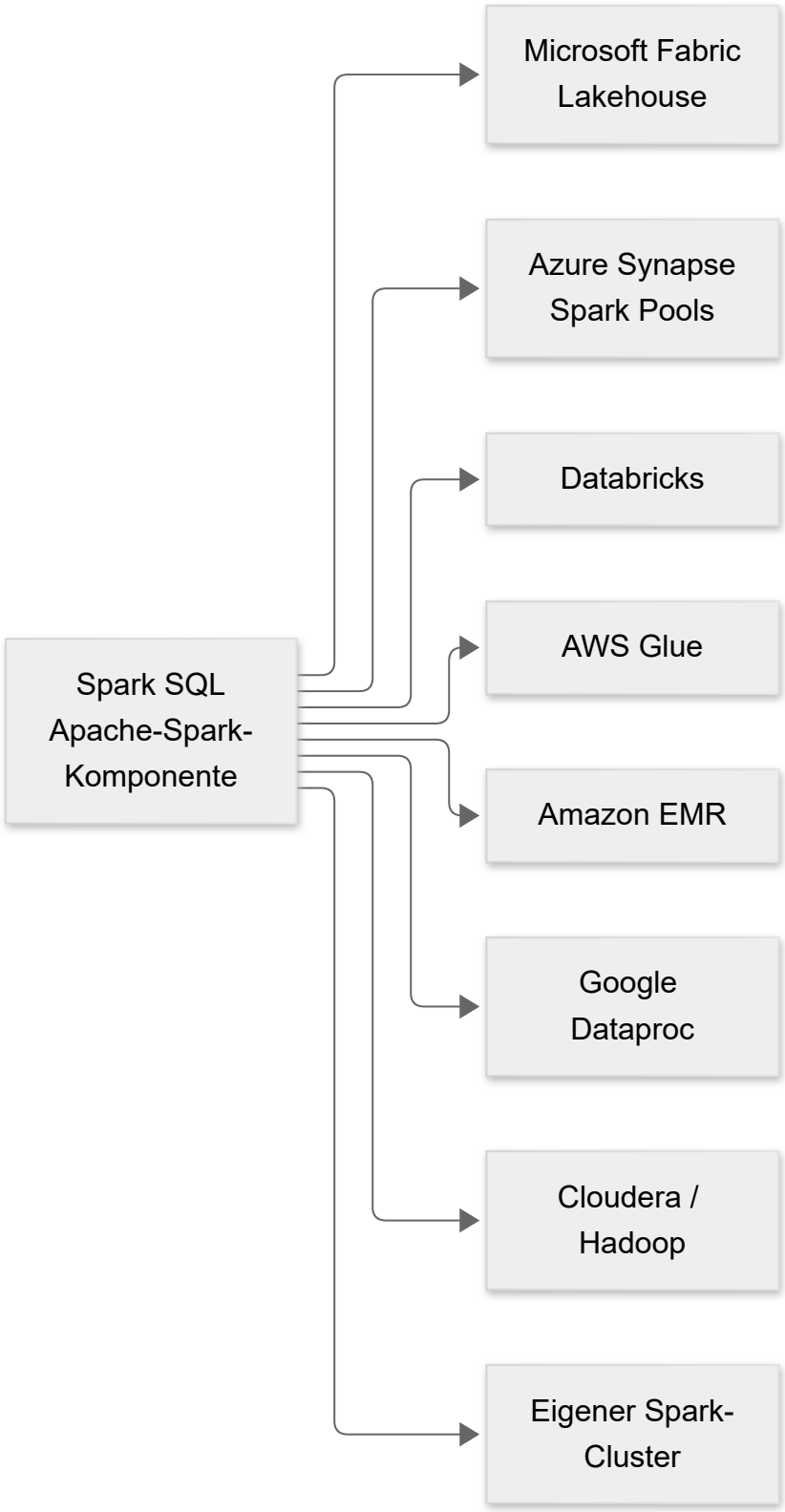
Spark SQL wäre dagegen nicht der richtige Ort für:

- den Live-Checkout des Webshops,
- Produktbestandsbuchungen mit strikter Einzeltransaktion,
- synchrone API-Abfragen pro Nutzerklick.

1.8. Produktbezug: Wo Spark SQL verwendet wird

Spark SQL gehört fachlich zu Apache Spark. Produkte wie Microsoft Fabric oder Databricks "besitzen" Spark SQL also nicht, sondern stellen Spark-Laufzeiten, Kataloge, Notebooks, Security, Storage-Integration und Verwaltungsoberflaechen bereit. Die SQL-Sprache und der Optimizer kommen aus der Spark-Welt; die konkrete Nutzererfahrung hängt aber stark von der Plattform ab.

Typische Umgebungen:

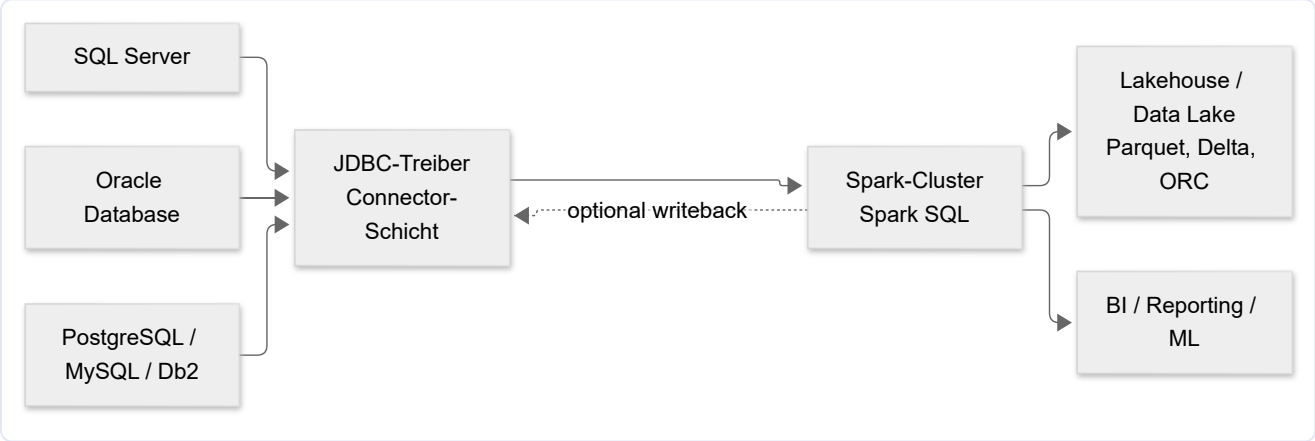


Umgebung	Rolle von Spark SQL	Typischer Kontext
Microsoft Fabric Lakehouse	SQL in Spark-Notebooks und Spark-Jobs	Transformationen auf Lakehouse-Tabellen und Dateien in OneLake
Azure Synapse Spark Pools	Spark-SQL-Abfragen in Notebooks und Jobs	Data Engineering im Azure-Analytics-Stack

Umgebung	Rolle von Spark SQL	Typischer Kontext
Databricks	Spark SQL, DataFrames und Databricks SQL-nahe Workloads	Lakehouse, Delta-Tabellen, BI und Data Engineering
AWS Glue	Spark-basierte ETL-Jobs mit SQL/DataFrames	serverless oder verwaltete Datenintegration
Amazon EMR	Spark auf verwalteten Clustern	Hadoop-/Spark-nahe Plattformen
Google Dataproc	Spark auf Google Cloud Clustern	Batch, ETL, Analytics
Cloudera/Hadoop-Distributionen	Spark SQL neben Hive, Impala und anderen Engines	klassische Big-Data-Plattformen
eigener Spark-Cluster	direkte Spark-SQL-Nutzung	Kubernetes, YARN oder Standalone

1.9. Kann Spark SQL mit SQL Server, Oracle und anderen Datenbanken arbeiten?

Ja, aber mit einer wichtigen Praezisierung: Spark SQL läuft **nicht innerhalb** von SQL Server oder Oracle. Spark startet einen eigenen verteilten Job und kann relationale Datenbanken über Connectoren, besonders JDBC, als Quelle oder Ziel verwenden.



Typisches Muster:

```
jdbc_df = (
    spark.read
    .format("jdbc")
    .option("url", "jdbc:sqlserver://server.example.com:1433;databaseName=Sales")
    .option("dbtable", "dbo.Orders")
    .option("user", "<user>")
    .option("password", "<password>")
    .load()
)

jdbc_df.createOrReplaceTempView("orders_from_sql_server")

spark.sql("""
    SELECT customer_id, count(*) AS orders
    FROM orders_from_sql_server
    GROUP BY customer_id
""").show()
```

Dasselbe Grundprinzip funktioniert auch mit Oracle, PostgreSQL, MySQL, Db2 und vielen anderen Systemen, sofern ein passender JDBC-Treiber und Netzwerkzugriff vorhanden sind.

Dabei gelten aber klare Grenzen:

- Spark SQL verwendet den Spark-SQL-Dialekt, nicht automatisch T-SQL oder Oracle SQL.
- Nicht jede SQL-Funktion wird in die Quelldatenbank "gepusht".
- Grosse Tabellen per JDBC zu lesen kann die Quelldatenbank stark belasten.
- Paralleles Lesen braucht eine geeignete Partitionierungsspalte.
- Schreibvorgaenge über JDBC sind nicht dasselbe wie native OLTP-Transaktionen.
- Rechte, Secrets, Netzwerk und Treiber müssen sauber betrieben werden.

Für größere JDBC-Lesevorgaenge sind insbesondere diese Optionen wichtig:

```
orders_from_oracle = (
    spark.read
    .format("jdbc")
    .option("url", "jdbc:oracle:thin:@//oracle.example.com:1521/SALES")
    .option("dbtable", "SALES.ORDERS")
    .option("user", "<user>")
    .option("password", "<password>")
    .option("partitionColumn", "ORDER_ID")
    .option("lowerBound", "1")
    .option("upperBound", "50000000")
    .option("numPartitions", "16")
    .load()
)
```

Diese Optionen machen aus einem seriellen Datenbankscan mehrere parallele JDBC-Abfragen. Das kann stark beschleunigen, kann aber auch die Quelldatenbank massiv belasten. `lowerBound` und `upperBound` definieren dabei vor allem die Aufteilung der Strides; sie sind kein fachlicher Filterersatz. Für produktive Systeme gilt daher:

- erst mit DBA oder Plattformteam abstimmen,
- nur geeignete numerische, date- oder timestamp-nahe Partitionierungsspalten nutzen,
- Quellsystem-Indizes und Isolation beachten,
- nachts oder in definierten Ladefenstern lesen,
- inkrementelle Muster bevorzugen,
- Full Loads nur bewusst und selten fahren.

Ein gutes mentales Modell:

- SQL Server oder Oracle bleiben die transaktionalen Systeme.
- Spark SQL ist die verteilte Analyse- und Transformations-Engine.
- JDBC ist die Brücke, nicht die Verschmelzung beider Welten.

1.10. Abgrenzung zu T-SQL, Oracle SQL und Fabric SQL Endpoint

Gerade in Microsoft Fabric ist die Unterscheidung wichtig:

- Spark SQL läuft im Spark-Kontext, typischerweise in Notebooks oder Spark Jobs.
- Der Fabric SQL Analytics Endpoint spricht T-SQL-nahe Abfragen gegen Lakehouse-Tabellen.
- Ein klassischer SQL Server führt T-SQL in der SQL-Server-Engine aus.
- Oracle Database führt Oracle SQL und PL/SQL in der Oracle-Engine aus.

Die Syntax kann ähnlich aussehen, aber die Ausfuehrungsorte sind verschieden. Ein `SELECT` in Spark SQL plant Tasks im Spark-Cluster. Ein `SELECT` in SQL Server nutzt den SQL-Server-Optimizer. Ein `SELECT` in Oracle nutzt den Oracle-Optimizer. Diese Engines haben unterschiedliche Funktionen, Kostenmodelle, Transaktionsregeln und Betriebsmetriken.

1.11. Vergleich: Spark SQL, T-SQL, Oracle SQL und Databricks SQL

Die folgende Tabelle ist bewusst grob. Sie hilft, falsche Erwartungen früh zu stoppen:

Frage	Spark SQL	T-SQL in SQL Server / Fabric SQL Endpoint	Oracle SQL / PL/SQL	Databricks SQL
Engine	Apache Spark	SQL-Server- bzw. Fabric-Warehouse-nahe Engine	Oracle Database Engine	Databricks SQL Warehouses
Typische Daten	Dateien, Lakehouse-Tabellen, DataFrames, JDBC-Quellen	relationale Tabellen, Warehouse/Lakehouse-Endpunkt	relationale Oracle-Schemas	Delta-Tabellen im Lakehouse
Stärke	verteilte Transformation grosser Datenmengen	BI, DWH, relationale SQL-Abfragen	OLTP, Enterprise-RDBMS, PL/SQL, Recovery	interaktive SQL-Analyse auf Lakehouse-Daten
Transaktionen	abhängig von Tabellenformat und Plattform	stark im relationalen Kontext	sehr stark im Oracle-Kontext	abhängig von Delta/Plattform
Dialekt	Spark SQL	T-SQL	Oracle SQL und PL/SQL	Databricks SQL, Spark-nah mit Plattform-Erweiterungen
Performance-Modell	Jobs, Tasks, Shuffles, Dateilayout	relationaler Optimizer, Indizes, Statistiken	Oracle Optimizer, Indizes, Partitionierung, Speicherstrukturen	SQL Warehouse, Delta-Optimierungen, Caching
Typischer Fehler	Spark als OLTP-Datenbank missverstehen	Spark-Operationen im SQL Endpoint erwarten	Oracle-Spezifika nach Spark übertragen	Databricks-Spezifika mit Apache Spark gleichsetzen

1.12. Spark SQL in Microsoft Fabric richtig einordnen

In Microsoft Fabric treffen mehrere SQL-Welten aufeinander. Genau deshalb entstehen viele Missverständnisse.

Fabric Lakehouse kombiniert aus Nutzersicht:

- OneLake als Speicher,
- Delta-Tabellen als Tabellenformat,
- Spark-Notebooks und Spark Jobs für Data Engineering,
- einen SQL Analytics Endpoint für T-SQL-nahe Leseabfragen,
- Power BI und semantische Modelle für Konsum.

Ein praktisches Entscheidungsmodell:

Aufgabe in Fabric	Besserer Einstieg	Warum
Rohdaten bereinigen, casten, deduplizieren	Spark Notebook mit PySpark oder Spark SQL	Schreibende Transformationen und verteilte Verarbeitung
Lakehouse-Tabellen in Silver/Gold schreiben	Spark Notebook oder Spark Job Definition	Spark arbeitet direkt auf Lakehouse-Dateien und Tabellen

Aufgabe in Fabric	Besserer Einstieg	Warum
Gold-Tabellen explorativ per SQL lesen	SQL Analytics Endpoint	T-SQL-nahe Abfrageoberflaeche für Lakehouse-Daten
BI-Berichte bauen	Power BI / Semantic Model	Konsum- und Modellierungsschicht
klassisches relationales DWH bauen	Fabric Warehouse	Warehouse-Engine statt Spark-Job-Modell

Wichtig: Der SQL Analytics Endpoint ist nicht einfach "Spark SQL mit anderer Oberfläche". Er dient vor allem dem T-SQL-nahen Lesen von Lakehouse-Daten. Wer Delta-Tabellen transformiert, schreibt, dedupliziert, partitioniert oder komplexe Data-Engineering-Logik ausführt, bleibt typischerweise im Spark-Kontext.

Ein Fabric-Merksatz:

- **Notebook** `%%sql`: Spark SQL im Spark-Kontext.
- **Lakehouse SQL Analytics Endpoint**: T-SQL-nahe Abfragen auf Lakehouse-Daten.
- **Fabric Warehouse**: Warehouse-Engine für relationale DWH-Szenarien.
- **SQL Server**: eigenständige relationale Datenbank ausserhalb des Spark-Kontexts.

---

2. Architektur und Kernkonzepte

SPARK SQL ALS INFRASTRUKTUR

Architektur, Lakehouse, Betrieb, Monitoring und Sicherheit – nicht die Abfragesprache

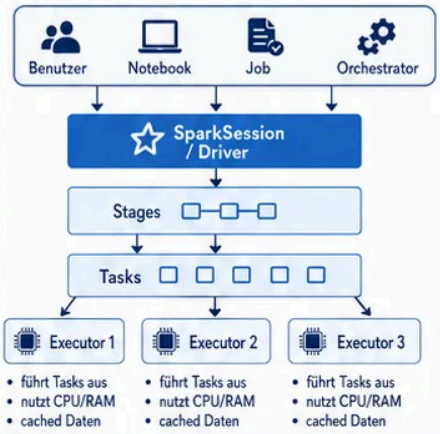
Driver, Executor, Tasks, Cluster, Storage, Catalog, Orchestrierung, Governance

1 Was Spark SQL infrastrukturell ist

Ist	Ist nicht
Komponente von Apache Spark	Kein einzelnes Produkt
Verteilte Engine für strukturierte Daten	Keine klassische OLTP-Datenbank
Häufig im Lakehouse-Umfeld	Nicht für Millisekunden-Einzeltransaktionen gedacht

Ein SELECT ist hier kein Blick in eine Datenbankinstanz, sondern ein Cluster-Job.

2 Kernarchitektur



Partitionen sind die Grundlage der Parallelität. Jede Partition kann unabhängig verarbeitet werden.

Shuffle = teure Netzwerk-Umverteilung bei Joins, Aggregationen, Repartitionierung.

3 Lakehouse-Stack

5	Konsumenten	BI, Notebooks, Data Science, Downstream-Systeme
4	Spark SQL Engine	Planung, Optimierung, verteilte Ausführung
3	Catalog / Metastore	Tabellen, Schemas, Orte, Metadaten
2	Tabellenschicht	Parquet, Delta Lake, ORC, Avro
1	Storage	Object Storage / HDFS / OneLake / Dateien

Lakehouse-Datenmodell

- **Bronze** = Rohdaten
- **Silver** = bereinigt und integriert
- **Gold** = fachlich kuratiert

4 Wo Spark SQL typischerweise läuft

Microsoft Fabric Lakehouse · Azure Synapse Spark Pools · Databricks · AWS Glue / EMR · Google Dataproc · Cloudera · Eigener Spark-Cluster

Die Engine ist dieselbe Grundidee – die Plattform liefert Laufzeit, Katalog, Security und Management.

5 Betrieb & Monitoring

- **Spark UI:** Jobs, Stages, Tasks, SQL
- **History Server:** vergangene Läufe analysieren
- **Event Logs:** Nachvollziehbarkeit und Fehleranalyse
- **Runbooks / Tests:** Betrieb standardisieren

Erste Pflichtsignale

lange Laufzeiten · viele kleine Dateien · große Shuffles · Skew / ungleiche Verteilung · zu hohe Clusterkosten

EXPLAIN, Spark UI und Event Logs gehören zum Alltag – nicht nur zum Incident.

6 Sicherheit & Governance

- Zugriff nach Datenzonen steuern
- Secrets und Credentials nicht im Code speichern
- Web UI / Logs schützen
- Tabellen dokumentieren: Owner, Zweck, Refresh, Klassifikation

- Reproduzierbarkeit
- Auditierbarkeit
- Datenklassifikation
- Least Privilege

7 Wann Spark SQL passt – und wann nicht

Passt gut	Passt eher nicht
große dateibasierte Datenmengen	OLTP und Web-Requests
Batch / ELT / Analytics	viele kleine Einzelupdates
Lakehouse-Transformationen	strikte transaktionale Punktzugriffe
Sekunden bis Minuten Latenz akzeptabel	Millisekunden-Latenz nötig
verteilbare Workloads	kein Team für Plattformbetrieb

VS.

8 Die Infrastruktur-Denkweise

1. Performance beginnt beim Storage
2. SQL, Storage und Laufzeit zusammen denken
3. Clustergröße ersetzt kein sauberes Datenlayout
4. Monitoring und Kostenkontrolle früh verankern
5. Gute Spark-Nutzung braucht Owner, Standards und Betrieb

KERNAUSSAGE

Spark SQL sollte man infrastrukturell lesen: als verteilte Ausführungsengine zwischen Storage, Catalog, Cluster-Ressourcen, Orchestrierung und Governance. Wer nur die SQL-Syntax sieht, übersieht die eigentlichen Erfolgsfaktoren im Betrieb.

2.1. SparkSession als Einstiegspunkt

Die `SparkSession` ist der zentrale Einstieg in Spark SQL. Sie verbindet:

- SQL-Ausfuehrung,
- DataFrame-Erzeugung,
- Catalog-Zugriffe,
- Konfiguration,
- temporäre Views,
- Lese- und Schreiboperationen.

Typisches Muster in PySpark:

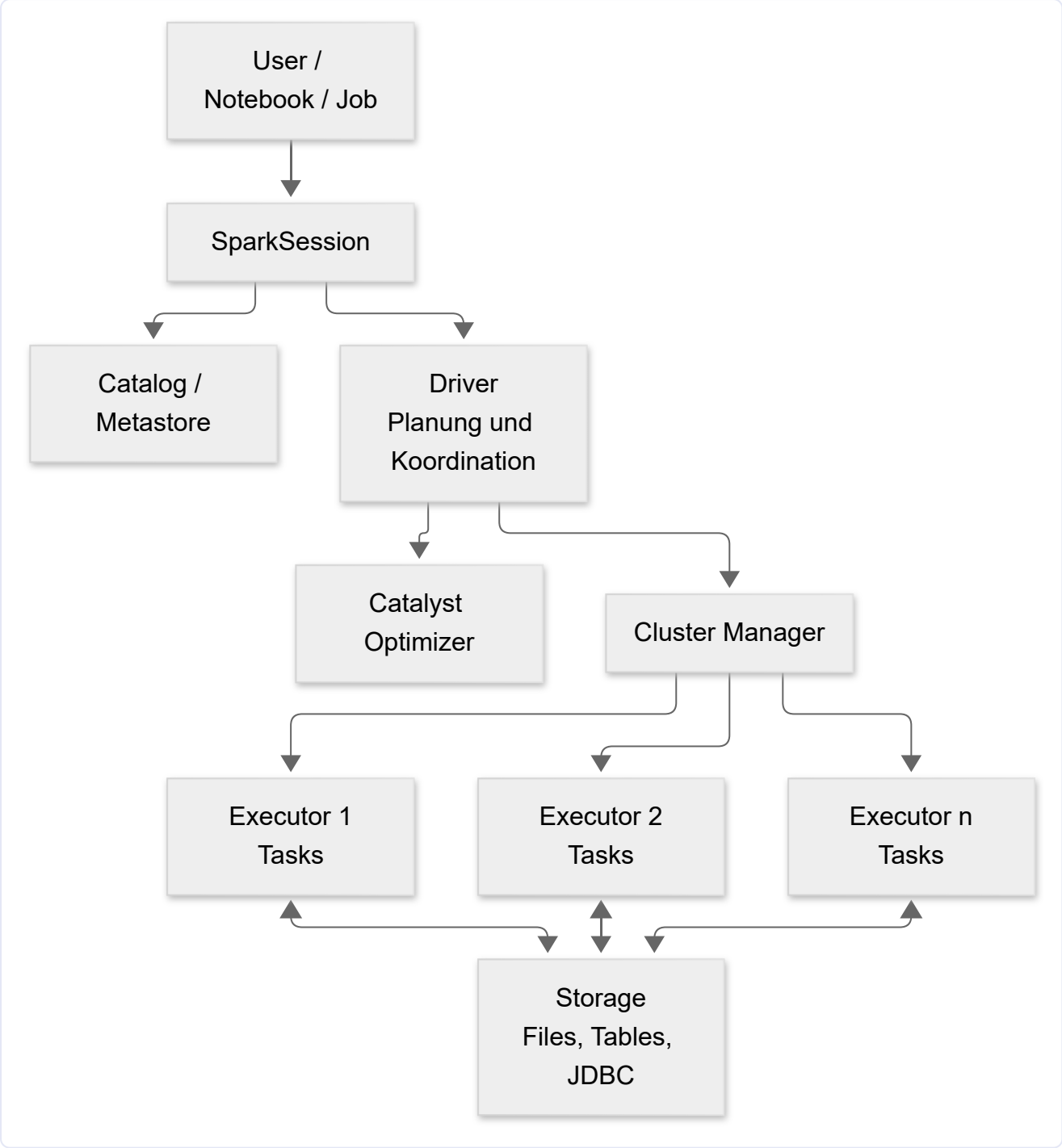
```
from pyspark.sql import SparkSession

spark = (
    SparkSession.builder
    .appName("sales-lakehouse-transform")
    .getOrCreate()
)

orders = spark.read.parquet("/lake/bronze/orders/")
orders.createOrReplaceTempView("orders_raw")

result = spark.sql("""
    SELECT customer_id, count(*) AS order_count
    FROM orders_raw
    GROUP BY customer_id
""")
```

Der wichtige Punkt: `spark.sql(...)` erzeugt zunächst einen Plan. Ausgeführt wird die Arbeit erst, wenn eine Action oder ein Schreibvorgang tatsächlich Ergebnisse verlangt.



2.2. Lazy Evaluation

Spark arbeitet weitgehend lazy. Transformationen wie `select`, `filter`, `join` oder `groupBy` beschreiben Arbeit, führen sie aber nicht sofort aus.

Ausführung entsteht erst bei Actions, zum Beispiel:

- `show()`
- `count()`
- `collect()`
- `write...`
- materialisierte Tabellenoperationen.

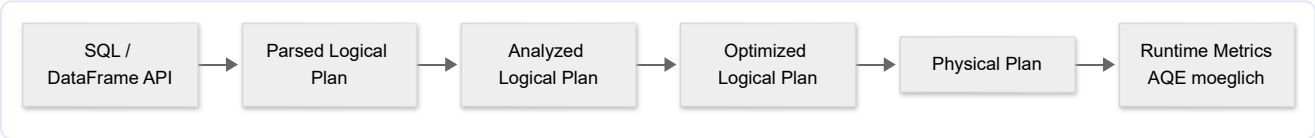
Praxisfolgen:

- Ein Fehler in einer Transformation kann erst spät sichtbar werden.
- Mehrere Transformationen können gemeinsam optimiert werden.
- Unbedachtes `collect()` kann grosse Datenmengen in den Driver ziehen.
- `show()` ist nicht kostenlos, wenn vorher teure Filter, Joins oder Scans stehen.

2.3. Logical Plan, Optimized Plan, Physical Plan

Spark SQL übersetzt eine Abfrage grob in:

1. **Parsed Logical Plan**: SQL wurde syntaktisch gelesen.
2. **Analyzed Logical Plan**: Tabellen, Spalten und Typen sind aufgelöst.
3. **Optimized Logical Plan**: Regeln wie Predicate Pushdown, Projection Pruning oder Vereinfachungen wurden angewendet.
4. **Physical Plan**: konkrete Ausfuehrungsstrategie mit Scans, Joins, Exchanges und Aggregationen.



Mit `EXPLAIN` wird diese Kette sichtbar:

```
EXPLAIN FORMATTED
SELECT
    customer_id,
    sum(net_amount) AS revenue
FROM silver_orders
WHERE order_date >= DATE '2026-01-01'
GROUP BY customer_id;
```

In der Praxis liest man besonders:

- Welche Datenquelle wird gescannt?
- Werden Filter früh angewendet?
- Gibt es `Exchange`-Operationen, also Shuffle?
- Welche Join-Strategie wurde gewählt?
- Stimmen geschätzte und reale Zeilenmengen ungefaehr?

2.4. Driver, Executor, Stage und Task

Spark SQL verteilt Arbeit in einem Cluster:

- Der **Driver** plant und koordiniert.
- **Executor** führen Tasks aus.
- Eine **Stage** fasst Tasks ohne Shuffle-Grenze zusammen.
- Ein **Task** bearbeitet eine Partition.

Wenn eine Abfrage langsam ist, ist die Ursache selten "SQL ist langsam". Häufiger sind:

- zu wenige oder zu viele Partitionen,
- Daten-Skew,
- teure Shuffles,

- schlechte Dateigroessen,
- fehlende Statistiken,
- Executor-Speicherprobleme,
- unpassende Join-Strategien.

2.5. Shuffle als zentrale Kostenstelle

Ein Shuffle verteilt Daten über das Netzwerk neu. Er entsteht typischerweise bei:

- `GROUP BY`
- `DISTINCT`
- grossen Joins
- `ORDER BY`
- Window Functions mit breiten Partitionen
- explizitem `repartition`

Shuffle ist nicht grundsätzlich schlecht. Viele analytische Abfragen brauchen ihn. Aber ein unkontrollierter Shuffle ist oft der größte Kosten- und Laufzeittreiber.

Praxisfragen:

- Kann ein Filter vor dem Join Daten reduzieren?
- Ist eine Join-Seite klein genug für Broadcast?
- Sind Partitionen extrem ungleich verteilt?
- Muss global sortiert werden oder reicht lokale Sortierung?
- Wird dieselbe teure Zwischenmenge mehrfach berechnet?

2.6. Data Sources und Formate

Spark SQL kann über DataFrames und SQL mit vielen Quellen arbeiten. Die offizielle Dokumentation nennt unter anderem:

- Parquet,
- ORC,
- JSON,
- CSV,
- Text,
- XML,
- Hive-Tabellen,
- JDBC,
- Avro,
- Protobuf,
- Binary Files.

Für analytische Tabellen sind spaltenorientierte Formate wie Parquet oder ORC meist deutlich besser geeignet als CSV oder JSON:

- Spalten können selektiv gelesen werden.
- Kompression ist effizienter.
- Typinformationen sind stabiler.

- Praedikate können häufig besser in den Scan gedrueckt werden.

CSV und JSON bleiben wichtig für Rohdaten und Schnittstellen, sollten aber für wiederholte analytische Abfragen meist in kuratierte Tabellenformate ueberfuehrt werden.

2.7. Catalog, Tabellen und Temp Views

Spark SQL unterscheidet:

- **Temp Views** für sessionnahe SQL-Sichten auf DataFrames.
- **Global Temp Views** für Spark-anwendungsweite temporäre Sichtbarkeit.
- **Managed Tables** mit durch den Catalog verwaltetem Speicherort.
- **External Tables** auf bestehenden Speicherpfaden.
- **Views** als gespeicherte Abfragedefinitionen, je nach Catalog-Unterstuetzung.

Temp Views sind ideal für Notebook- und Job-Zwischenschritte:

```
events_df.createOrReplaceTempView("events_stage")
```

Permanente Tabellen sind besser für wiederverwendbare Datenprodukte:

```
CREATE TABLE IF NOT EXISTS gold_daily_revenue
USING parquet
AS
SELECT
    order_date,
    sum(net_amount) AS revenue
FROM silver_orders
GROUP BY order_date;
```

2.8. Distributed SQL Engine

Spark SQL kann auch als verteilter SQL-Engine über JDBC/ODBC oder Kommandozeile verwendet werden. Der Spark Thrift Server erlaubt Clients, SQL gegen Spark auszufuehren, ohne selbst Spark-Code zu schreiben.

Das ist nützlich für:

- BI-nahe Abfragen,
- Ad-hoc-Analyse,
- Migration von Hive-nahen Workloads,
- zentrale SQL-Zugriffe auf Lakehouse-Daten.

Aber es braucht klare Grenzen:

- Lang laufende Queries müssen quotiert und beobachtet werden.
- Ressourcenpools und Warteschlangen sind wichtig.
- Security und Identitaetsdurchreichung müssen sauber gelöst sein.
- Nicht jeder BI-User sollte beliebig grosse Rohdaten scannen können.

2.9. Mini-Projekt: Architektur

Für das Lakehouse ergibt sich ein klares Modell:

- Bronze: rohe Dateien, möglichst unverändert, schemaarm, nachvollziehbar.
- Silver: bereinigte, typisierte, deduplizierte Tabellen.

- Gold: fachliche Modelle für Umsatz, Kampagnen, Kundenverhalten und BI.

Spark SQL liest Bronze-Daten, transformiert sie in Silver, berechnet Gold-Modelle und schreibt Tabellen zurück in den Storage. Ein Catalog beschreibt Tabellen, Spalten und Speicherorte. Orchestrierung startet die Jobs. Monitoring zeigt Laufzeiten, Kosten und Fehler.

---

3. Datenmodellierung und Lakehouse-Design

3.1. Tabellen als Datenprodukte verstehen

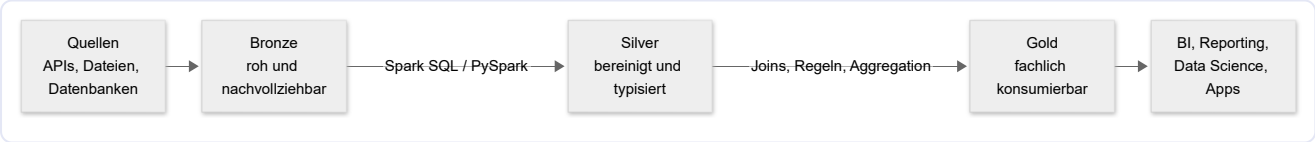
Eine Spark-SQL-Tabelle ist nicht nur ein Speicherort. Sie ist ein Datenprodukt mit:

- fachlichem Zweck,
- Schema,
- Aktualisierungslogik,
- Qualitaetsregeln,
- Owner,
- Zugriffsschutz,
- Kostenprofil,
- Lebenszyklus.

Ohne diese Einordnung entstehen Tabellenfriedhoefe: viele Dateien, unklare Semantik, doppelte Logik und Abfragen, die niemand mehr belastbar erklären kann.

3.2. Bronze, Silver, Gold

Ein bewaehrtes Zonenmodell:



Zone	Zweck	Typische Spark-SQL-Arbeit
Bronze	rohe, nachvollziehbare Eingangsdaten	Laden, Schema-Erkennung, technische Metadaten
Silver	bereinigt, typisiert, dedupliziert	Casting, Deduplizierung, Normalisierung, Qualitaetsregeln
Gold	fachlich konsumierbar	Aggregation, Business-Logik, BI-Modelle

Wichtig:

- Bronze sollte nicht durch fachliche Korrekturen zerstoert werden.
- Silver sollte klare technische Qualität liefern.
- Gold sollte fachlich erklärbar und stabil sein.

3.3. Schema-Design

Spark SQL ist tolerant gegenüber semi-strukturierten Daten, aber produktive Tabellen brauchen stabile Schemas.

Praxisregeln:

- Datentypen explizit setzen, nicht dauerhaft aus CSV oder JSON erraten lassen.
- Zeitspalten sauber als `date` oder `timestamp` modellieren.
- Geldbetraege nicht als `double`, sondern als Dezimaltypen speichern.
- technische Ladezeit, Quellsystem und Batch-ID mitfuehren.
- fachliche Schlüssel getrennt von technischen Zeilen-IDs denken.

Beispiel:

```
CREATE TABLE silver_orders (
    order_id STRING,
    customer_id STRING,
    order_ts TIMESTAMP,
    order_date DATE,
    currency STRING,
    net_amount DECIMAL(18, 2),
    source_system STRING,
    ingestion_batch_id STRING,
    ingestion_ts TIMESTAMP
)
USING parquet
PARTITIONED BY (order_date);
```

3.4. Partitionierung im Storage

Partitionierung im Storage ist ein grober Schneidemechanismus. Sie hilft, wenn Abfragen regelmäßig nach denselben Spalten filtern, etwa Datum, Land oder Mandant.

Gute Partitionierung:

- reduziert gescannte Dateien,
- passt zu häufigen Filtern,
- erzeugt keine unzaehligen Mini-Verzeichnisse,
- ist stabil über Zeit.

Schlechte Partitionierung:

- nach hochkardinalen Spalten wie `customer_id`,
- nach Spalten ohne Filterverwendung,
- zu granular, etwa pro Minute bei kleinen Datenmengen,
- rein technisch gewählt, ohne Query-Profil.

3.5. Kleine Dateien als Betriebsproblem

Viele kleine Dateien sind in Spark-SQL-Landschaften ein klassisches Problem.

Folgen:

- hoher Planungsaufwand,
- viele Tasks mit wenig Arbeit,
- langsame Listings im Object Storage,
- ineffiziente Scans,
- unruhige Laufzeiten.

Gegenmassnahmen:

- Batch-Ausgabe bewusst steuern.

- kleine Mikro-Batches regelmäßig kompaktieren.
- Zielgroessen für Dateien definieren.
- Partitionierung nicht zu fein wählen.
- Schreibjobs auf sinnvolle Parallelitaet begrenzen.

3.6. Normalisierung und Denormalisierung

Spark SQL kann relationale Modelle verarbeiten, aber analytische Lakehouse-Tabellen sind oft anders geschnitten als OLTP-Schemas.

Sinnvolle Muster:

- Roh- und Silver-Zonen eher nah an Quellsystemen und sauber typisiert.
- Gold-Zonen denormalisieren, wenn BI-Abfragen dadurch stabiler und guenstiger werden.
- Dimensionen und Fakten klar trennen, wenn Reporting und Historisierung davon profitieren.
- Stammdaten historisieren, wenn fachliche Rueckblicke korrekte Gültigkeit brauchen.

3.7. Umgang mit Änderungen

Spark-SQL-Pipelines müssen Schema- und Datenveraenderungen bewusst behandeln:

- neue Spalten,
- geänderte Typen,
- gelöschte Felder,
- Quellsystemwechsel,
- Duplikate,
- verspaetete Daten,
- Korrekturen.

Besonders wichtig ist die Unterscheidung:

- **append-only**: neue Daten werden nur angehaengt.
- **overwrite**: Datenbereich wird ersetzt.
- **merge/upsert**: neue und bestehende Daten werden zusammengefuehrt.
- **SCD2**: historische Gültigkeit wird bewahrt.

Spark SQL allein beschreibt die Transformation. Tabellenformat, Catalog und Plattform entscheiden, welche ACID-, Merge- oder History-Faehigkeiten tatsächlich verfügbar sind.

3.8. Mini-Projekt: Tabellenmodell

Für das Verkaufs-Lakehouse:

- `bronze_orders_raw`: rohe Bestelldateien mit Lade-Metadaten.
- `silver_orders`: typisierte, deduplizierte Bestellungen.
- `silver_order_items`: Positionsdaten mit Preis- und Mengenlogik.
- `silver_product`: Produktstammdaten mit Gültigkeit.
- `gold_daily_revenue`: Tagesumsatz je Markt und Produktgruppe.
- `gold_campaign_effect`: Kampagnenwirkung aus Orders und Nutzungsereignissen.

3.9. So setzt du es um - Checkliste

- Definiere pro Tabelle Zone, Owner und Aktualisierungsmodus.
- Lege Partitionierung aus Query-Mustern ab, nicht aus Bauchgefuehl.
- Verwende spaltenorientierte Formate für kuratierte Analytics-Tabellen.
- Plane Kompaktierung und Dateigroessen als Betriebsaufgabe.
- Dokumentiere Schema-Aenderungen und erwartete Vertrage mit Konsumenten.

---

4. Spark-SQL-Praxis: Syntax, Patterns und Grenzen

4.1. Grundmuster für Abfragen

Spark SQL folgt weitgehend vertrauten SQL-Mustern:

SPARK SQL – SYNTAX SCHNELLÜBERSICHT

SELECT, JOINS, CTEs, Window Functions, Views, DDL und typische Muster

Für den schnellen Einstieg – mit Beispielen und Hinweisen zu Spark-spezifischen Besonderheiten

SQL JOIN WINDOW CODE

1 Grundmuster einer Abfrage

```
SELECT
      customer_id,
      count(*) AS orders,
      sum(net_amount) AS revenue
FROM silver_orders
WHERE order_date >= DATE '2026-01-01'
GROUP BY customer_id
HAVING sum(net_amount) > 1000
ORDER BY revenue DESC;
```

SELECT → FROM → WHERE → GROUP BY → HAVING → ORDER BY

Merke: Spark SQL sieht oft vertraut aus – die Kosten entstehen aber im verteilten Ausführungsmodell.

2 Häufige SELECT-Bausteine

Spaltenwahl
```
SELECT col1, col2
```

Alias
```
SELECT net_amount AS revenue
```

DISTINCT
```
SELECT DISTINCT customer_id
```

Filter
```
WHERE status = 'OK'
```

Sortierung
```
ORDER BY revenue DESC
```

Begrenzung
```
LIMIT 100
```

Ausdruck
```
SELECT price * quantity AS amount
```

CASE WHEN
```
CASE
  WHEN revenue > 1000 THEN 'hoch'
  ELSE 'normal'
END
```

Gut für schnelle, lesbare Projektionen.

3 JOINS im Überblick

INNER JOIN	LEFT JOIN	FULL OUTER JOIN	LEFT SEMI JOIN	LEFT ANTI JOIN
ON a.key = b.key	ON a.key = b.key	ON a.key = b.key	ON a.key = b.key	ON a.key = b.key

```
SELECT
    o.order_id,
    p.product_group
FROM silver_order_items o
LEFT JOIN silver_product p
     ON o.product_id = p.product_id;
```

Wichtig: Vor dem Join filtern, Join-Schlüssel sauber typisieren und unnötige Spalten früh entfernen.

`/*+ BROADCAST(p) */` Broadcast Hint für kleine Dimensionen

4 CTEs, Temp Views und Strukturierung

CTE = WITH-Block für Lesbarkeit und Zwischenschritte

```
WITH ranked AS (
 SELECT *,
        row_number() OVER (
         PARTITION BY order_id
         ORDER BY ingestion_ts DESC
        ) AS rn
 FROM bronze_orders_raw
)
SELECT *
FROM ranked
WHERE rn = 1;
```

Temp Views verbinden DataFrame-Welt und SQL

```
CREATE OR REPLACE TEMP VIEW orders_stage AS
SELECT *
FROM bronze_orders_raw;
```

↓

```
SELECT customer_id, sum(net_amount)
FROM orders_stage
GROUP BY customer_id;
```

5 Window Functions

```
SELECT
    customer_id,
    order_id,
    order_ts,
    net_amount,
    row_number() OVER (
        PARTITION BY customer_id
        ORDER BY order_ts DESC
    ) AS order_rank
FROM silver_orders;
```

Häufige Window Functions

- row_number()
- rank()
- dense_rank()
- lag() /lead()
- sum() OVER (...)
- avg() OVER (...)

Typische Einsätze: letzte Bestellung, Ranglisten, Deduplizierung, gleitende Werte.

6 DDL, Tabellen und Views

1 Tabelle anlegen:
```
CREATE TABLE sales_gold (
  order_id STRING,
  revenue DECIMAL(18,2)
);
```

2 View anlegen:
```
CREATE VIEW v_sales AS
SELECT * FROM sales_gold;
```

3 Daten einfügen:
```
INSERT INTO sales_gold
SELECT order_id, net_amount
FROM silver_orders;
```

4 Partition overwrite example:
```
INSERT OVERWRITE TABLE gold_daily_revenue
PARTITION (order_date)
SELECT market, sum(net_amount), order_date
FROM silver_orders_enriched
GROUP BY market, order_date;
```

DDL-Fähigkeiten und MERGE-Unterstützung hängen von Catalog und Tabellenformat ab.

7 Datentypen, Casts und Datumsfunktionen

Häufige Typen	Wichtige Funktionen
STRING	`cast(order_ts AS timestamp)`
INT / BIGINT	`cast(order_ts AS date)`
DECIMAL(18,2)	`current_date()`
DATE	`current_timestamp()`
TIMESTAMP	`date(order_ts)`
BOOLEAN	`upper(currency)`
	`coalesce(col1, col2)`

Dialektunterschiede beachten: Spark SQL ist nicht identisch mit T-SQL oder Oracle SQL.

8 Praxis-Hinweise & Denkweise

1. Erst lesbar schreiben, dann mit EXPLAIN prüfen.
2. Große Shuffles entstehen oft durch JOIN + GROUP BY + Sortierung.
3. SQL und DataFrame-API nutzen denselben Optimizer.
4. Window Functions und globale ORDER BY sind oft teuer.
5. Inkrementelle Muster sparen viel Laufzeit.

SQL passt gut für:
- fachliche Aggregationen
- lesbare Transformationen
- BI-nahe Logik

API passt gut für:
- technische Bereinigung
- dynamische Spaltenlogik
- Wiederverwendung in Python/Scala

KERNAUSSAGE

Spark SQL-Syntax ist gut lesbar und für analytische Transformationen sehr stark.
Wer Spark SQL produktiv nutzt, sollte aber nicht nur die Syntax kennen, sondern auch Joins, Window Functions, Datenlayout und plattformspezifische Unterschiede bewusst einordnen.

```
SELECT
    customer_id,
    count(*) AS orders,
    sum(net_amount) AS revenue
FROM silver_orders
WHERE order_date >= DATE '2026-01-01'
GROUP BY customer_id
HAVING sum(net_amount) > 1000
ORDER BY revenue DESC;
```

Der Unterschied liegt im Ausfuehrungsmodell. Diese Abfrage kann:

- viele Dateien scannen,
- Filter pushen,
- Daten shufflen,
- Aggregationen verteilt berechnen,
- Ergebnisse global sortieren.

4.2. Temp Views als Brücke zwischen PySpark und SQL

In der Praxis mischen Teams oft PySpark und Spark SQL:

```
clean_orders = (
    raw_orders
    .filter("order_id IS NOT NULL")
    .withColumnRenamed("amount", "net_amount")
)

clean_orders.createOrReplaceTempView("orders_stage")
```

Danach kann SQL die fachliche Logik ausdruecken:

```
SELECT
    customer_id,
    date(order_ts) AS order_date,
    sum(net_amount) AS revenue
FROM orders_stage
GROUP BY customer_id, date(order_ts);
```

Dieses Muster ist stark, wenn technische Bereinigung in PySpark und fachliche Aggregation in SQL besser lesbar ist.

4.3. Joins bewusst schreiben

Joins sind in Spark SQL oft der kritische Punkt.

Typische Join-Arten:

- `INNER JOIN`
- `LEFT JOIN`
- `RIGHT JOIN`
- `FULL OUTER JOIN`
- `LEFT SEMI JOIN`
- `LEFT ANTI JOIN`
- Cross Join nur bewusst und kontrolliert.

Beispiel:

```
SELECT
    o.order_id,
    o.customer_id,
    p.product_group,
    oi.quantity,
    oi.net_amount
FROM silver_orders o
JOIN silver_order_items oi
    ON o.order_id = oi.order_id
LEFT JOIN silver_product p
    ON oi.product_id = p.product_id
WHERE o.order_date >= DATE '2026-01-01';
```

Optimierungsfragen:

- Wird vor dem Join gefiltert?
- Ist eine Seite klein genug für Broadcast?
- Sind Join-Schluessel sauber typisiert?
- Gibt es Daten-Skew auf einzelnen Schlüsseln?
- Werden nach dem Join nur benötigte Spalten weitergetragen?

4.4. Broadcast Hints gezielt einsetzen

Spark kann kleine Tabellen automatisch broadcasten. Manchmal hilft ein expliziter Hint:

```
SELECT /*+ BROADCAST(p) */
    o.order_id,
    p.product_group
FROM silver_order_items o
JOIN silver_product p
    ON o.product_id = p.product_id;
```

Das ist sinnvoll, wenn:

- die Dimension klein und stabil ist,
- Statistiken unvollständig sind,
- ein grosser Shuffle vermieden werden soll.

Es ist riskant, wenn:

- die Dimension unerwartet waechst,
- Executor-Speicher knapp ist,
- mehrere Broadcasts parallel laufen,
- Hints unreflektiert kopiert werden.

4.5. Window Functions

Window Functions sind für analytische Aufgaben sehr wertvoll:

```
SELECT
    customer_id,
    order_id,
    order_ts,
    net_amount,
    row_number() OVER (
        PARTITION BY customer_id
        ORDER BY order_ts DESC
```

```
    ) AS order_rank
FROM silver_orders;
```

Typische Anwendungen:

- letzte Bestellung pro Kunde,
- Deduplizierung,
- Ranglisten,
- gleitende Summen,
- SCD-nahe Gueltigkeitslogik.

Aber Window Functions können teuer sein, weil Partitionierung und Sortierung Daten bewegen. Wichtig sind passende Filter, sinnvolle Partition Keys und begrenzte Datenbereiche.

4.6. Deduplizierung

Ein robustes SQL-Muster für Deduplizierung:

```
WITH ranked AS (
    SELECT
        *,
        row_number() OVER (
            PARTITION BY order_id
            ORDER BY ingestion_ts DESC
        ) AS rn
    FROM bronze_orders_raw
)
SELECT
    order_id,
    customer_id,
    cast(order_ts AS timestamp) AS order_ts,
    cast(order_ts AS date) AS order_date,
    cast(net_amount AS decimal(18, 2)) AS net_amount,
    source_system,
    ingestion_batch_id,
    ingestion_ts
FROM ranked
WHERE rn = 1;
```

Dieses Muster ist gut lesbar, aber bei sehr grossen Datenmengen muss man Partitionierung, Sortierung und inkrementelle Verarbeitung mitdenken.

4.7. Inkrementelle Verarbeitung

Viele Spark-SQL-Pipelines sollten nicht täglich alles neu rechnen.

Mögliche Muster:

- nur neue Dateien laden,
- nach `ingestion_date` oder fachlichem Datum filtern,
- betroffene Partitionen ersetzen,
- Upserts über Tabellenformat-Funktionen nutzen,
- Gold-Tabellen aus veränderten Silver-Bereichen aktualisieren.

Beispiel für ersetzbare Tagesaggregation:

```
INSERT OVERWRITE TABLE gold_daily_revenue
PARTITION (order_date)
SELECT
```

```
    market,
    product_group,
    sum(net_amount) AS revenue,
    count(DISTINCT order_id) AS orders,
    order_date
FROM silver_orders_enriched
WHERE order_date BETWEEN DATE '2026-04-01' AND DATE '2026-04-07'
GROUP BY market, product_group, order_date;
```

Ob `INSERT OVERWRITE`, dynamische Partitionen oder `MERGE` passend sind, hängt stark von Tabellenformat, Catalog und Plattform ab.

4.8. SQL versus DataFrame-API

SQL ist oft besser für:

- fachlich lesbare Aggregationen,
- Joins,
- BI-nahe Logik,
- dokumentierbare Transformationen.

DataFrame-API ist oft besser für:

- komplexe technische Bereinigung,
- dynamische Spaltenlogik,
- wiederverwendbare Funktionen,
- Integration mit Python- oder Scala-Code,
- kontrollierte Tests.

Gute Teams streiten nicht dogmatisch über SQL gegen API. Sie entscheiden pro Transformationsschritt nach Lesbarkeit, Testbarkeit und Betriebsrisiko.

4.9. Grenzen und Dialektunterschiede

Spark SQL ist nicht identisch mit T-SQL, Oracle SQL, PostgreSQL SQL oder ANSI SQL in jedem Detail.

Typische Unterschiede:

- Datentypen und Cast-Verhalten.
- Null-Semantik in Randfaellen.
- Datums- und Zeitfunktionen.
- DDL-Faehigkeiten je Catalog.
- `MERGE`-Unterstuetzung je Tabellenformat.
- Rechte- und Transaktionsmodell je Plattform.

Deshalb sollten Teams Dialektannahmen dokumentieren, vor allem wenn Workloads aus klassischen Datenbanken nach Spark migriert werden.

4.10. Mini-Projekt: Silver-Transformation

Eine typische Silver-Transformation:

```
CREATE OR REPLACE TEMP VIEW orders_dedup AS
WITH ranked AS (
    SELECT
        *,
```

```
        row_number() OVER (
            PARTITION BY order_id
            ORDER BY ingestion_ts DESC
        ) AS rn
    FROM bronze_orders_raw
    WHERE ingestion_date = DATE '2026-04-28'
)
SELECT
    order_id,
    customer_id,
    cast(order_ts AS timestamp) AS order_ts,
    cast(order_ts AS date) AS order_date,
    upper(currency) AS currency,
    cast(net_amount AS decimal(18, 2)) AS net_amount,
    source_system,
    ingestion_batch_id,
    ingestion_ts
FROM ranked
WHERE rn = 1;
```

Danach kann ein Schreibjob die Daten in `silver_orders` übernehmen. Entscheidend ist, dass Deduplizierungsregel, Ladefenster und Fehlerbehandlung explizit dokumentiert sind.

4.11. End-to-End-Beispiel: Von Bronze zu Gold

Das folgende Beispiel zeigt einen kleinen, aber realistischen Ablauf. Es ist bewusst kompakt gehalten und soll das Zusammenspiel aus Laden, Bereinigen, Persistieren, Aggregieren und Optimieren zeigen.

Schritt 1: Rohdaten in Bronze lesen

```
raw_orders = (
    spark.read
    .option("header", "true")
    .option("mode", "PERMISSIVE")
    .csv("/lake/landing/orders/2026-04-28/")
)

raw_orders.createOrReplaceTempView("orders_raw")
```

In produktiven Pipelines sollte das Schema explizit definiert werden. Automatische Schema-Inferenz ist für Exploration bequem, aber für stabile Jobs riskant.

Schritt 2: Bronze mit technischen Metadaten schreiben

```
CREATE OR REPLACE TEMP VIEW bronze_orders_today AS
SELECT
    *,
    current_timestamp() AS ingestion_ts,
    'orders_2026_04_28' AS ingestion_batch_id,
    DATE '2026-04-28' AS ingestion_date
FROM orders_raw;
```

Je nach Plattform wird diese View anschliessend als Tabelle oder Partition geschrieben. In Fabric, Databricks oder anderen Lakehouse-Umgebungen ist das Ziel typischerweise eine Delta- oder Parquet-nahe Tabelle.

Schritt 3: Silver typisieren und deduplizieren

```
CREATE OR REPLACE TEMP VIEW silver_orders_today AS
WITH typed AS (
    SELECT
        cast(order_id AS string) AS order_id,
```

```
        cast(customer_id AS string) AS customer_id,
        cast(order_ts AS timestamp) AS order_ts,
        cast(order_ts AS date) AS order_date,
        upper(cast(currency AS string)) AS currency,
        cast(net_amount AS decimal(18, 2)) AS net_amount,
        source_system,
        ingestion_batch_id,
        ingestion_ts
    FROM bronze_orders_today
    WHERE order_id IS NOT NULL
),
ranked AS (
    SELECT
        *,
        row_number() OVER (
            PARTITION BY order_id
            ORDER BY ingestion_ts DESC
        ) AS rn
    FROM typed
)
SELECT
    order_id,
    customer_id,
    order_ts,
    order_date,
    currency,
    net_amount,
    source_system,
    ingestion_batch_id,
    ingestion_ts
FROM ranked
WHERE rn = 1;
```

Schritt 4: Silver persistieren

```
INSERT OVERWRITE TABLE silver_orders
PARTITION (order_date)
SELECT
    order_id,
    customer_id,
    order_ts,
    currency,
    net_amount,
    source_system,
    ingestion_batch_id,
    ingestion_ts,
    order_date
FROM silver_orders_today;
```

Ob `INSERT OVERWRITE`, Append, `MERGE` oder ein anderer Schreibmodus richtig ist, hängt vom Tabellenformat und von der fachlichen Korrekturlogik ab. Für Tagespartitionen ist ersetzendes Schreiben oft einfacher zu kontrollieren als zeilenweise Upsert-Logik.

Schritt 5: Gold-Aggregation erzeugen

```
CREATE OR REPLACE TABLE gold_daily_revenue
USING parquet
AS
SELECT
    order_date,
    currency,
    count(DISTINCT order_id) AS order_count,
    count(DISTINCT customer_id) AS customer_count,
    sum(net_amount) AS revenue
```

```
FROM silver_orders
GROUP BY order_date, currency;
```

Schritt 6: Plan prüfen

```
EXPLAIN FORMATTED
SELECT
    order_date,
    currency,
    sum(revenue) AS revenue
FROM gold_daily_revenue
WHERE order_date >= DATE '2026-04-01'
GROUP BY order_date, currency;
```

Bei der Prüfung interessiert:

- Werden nur benötigte Spalten gelesen?
- Greift Partition Pruning auf `order_date`?
- Entsteht ein unerwartet grosser Shuffle?
- Sind Gold-Daten klein genug für interaktive BI-Abfragen?
- Ist die Datenqualität vor dem Schreiben geprüft?

Schritt 7: Mindestchecks

```
SELECT
    count(*) AS rows_total,
    count(DISTINCT order_id) AS orders_distinct,
    sum(CASE WHEN net_amount IS NULL THEN 1 ELSE 0 END) AS missing_amounts
FROM silver_orders_today;
```

Ein End-to-End-Workflow ist erst dann produktionsnah, wenn er nicht nur Daten schreibt, sondern auch erklärt, welche Daten verarbeitet wurden, welche Qualitaetsregeln gelten und wie ein fehlerhafter Lauf wiederholt oder zurueckgerollt wird.

---

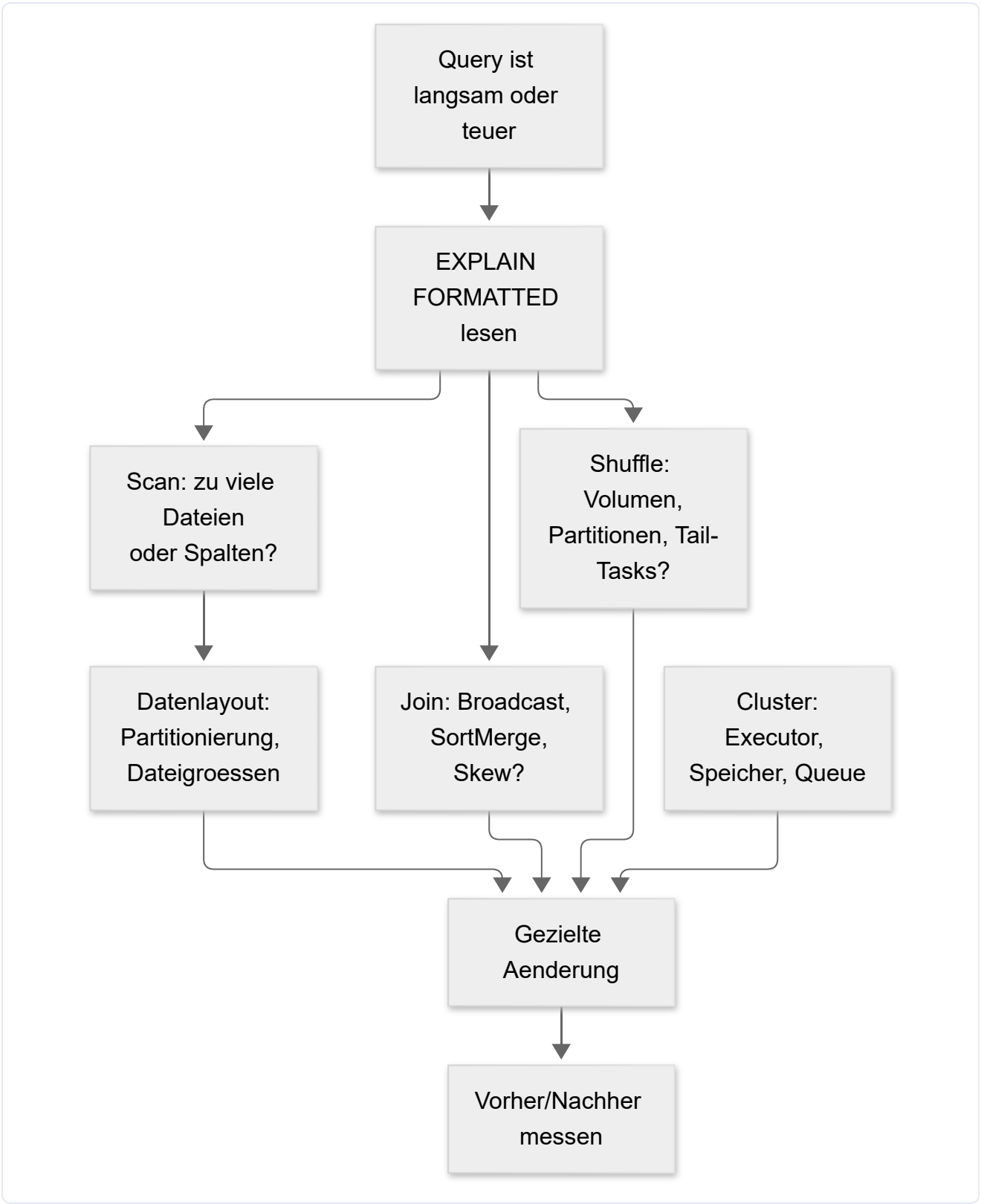
5. Performance, Skalierung und Kostenkontrolle

5.1. Performance in Spark SQL systematisch lesen

Spark-SQL-Performance entsteht aus mehreren Ebenen:

- SQL-Plan,
- Datenlayout,
- Partitionierung,
- Join-Strategie,
- Shuffle-Verhalten,
- Executor-Speicher,
- Cluster-Ressourcen,
- Storage-Latenz,
- Dateianzahl und Dateigroessen.

Ein einzelner Parameter löst selten das Grundproblem. Wer nur `spark.sql.shuffle.partitions` verändert, ohne Plan, Datenmenge und Shuffle zu verstehen, optimiert blind.



5.2. `EXPLAIN` als Pflichtwerkzeug

`EXPLAIN FORMATTED` sollte für kritische Queries Standard sein:

```
EXPLAIN FORMATTED
SELECT /*+ BROADCAST(p) */
    p.product_group,
    sum(oi.net_amount) AS revenue
```

```
FROM silver_order_items oi
JOIN silver_product p
    ON oi.product_id = p.product_id
WHERE oi.order_date >= DATE '2026-01-01'
GROUP BY p.product_group;
```

Achte auf:

- `FileScan`: welche Dateien und Spalten werden gelesen?
- `PushedFilters`: werden Filter in die Quelle gedrueckt?
- `Exchange`: wo entsteht Shuffle?
- `BroadcastHashJoin`, `SortMergeJoin`, `ShuffledHashJoin`: welche Join-Strategie wurde gewählt?
- `AdaptiveSparkPlan`: ob AQE aktiv ist.

Die offizielle Spark-Dokumentation zu SQL Performance Tuning beschreibt genau diese Optimierungsfelder: Caching, Dateipartitionen, Join-Hints, Adaptive Query Execution, Coalescing von Shuffle-Partitionen und Skew-Behandlung. In der Praxis sollte man diese Punkte nicht als isolierte Trickliste lesen, sondern als Diagnosebaum: erst Plan und Datenlayout verstehen, dann konkrete Hebel setzen.

5.3. Predicate Pushdown und Projection Pruning

Zwei einfache, aber starke Prinzipien:

- **Predicate Pushdown**: Filter möglichst früh in den Scan druecken.
- **Projection Pruning**: nur benötigte Spalten lesen.

Schlecht:

```
SELECT *
FROM silver_orders;
```

Besser:

```
SELECT
    order_id,
    customer_id,
    net_amount
FROM silver_orders
WHERE order_date = DATE '2026-04-28';
```

Bei spaltenorientierten Formaten kann das massiv weniger I/O bedeuten.

5.4. Joins optimieren

Wichtige Join-Hebel:

- kleine Dimensionen broadcasten,
- grosse Tabellen vor dem Join filtern,
- nur benötigte Spalten joinen,
- Join-Schluessel typgleich halten,
- Skew-Schluessel erkennen,
- Statistiken aktuell halten.

Typische Anti-Patterns:

- Join auf gecasteten Ausdruecken statt sauberen Spalten.

- Join nach `SELECT *`.
- Cross Join durch fehlende Bedingung.
- riesige Dimension versehentlich broadcasten.
- unklare Duplikate, die Zeilenmengen explodieren lassen.

5.5. Adaptive Query Execution

Adaptive Query Execution kann Spark-SQL-Plaene während der Laufzeit verbessern. Typische AQE-Faehigkeiten sind:

- Shuffle-Partitionen zusammenfassen,
- Skew besser behandeln,
- Join-Strategien anhand echter Laufzeitstatistiken anpassen.

AQE ist wichtig, weil statische Schätzungen in verteilten Datenlandschaften oft ungenau sind. Trotzdem bleibt AQE ein Optimierungsrahmen, kein Ersatz für saubere Datenmodelle.

5.6. Statistiken und Metadaten

Spark kann bessere Plaene erzeugen, wenn Statistiken verfügbar sind:

- Tabellen- und Spaltengroessen,
- Zeilenanzahl,
- Min/Max-Werte aus Dateimetadaten,
- Catalog-Statistiken,
- Laufzeitstatistiken.

Praxis:

- `ANALYZE TABLE` dort nutzen, wo Catalog und Plattform es sinnvoll unterstützen.
- `DESCRIBE EXTENDED` und `EXPLAIN COST` für kritische Tabellen prüfen.
- Statistiken nach grossen Datenveraenderungen aktualisieren.

5.7. Partitionen: nicht zu viele, nicht zu wenige

Spark braucht Parallelitaet, aber zu viele Mini-Tasks kosten Overhead.

Leitfragen:

- Wie gross sind Eingabedateien?
- Wie viele Tasks entstehen?
- Sind Tasks ungefaehr gleich gross?
- Gibt es lange Tail-Tasks?
- Entsteht unnötig viel Shuffle?

Typische Werkzeuge:

- `coalesce(n)` zum Reduzieren ohne vollständigen Shuffle.
- `repartition(n)` für neue Verteilung mit Shuffle.
- Partitionierung beim Schreiben bewusst steuern.
- `spark.sql.shuffle.partitions` workloadabhaengig setzen.

5.8. Caching bewusst einsetzen

Caching lohnt sich, wenn dieselbe Zwischenmenge mehrfach gelesen wird und teuer zu berechnen ist.

Beispiel:

```
enriched = spark.sql("""
    SELECT
        o.*,
        p.product_group
    FROM silver_orders o
    JOIN silver_product p
        ON o.product_id = p.product_id
""")

enriched.cache()
enriched.count()
```

Wichtig:

- Cache materialisieren, sonst bleibt er nur vorgemerkt.
- Speicherverbrauch beobachten.
- Cache wieder entfernen, wenn er nicht mehr gebraucht wird.
- Nicht jede Tabelle reflexhaft cachen.

5.9. Kostenkontrolle

Spark SQL kann sehr teuer werden, wenn Jobs grosse Datenbereiche wiederholt scannen oder Cluster zu lange laufen.

Kostenhebel:

- Datenbereich früh filtern.
- Gold-Tabellen für wiederholte BI-Abfragen vorberechnen.
- Rohdaten nicht direkt für Standardberichte nutzen.
- kleine Dateien kompaktieren.
- Cluster automatisch beenden oder skalieren.
- fehlgeschlagene Jobs schnell erkennen.
- teure Explorationsqueries begrenzen.

5.10. Mini-Projekt: Tuning-Loop

Für das Lakehouse:

1. Kritische Gold-Abfrage identifizieren.
2. `EXPLAIN FORMATTED` ausführen.
3. Spark UI auf Shuffle, Skew, Laufzeit und Task-Verteilung prüfen.
4. Filter, Spaltenauswahl, Join-Strategie und Partitionierung anpassen.
5. Datenlayout und Dateigroessen prüfen.
6. Laufzeit und Kosten vor/nach der Änderung dokumentieren.

5.11. Performance-Regressionen früh stoppen

Führe Gates für produktive Pipelines ein:

- Laufzeit steigt um mehr als definierten Schwellwert.

- gescannte Datenmenge steigt unerwartet.
- Shuffle-Volumen explodiert.
- Anzahl kleiner Dateien waechst stark.
- Skew erzeugt lange Tail-Tasks.
- Gold-Tabelle liefert andere Zeilenanzahl ohne erklärte Ursache.

---

6. Betrieb, Monitoring und Troubleshooting

6.1. Erste Pflichtsignale im Alltag

Produktive Spark-SQL-Workloads brauchen Sicht auf:

- Job-Laufzeit,
- Stage- und Task-Verteilung,
- Shuffle Read/Write,
- Input- und Output-Datenmenge,
- Executor-Speicher und GC,
- fehlgeschlagene Tasks,
- Daten-Skew,
- Dateianzahl und Dateigroessen,
- Queue-Wartezeit,
- Kosten pro Pipeline oder Datenprodukt.

6.2. Spark UI und SQL UI

Jede Spark-Anwendung stellt während der Laufzeit eine Web UI bereit, standardmaessig auf Port 4040 des Driver-Knotens. Dort sieht man unter anderem:

- Jobs,
- Stages,
- Tasks,
- Storage,
- Environment,
- Executors,
- SQL-Abfragen und Plaene.

Für Spark SQL ist die SQL-Ansicht besonders wichtig. Sie zeigt physische Plaene, Laufzeitmetriken und Details wie Runtime Statistics.

6.3. History Server und Event Logs

Da viele Jobs nach kurzer Zeit beendet sind, reichen Live-UIs nicht. Event Logs und Spark History Server machen abgeschlossene Anwendungen auswertbar.

Praxisregel:

- Produktive Jobs sollten Event Logging aktivieren.
- Logs müssen persistent gespeichert werden.
- History Server oder Plattform-UI muss für Betriebsteams erreichbar sein.

- Job-ID, Pipeline-Run und Datenbereich sollten korrelierbar sein.

6.4. Typischer Incident-Workflow

Ein robuster Spark-SQL-Incident-Workflow:

1. Symptom klären: Job langsam, falsch, fehlgeschlagen oder zu teuer?
2. Betroffenen Run, Datenbereich und Code-Stand identifizieren.
3. SQL UI und Stages prüfen.
4. Shuffle, Skew, Executor-Verluste und Speicherfehler ansehen.
5. Input-Dateien, Dateianzahl und Partitionen gegen Vorlauf vergleichen.
6. Datenqualität und Schema-Aenderungen prüfen.
7. Fix testen, Laufzeit und Ergebnis validieren.

6.5. Typische Fehlerbilder

- **OutOfMemory**: Broadcast zu gross, Cache zu breit, Skew, zu grosse Partitionen oder Driver-Collect.
- **Sehr viele kleine Tasks**: kleine Dateien, zu feine Partitionierung, zu hohe Parallelitaet.
- **Einzelne Tasks laufen ewig**: Daten-Skew oder ungleich grosse Dateien.
- **Langsame Joins**: falsche Join-Strategie, fehlende Filter, fehlende Statistiken.
- **Unerwartete Ergebnisse**: Duplikate, Null-Semantik, falsche Casts, Schema Drift.
- **Hohe Kosten**: wiederholte Full Scans, fehlende Gold-Modelle, ineffizientes Datenlayout.

6.6. Runbook-Vorlage pro Pipeline

Eine einfache Vorlage:

- Pipeline-Name.
- Owner und Eskalation.
- Quelltabellen und Zieltabellen.
- Erwartete Laufzeit.
- Erwartete Input-Zeilen und Output-Zeilen.
- Kritische Partitionen oder Datenbereiche.
- Wichtige Konfigurationen.
- Bekannte teure Joins.
- Wiederanlauf-Strategie.
- Datenqualitaetspruefungen.

6.7. Tests für Spark SQL

Sinnvolle Testebenen:

- Unit-Tests für Transformationslogik mit kleinen DataFrames.
- SQL-Golden-Tests für erwartete Ergebnisse.
- Schema-Tests für Pflichtspalten und Datentypen.
- Datenqualitaetschecks für Duplikate, Nulls und Wertebereiche.
- Performance-Smoke-Tests für kritische Pfade.
- End-to-End-Tests für Orchestrierung und Wiederanlauf.

6.8. Mini-Projekt: Betrieb

Für das Verkaufs-Lakehouse:

- Jeder Tageslauf schreibt Batch-ID, Quellbereich und Ergebnisanzahl.
- Silver-Jobs prüfen Duplikate und Pflichtfelder.
- Gold-Jobs prüfen Umsatzsummen gegen Vortag und erwartete Schwankungsbereiche.
- Event Logs sind dauerhaft verfügbar.
- Teure Gold-Abfragen werden quartalsweise geprüft.

6.9. Post-Incident-Lernen

Nach jedem größeren Spark-SQL-Incident:

1. Welche Daten- oder Codeaenderung hat das Verhalten ausgelöst?
2. Welche Metrik hätte früher gewarnt?
3. War das Problem SQL, Datenlayout, Cluster, Storage oder Governance?
4. Welche Query oder Tabelle braucht ein dauerhaftes Gate?
5. Welche Kosten wurden verursacht und wie lassen sie sich künftig begrenzen?

---

7. Sicherheit, Governance und Compliance

7.1. Sicherheitsgrundsatz

Spark ist nicht automatisch sicher, nur weil es in einer geschlossenen Plattform läuft. Die offizielle Spark-Sicherheitsdokumentation betont, dass viele Security-Funktionen nicht per Default aktiv sind und je Deployment bewertet werden müssen.

Produktive Spark-SQL-Umgebungen brauchen mindestens:

- Authentifizierung,
- Autorisierung,
- Netzwerkverschluesselung,
- Schutz der Web UIs,
- Secrets-Management,
- Zugriffskontrolle auf Storage,
- Auditing und Event Logs,
- Datenklassifikation.

7.2. Zugriff auf Datenzonen

Ein einfaches Modell:

- Bronze: Zugriff stark begrenzen, weil Rohdaten sensible Felder enthalten können.
- Silver: Zugriff für Data Engineering und kontrollierte Datenprodukte.
- Gold: Zugriff für BI und Fachbereiche, möglichst kuratiert.

Nicht jeder, der Gold-Berichte lesen darf, sollte Bronze-Rohdaten scannen können.

7.3. Secrets und Credentials

Spark-Jobs brauchen oft Zugriff auf:

- Object Storage,
- Datenbanken über JDBC,
- Catalogs,
- APIs,
- Key Vaults oder Secret Stores.

Anti-Patterns:

- Passwoerter im Notebook.
- Credentials in Git.
- Shared Accounts ohne Owner.
- Secrets in Logs.
- breite Storage-Rechte für alle Jobs.

Saubere Praxis:

- Secret Stores nutzen.
- technische Identitäten pro Pipeline oder Bereich trennen.
- Least Privilege für Storage-Pfade und Tabellen.
- Rotationsprozesse testen.

7.4. Web UI, History Server und Logs

Spark UI und History Server können sensible Informationen enthalten:

- SQL-Text,
- Pfade,
- Konfigurationen,
- Fehlermeldungen,
- Nutzerinformationen,
- teilweise Datenbeispiele in Logs.

Deshalb brauchen auch Diagnoseoberflaechen:

- Authentifizierung,
- Autorisierung,
- TLS,
- Log-Aufbewahrungsregeln,
- Maskierung sensibler Parameter.

7.5. Governance für Tabellen

Governance-Fragen pro Spark-SQL-Tabelle:

- Wer ist Owner?
- Welche Datenklasse liegt vor?
- Welche Quellsysteme fliessen ein?
- Welche Transformation erzeugt die Tabelle?

- Welche Consumers hängen daran?
- Wie lange werden Daten gespeichert?
- Welche SLA oder SLO gelten?
- Welche Qualitaetschecks sind verpflichtend?

Ohne diese Antworten wird Spark SQL schnell zu einer Sammlung technischer Tabellen ohne fachliche Verbindlichkeit.

7.6. Compliance und Reproduzierbarkeit

Für auditierbare Pipelines wichtig:

- Code-Version je Lauf.
- Input-Datenbereich je Lauf.
- erzeugte Output-Partitionen.
- Zeilenanzahlen und Summen.
- Fehler- und Wiederanlaufprotokolle.
- Berechtigungsmodell.
- Aufbewahrung und Loeschkonzepte.

7.7. Mini-Projekt: Security-Modell

Für das Verkaufs-Lakehouse:

- Bronze-Zugriff nur für Data Engineering und Plattformteam.
- Silver-Zugriff für validierte Transformationsjobs.
- Gold-Zugriff für BI-Gruppen nach Fachbereich.
- Produkt- und Kundendaten werden klassifiziert.
- Notebooks dürfen keine Secrets enthalten.
- History-Server-Zugriff ist rollenbasiert.

7.8. So setzt du es um - Checkliste

- Aktiviere Authentifizierung und Verschluesselung passend zum Deployment.
- Schuetze Spark UI, History Server und Event Logs.
- Trenne technische Identitäten nach Workload.
- Klassifiziere Tabellen und Spalten mit sensiblen Daten.
- Dokumentiere Owner, Lineage und Aufbewahrung.
- Prüfe regelmäßig, ob Explorationsrechte zu breit geworden sind.

---

8. Weiterführende Links und Uebungsaufgaben

8.1. Offizielle Kernquellen (Apache Spark)

Dokumentationsueberblick:

- Apache Spark Documentation
- Spark SQL Reference
- Spark SQL and DataFrames Guide

- Spark SQL Data Sources
- Spark JDBC Data Source
- Spark SQL Performance Tuning
- Distributed SQL Engine
- Monitoring and Instrumentation
- Spark Security

8.2. Offizielle Kernquellen (Microsoft Fabric)

Fabric-Kontext:

- Microsoft Fabric Lakehouse Overview
- Create a Lakehouse in Microsoft Fabric
- SQL Analytics Endpoint for a Lakehouse
- Lakehouse and Warehouse together
- Data Ingestion Options for a Lakehouse

8.3. Verwandte Themen auf dieser Website

- Für SQL-Grundlagen und Sprachebenen siehe SQL.
- Für Datenbank-Grundlagen siehe Datenbank.
- Für Datenintegration und Ladeprozesse siehe Datenintegration - ETL.
- Für Microsoft-Fabric-Notebooks siehe Microsoft Fabric Notebook.

8.4. Uebungsaufgaben (steigende Schwierigkeit)

1. **Basisabfrage (leicht)**
   - Lade eine kleine CSV-Datei als DataFrame.
   - Erzeuge eine Temp View.
   - Führe eine einfache Aggregation mit Spark SQL aus.
2. **Schema und Typen (leicht bis mittel)**
   - Definiere ein explizites Schema für Bestelldaten.
   - Vergleiche das Ergebnis mit automatischer Schema-Inferenz.
3. **Deduplizierung (mittel)**
   - Baue eine Deduplizierung mit `row_number()`.
   - Dokumentiere, welche Zeile gewinnt und warum.
4. **Join-Analyse (mittel)**
   - Join eine grosse Faktentabelle mit einer kleinen Dimension.
   - Vergleiche Plan und Laufzeit mit und ohne Broadcast Hint.
5. **Partitionierung (mittel)**
   - Schreibe eine Tabelle partitioniert nach Datum.
   - Miss, ob ein Datumsfilter weniger Dateien scannt.
6. **Performance-Diagnose (schwer)**
   - Erzeuge absichtlich Daten-Skew.
   - Analysiere Task-Laufzeiten in der Spark UI.

- Entwickle eine Gegenmassnahme.

7. **Betriebsmodell (schwer)**
   - Definiere ein Runbook für einen täglichen Gold-Job.
   - Nenne Metriken, Alerts, Wiederanlauf und Datenqualitaetschecks.

8.5. Abnahmekriterien für "produktionsreif"

- Tabellen haben Owner, Zweck, Zone und Datenklassifikation.
- Kritische Queries sind mit `EXPLAIN` und Laufzeitmetriken dokumentiert.
- Dateiformat, Partitionierung und Dateigroessen sind bewusst gewählt.
- Event Logs und History-Auswertung sind verfügbar.
- Security für Storage, Catalog, UI und Jobs ist eingerichtet.
- Wiederanlauf und Datenqualitaetspruefungen sind getestet.
- Kosten und Laufzeiten werden aktiv beobachtet.

8.6. Abschluss: Was "Pro-Level Spark SQL" in der Praxis bedeutet

Pro-Level heisst nicht, jede SQL-Funktion auswendig zu kennen. Pro-Level heisst:

- Du erkennst Spark SQL als verteiltes Ausfuehrungsmodell.
- Du liest Query-Plaene, Shuffles, Joins und Task-Verteilungen.
- Du modellierst Tabellen nach Datenprodukt, Zugriffsmuster und Kostenprofil.
- Du behandelst Dateigroessen, Partitionierung und Statistiken als Teil der Architektur.
- Du sicherst Betrieb, Security und Governance ab, bevor Datenprodukte kritisch werden.

Wenn diese Punkte erfüllt sind, ist Spark SQL nicht nur ein bequemes SQL-Interface auf grosse Dateien, sondern ein belastbares Werkzeug für moderne Datenplattformen.

8.7. 90-Tage-Umsetzungsfahrplan

Tage 1-30

- Kritische Datenprodukte, Owner und Zonenmodell definieren.
- Wichtigste Tabellen auf Format, Partitionierung und Dateigroessen prüfen.
- Spark UI, Event Logs und History-Auswertung verfügbar machen.

Tage 31-60

- Kritische Queries mit `EXPLAIN` dokumentieren.
- Join-Strategien, Statistiken und Shuffle-Verhalten optimieren.
- Datenqualitaetschecks und Wiederanlauf-Runbooks etablieren.

Tage 61-90

- Security- und Governance-Modell nachhaerten.
- Kostenmetriken pro Pipeline einfuehren.
- Performance-Regressionen und Dateigroessen als Release-Gates verankern.

8.8. Self-Assessment für Teams

Bewerte euer Team ehrlich mit 0 bis 2 Punkten je Aussage:

- Wir können erklären, welche Spark-SQL-Jobs für welche Datenprodukte verantwortlich sind.
- Wir lesen `EXPLAIN` und Spark UI regelmäßig.
- Wir kennen die teuersten Shuffles und Joins.
- Unsere Tabellen haben sinnvolle Partitionierung und Dateigroessen.
- Event Logs und History Server sind nutzbar.
- Secrets liegen nicht in Code oder Notebooks.
- Gold-Tabellen haben Datenqualitaetschecks.

Auswertung:

- 0-5 Punkte: hohe Betriebs- und Kostenrisiken.
- 6-10 Punkte: solide Basis, aber inkonsistent.
- 11-14 Punkte: gutes Produktionsniveau.

Über den Autor

Erhard Rainer berät kleine und mittlere Unternehmen in Österreich zu Business Intelligence, Machine Learning und KI-Agenten. Er hat 15 Jahre Erfahrung in Data Analytics, davon 10 Jahre in der Leitung von Teams und Departments in Konzernen, und ist Dozent für betriebswirtschaftliche und Data-Analytics-Fächer.

Sprechen Sie mit mir über Ihre Daten: Das Erstgespräch ist kostenlos.

https://www.thinkbigger.at/contact.html#anfrage

© 2026 Think BIGGER – Erhard Rainer. Dieses PDF gibt den Stand des Online-Artikels vom 28.04.2026 wieder; maßgeblich ist die Online-Fassung unter https://www.thinkbigger.at/wissen/spark-sql/