

An Introduction to Delta Lakes

& Delta Lake-Houses

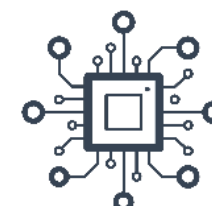
Paul Andrew | Technical Architect in Azure CoE



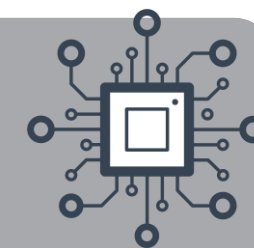
@MrPaulAndrew



In/MrPaulAndrew



MrPaulAndrew.com



An Introduction to Delta^{*} Lakes

& Delta^{*} Lake-Houses

* We are not talking about the delta of changed records since our data processing last ran.

Uppercase Δ
Lowercase δ or δ

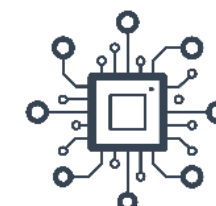
Paul Andrew | Technical Architect in Azure CoE



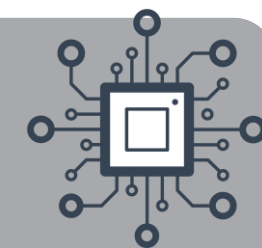
@MrPaulAndrew



In/MrPaulAndrew



MrPaulAndrew.com



<https://github.com/mrpaulandrew>

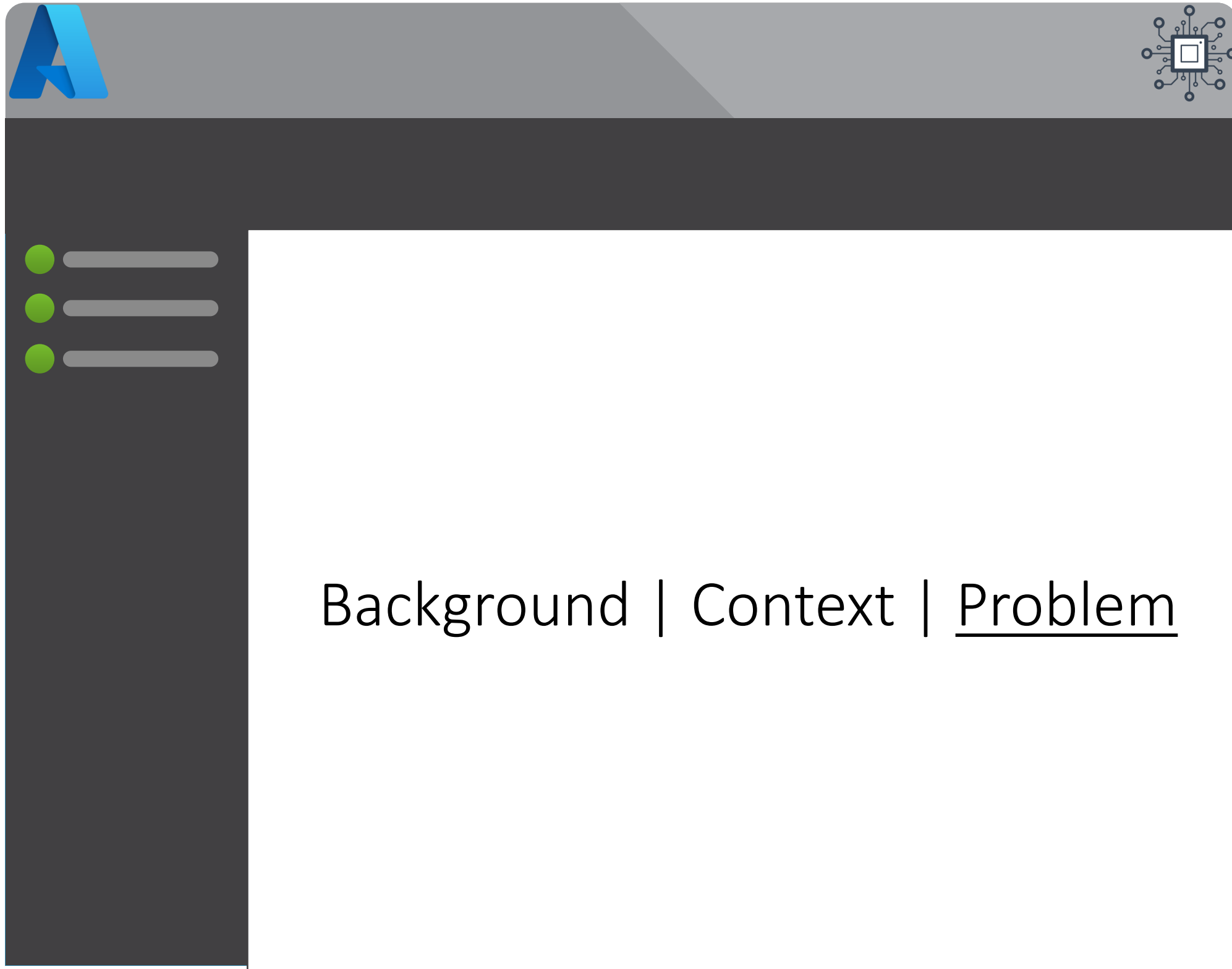
CommunityEvents

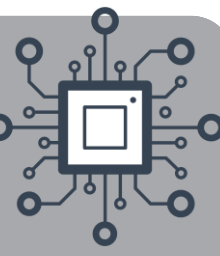
Demo code, content and slides from various community events.

● C++

[{Event/Location}-{Month}-{Year}](#)

Part 1.0 – Theory



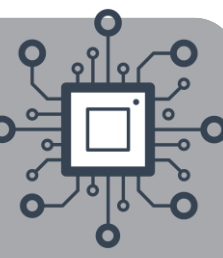


Human Traffic (1999)





ACID – In Computer Science



Atomicity
Consistency
Isolation
Durability

“is a set of properties of [database transactions](#) intended to guarantee data validity”

The screenshot shows the Wikipedia page for "ACID". The left sidebar contains the Wikipedia logo and navigation links. The main content area has tabs for "Article" and "Talk", and buttons for "Read", "Edit", and "View history". A search bar is at the top right. The article title "ACID" is prominently displayed, followed by the subtitle "From Wikipedia, the free encyclopedia". A notice box states that the article needs additional citations for verification. The main text defines ACID in computer science and mentions its origin in 1983 by Andreas Reuter and Theo Härder, building on Jim Gray's earlier work. A reference to the IBM Information Management System is also present.

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file

Tools

Article Talk

Read Edit View history

Search Wikipedia

ACID

From Wikipedia, the free encyclopedia

For other uses, see [Acid \(disambiguation\)](#).

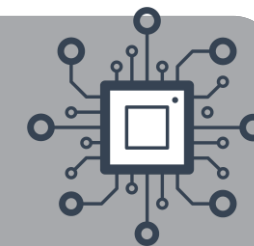
This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
Find sources: "ACID" – news · newspapers · books · scholar · JSTOR (May 2018) (Learn how and when to remove this template message)

In computer science, **ACID** (atomicity, consistency, isolation, durability) is a set of properties of [database transactions](#) intended to guarantee data validity despite errors, power failures, and other mishaps. In the context of [databases](#), a sequence of database operations that satisfies the ACID properties (which can be perceived as a single logical operation on the data) is called a *transaction*. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

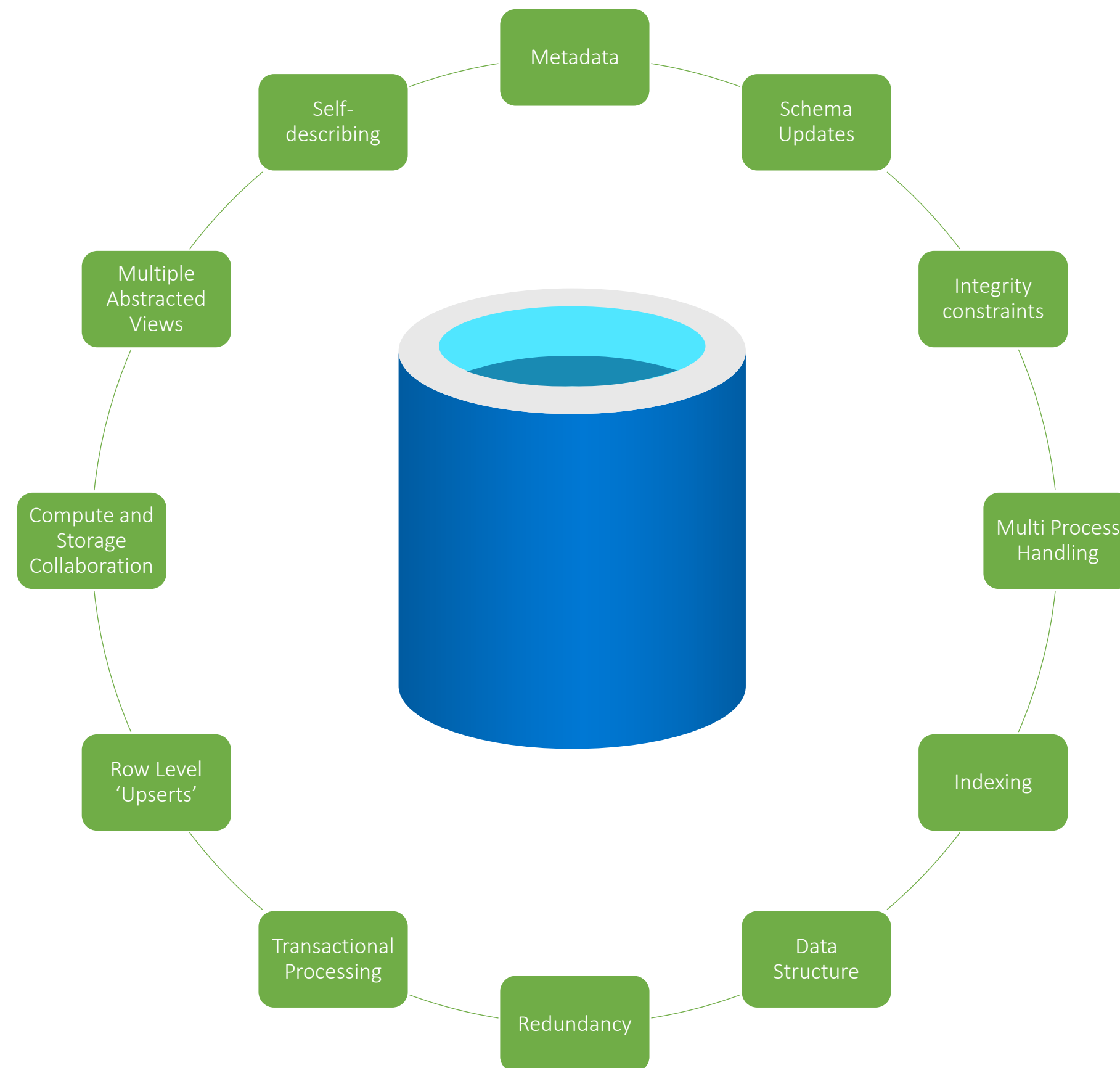
In 1983,^[1] [Andreas Reuter](#) and [Theo Härder](#) coined the acronym *ACID*, building on earlier work by [Jim Gray](#)^[2] who named atomicity, consistency, and durability, but not isolation, when characterizing the transaction concept. These four properties are the major guarantees of the transaction paradigm, which has influenced many aspects of development in database systems.

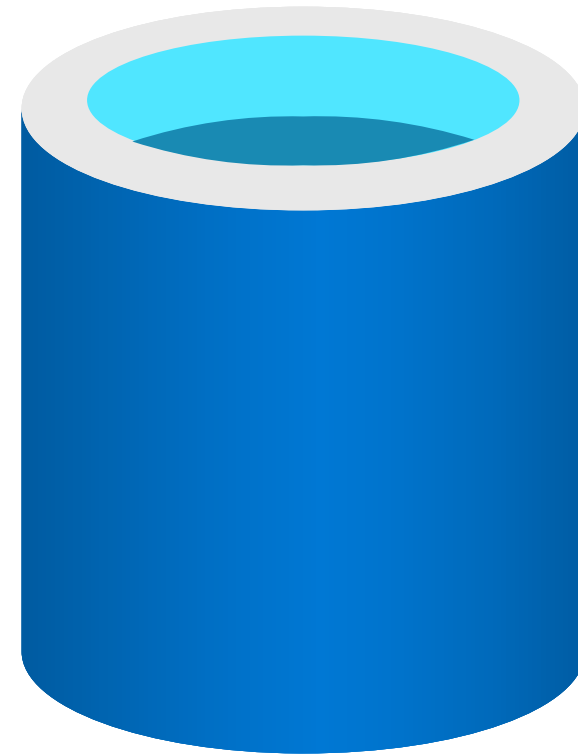
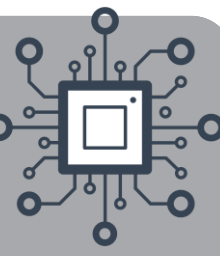
According to Gray and Reuter, the [IBM Information Management System](#) supported ACID transactions as early as 1973 (although the acronym was created later).^[3]

<https://en.wikipedia.org/wiki/ACID>



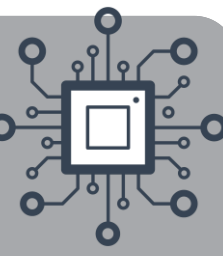
DataBase Management System







Creating a Data Warehouse



Online
Line
Transactional
Processing



Application
Data

Extract
Transform
Load



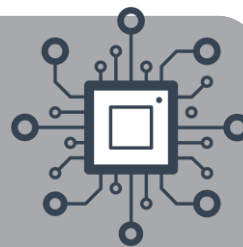
Data
Warehouse

Offline
Analytical
Transactional
Processing





Creating a Data Warehouse



“a system for reporting and data analysis”



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute

Help
Learn to edit
Community portal
Recent changes
Upload file

Tools

What links here
Related changes
Special pages
Permanent link

Article **Talk**

Read **Edit** View history

Search Wikipedia

Data warehouse

From Wikipedia, the free encyclopedia

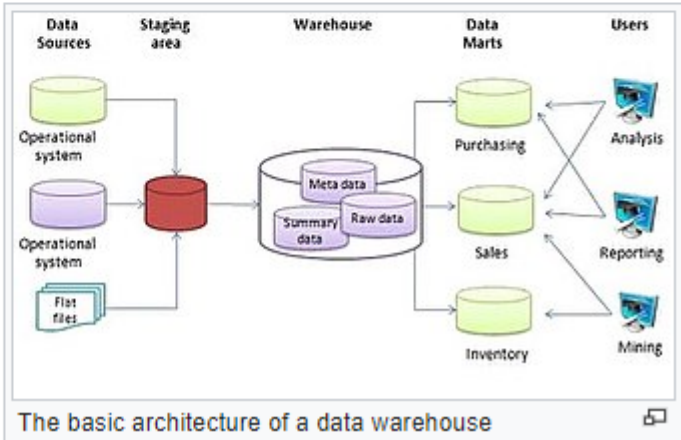
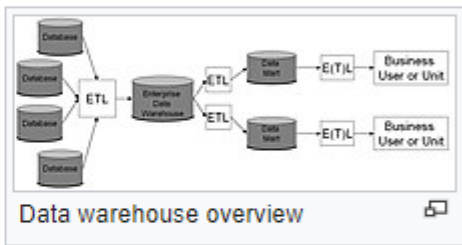
In **computing**, a **data warehouse** (**DW** or **DWH**), also known as an **enterprise data warehouse** (**EDW**), is a system used for **reporting** and data analysis and is considered a core component of **business intelligence**.^[1] DWs are central repositories of integrated data from one or more disparate sources. They store current and historical data in one single place^[2] that are used for creating analytical reports for workers throughout the enterprise.^[3]

The data stored in the warehouse is **uploaded** from the **operational systems** (such as marketing or sales). The data may pass through an **operational data store** and may require **data cleansing**^[2] for additional operations to ensure **data quality** before it is used in the DW for reporting.

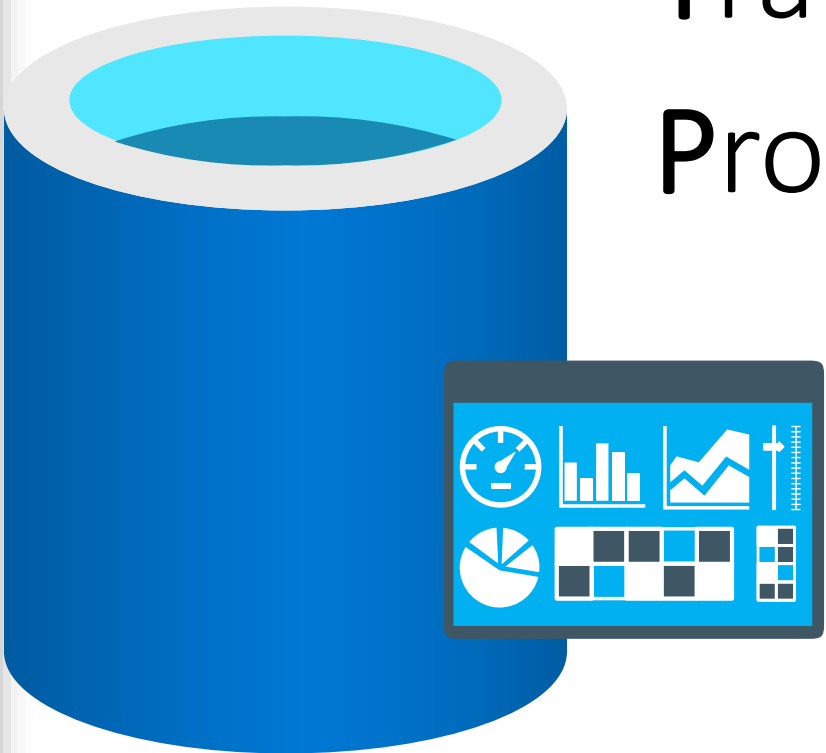
Extract, transform, load (ETL) and extract, load, transform (ELT) are the two main approaches used to build a data warehouse system.

Contents [hide]

- ETL-based data warehousing
- ELT-based data warehousing
- Benefits
- Generic
- Related systems (data mart, OLAPS, OLTP, predictive analytics)

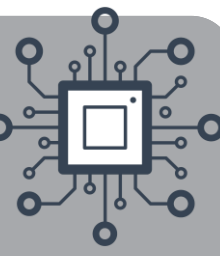


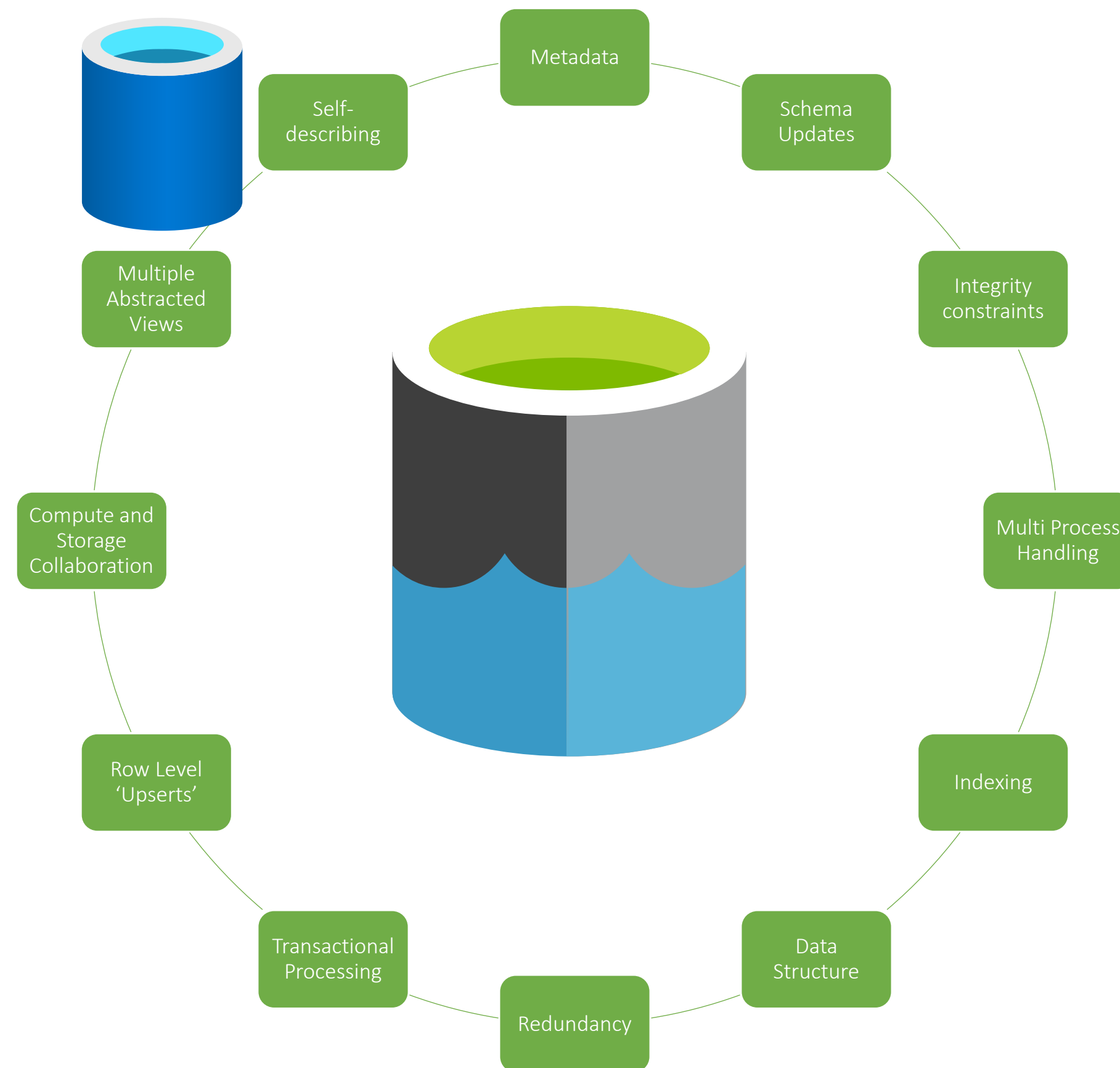
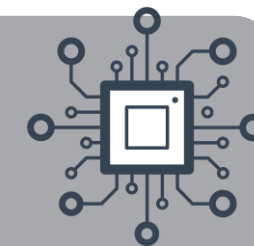
Offline
Analytical
Transactional
Processing

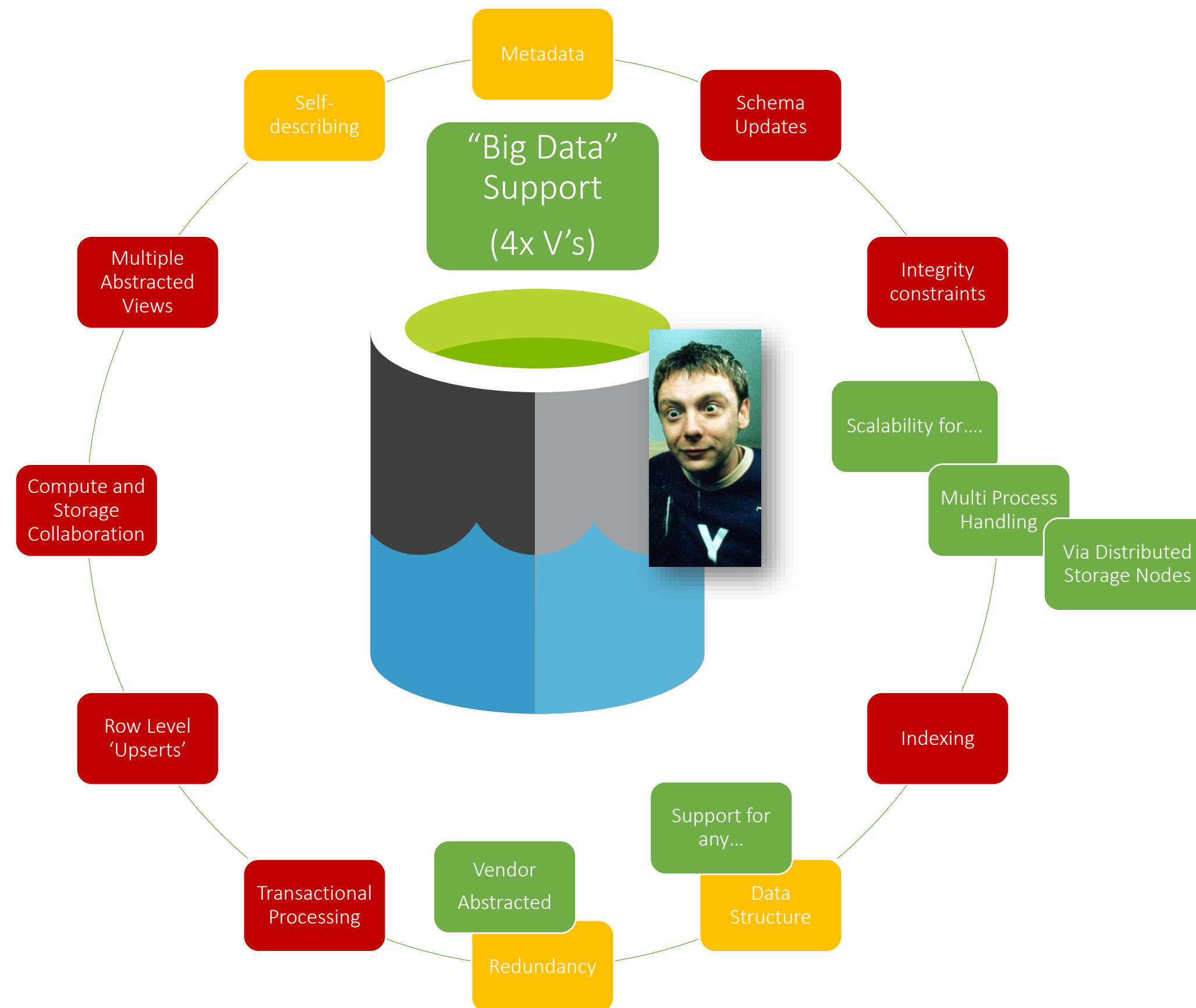
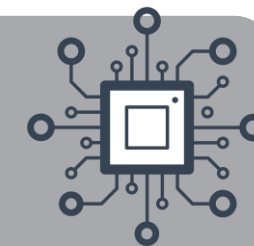


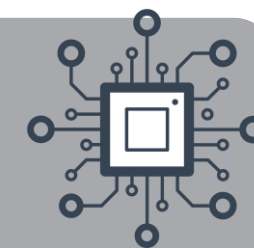
Data Warehouse

https://en.wikipedia.org/wiki/Data_warehouse









Data Lakes are good, but they still lack some of the basic ACID functionality needed for data processing.

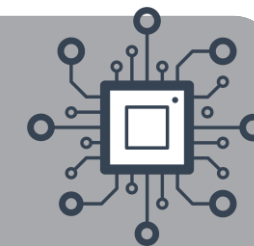
We are/were trying to use Data Lakes for everything (to replace Databases).



VS



Scales Up	Scales Out
Natural Home for Structured Data	Any Data Structure
Storage Limits	No Storage Limits
Transactional Resilience	No Transactional Handling
Storage & Compute Coupled	Storage & Compute Decoupled



Data Lakes are good, but they still lack some of the basic ACID functionality needed for data processing.

We are/were trying to use Data Lakes for everything (to replace Databases).

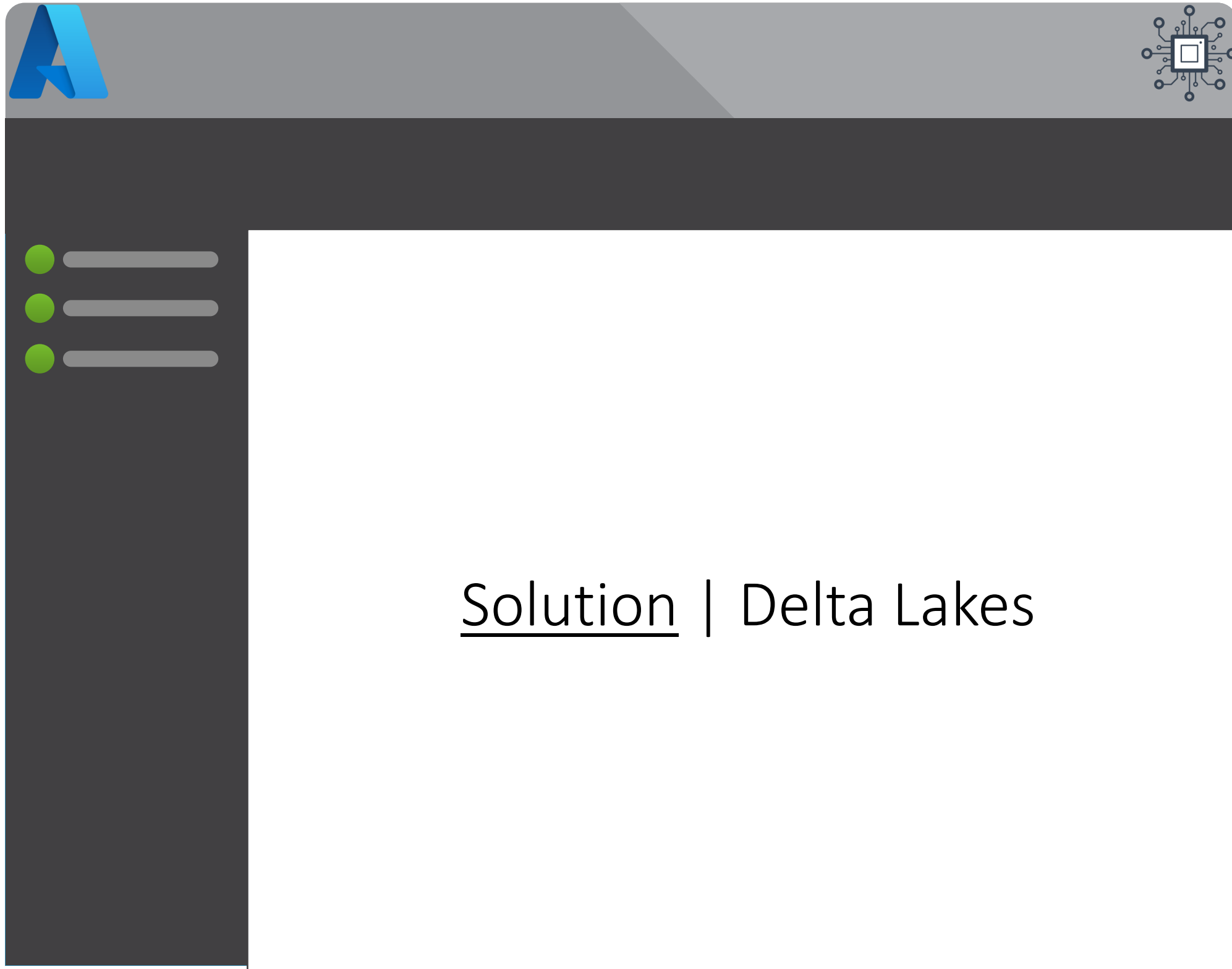


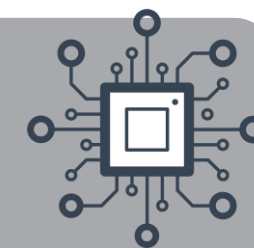
VS



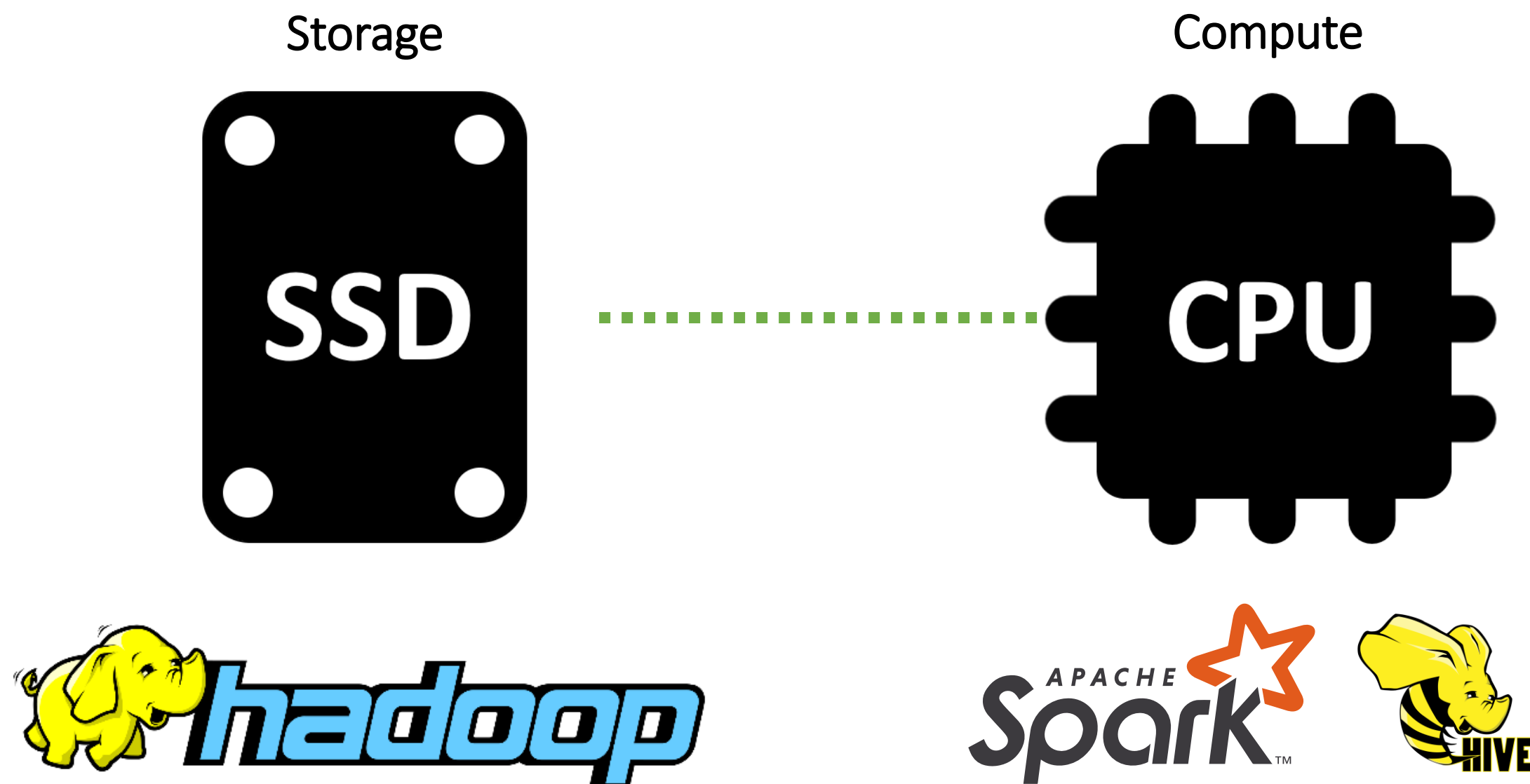
Scales Up	Scales Out
Natural Home for Structured Data	Any Data Structure
Storage Limits	No Storage Limits
Transactional Resilience	No Transactional Handling
Storage & Compute Coupled	Storage & Compute Decoupled

Part 1.1 – Theory

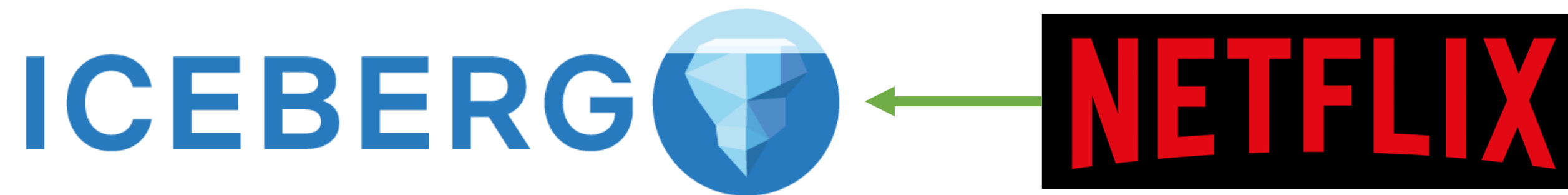
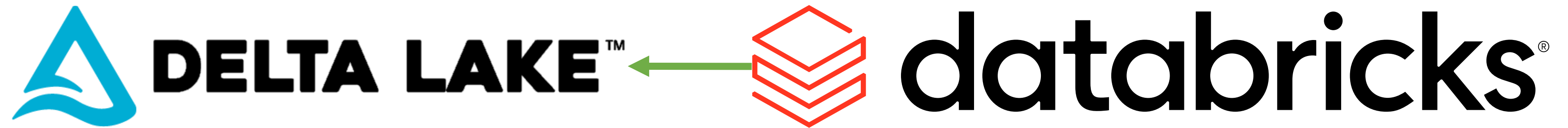
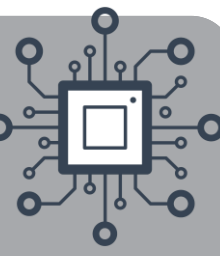




Enable ACID transactional support for Data Lakes...

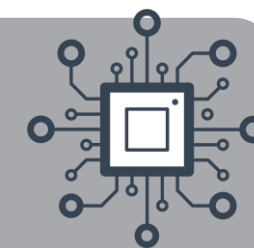


Storage & Compute ~~Decoupled~~ Working Together Again As Friends!





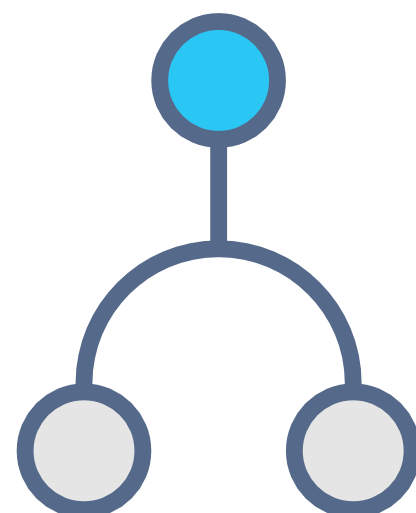
What is Delta Lake?



DELTA LAKE™

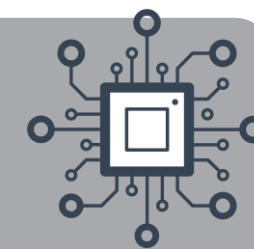


databricks®

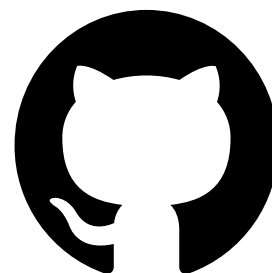




What is Delta Lake?

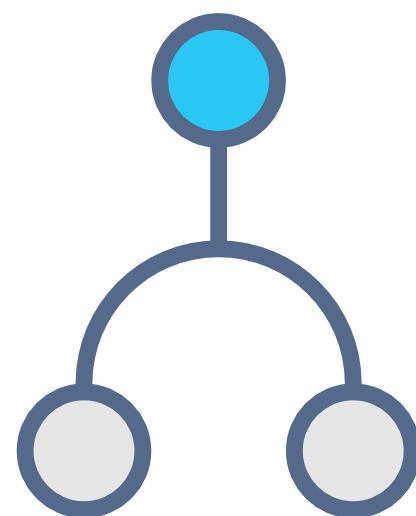


DELTA LAKE™



<https://delta.io>

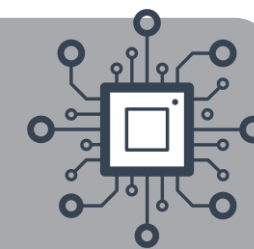
<https://github.com/delta-io/delta>



databricks®



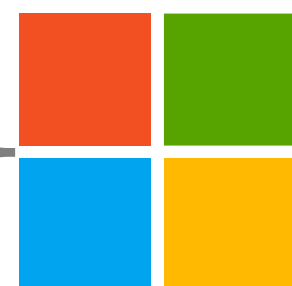
What is Delta Lake?



Open Source

Microsoft
Implementation

Databricks
Implementation

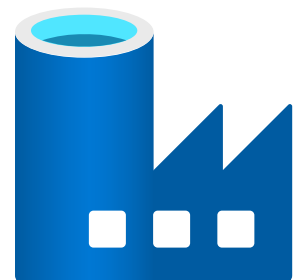


"Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark™ and big data workloads."

Synapse
Analytics



Data Factory
Data Flows

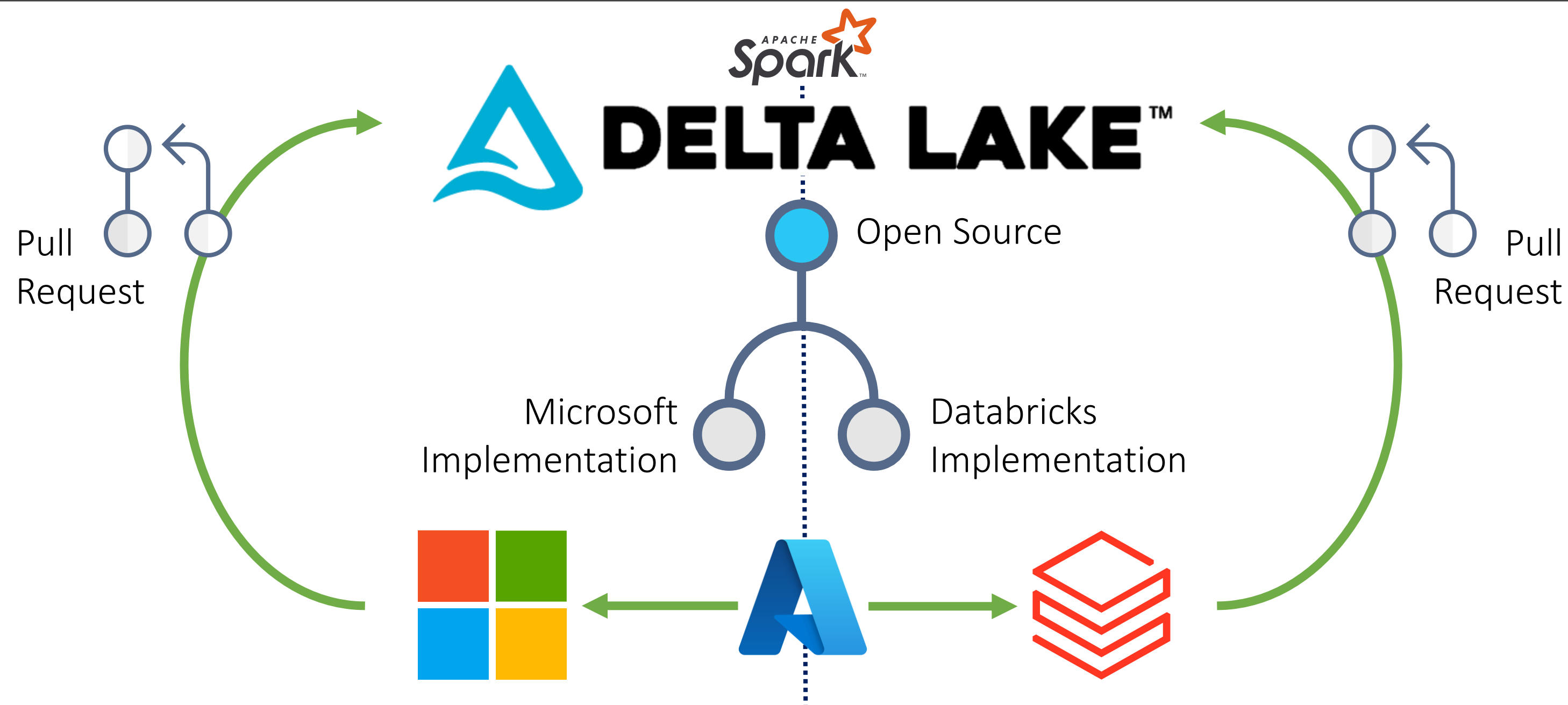
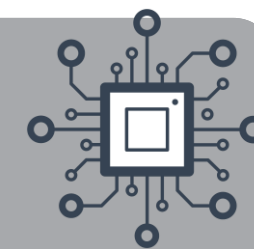


HD Insight
Spark





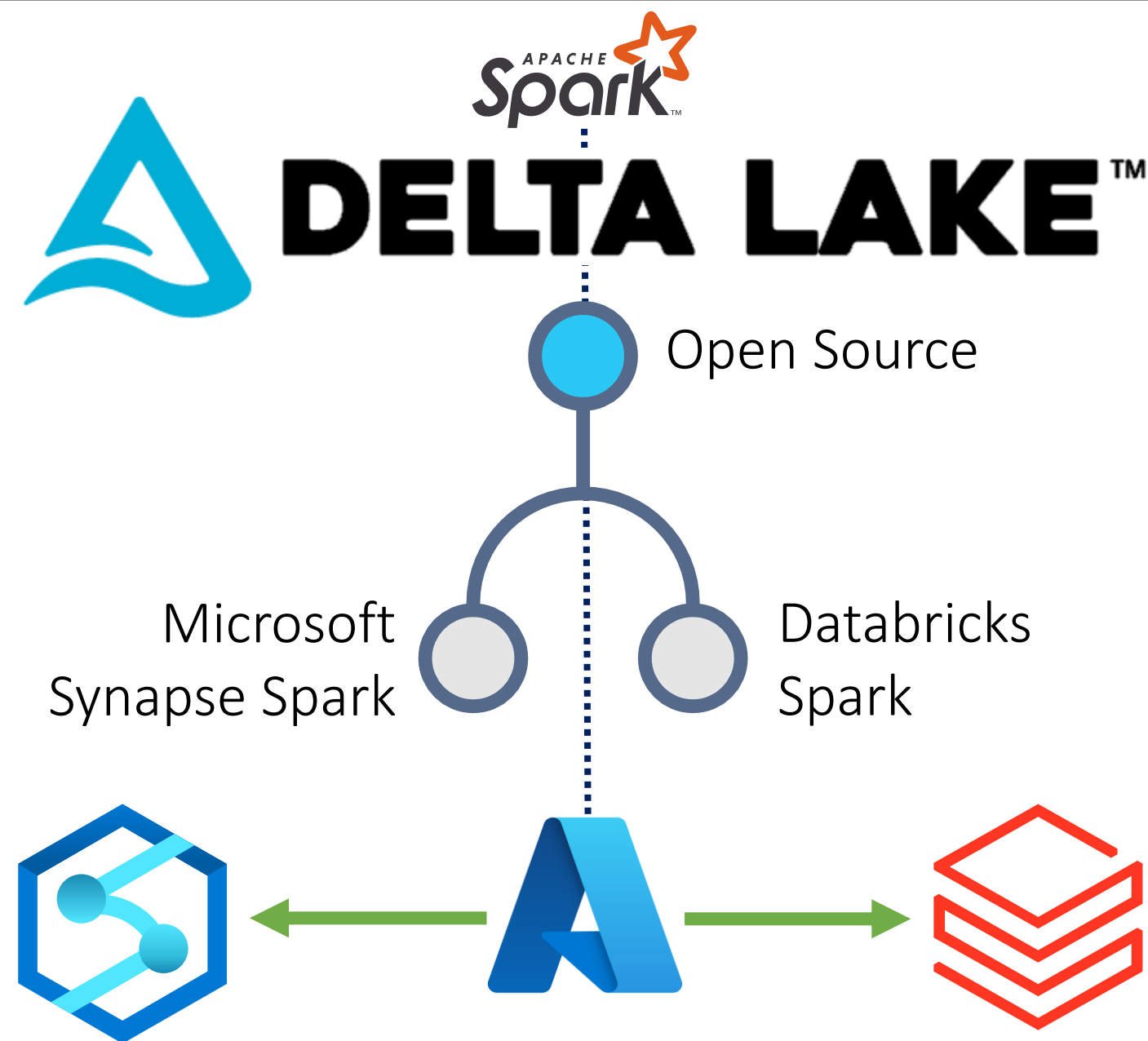
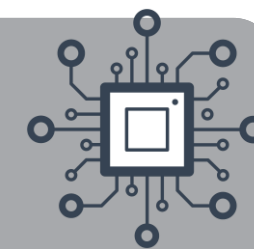
What is Delta Lake?



"Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark™ and big data workloads."

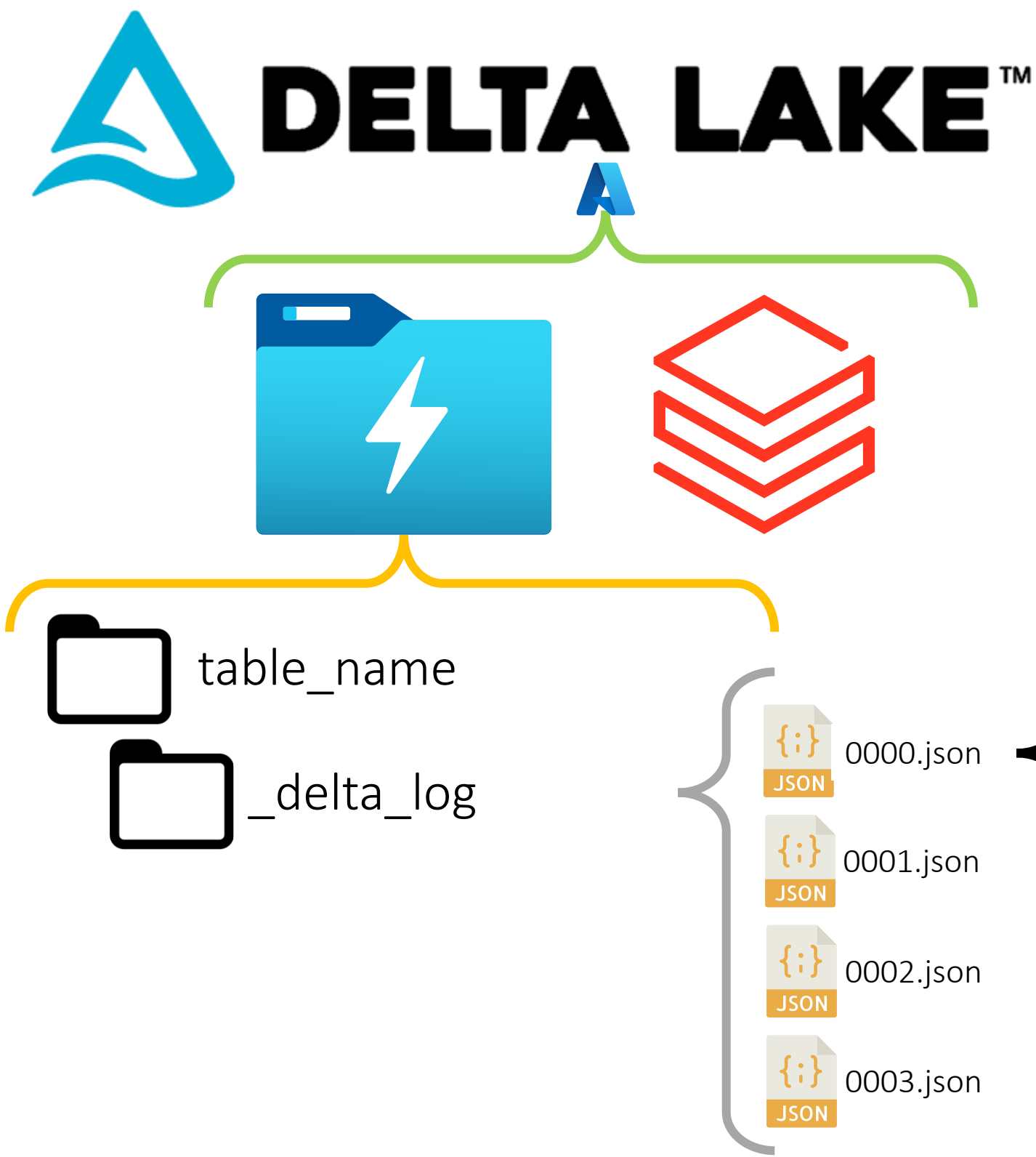
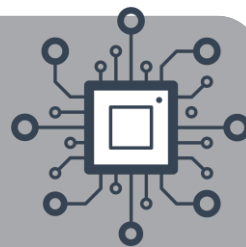


Which Spark Implementation is Better?



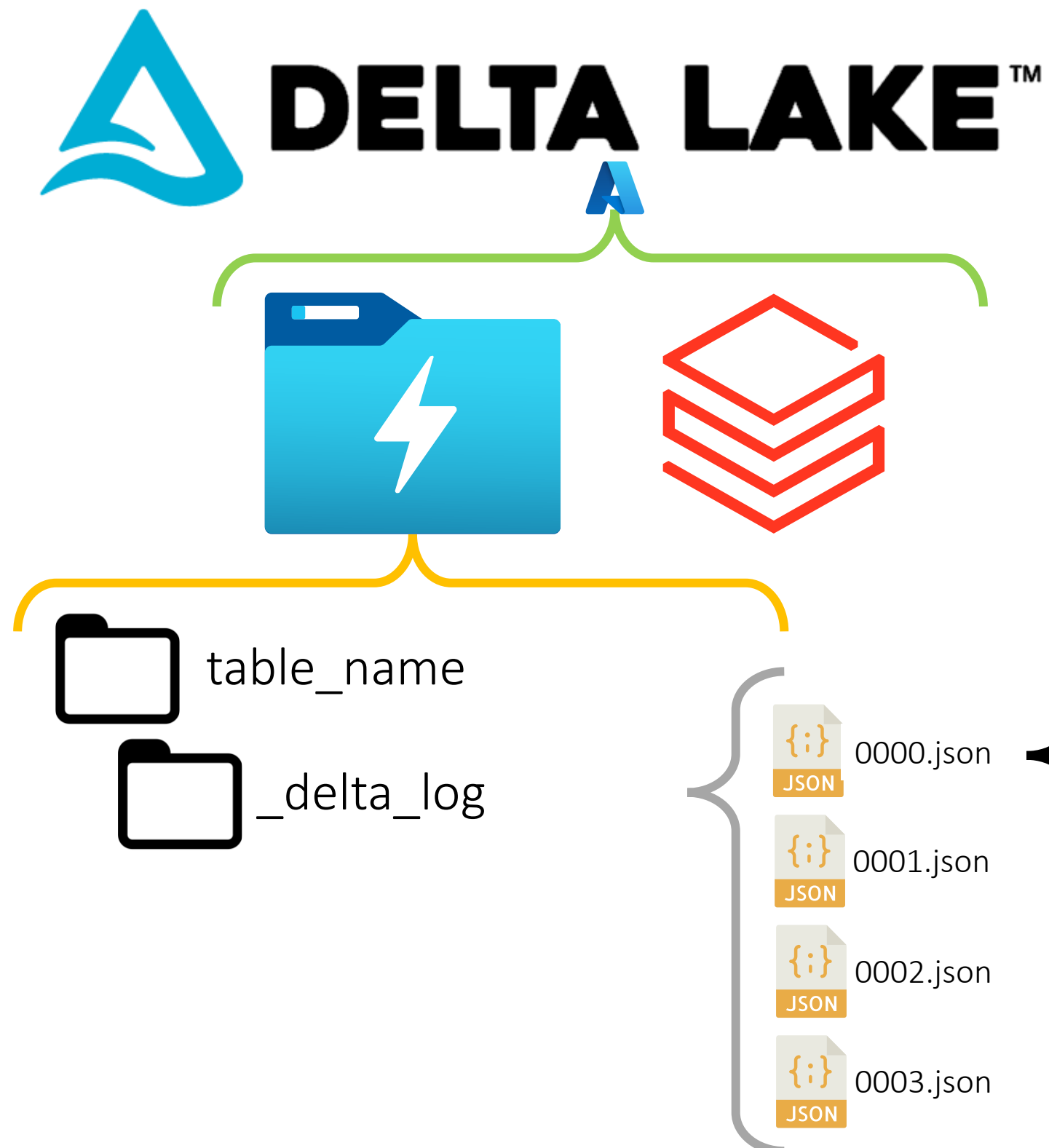
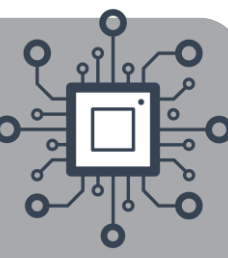


Delta Table - Transaction Log



```
{
  "commitInfo":
  {
    "timestamp":1628596034417,
    "operation":"WRITE",
    "operationParameters":
    {
      "mode":"ErrorIfExists",
      "partitionBy":"[]"
    },
    "isBlindAppend":true,
    "operationMetrics":
    {
      "numFiles":"6",
      "numOutputBytes":"2407",
      "numOutputRows":"5"
    }
  }
}
{"protocol":{"minReaderVersion":1,"minWriterVersion":2}}
{
  "metaData":
  {
    "id":"58e5de01-de72-4d5b-a208-d0b4ae919efe",
    "format":
    {
      "provider":"parquet",
      "options":{}
    },
    "schemaString":
      "{\n  \"type\": \"struct\",\n  \"fields\": [\n    {\n      \"name\": \"id\",\n      \"type\": \"long\",\n      \"nullable\": true,\n      \"metadata\": {}\n    }\n  ]\n}",
    "partitionColumns":[],
    "configuration":{},
    "createTime":1628596029470
  }
}
{"add":{"path":"part-00000.snappy.parquet","size":262,"modificationTime":1628596034000}}
{"add":{"path":"part-00001.snappy.parquet","size":429,"modificationTime":1628596034000}}
{"txn":{"appId":"731b2c96-bf64-445c-8ca8-cd6cad6735e2","lastUpdated":1628596094191}}
{"add":{"path":"part-00000.snappy.parquet","size":429,"modificationTime":1628596094000}}
{"add":{"path":"part-00001.snappy.parquet","size":429,"modificationTime":1628596094000}}
{"remove":{"path":"part-00150.snappy.parquet","deletionTimestamp":1628596098597}}
{"remove":{"path":"part-00128.snappy.parquet","deletionTimestamp":1628596098597}}
```


Delta Table - Transaction Log



Breaking Down Transactions Into Atomic Commits

Whenever a user performs an operation to modify a table (such as an INSERT, UPDATE or DELETE), Delta Lake breaks that operation down into a series of discrete steps composed of one or more of the actions below.

Add file – adds a data file.

Remove file – removes a data file.

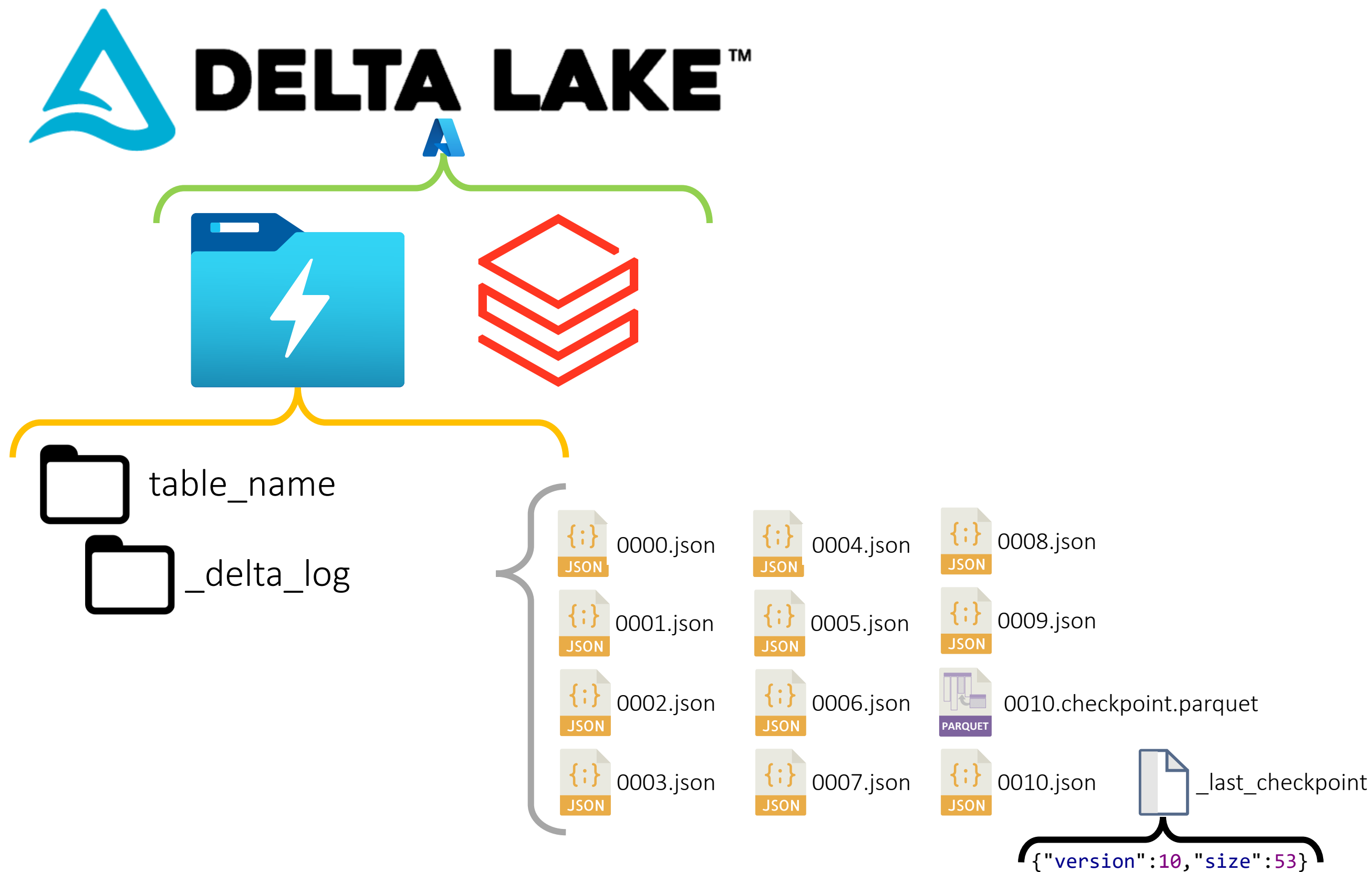
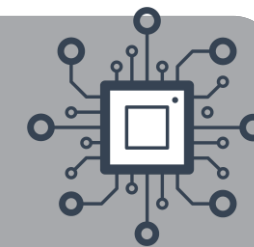
Update metadata – Updates the table's metadata (e.g., changing the table's name, schema or partitioning).

Set transaction – Records that a structured streaming job has committed a micro-batch with the given ID.

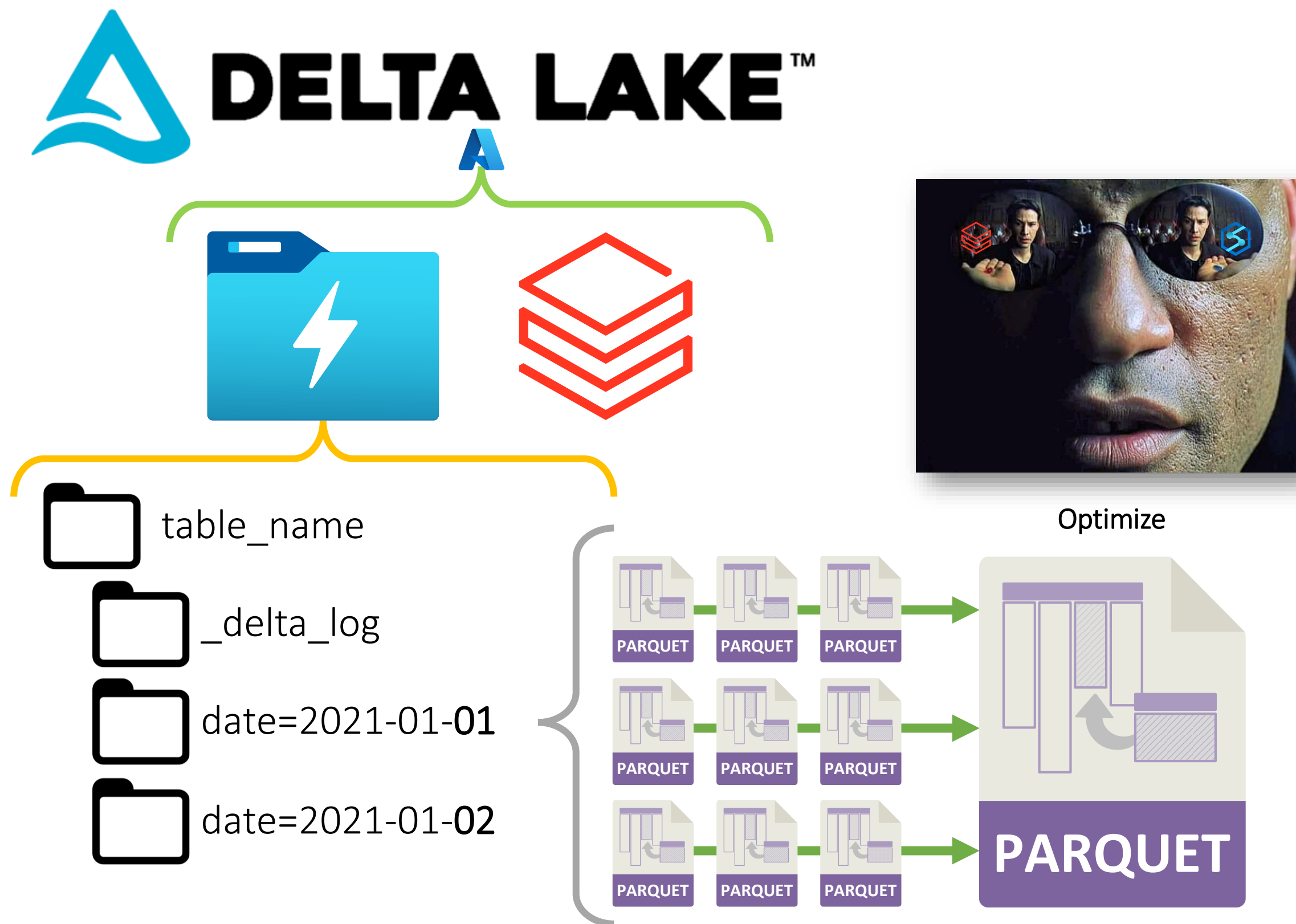
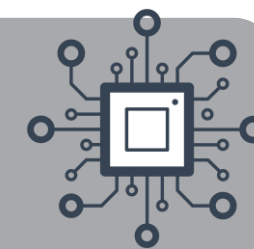
Change protocol – enables new features by switching the Delta Lake transaction log to the newest software protocol.

Commit info – Contains information around the commit, which operation was made, from where and at what time.

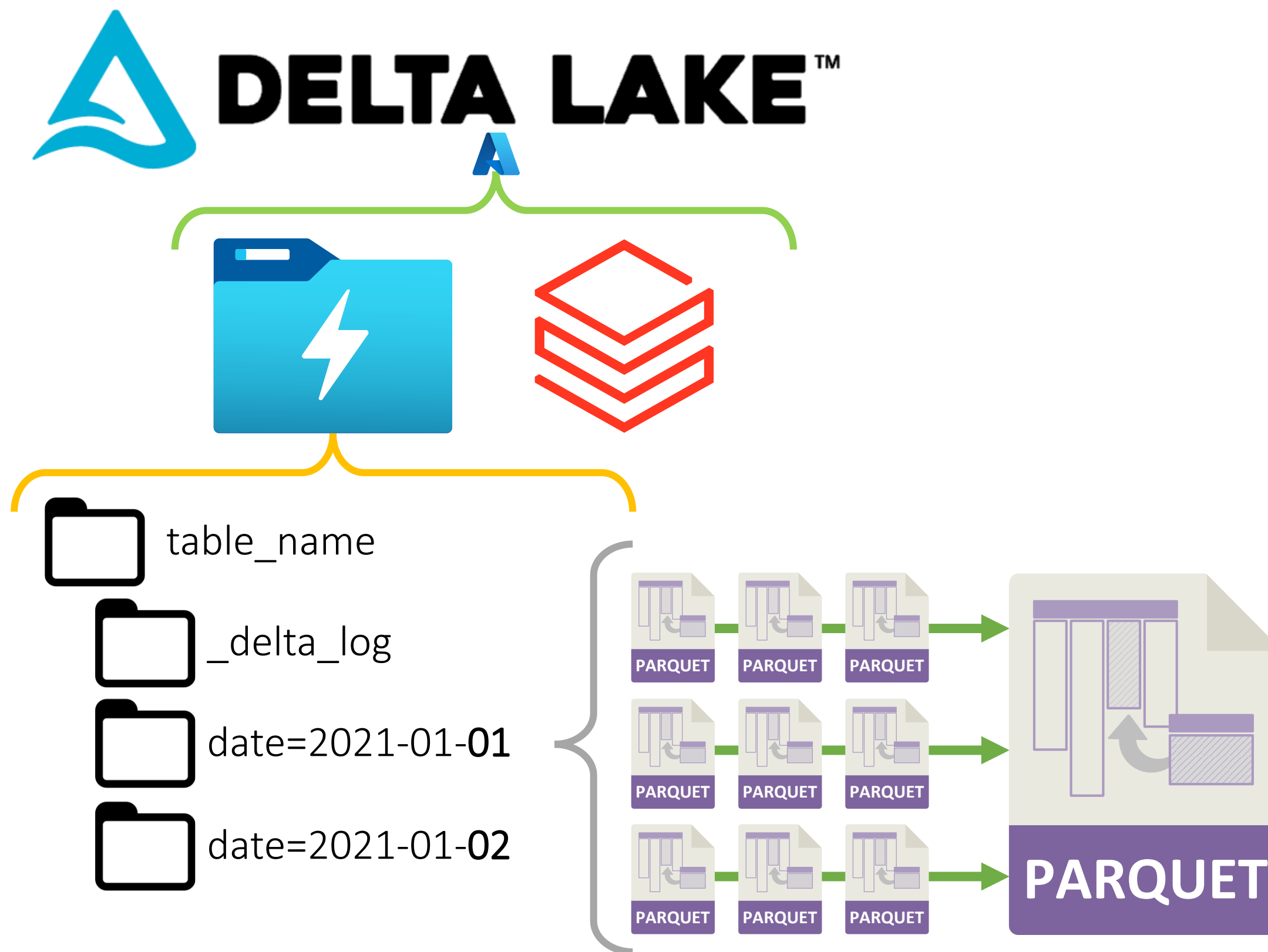
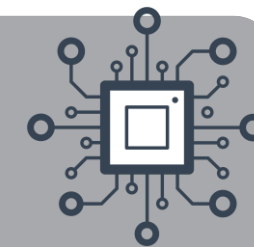
Delta Table - Transaction Log



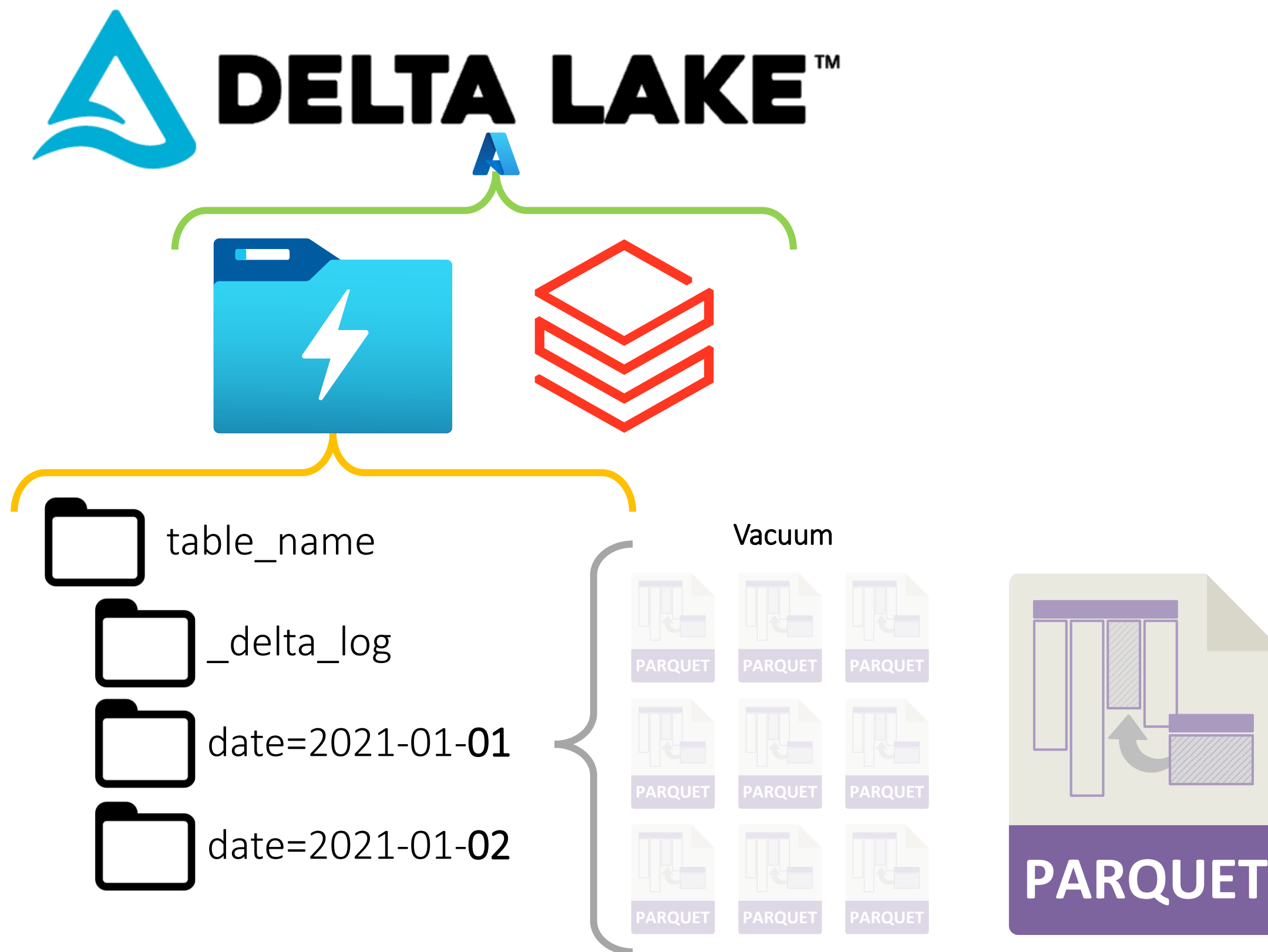
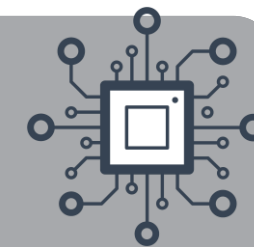
Delta Table - Transaction Log



Delta Table - Transaction Log

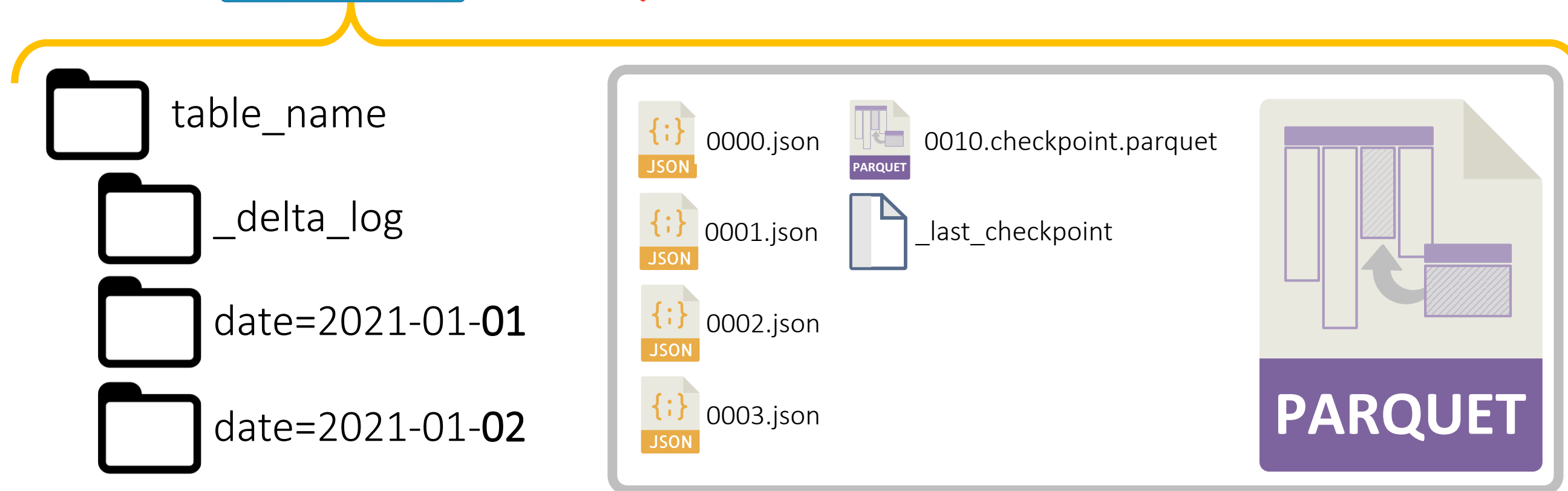
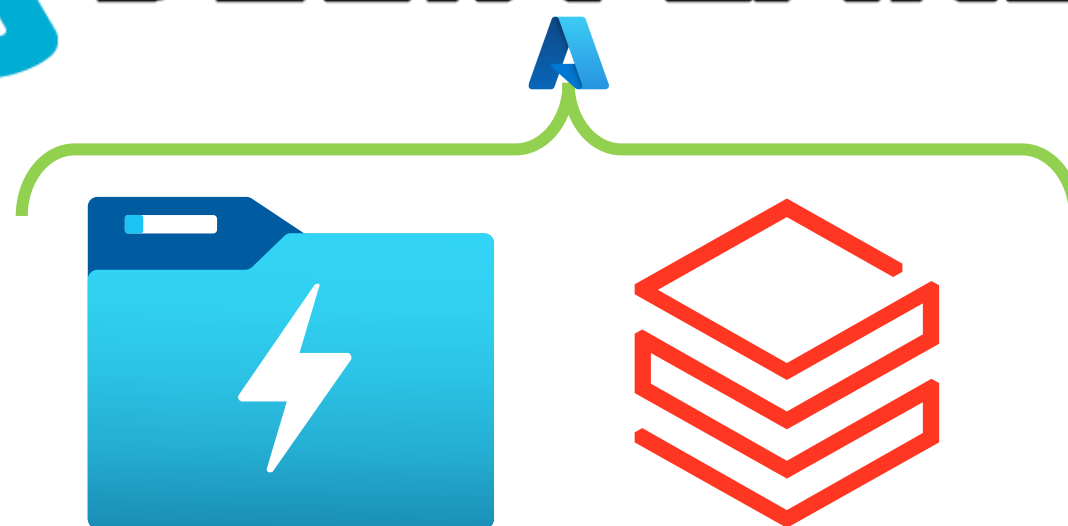
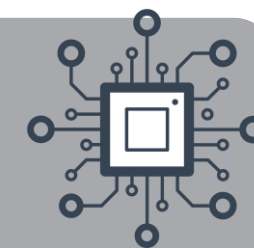


Delta Table - Transaction Log



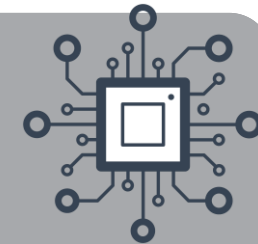


Delta Table – On Disk






Delta Table - Querying



DELTA LAKE™



**DELTA LAKE**
WITH SPARK SQL

Delta Lake is an open source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.

[delta.io](#) | [Documentation](#) | [GitHub](#) | [Delta Lake on Databricks](#)

CREATE AND QUERY DELTA TABLES

Create and use managed database

```
-- Managed database is saved in the Hive metastore.
-- Default database is named 'default'.
DROP DATABASE IF EXISTS dbName;
CREATE DATABASE dbName;
USE dbName -- This command avoids having to specify
dbName.tableName every time instead of just tableName.
```

Query Delta Lake table by table name (preferred)

```
/* You can refer to Delta Tables by table name, or by
path. Table name is the preferred way, since named tables
are managed in the Hive Metastore (i.e., when you DROP a
named table, the data is dropped also -- not the case for
path-based tables.) */
SELECT * FROM {dbName} tableName
```

Query Delta Lake table by path

```
SELECT * FROM delta.`path/to/delta_table` -- note backticks
```

Convert Parquet table to Delta Lake format in place

```
-- by table name
CONVERT TO DELTA {dbName}.tableName
[LOCATION 'path/to/table']
-- path-based tables
CONVERT TO DELTA parquet.`path/to/table` -- note backticks
[PARTITIONED BY (col_name1 col_type1, col_name2
col_type2)]
```

Create Delta Lake table as SELECT* with no upfront schema definition

```
CREATE TABLE {dbName} tableName
USING DELTA
AS SELECT * FROM tableName | parquet.`path/to/data`
[LOCATION 'path/to/table']
-- using location = unmanaged table
```

Create table, define schema explicitly with SQL DDL

```
CREATE TABLE {dbName} tableName (
  id INT NOT NULL,
  name STRING,
  date DATE,
  last_price FLOAT)
USING DELTA
[PARTITIONED BY (time, date)] -- optional
```

Copy new data into Delta Lake table (with idempotent retries)

```
COPY INTO {dbName} targetTable
FROM (SELECT * FROM "path/to/table")
FILEFORMAT = DELTA -- or CSV, Parquet, ORC, JSON, etc.
```

Provided to the open source community by Databricks
© Databricks 2021. All rights reserved. Apache, Apache Spark, Spark and the Spark logo are
trademarks of the Apache Software Foundation.

DELTA LAKE DDL/DML: UPDATE, DELETE, MERGE, ALTER TABLE

Update rows that match a predicate condition

```
UPDATE tableName SET event = 'click' WHERE event = 'clk'
```

Delete rows that match a predicate condition

```
DELETE FROM tableName WHERE "date" < "2017-01-01"
```

Insert values directly into table

```
INSERT INTO TABLE tableName VALUES (
  (8003, "Tim Jones", "2020-12-18", 3.875),
  (8004, "Tim Jones", "2020-12-20", 3.750)
);
-- Insert using SELECT statement
INSERT INTO tableName SELECT * FROM sourceTable
-- Atomically replace all data in table with new values
INSERT OVERWRITE tableName loan_by_state_delta VALUES (...)
```

Upsert (update + insert) using MERGE

```
MERGE INTO target
USING updates
ON target.id = updates.id
WHEN MATCHED AND target.delete_flag = "true" THEN
  DELETE
WHEN NOT MATCHED THEN
  UPDATE SET * -- star notation means all columns
WHEN NOT MATCHED THEN
  INSERT (date, id, data) -- or, use INSERT *
VALUES (date, id, data)
```

Insert with Deduplication using MERGE

```
MERGE INTO logs
USING dedupedLogs
ON logs.uniqueId = dedupedLogs.uniqueId
WHEN NOT MATCHED
THEN INSERT *
```

Alter table schema—add columns

```
ALTER TABLE tableName ADD COLUMNS (
  col_name data_type
  [FIRST|AFTER col_name])
```

Alter table—add constraint

```
-- Add "not null" constraint:
ALTER TABLE tableName CHANGE COLUMN col_name SET NOT NULL
-- Add "check" constraint:
ALTER TABLE tableName
ADD CONSTRAINT dateWithinRange CHECK date > "1900-01-01"
-- Drop constraint:
ALTER TABLE tableName DROP CONSTRAINT dateWithinRange
```

TIME TRAVEL (CONTINUED)

Rollback a table to an earlier version

```
-- RESTORE requires Delta Lake version 0.7.0+ & DBR 7.4+.
RESTORE tableName VERSION AS OF 0
RESTORE tableName TIMESTAMP AS OF "2020-12-18"
```

UTILITY METHODS

View table details

```
DESCRIBE DETAIL tableName
DESCRIBE FORMATTED tableName
```

Delete old files with Vacuum

```
VACUUM tableName [RETAIN num HOURS] [DRY RUN]
```

Clone a Delta Lake table

```
-- Deep clones copy data from source, shallow clones don't.
CREATE TABLE {dbName} targetName
[SHALLOW | DEEP] CLONE sourceName [VERSION AS OF 0]
[LOCATION 'path/to/table']
-- specify location only for path-based tables
```

Interoperability with Python / DataFrames

```
-- Read name-based table from Hive metastore into DataFrame
df = spark.table("tableName")
-- Read path-based table into DataFrame
df = spark.read.format("delta").load("path/to/delta_table")
```

Run SQL queries from Python

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`path/to/delta_table`")
```

Modify data retention settings for Delta Lake table

```
-- logRetentionDuration -> how long transaction log history
is kept, deletedFileRetentionDuration -> how long ago a file
must have been deleted before being a candidate for VACUUM.
ALTER TABLE tableName
SET TBLPROPERTIES(
  delta.logRetentionDuration = "interval 30 days",
  delta.deletedFileRetentionDuration = "interval 7 days"
);
SHOW TBLPROPERTIES tableName;
```

PERFORMANCE OPTIMIZATIONS

Compact data files with Optimize and Z-Order

```
>Databricks Delta Lake feature
OPTIMIZE tableName
[ORDER BY (colNameA, colNameB)]
```

Auto-optimize tables

```
>Databricks Delta Lake feature
ALTER TABLE [table_name | delta.`path/to/delta_table`]
SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true)
```

Cache frequently queried data in Delta Cache

```
>Databricks Delta Lake feature
CACHE SELECT * FROM tableName
-- or:
CACHE SELECT colA, colB FROM tableName WHERE colNameA > 0
```





Delta Lake is an open source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.

delta.io | [Documentation](#) | [GitHub](#) | [API reference](#) | [Databricks](#)

READS AND WRITES WITH DELTA LAKE

Read data from pandas DataFrame

```
df = spark.createDataFrame(pdf)
# where pdf is a pandas DF
# then save DataFrame in Delta Lake format as shown below
```

Read data using Apache Spark™

```
# read by path
df = (spark.read.format("parquet"|"csv"|"json"|etc.)
      .load("/path/to/delta_table"))

# read by table name
df = spark.table("events")
```

Save DataFrame in Delta Lake format

```
(df.write.format("delta")
  .mode("append"|"overwrite")
  .partitionBy("date") # optional
  .option("mergeSchema", "true") # option - evolve schema
  .saveAsTable("events") | .save("/path/to/delta_table")
)
```

Streaming reads (Delta table as streaming source)

```
# by path or by table name
df = (spark.readStream
      .format("delta")
      .schema(schema)
      .table("events") | .load("/delta/events")
)
```

Streaming writes (Delta table as a sink)

```
streamingQuery = (
df.writeStream.format("delta")
  .outputMode("append"|"update"|"complete")
  .option("checkpointLocation", "/path/to/checkpoints")
  .trigger(once=True|processingTime="10 seconds")
  .table("events") | .start("/delta/events")
)
```

CONVERT PARQUET TO DELTA LAKE

Convert Parquet table to Delta Lake format in place

```
deltaTable = DeltaTable.convertToDelta(spark,
"parquet.`/path/to/parquet_table`")
```

```
partitionedDeltaTable = DeltaTable.convertToDelta(spark,
"parquet.`/path/to/parquet_table`", "part int")
```

Provided to the open source community by Databricks

© Databricks 2021. All rights reserved. Apache, Apache Spark, Spark and the Spark logo are trademarks of the Apache Software Foundation.

WORKING WITH DELTA TABLES

```
# A DeltaTable is the entry point for interacting with
tables programmatically in Python – for example, to
perform updates or deletes.
from delta.tables import *
```

```
deltaTable = DeltaTable.forName(spark, tableName)
deltaTable = DeltaTable.forPath(spark,
delta.`path/to/table`)
```

DELTA LAKE DDL/DML: UPDATES, DELETES, INSERTS, MERGES

Delete rows that match a predicate condition

```
# predicate using SQL formatted string
deltaTable.delete("date < '2017-01-01'")
# predicate using Spark SQL functions
deltaTable.delete(col("date") < "2017-01-01")
```

Update rows that match a predicate condition

```
# predicate using SQL formatted string
deltaTable.update(condition = "eventType = 'clk'",
set = { "eventType": "'click'" } )
# predicate using Spark SQL functions
deltaTable.update(condition = col("eventType") == "clk",
set = { "eventType": lit("click") } )
```

Upsert (update + insert) using MERGE

```
# Available options for merges [see documentation for
details]:
.whenMatchedUpdate(...) | .whenMatchedUpdateAll(...) |
.whenNotMatchedInsert(...) | .whenMatchedDelete(...)
(deltaTable.alias("target").merge(
  source = updatesDF.alias("updates"),
  condition = "target.eventId = updates.eventId")
  .whenMatchedUpdateAll()
  .whenNotMatchedInsert(
    values = {
      "date": "updates.date",
      "eventId": "updates.eventId",
      "data": "updates.data",
      "count": 1
    }
  ).execute()
)
```

Insert with Deduplication using MERGE

```
(deltaTable.alias("logs").merge(
  newDedupedLogs.alias("newDedupedLogs"),
  "logs.uniqueId = newDedupedLogs.uniqueId")
  .whenNotMatchedInsertAll()
  .execute()
)
```

TIME TRAVEL

View transaction log (aka Delta Log)

```
fullHistoryDF = deltaTable.history()
```

Query historical versions of Delta Lake tables

```
# choose only one option: versionAsOf, or timestampAsOf
df = (spark.read.format("delta")
      .option("versionAsOf", 0)
      .option("timestampAsOf", "2020-12-18")
      .load("/path/to/delta_table"))
```

TIME TRAVEL (CONTINUED)

Find changes between 2 versions of a table

```
df1 = spark.read.format("delta").load(pathToTable)
df2 = spark.read.format("delta").option("versionAsOf",
2).load("/path/to/delta_table")
df1.exceptAll(df2).show()
```

Rollback a table by version or timestamp

```
deltaTable.restoreToVersion(0)
deltaTable.restoreToTimestamp('2020-12-01')
```

UTILITY METHODS

Run Spark SQL queries in Python

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`/path/to/delta_table`")
spark.sql("DESCRIBE HISTORY tableName")
```

Compact old files with Vacuum

```
deltaTable.vacuum() # vacuum files older than default
retention period (7 days)
deltaTable.vacuum(100) # vacuum files not required by
versions more than 100 hours old
```

Clone a Delta Lake table

```
deltaTable.clone(target="/path/to/delta_table/",
isShallow=True, replace=True)
```

Get DataFrame representation of a Delta Lake table

```
df = deltaTable.toDF()
```

Run SQL queries on Delta Lake tables

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`/path/to/delta_table`")
```

PERFORMANCE OPTIMIZATIONS

Compact data files with Optimize and Z-Order

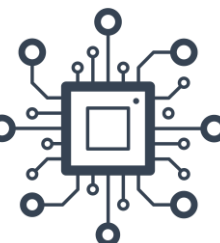
```
*Databricks Delta Lake feature
spark.sql("OPTIMIZE tableName [ZORDER BY (colA, colB)]")
```

Auto-optimize tables

```
*Databricks Delta Lake feature. For existing tables:
spark.sql("ALTER TABLE [table_name |
delta.`path/to/delta_table`]
SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true)
To enable auto-optimize for all new Delta Lake tables:
spark.sql("SET spark.databricks.delta.properties.
defaults.autoOptimize.optimizeWrite = true")
```

Cache frequently queried data in Delta Cache

```
*Databricks Delta Lake feature
spark.sql("CACHE SELECT * FROM tableName")
-- or:
spark.sql("CACHE SELECT colA, colB FROM tableName
WHERE colNameA > 0")
```





Delta Lake is an open source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.

delta.io | [Documentation](#) | [GitHub](#) | [Delta Lake on Databricks](#)

CREATE AND QUERY DELTA TABLES

Create and use managed database

```
-- Managed database is saved in the Hive metastore.
Default database is named "default".
DROP DATABASE IF EXISTS dbName;
CREATE DATABASE dbName;
USE dbName -- This command avoids having to specify
dbName.tableName every time instead of just tableName.
```

Query Delta Lake table by table name (preferred)

```
/* You can refer to Delta Tables by table name, or by
path. Table name is the preferred way, since named tables
are managed in the Hive Metastore (i.e., when you DROP a
named table, the data is dropped also – not the case for
path-based tables.) */
SELECT * FROM [dbName.] tableName
```

Query Delta Lake table by path

```
SELECT * FROM delta.`path/to/delta_table` -- note backticks
```

Convert Parquet table to Delta Lake format in place

```
-- by table name
CONVERT TO DELTA [dbName.]tableName
[PARTITIONED BY (col_name1 col_type1, col_name2
col_type2)]

-- path-based tables
CONVERT TO DELTA parquet.`/path/to/table` -- note backticks
[PARTITIONED BY (col_name1 col_type1, col_name2 col_type2)]
```

Create Delta Lake table as SELECT * with no upfront schema definition

```
CREATE TABLE [dbName.] tableName
USING DELTA
AS SELECT * FROM tableName | parquet.`path/to/data`
[LOCATION `/path/to/table`]
-- using location = unmanaged table
```

Create table, define schema explicitly with SQL DDL

```
CREATE TABLE [dbName.] tableName (
  id INT [NOT NULL],
  name STRING,
  date DATE,
  int_rate FLOAT)
USING DELTA
[PARTITIONED BY (time, date)] -- optional
```

Copy new data into Delta Lake table (with idempotent retries)

```
COPY INTO [dbName.] targetTable
FROM (SELECT * FROM "/path/to/table")
FILEFORMAT = DELTA -- or CSV, Parquet, ORC, JSON, etc.
```

Provided to the open source community by Databricks

© Databricks 2021. All rights reserved. Apache, Apache Spark, Spark and the Spark logo are trademarks of the Apache Software Foundation.

DELTA LAKE DDL/DML: UPDATE, DELETE, MERGE, ALTER TABLE

Update rows that match a predicate condition

```
UPDATE tableName SET event = 'click' WHERE event = 'clk'
```

Delete rows that match a predicate condition

```
DELETE FROM tableName WHERE "date < '2017-01-01"
```

Insert values directly into table

```
INSERT INTO TABLE tableName VALUES (
  (8003, "Kim Jones", "2020-12-18", 3.875),
  (8004, "Tim Jones", "2020-12-20", 3.750)
);
-- Insert using SELECT statement
INSERT INTO tableName SELECT * FROM sourceTable
-- Atomically replace all data in table with new values
INSERT OVERWRITE loan_by_state_delta VALUES (...)
```

Upsert (update + insert) using MERGE

```
MERGE INTO target
USING updates
ON target.Id = updates.Id
WHEN MATCHED AND target.delete_flag = "true" THEN
  DELETE
WHEN MATCHED THEN
  UPDATE SET * -- star notation means all columns
WHEN NOT MATCHED THEN
  INSERT (date, Id, data) -- or, use INSERT *
VALUES (date, Id, data)
```

Insert with Deduplication using MERGE

```
MERGE INTO logs
USING newDedupedLogs
ON logs.uniqueId = newDedupedLogs.uniqueId
WHEN NOT MATCHED
  THEN INSERT *
```

Alter table schema – add columns

```
ALTER TABLE tableName ADD COLUMNS (
  col_name data_type
  [FIRST|AFTER colA_name])
```

Alter table – add constraint

```
-- Add "Not null" constraint:
ALTER TABLE tableName CHANGE COLUMN col_name SET NOT NULL
-- Add "Check" constraint:
ALTER TABLE tableName
ADD CONSTRAINT dateWithinRange CHECK date > "1900-01-01"
-- Drop constraint:
ALTER TABLE tableName DROP CONSTRAINT dateWithinRange
```

TIME TRAVEL

View transaction log (aka Delta Log)

```
DESCRIBE HISTORY tableName
```

Query historical versions of Delta Lake tables

```
SELECT * FROM tableName VERSION AS OF 0
SELECT * FROM tableName@v0 -- equivalent to VERSION AS OF 0
SELECT * FROM tableName TIMESTAMP AS OF "2020-12-18"
```

Find changes between 2 versions of table

```
SELECT * FROM tableName VERSION AS OF 12
EXCEPT ALL SELECT * FROM tableName VERSION AS OF 11
```

TIME TRAVEL (CONTINUED)

Rollback a table to an earlier version

```
-- RESTORE requires Delta Lake version 0.7.0+ & DBR 7.4+.
RESTORE tableName VERSION AS OF 0
RESTORE tableName TIMESTAMP AS OF "2020-12-18"
```

UTILITY METHODS

View table details

```
DESCRIBE DETAIL tableName
DESCRIBE FORMATTED tableName
```

Delete old files with Vacuum

```
VACUUM tableName [RETAIN num HOURS] [DRY RUN]
```

Clone a Delta Lake table

```
-- Deep clones copy data from source, shallow clones don't.
CREATE TABLE [dbName.] targetName
[SHALLOW | DEEP] CLONE sourceName [VERSION AS OF 0]
[LOCATION "path/to/table"]
-- specify location only for path-based tables
```

Interoperability with Python / DataFrames

```
-- Read name-based table from Hive metastore into DataFrame
df = spark.table("tableName")
-- Read path-based table into DataFrame
df = spark.read.format("delta").load("/path/to/delta_table")
```

Run SQL queries from Python

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`/path/to/delta_table`")
```

Modify data retention settings for Delta Lake table

```
-- logRetentionDuration -> how long transaction log history
is kept, deletedFileRetentionDuration -> how long ago a file
must have been deleted before being a candidate for VACUUM.
ALTER TABLE tableName
SET TBLPROPERTIES(
  delta.logRetentionDuration = "interval 30 days",
  delta.deletedFileRetentionDuration = "interval 7 days"
);
SHOW TBLPROPERTIES tableName;
```

PERFORMANCE OPTIMIZATIONS

Compact data files with Optimize and Z-Order

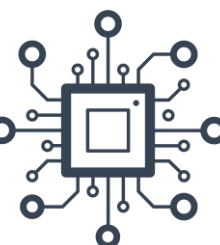
```
*Databricks Delta Lake feature
OPTIMIZE tableName
[ZORDER BY (colNameA, colNameB)]
```

Auto-optimize tables

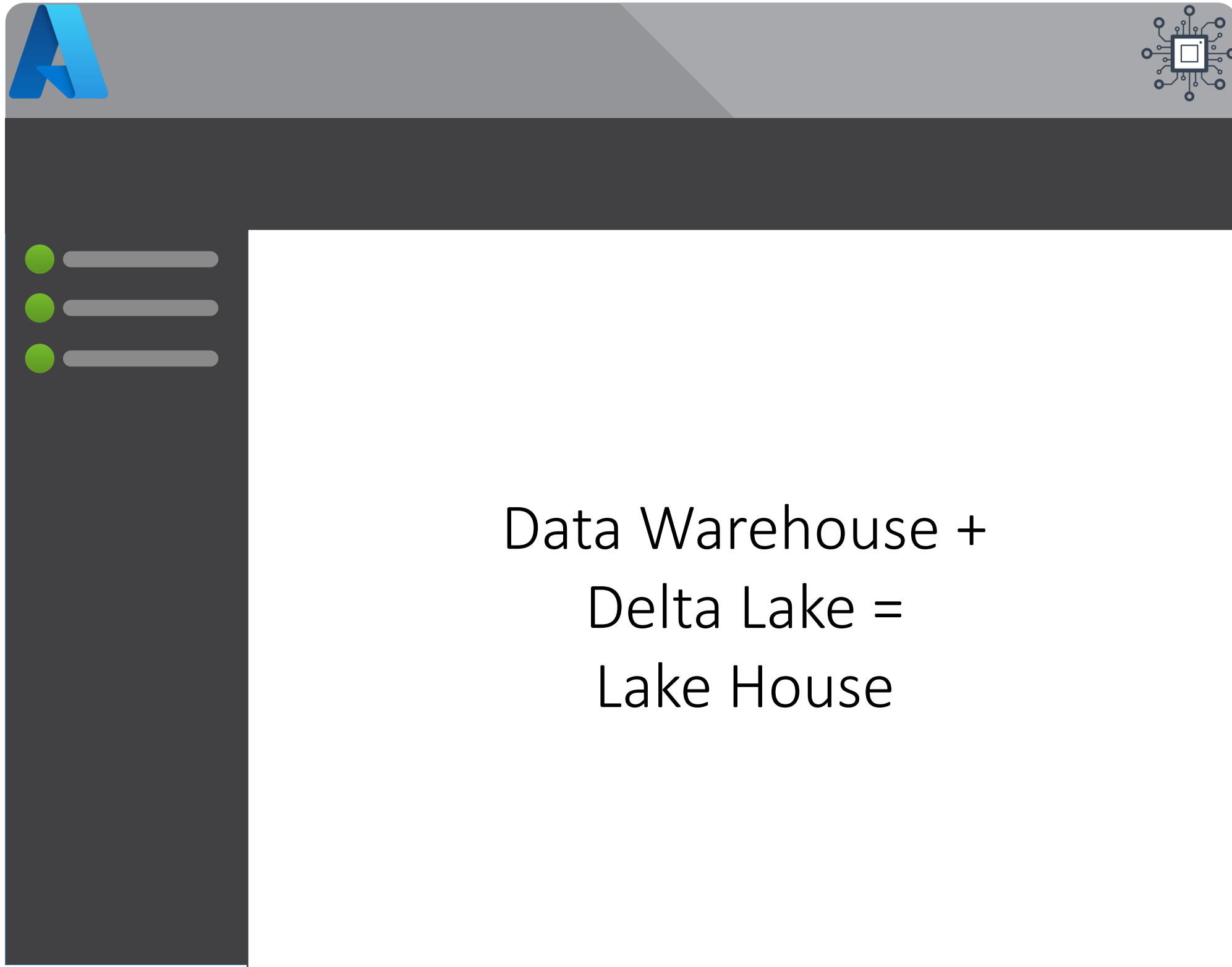
```
*Databricks Delta Lake feature
ALTER TABLE [table_name | delta.`path/to/delta_table`]
SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true)
```

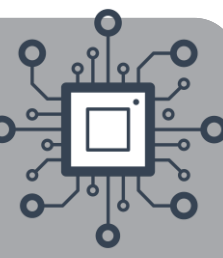
Cache frequently queried data in Delta Cache

```
*Databricks Delta Lake feature
CACHE SELECT * FROM tableName
-- or:
CACHE SELECT colA, colB FROM tableName WHERE colNameA > 0
```



Part 1.2 – Theory





Online
Line
Transactional
Processing



Application
Data

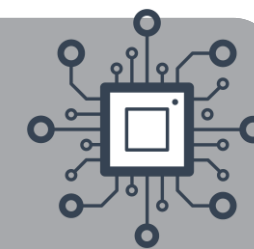
Extract
Transform
Load



Data
Warehouse

Offline
Analytical
Transactional
Processing





Online
Line
Transactional
Processing



Application
Data

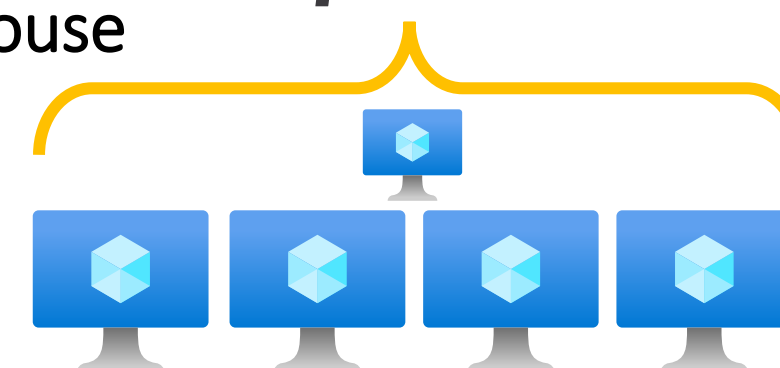
Extract
Transform
Load

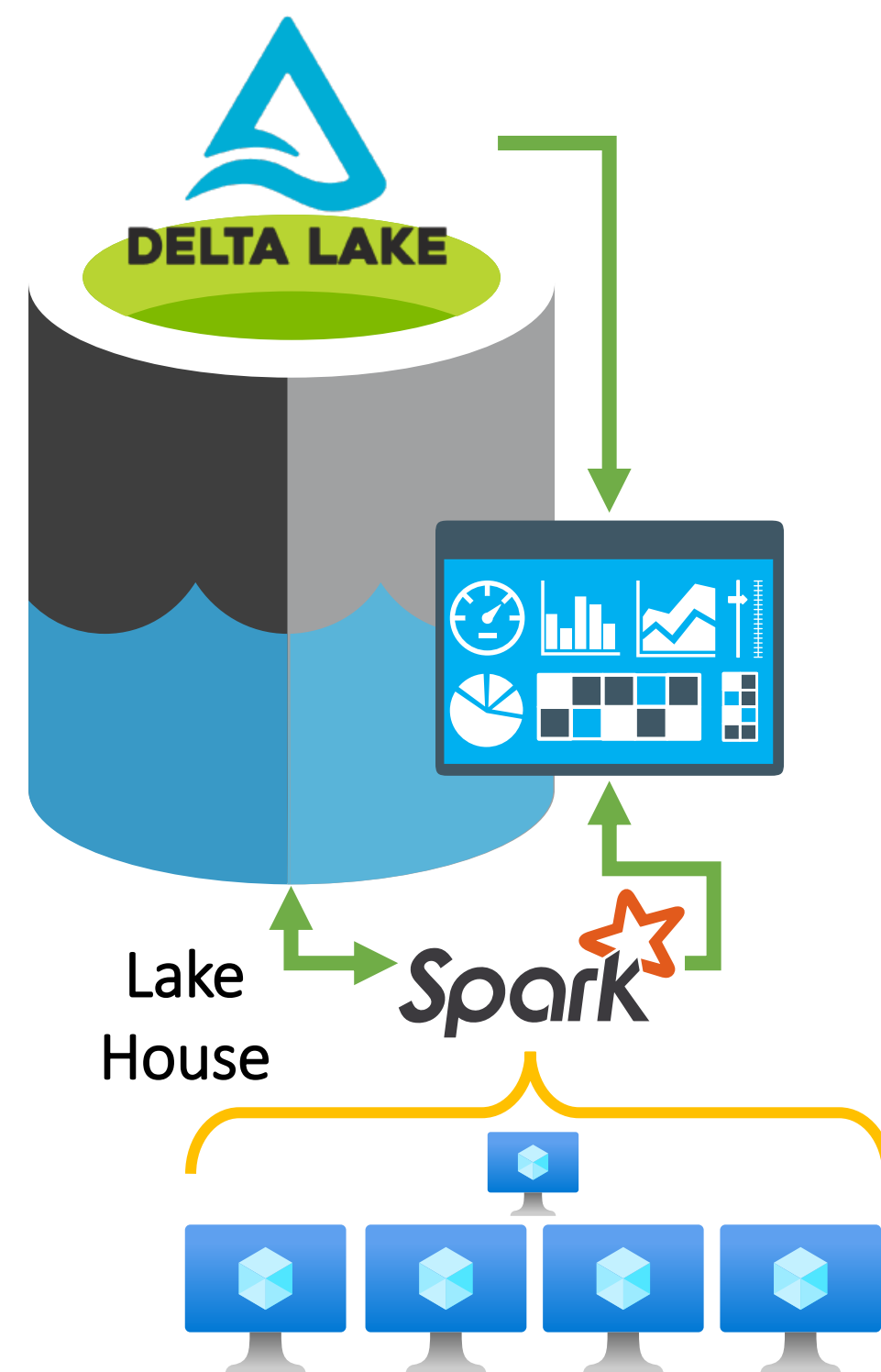
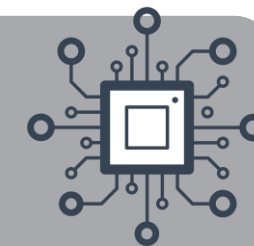


Lake
House

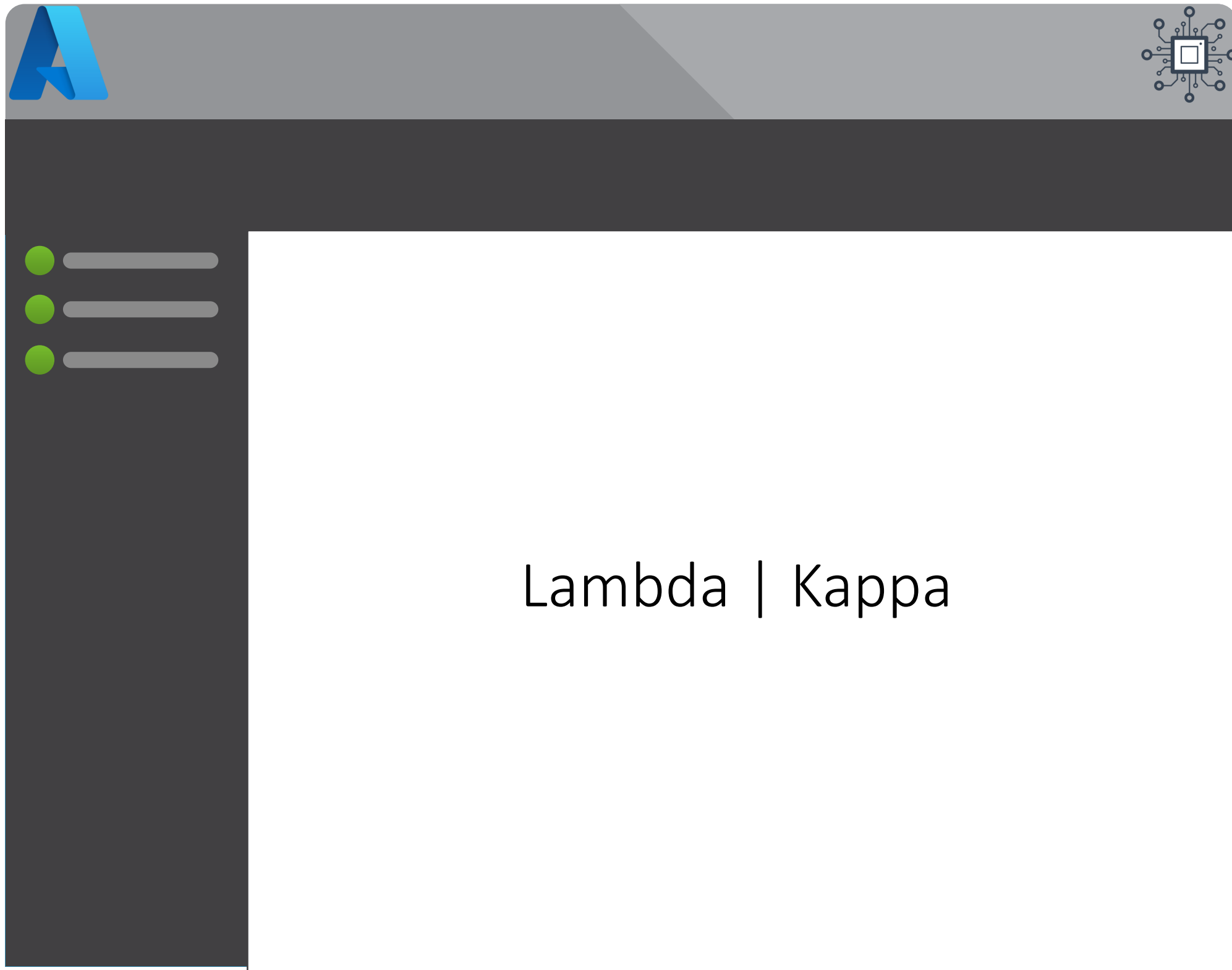
Spark

Offline
Analytical
Transactional
Processing



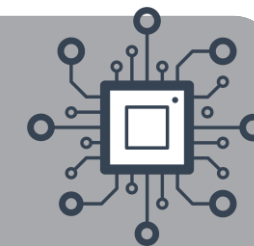


Part 1.3 – Theory

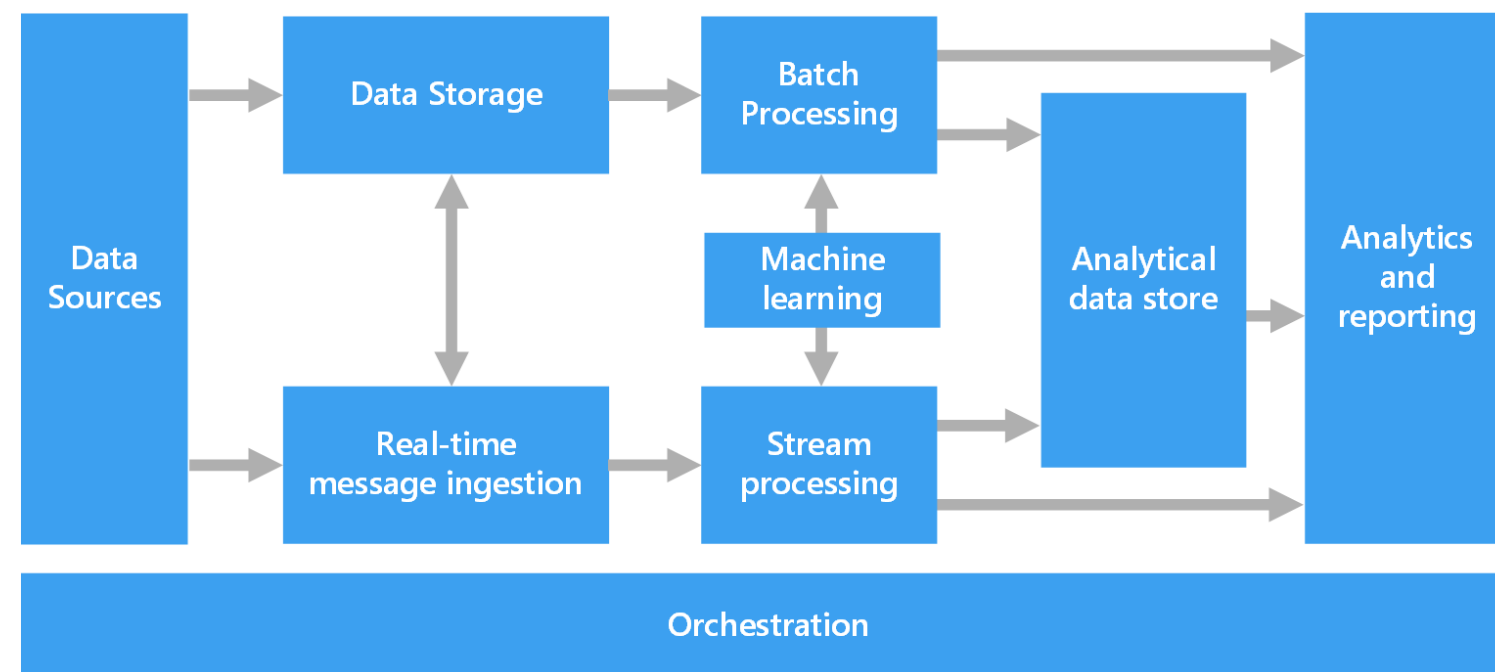




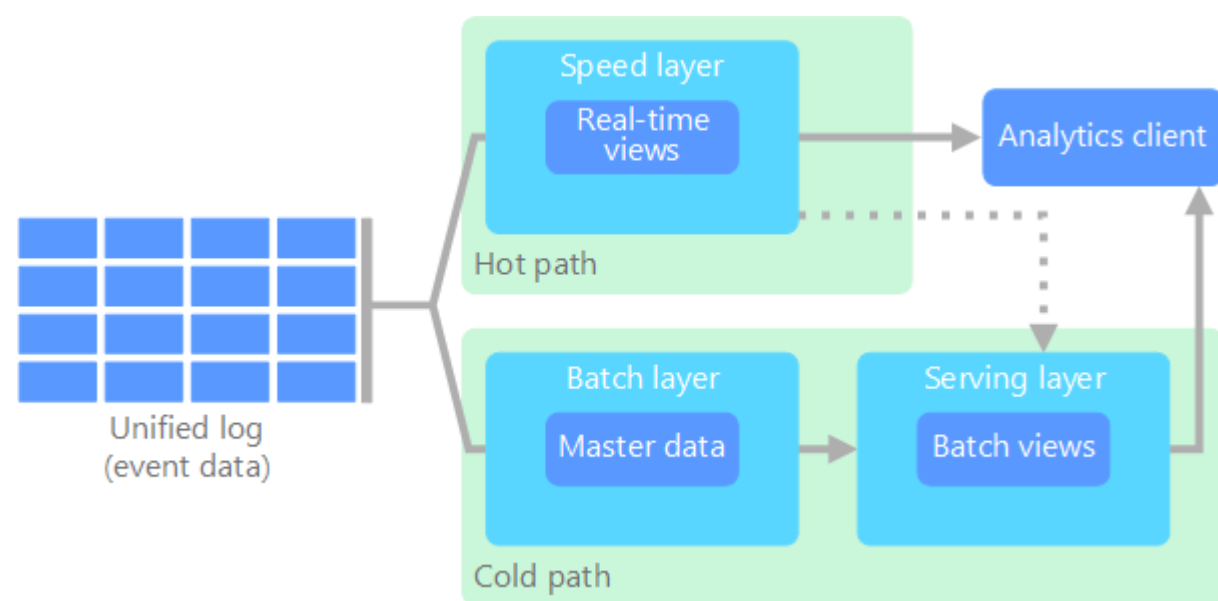
Lambda & Kappa Architectures



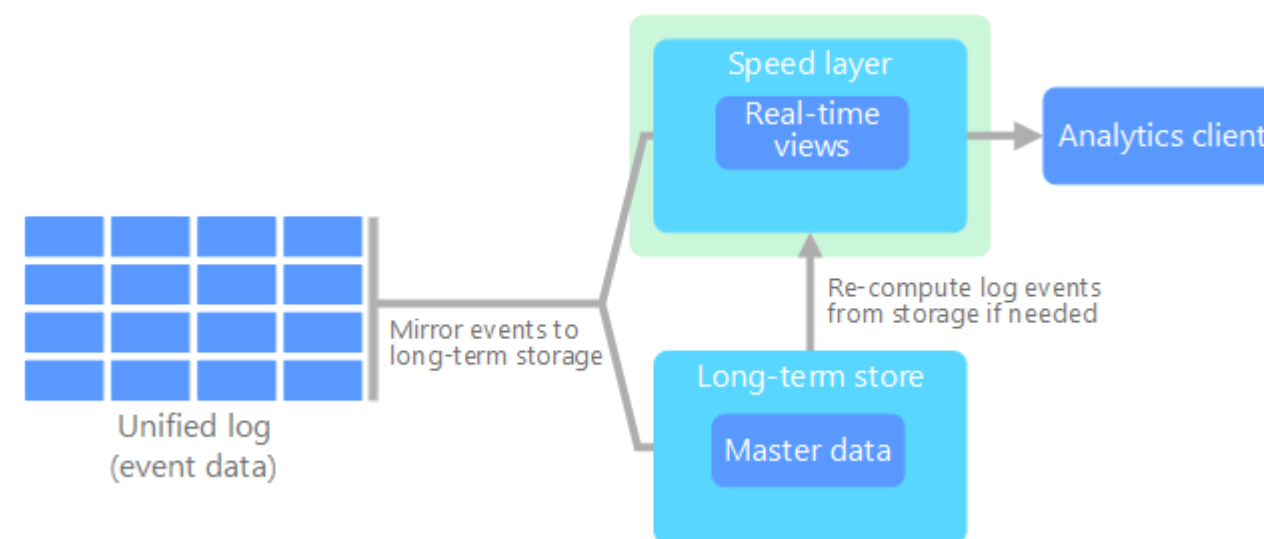
Components of a Big Data Architecture

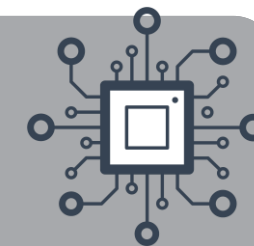


Lambda

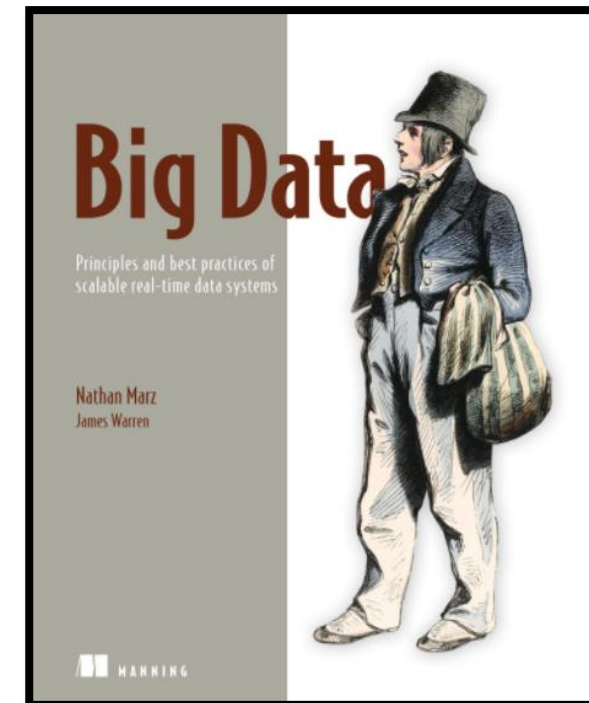
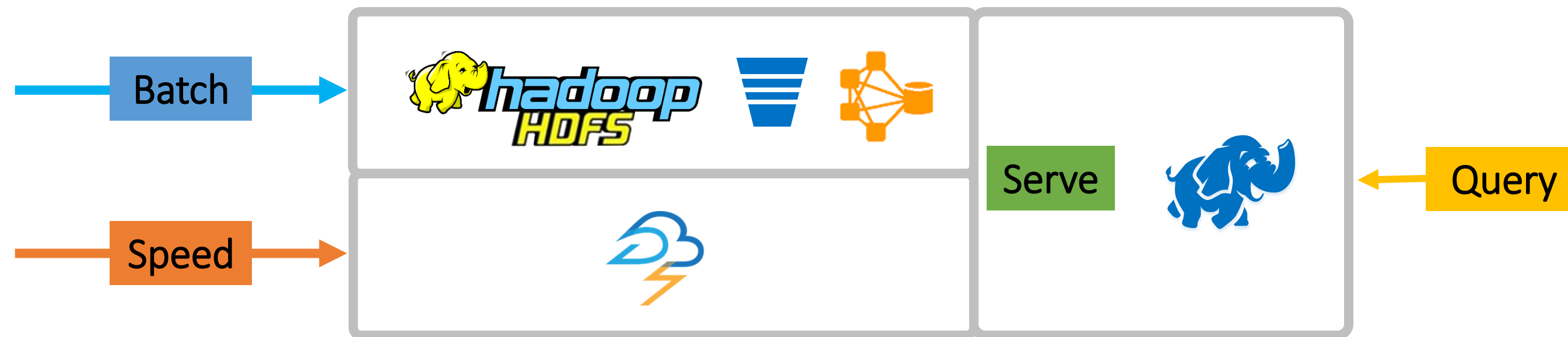
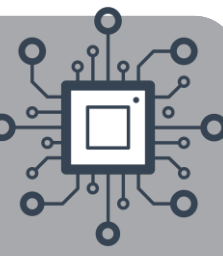


Kappa

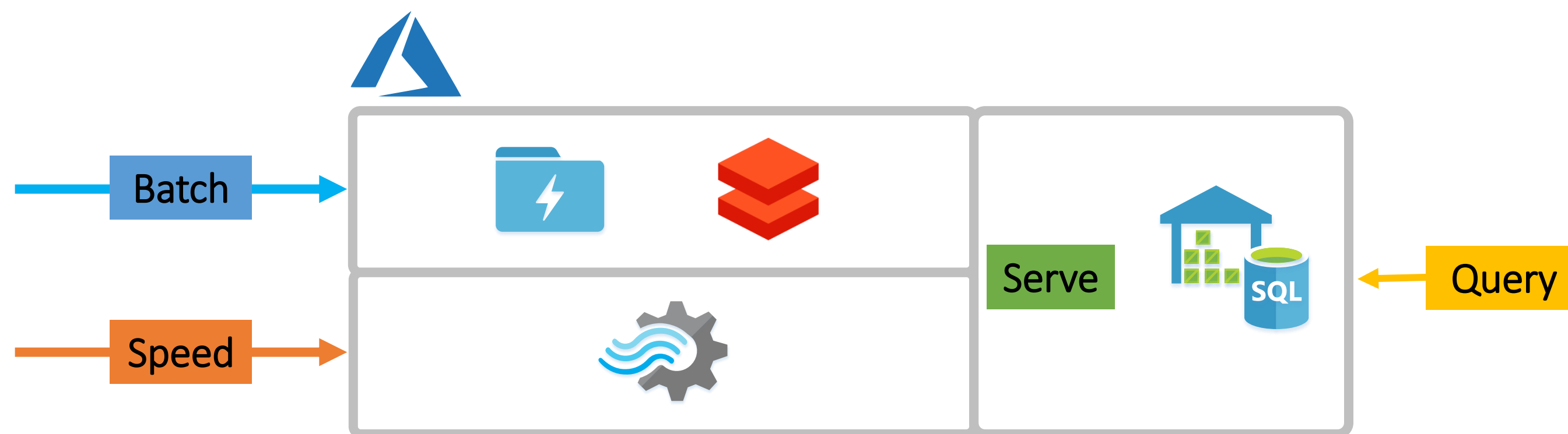
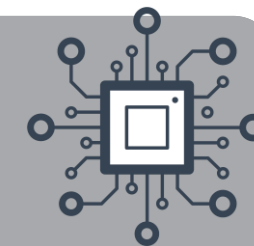




Lambda & Kappa Architectures

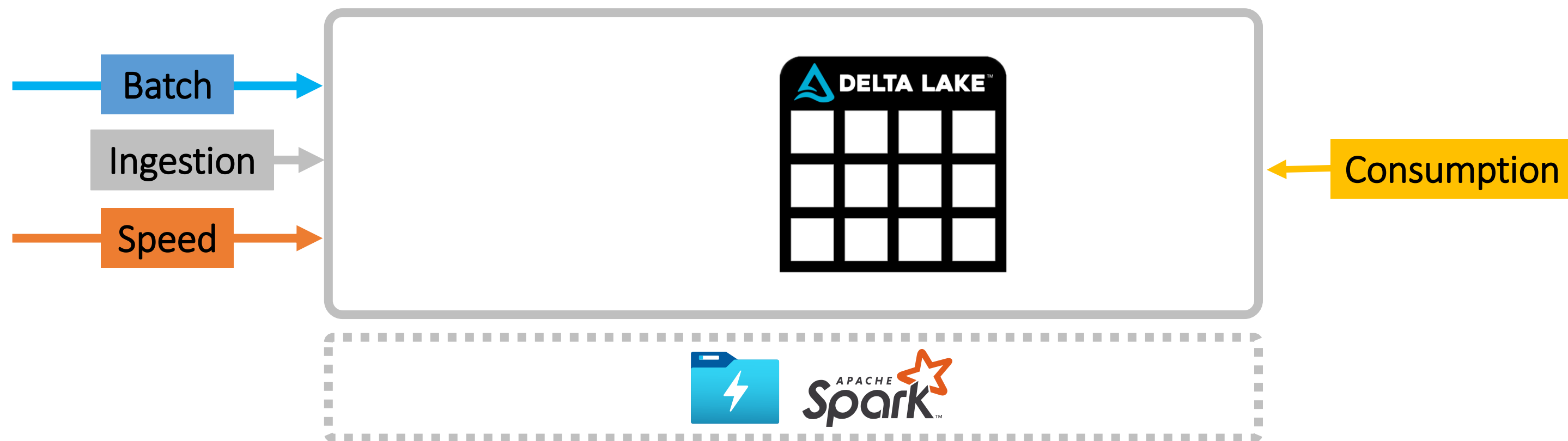
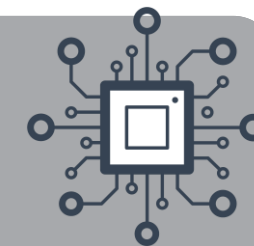


Lambda & Kappa Architectures

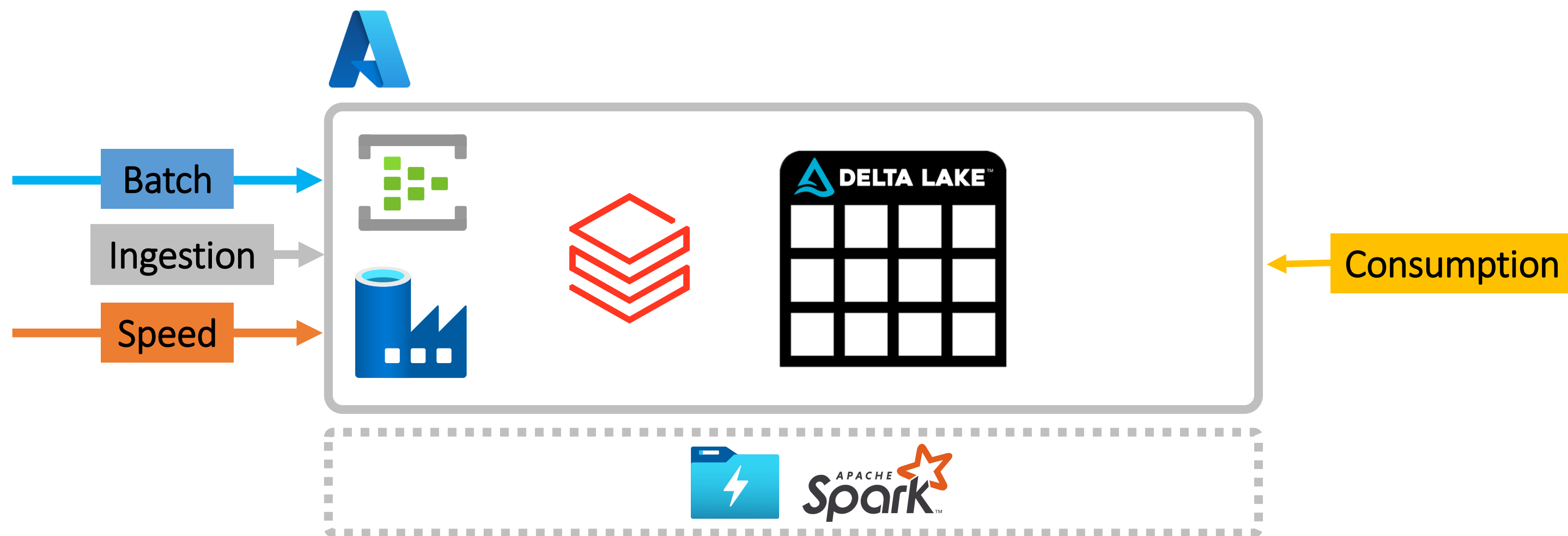
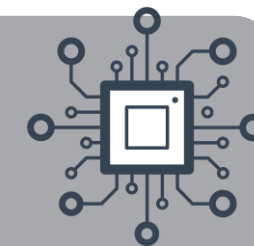


* Pre Delta-Lake

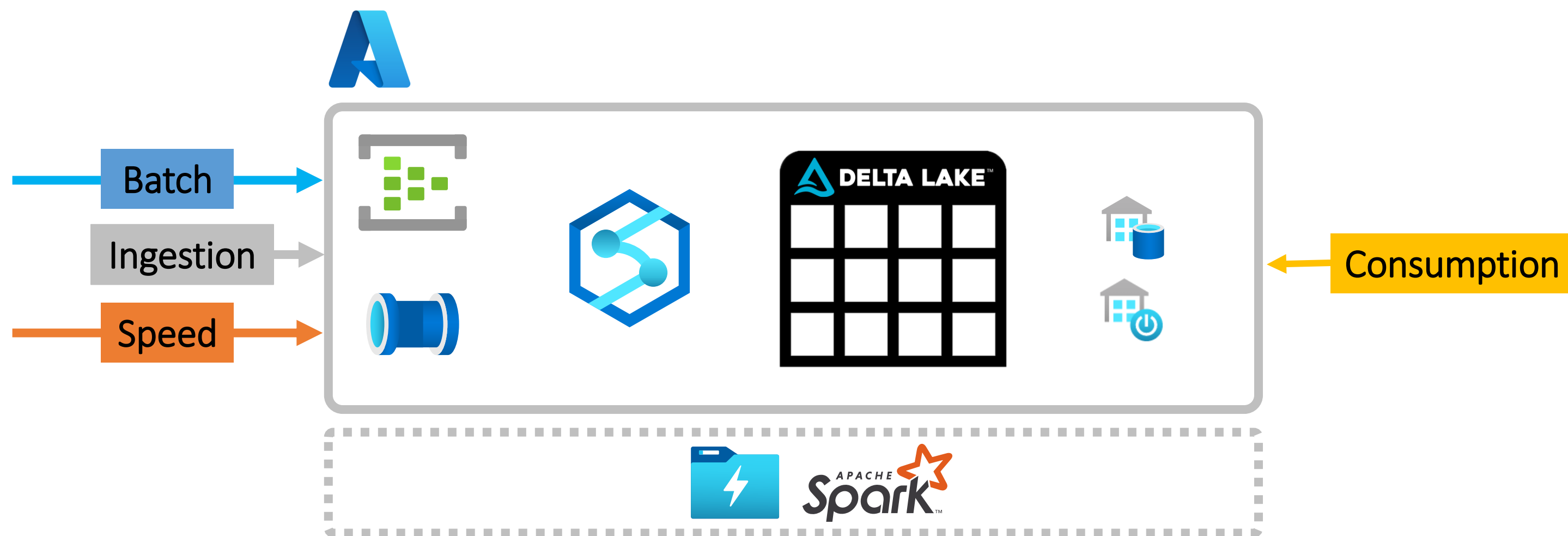
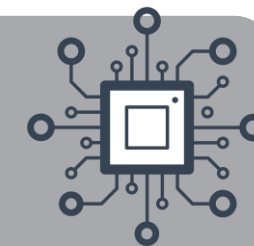
Lambda & Kappa Architectures



Lambda & Kappa Architectures

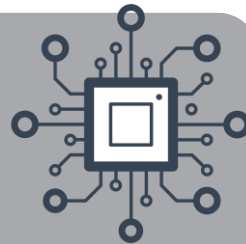


Lambda & Kappa Architectures

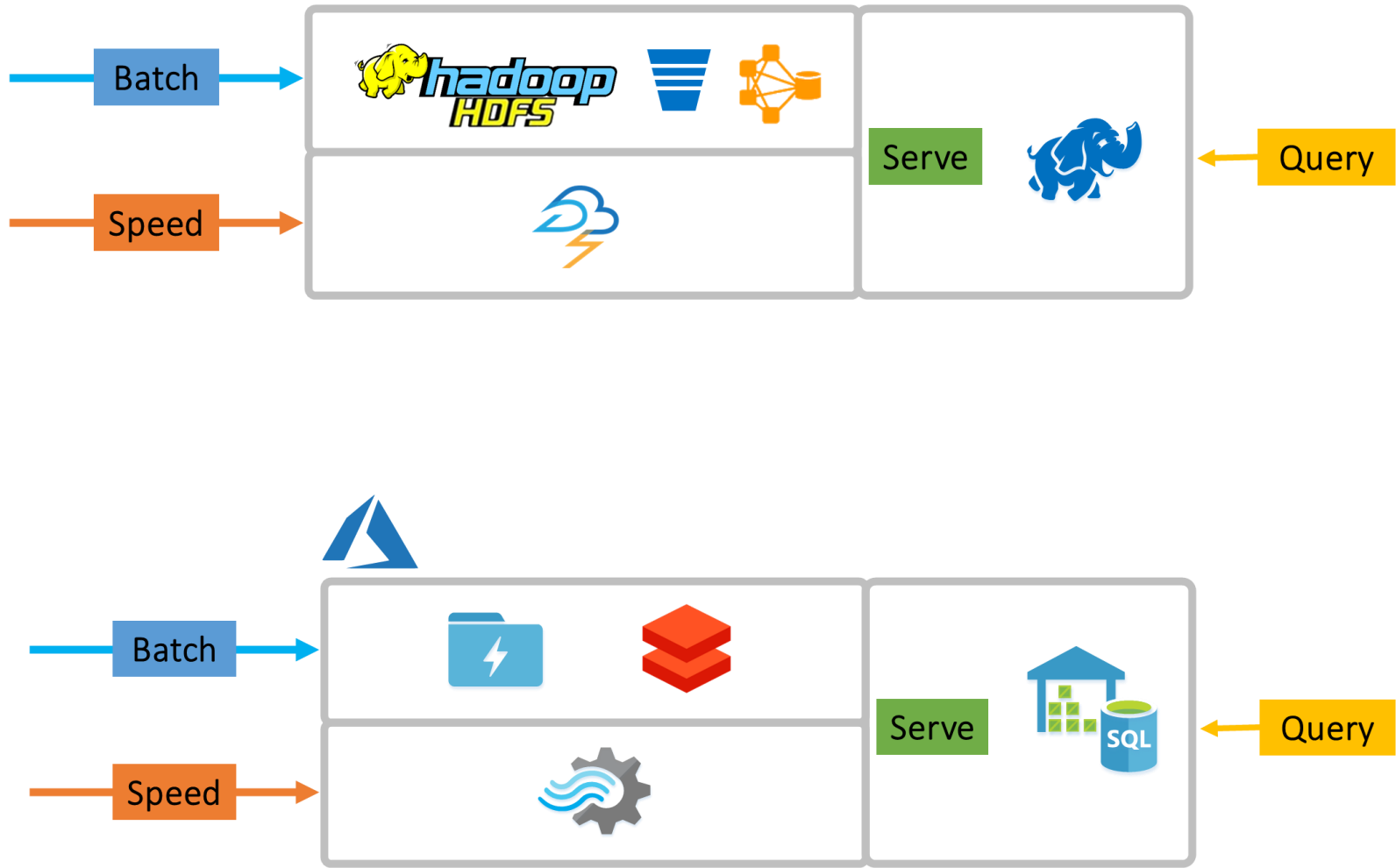




Delta Lake in the Context of Lambda & Kappa

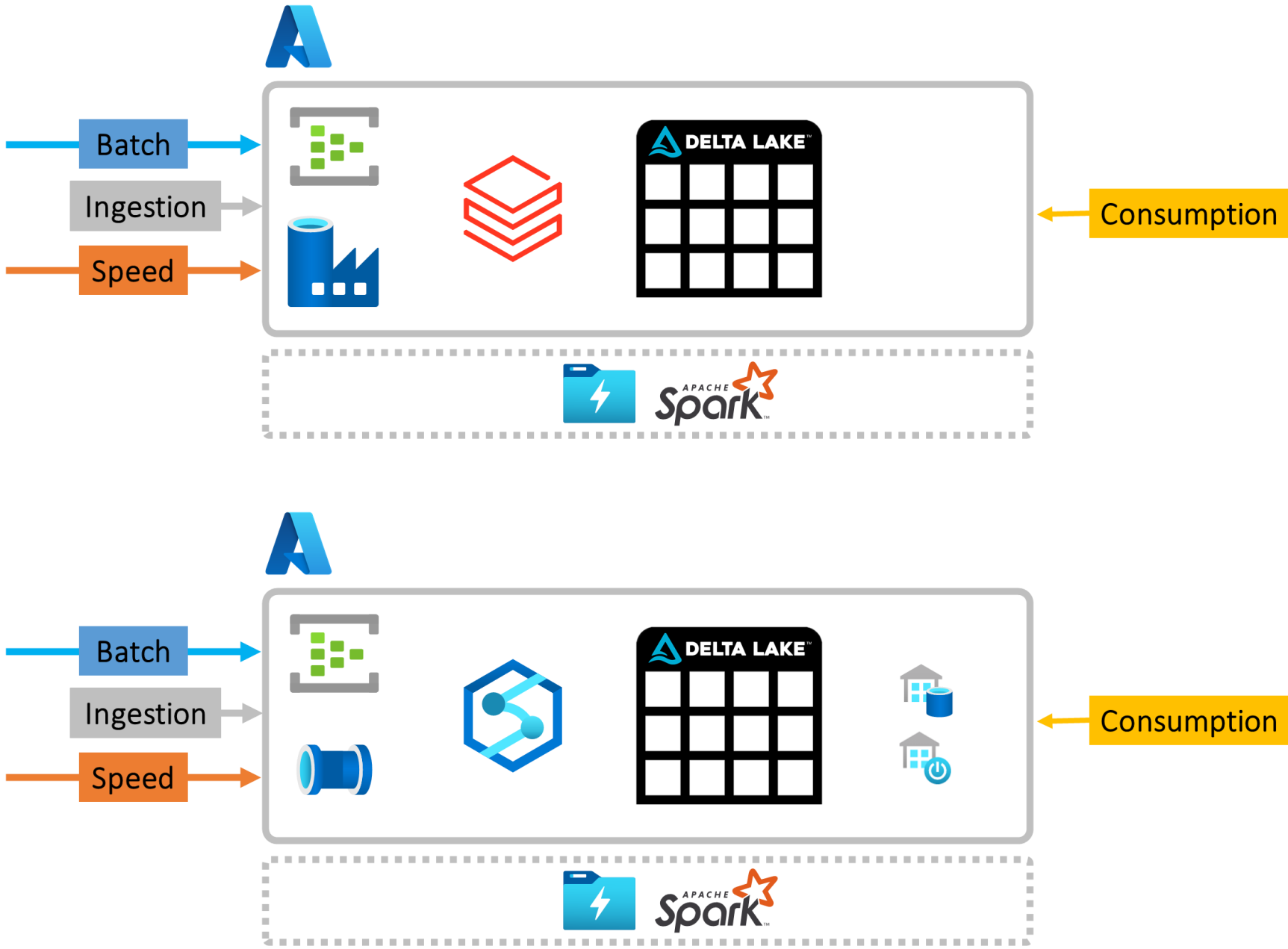


Lambda

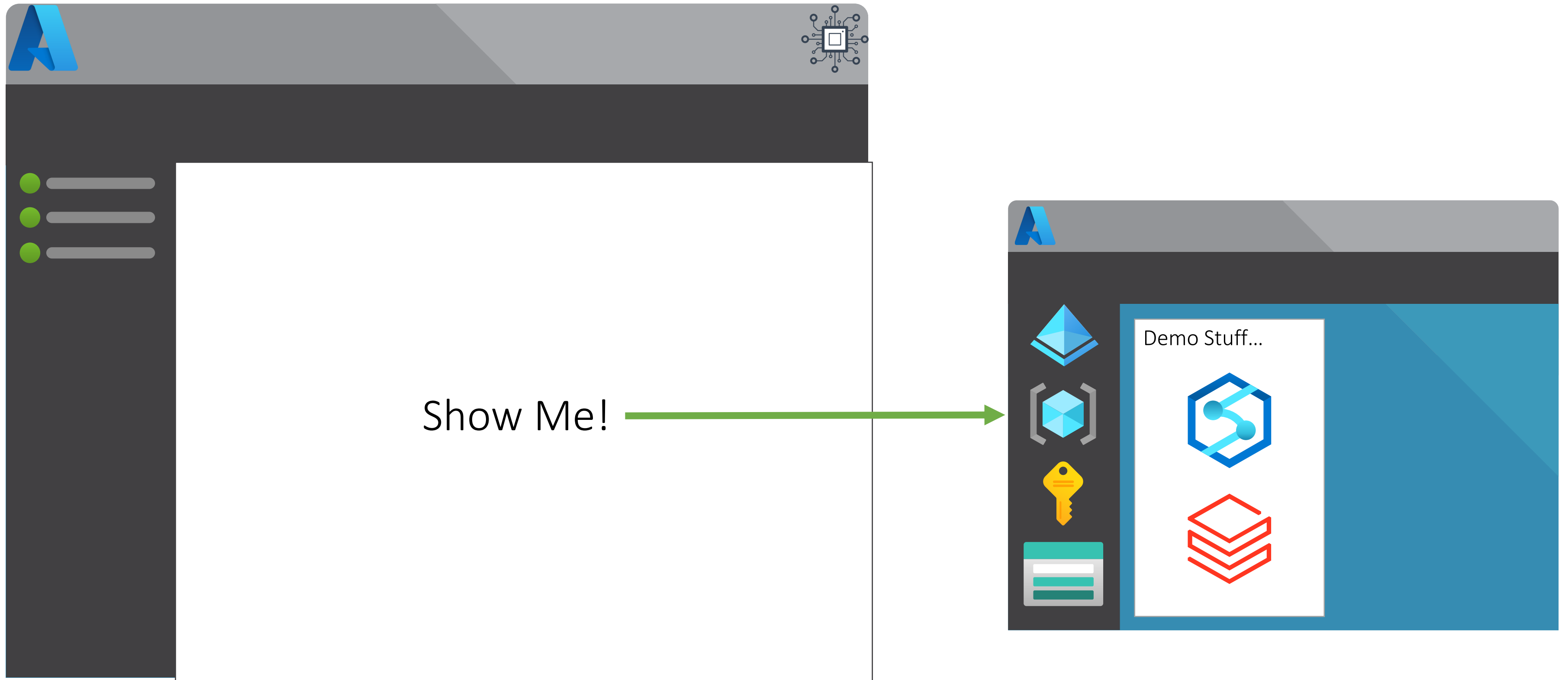


* Pre Delta-Lake

Kappa

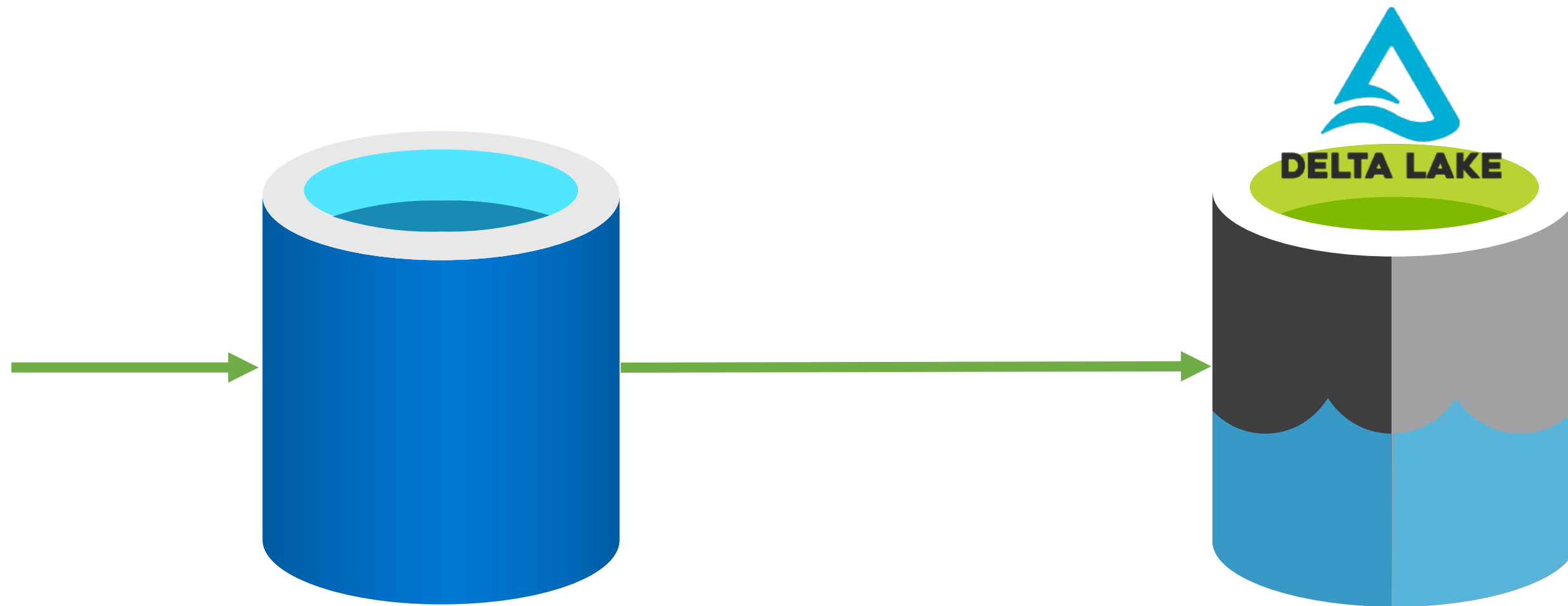
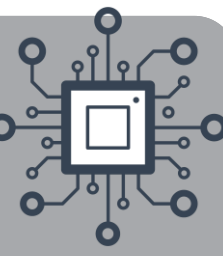


Part 2.0 – Practice

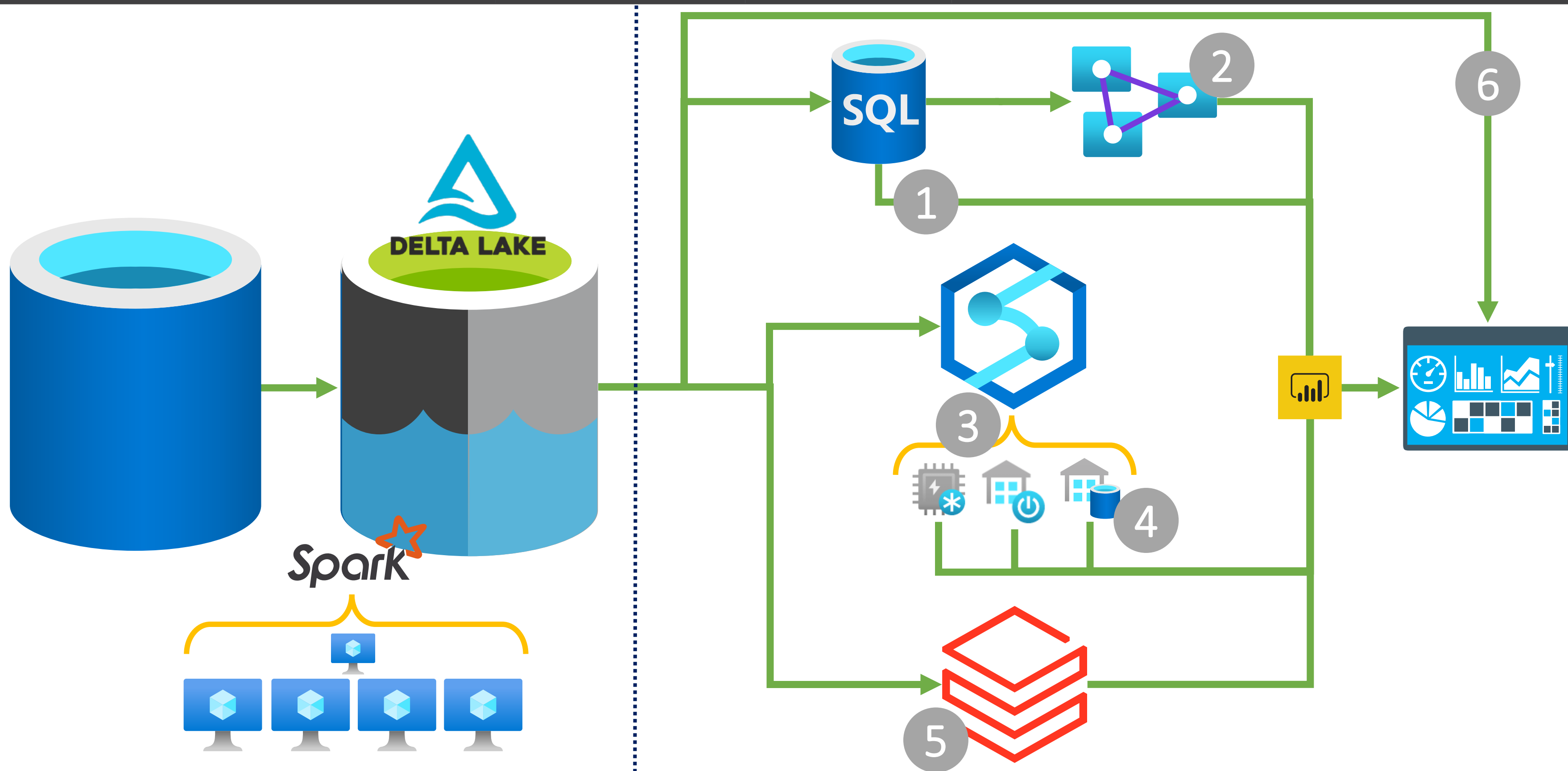
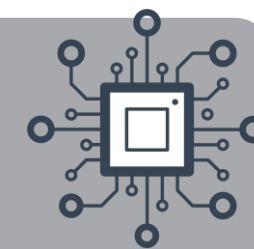




Consuming Our Lake House



Consuming Our Lake House in Azure





Diving Into Delta Lake: Unpacking The Transaction Log

<https://databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html>

Query Delta Lake files (preview) using serverless SQL pool in Azure Synapse Analytics

<https://docs.microsoft.com/en-us/azure/synapse-analytics/sql/query-delta-lake-format>

Delta Lake Cheat Sheet

[https://github.com/delta-io/delta/blob/master/examples/cheat sheet/delta lake cheat sheet.pdf](https://github.com/delta-io/delta/blob/master/examples/cheat%20sheet/delta%20lake%20cheat%20sheet.pdf)

How Interchangeable Are Delta Tables Between Azure Databricks and Azure Synapse Analytics?

<https://mrpaulandrew.com/2021/01/21/how-interchangeable-are-delta-tables-between-azure-databricks-and-azure-synapse-analytics/>

Lambda & Kappa Architecture with Azure Databricks

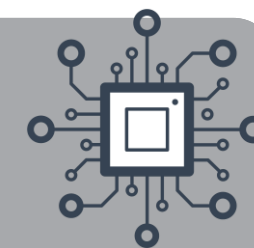
<https://nileshprajapati.net/blog/2020/lambda-and-kappa-architecture-with-databricks/>

Big Data Architectures

<https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/>

Simon Whitley on YouTube

<https://www.youtube.com/channel/UCmRI-X6XoeH2dQE4BShRU9Q>



Thank you for listening...

Paul Andrew



Blog: mrpaulandrew.com
YouTube: [c/mrpaulandrew](https://www.youtube.com/c/mrpaulandrew)
Email: paul@mrpaulandrew.com

Twitter: [@mrpaulandrew](https://twitter.com/mrpaulandrew)
LinkedIn: [In/mrpaulandrew](https://www.linkedin.com/company/mrpaulandrew)

GitHub: github.com/mrpaulandrew