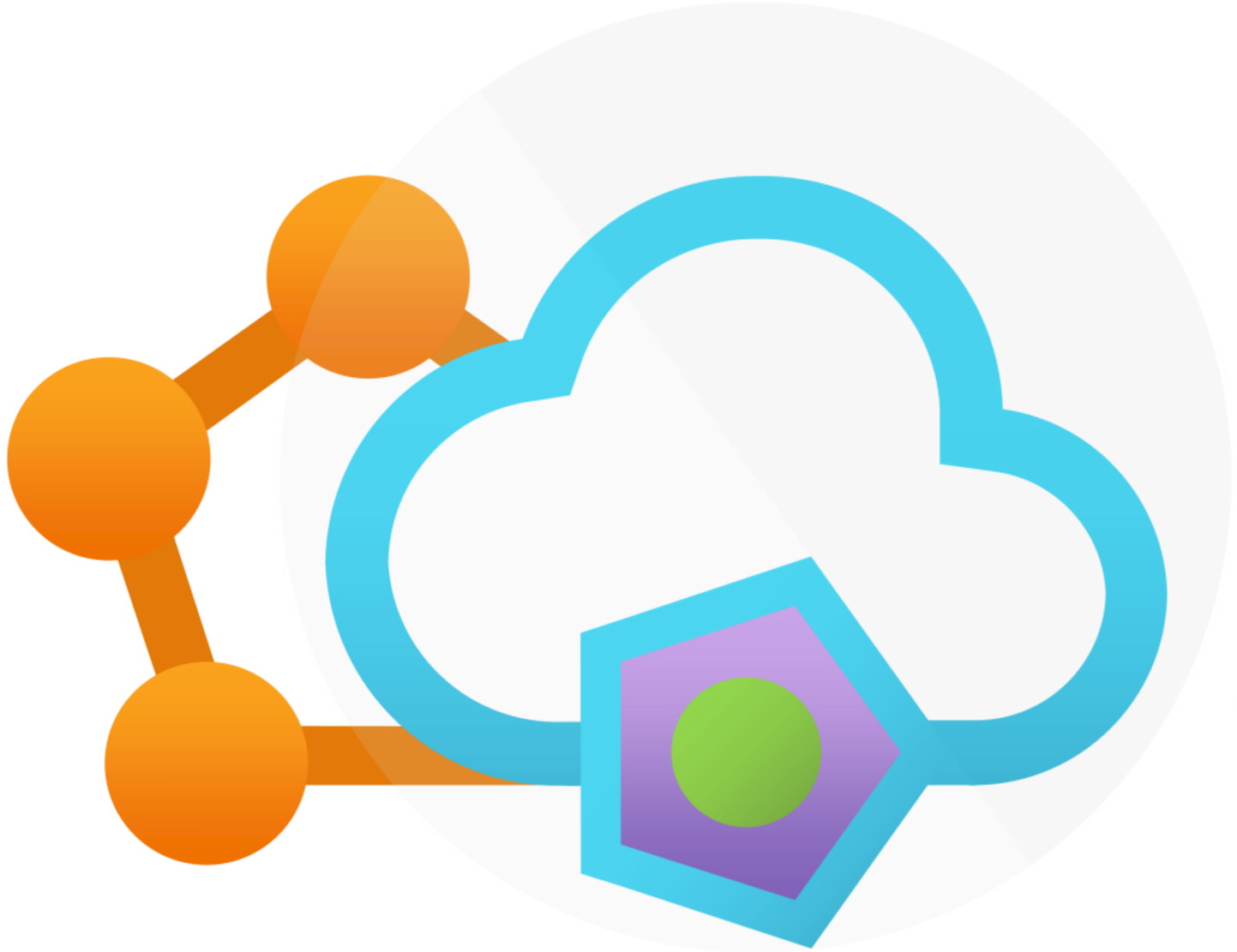


An Introduction to Delta Lake

& Delta Lake-Houses

Paul Andrew

Co-Founder | Chief Technology Officer

Microsoft Most Valuable Professional

Cloud Formations

Our Offerings

Supporting you on every level. Providing stakeholders with transparent reviews and feedback on planning, rollout and platform architecture.

Advisory

Strategy

Ensuring business value in everything, motivated by use cases, people and process. Aligned to the latest industry standards and concepts.

Partnering for the long term. From development to operations, monitoring and alerting. We'll support your business-critical platforms.

Lifecycle Support

Design

Scalable platform design using cloud native technology, reducing time to insight for batch, stream and event driven workloads for fabric and mesh architectures.

Knowledge sharing from the experts. Ensuring skills remain relevant for emerging cloud technologies and techniques.

Training

Delivery

Highly experienced engineers to support your implementation, with a full complement of continuous deployment practices and reusable assets.

Cloud Formations

Data Saturday Gothenburg 2023

Sponsors

SQLSERVICE
A PART OF CEGAL

b3
The after-party sponsor

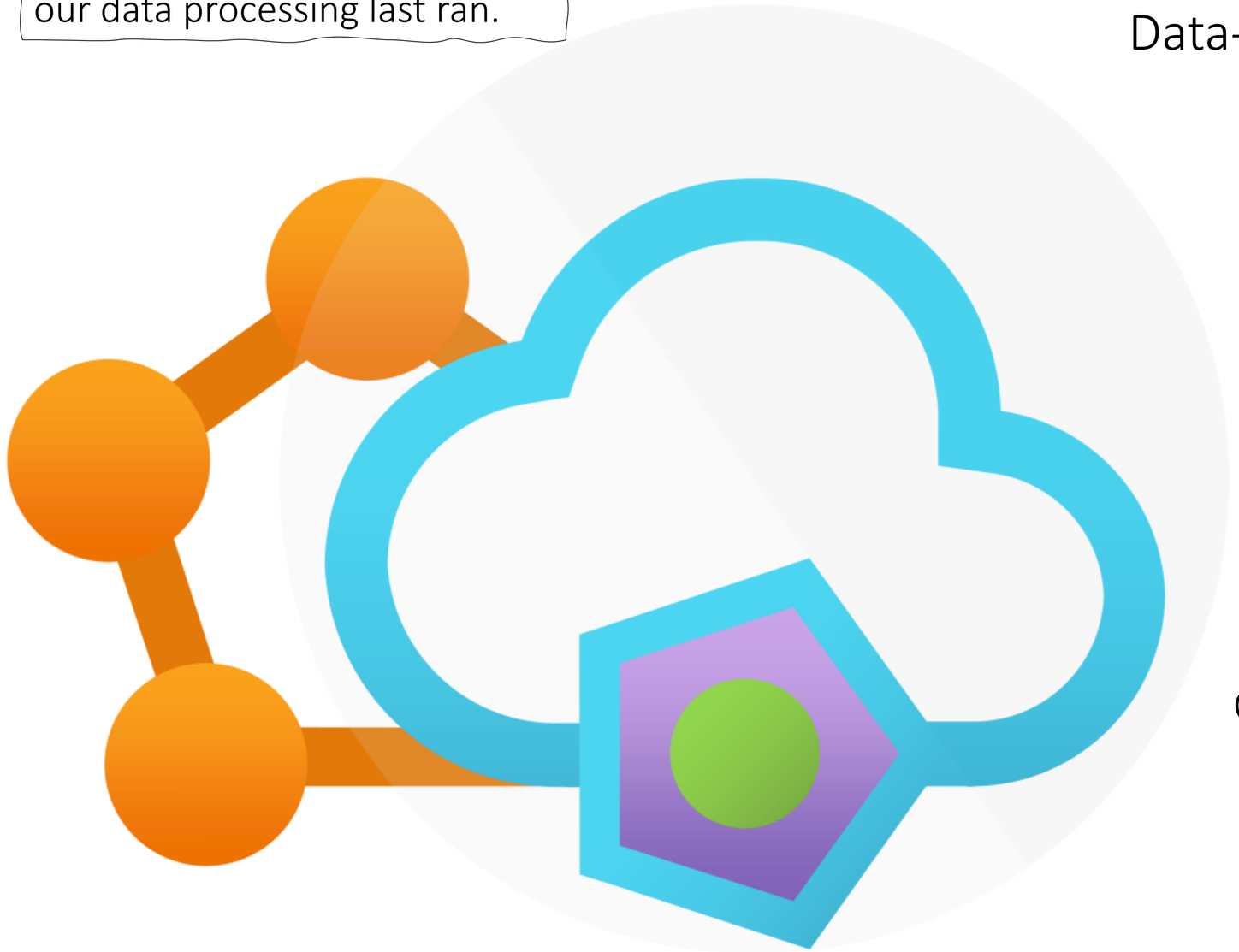
24

Republify
Part of Cilbuper Group

Tabular Editor

Pagesplit

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



Data-Ware-Lake-Delta-Beach-House-Lakes

An Introduction to Delta* Lake

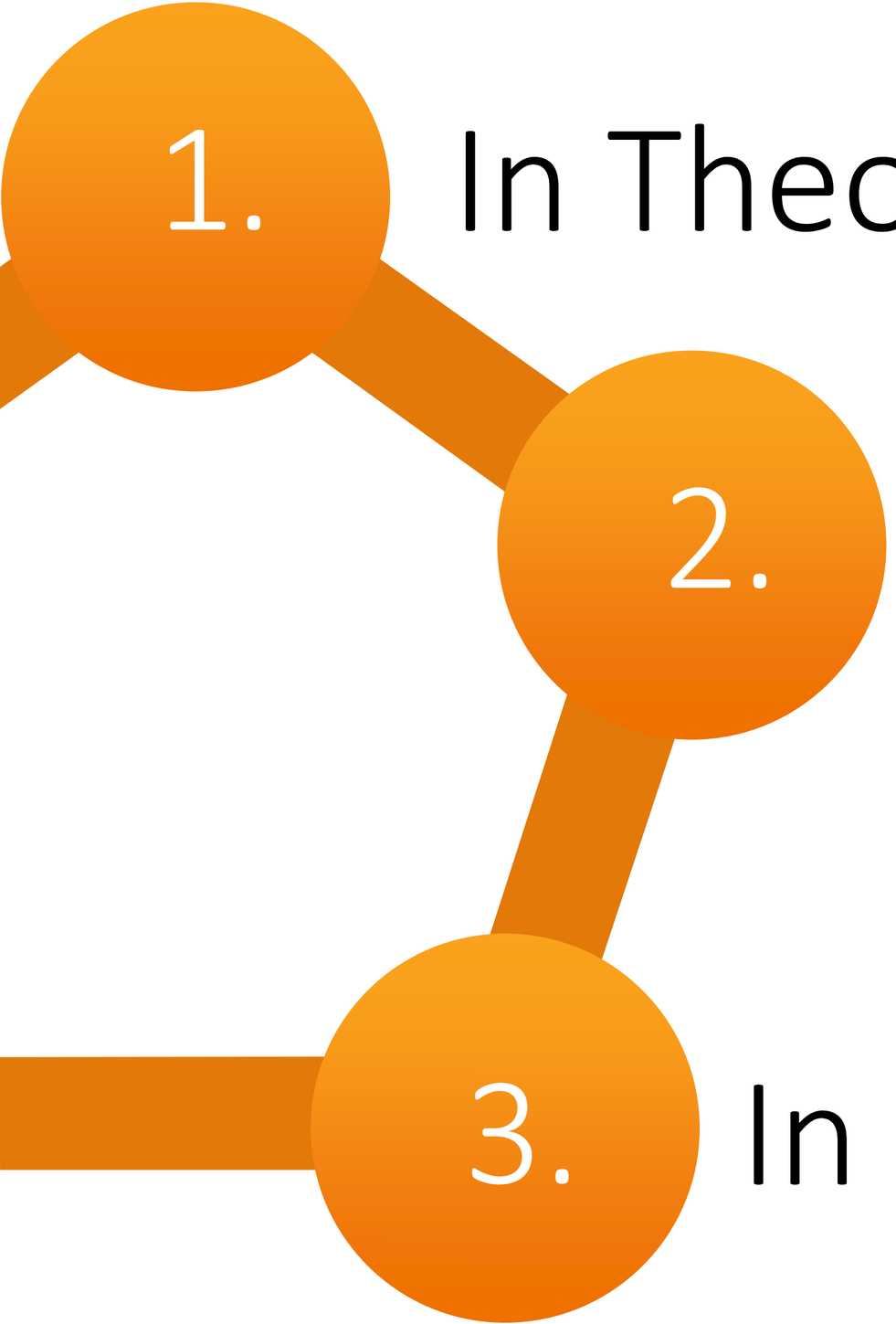
& Delta Lake-Houses

Paul Andrew

Co-Founder | Chief Technology Officer

MVP Microsoft Most Valuable Professional

Cloud Formations

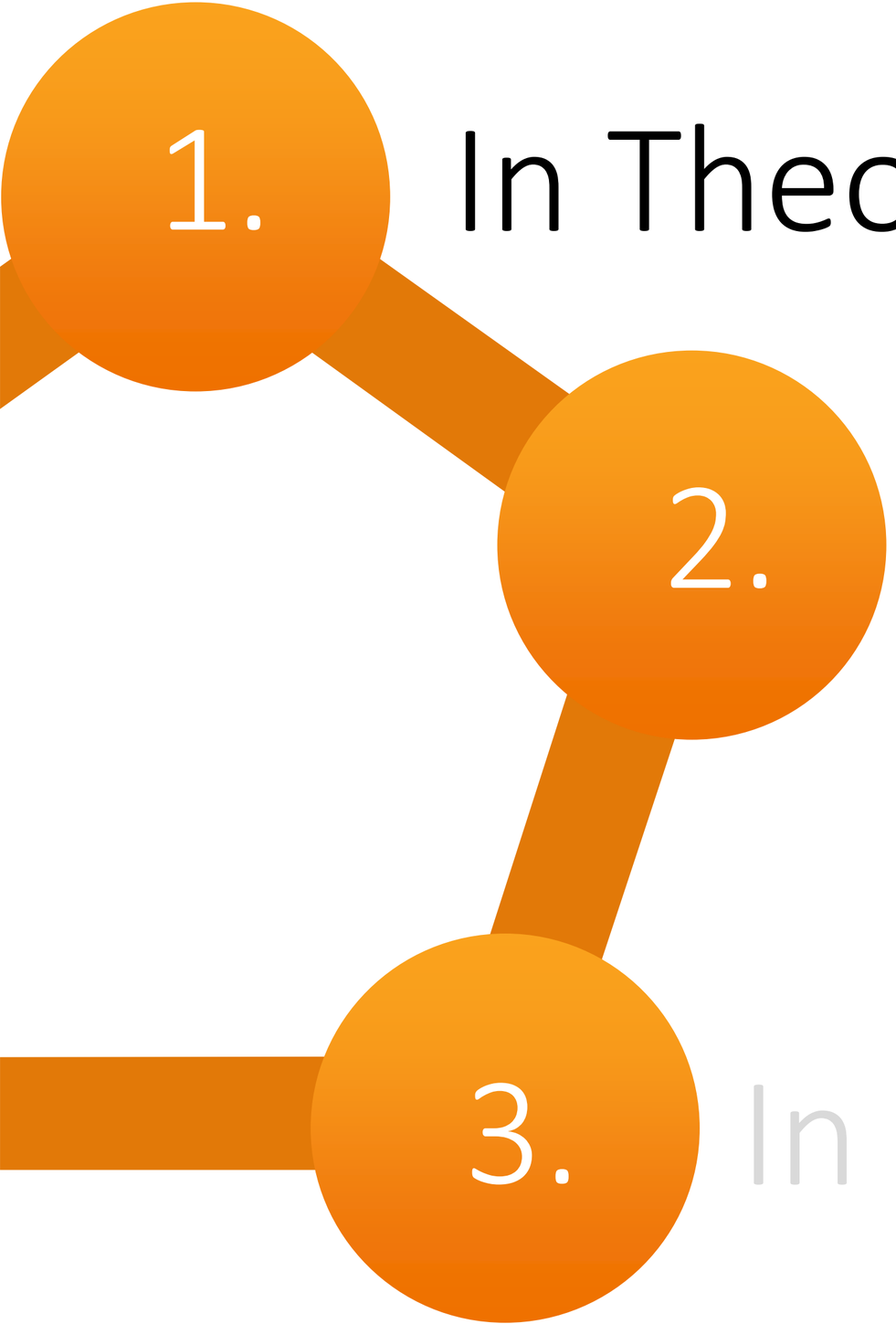


1. In Theory

2. In Practice

3. In Reality

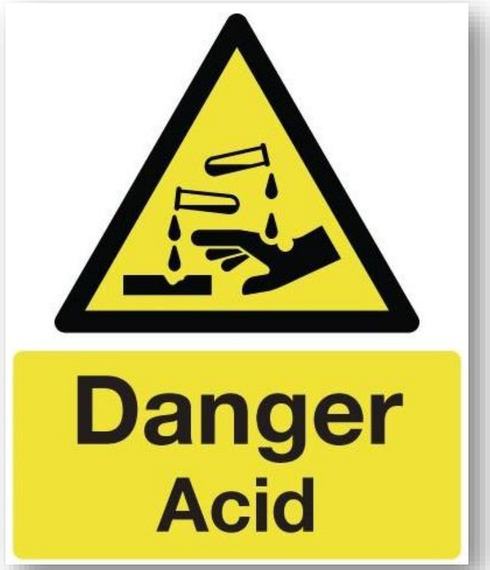
Agenda



1. In Theory
2. In Practice
3. In Reality

ACID!

Human Traffic (1999)



ACID – In Computer Science

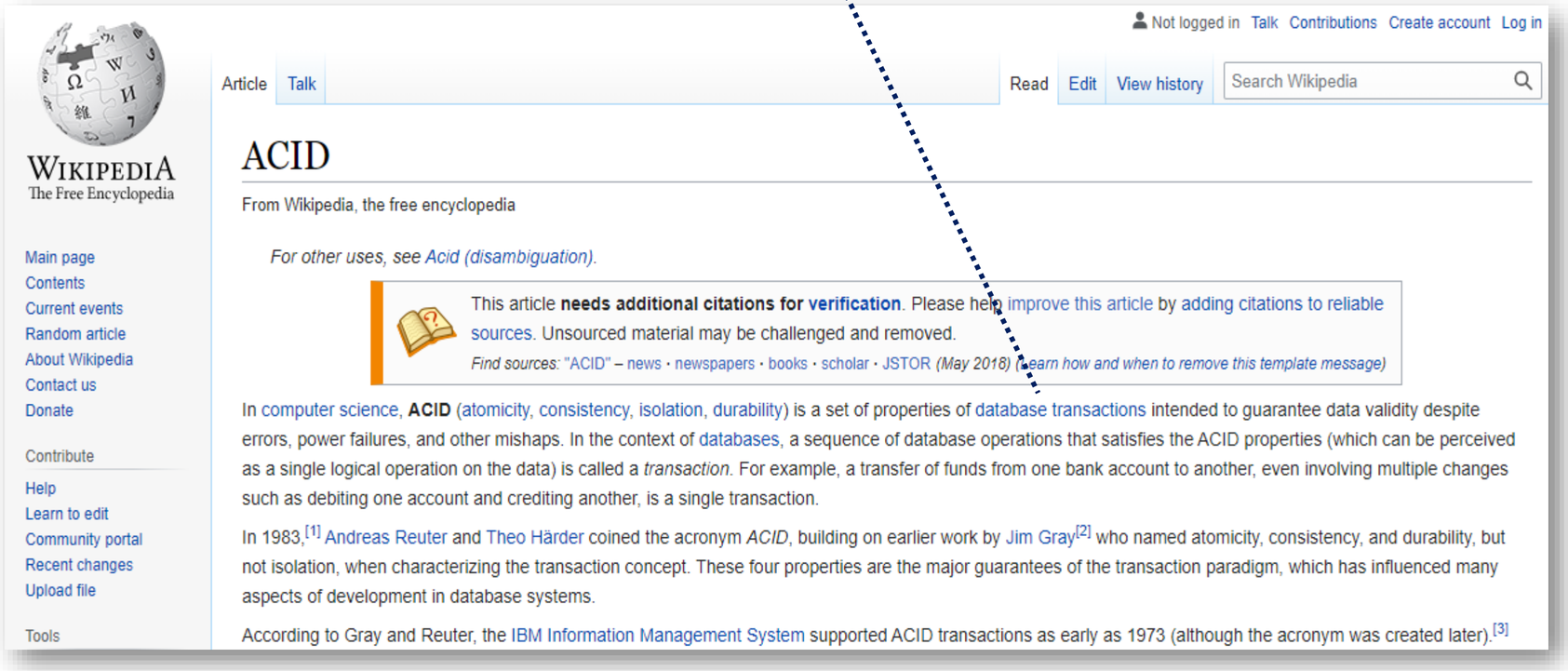
Atomicity

Consistency

Isolation

Durability

"is a set of properties of database transactions intended to guarantee data validity"



Not logged in Talk Contributions Create account Log in

Article Talk | Read Edit View history | Search Wikipedia

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

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.

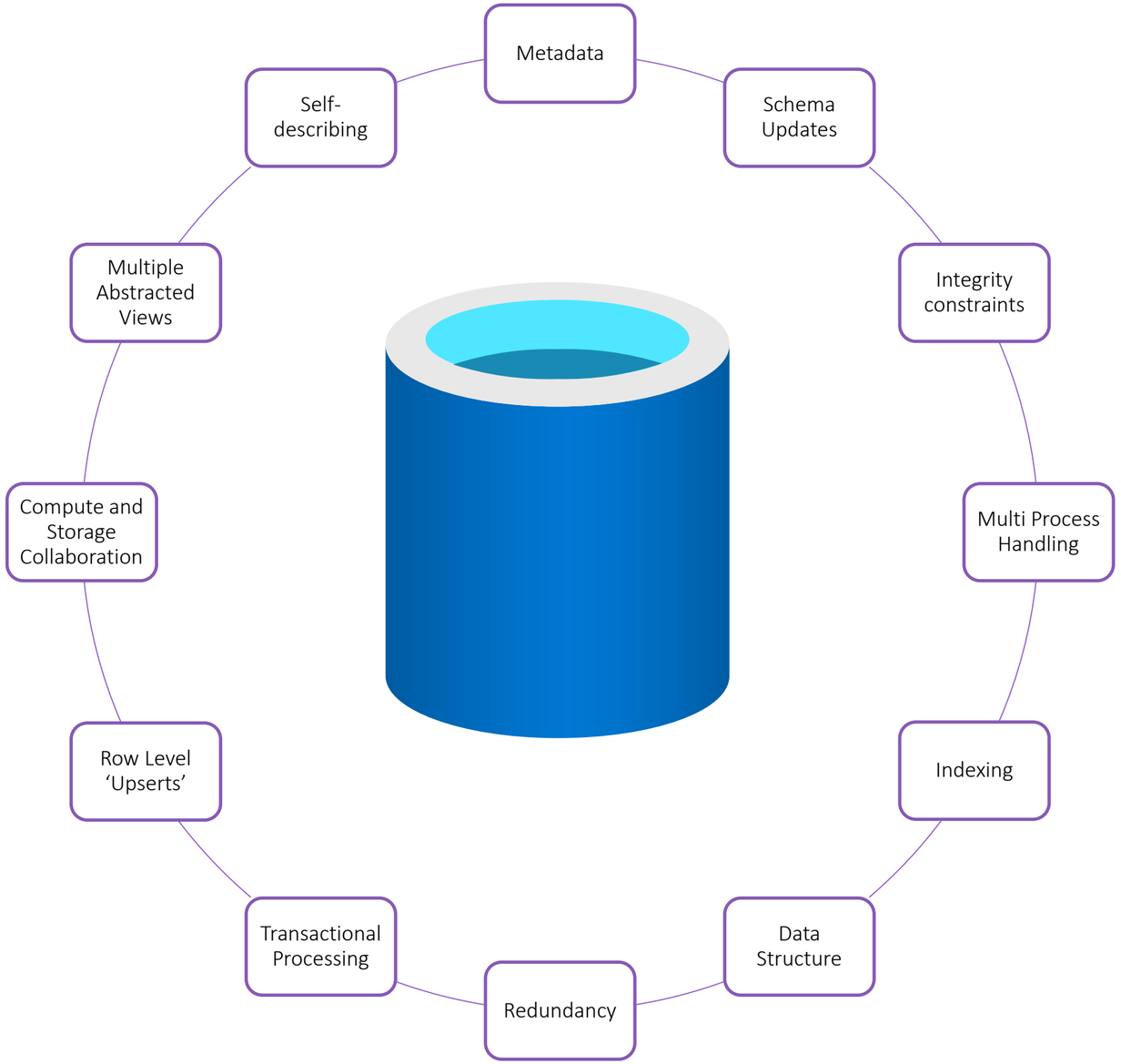
In 1983,[1] Andreas Reuter and Theo Härder coined the acronym *ACID*, building on earlier work by Jim Gray[2] who named atomicity, consistency, and durability, but not isolation, when characterizing the transaction concept. These four properties are the major guarantees of the transaction paradigm, which 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

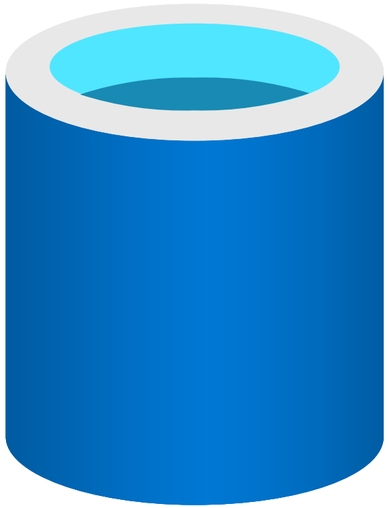
Databases

Data**B**ase
Management
System



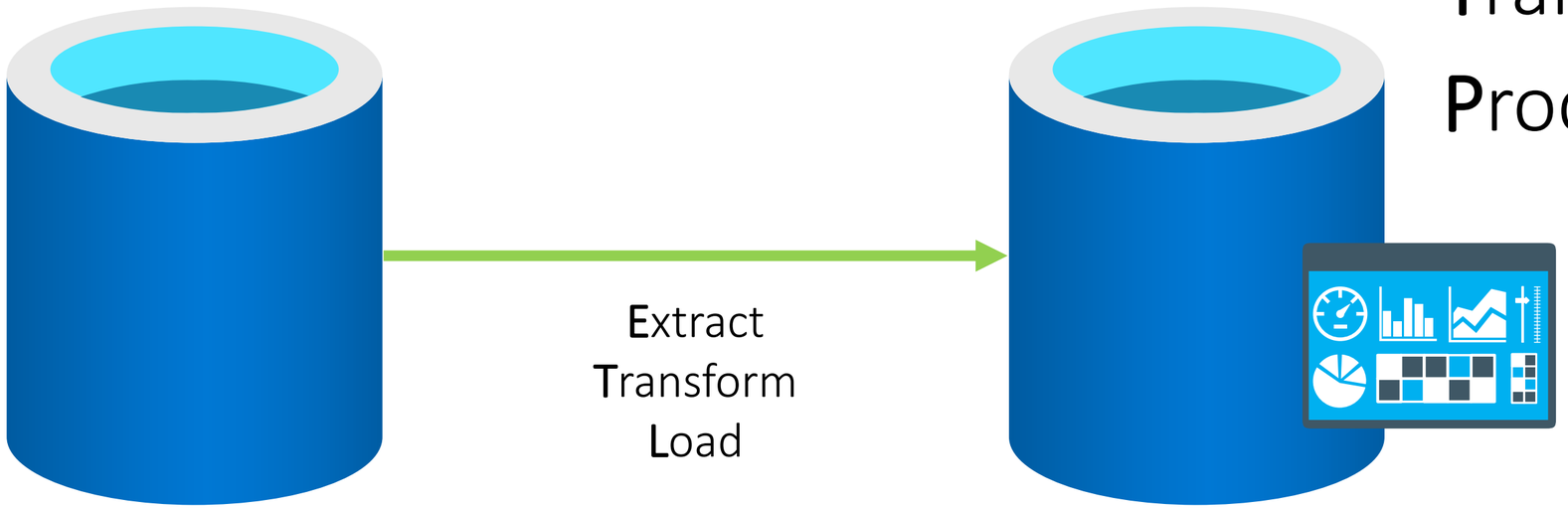
Cloud Formations - Knowledge Transfer & Training

Databases



Creating a Data Warehouse

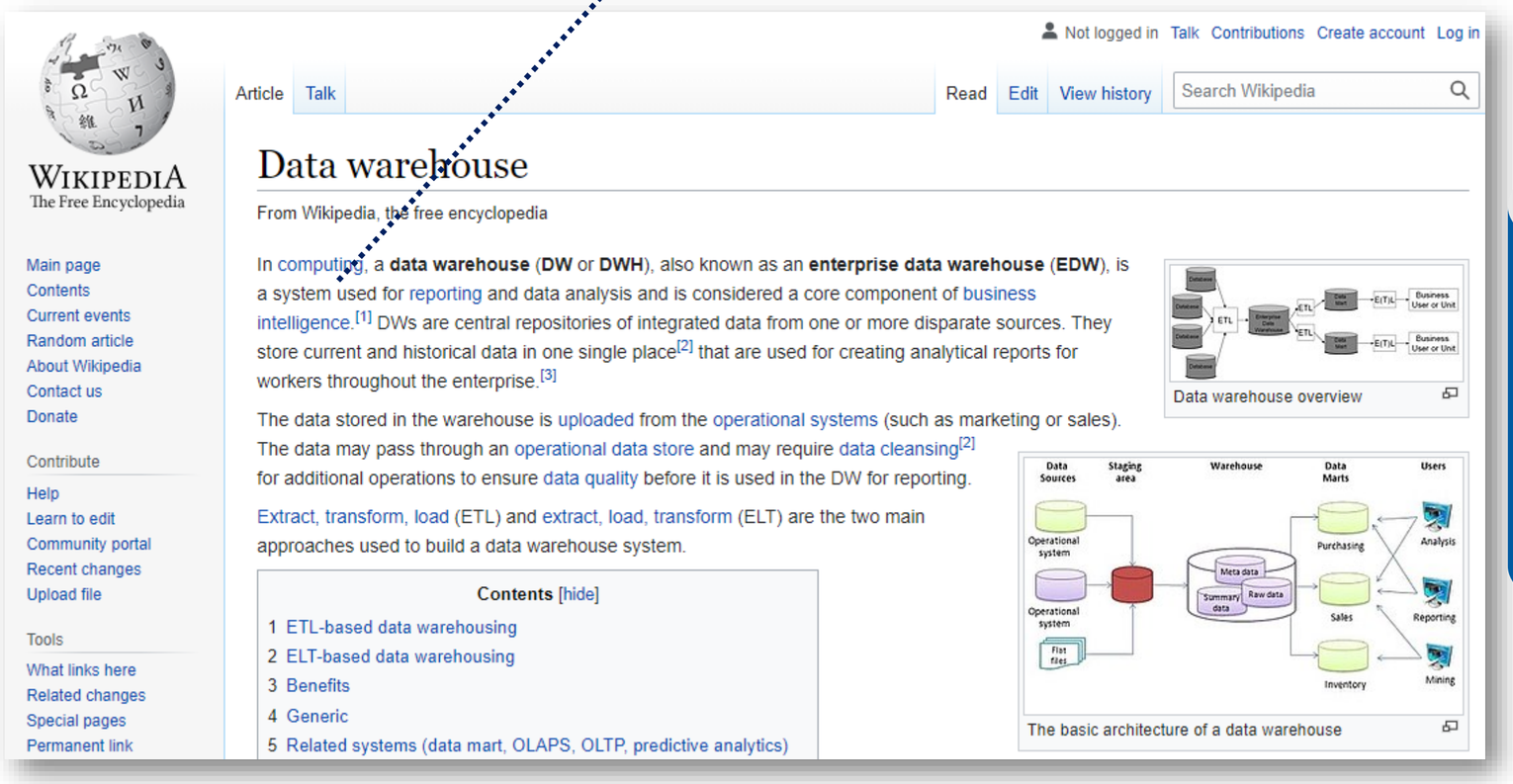
Online
Line
Transactional
Processing



Offline
Analytical
Transactional
Processing

Creating a Data Warehouse

"a system for reporting and data analysis"



Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

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

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]

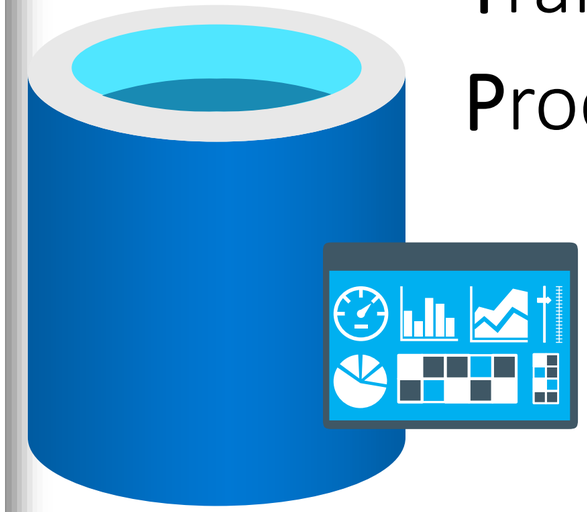
1 ETL-based data warehousing
2 ELT-based data warehousing
3 Benefits
4 Generic
5 Related systems (data mart, OLAPS, OLTP, predictive analytics)

Data warehouse overview

The basic architecture of a data warehouse

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

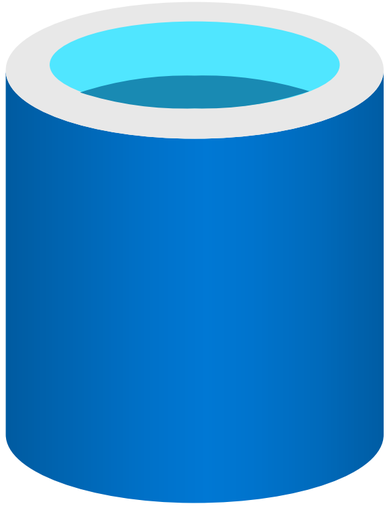
Offline
Analytical
Transactional
Processing



Data Warehouse

Cloud Formations - Knowledge Transfer & Training

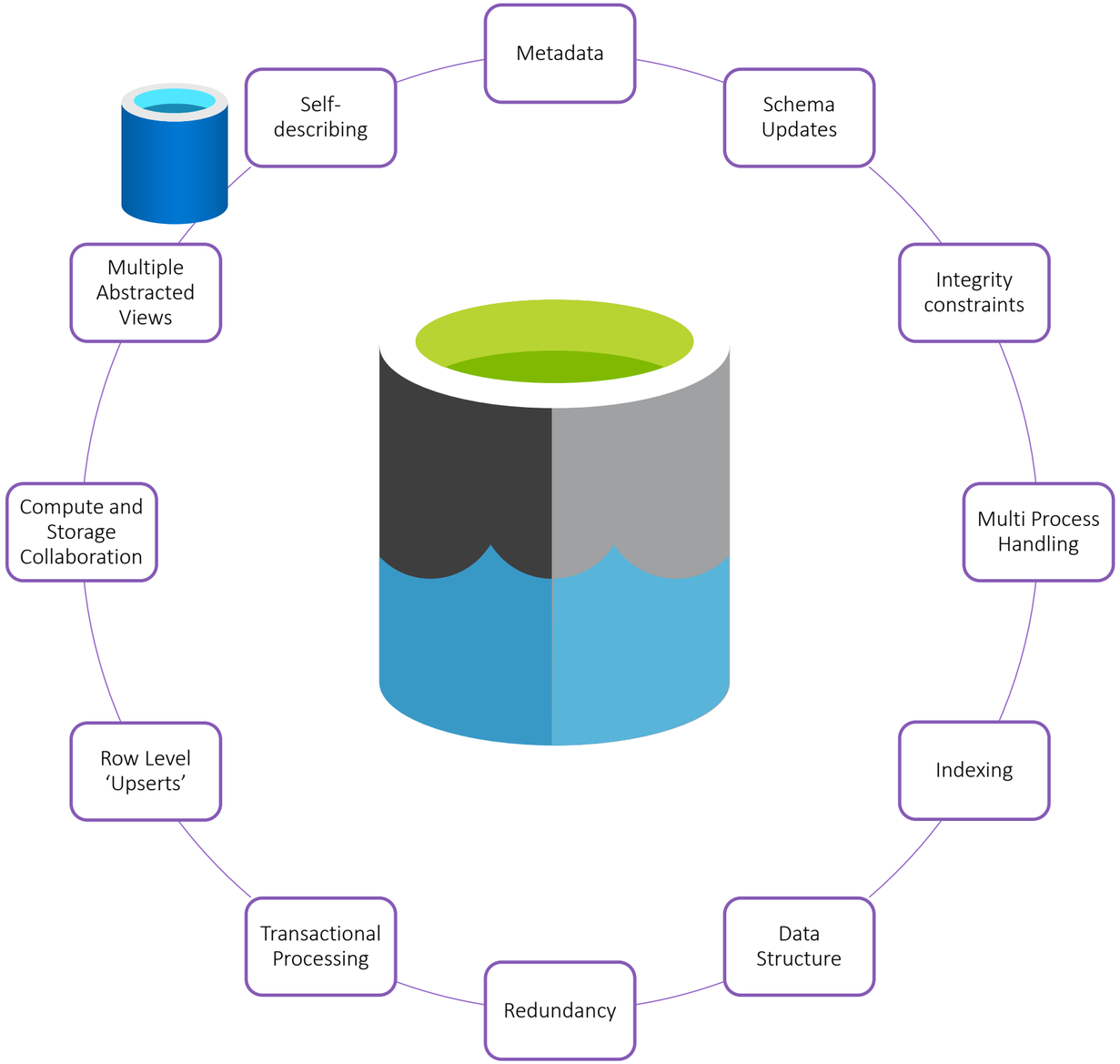
Databases



Data Lakes

Big Data:

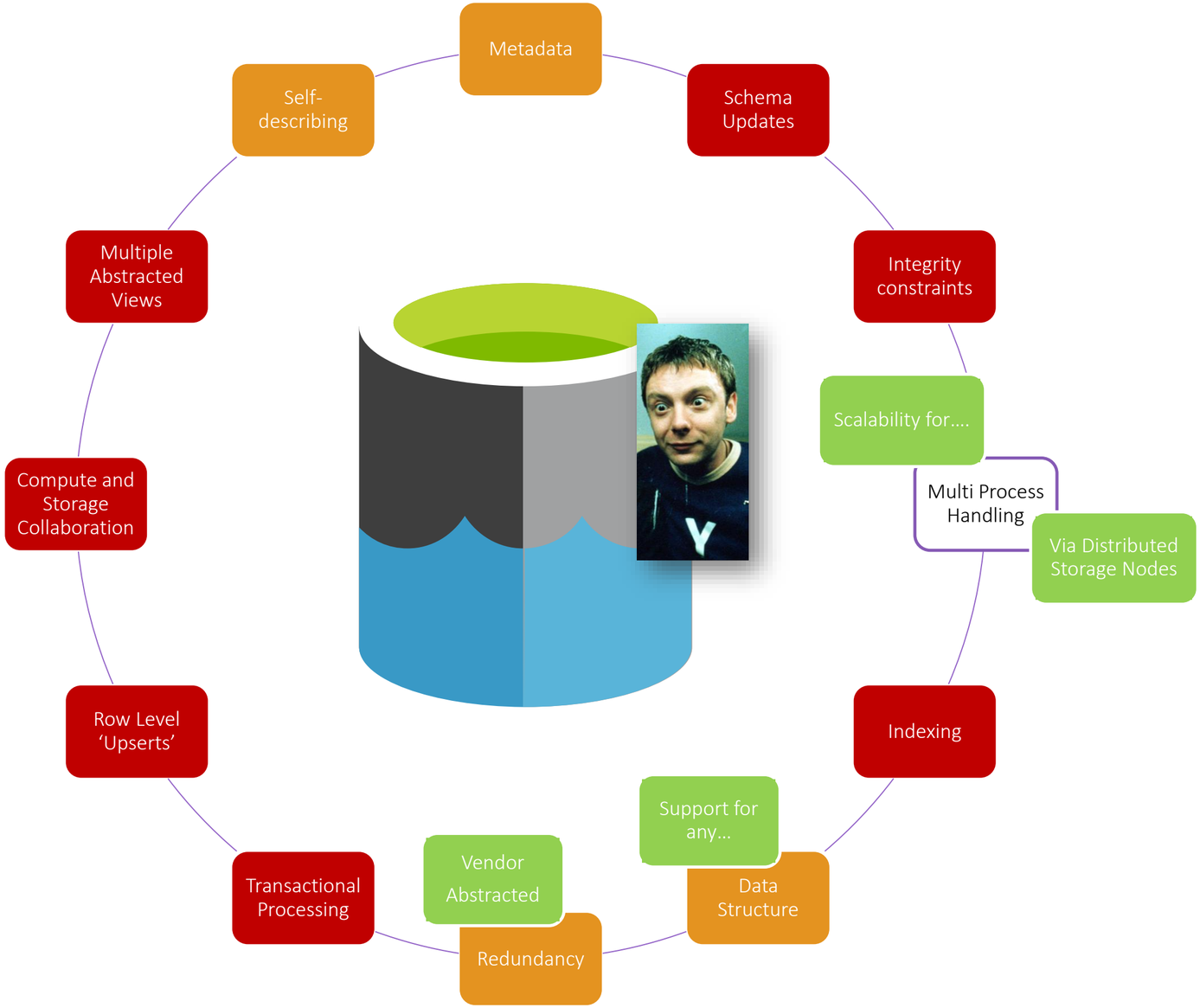
- **V**olume
- **V**elocity
- **V**ariety
- **V**eracity
- **V**alue



Data Lakes

Big Data:

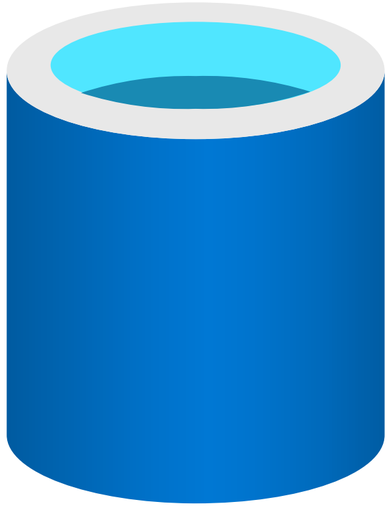
- **V**olume
- **V**elocity
- **V**ariety
- **V**eracity
- **V**alue



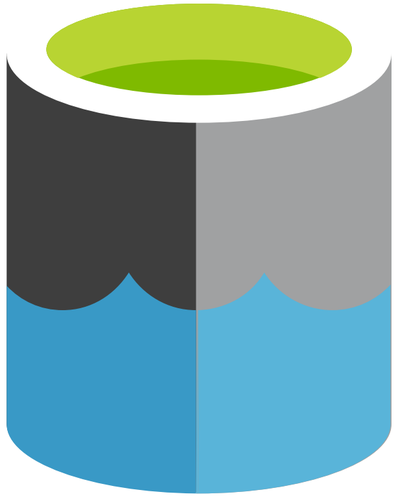
Problem Summary

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

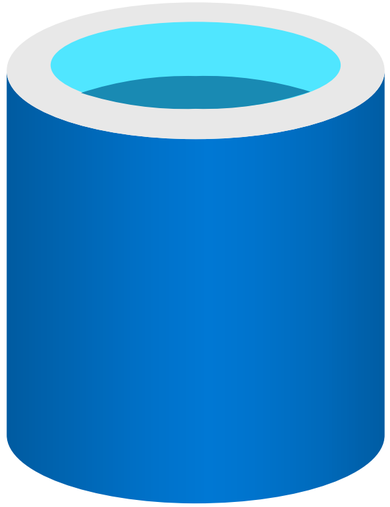


Scales Up	Scales Out
Natural Home for Structured Data	Any Data Structure
Storage Limits	No Storage Limits
Transactional Resilience	No Transactional Handling
Storage & Compute Coupled	Storage & Compute Decoupled

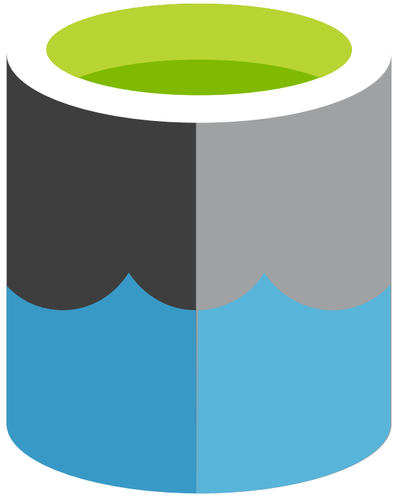
Problem Summary

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



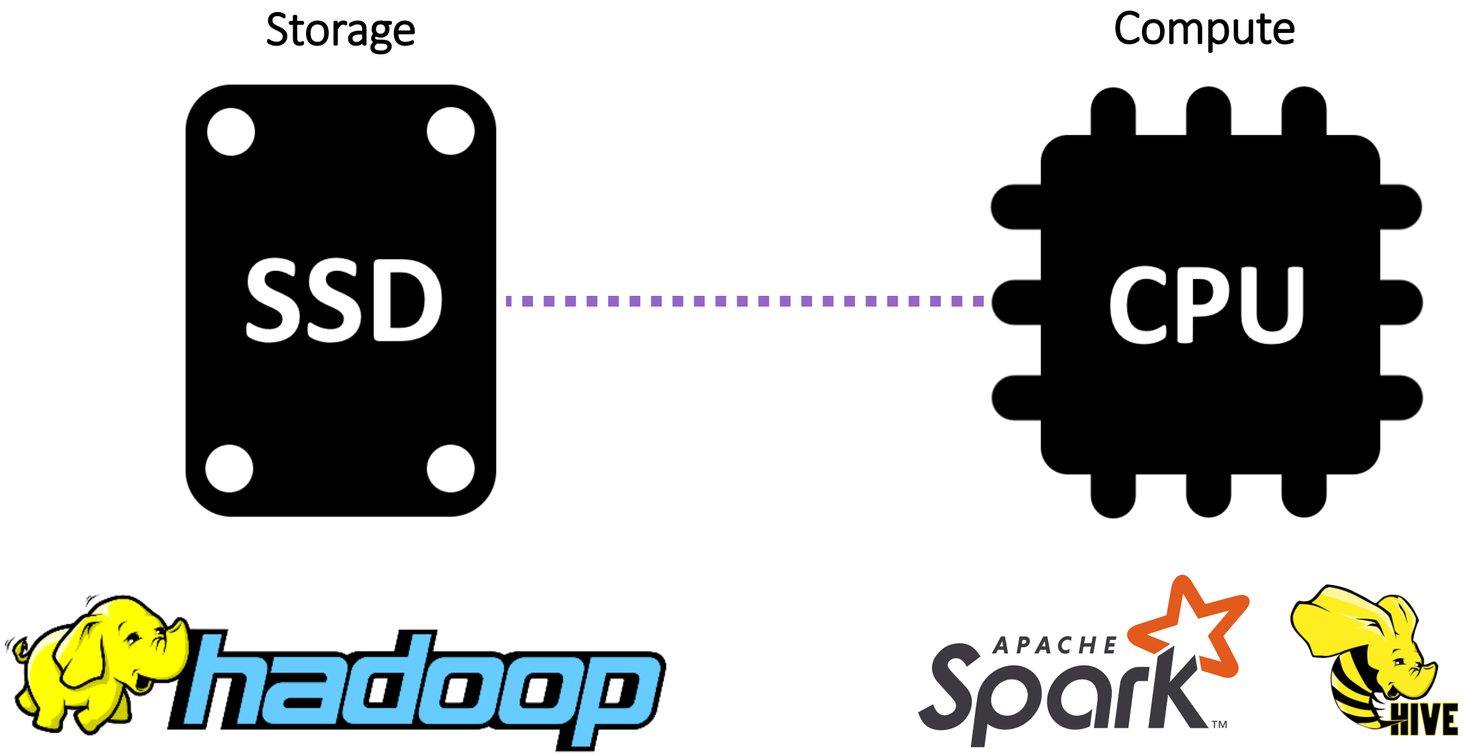
Scales Up	Scales Out
Natural Home for Structured Data	Any Data Structure
Storage Limits	No Storage Limits
Transactional Resilience	No Transactional Handling
Storage & Compute Coupled	Storage & Compute Decoupled

Solution

‘Just’ enable ACID transactional support for Data Lakes...

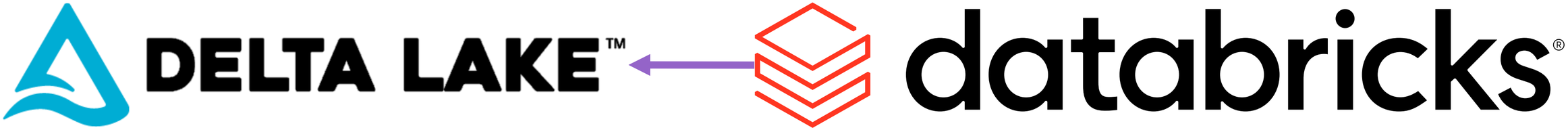
Big Data:

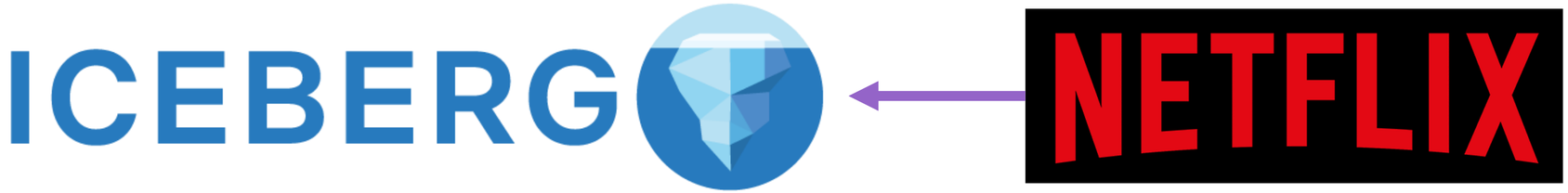
- Volume
- Velocity
- Variety
- Veracity
- Value



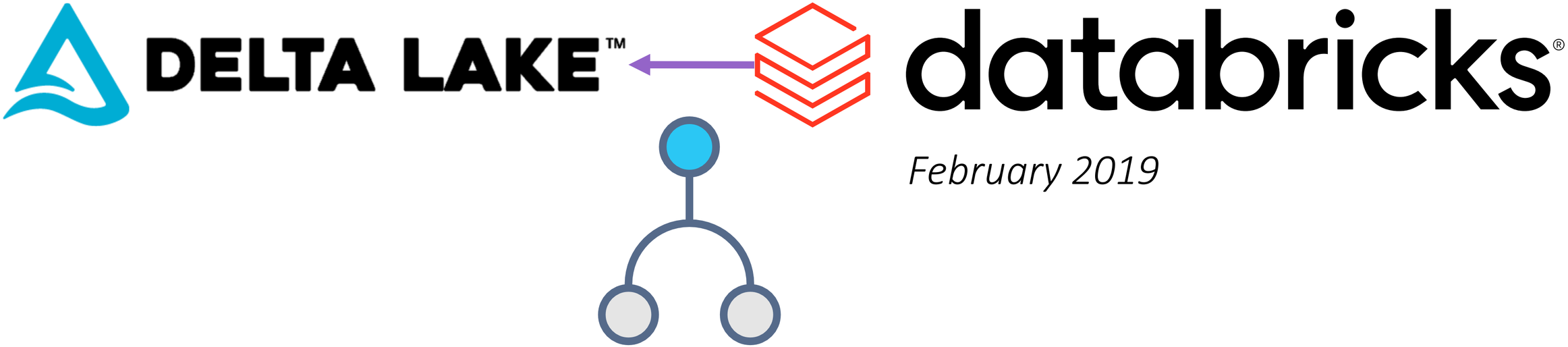
Storage & Compute ~~Decoupled~~ Working Together Again As Friends!

ACID Data Frameworks for Data Lakes





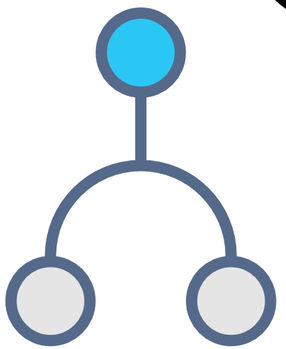
What is Delta Lake?



February 2019

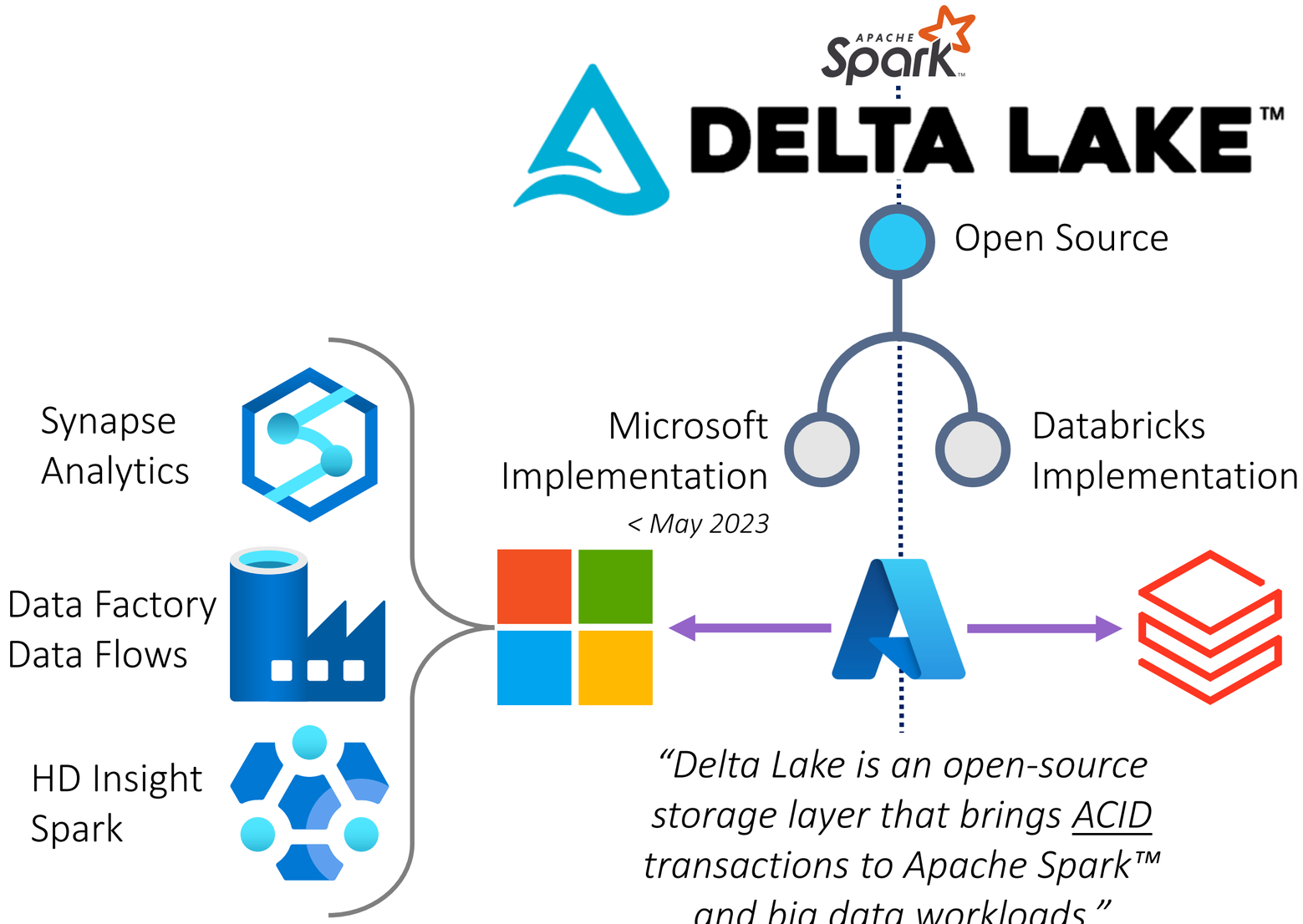
What is Delta Lake?

https://delta.io
https://github.com/delta-io/delta
April 2019

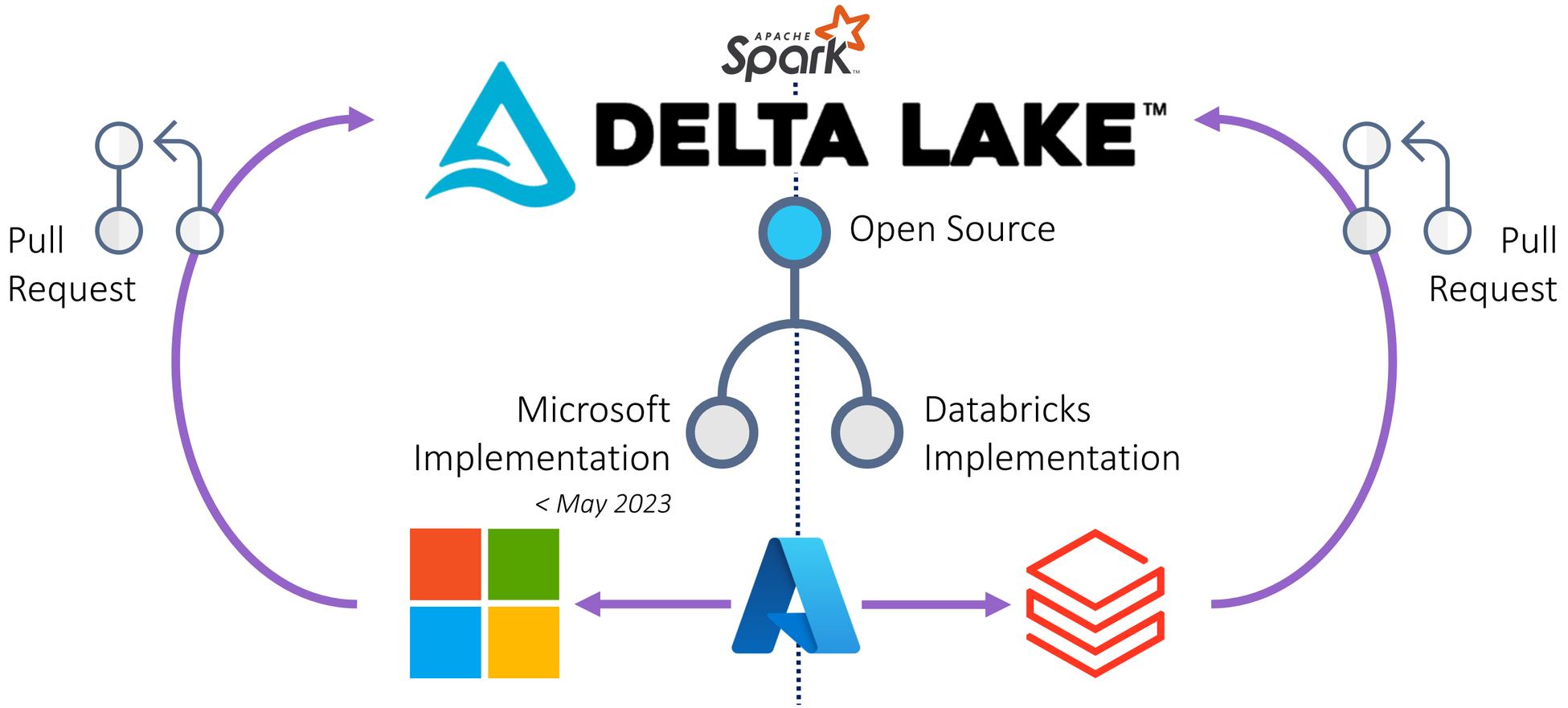


databricks®

What is Delta Lake?

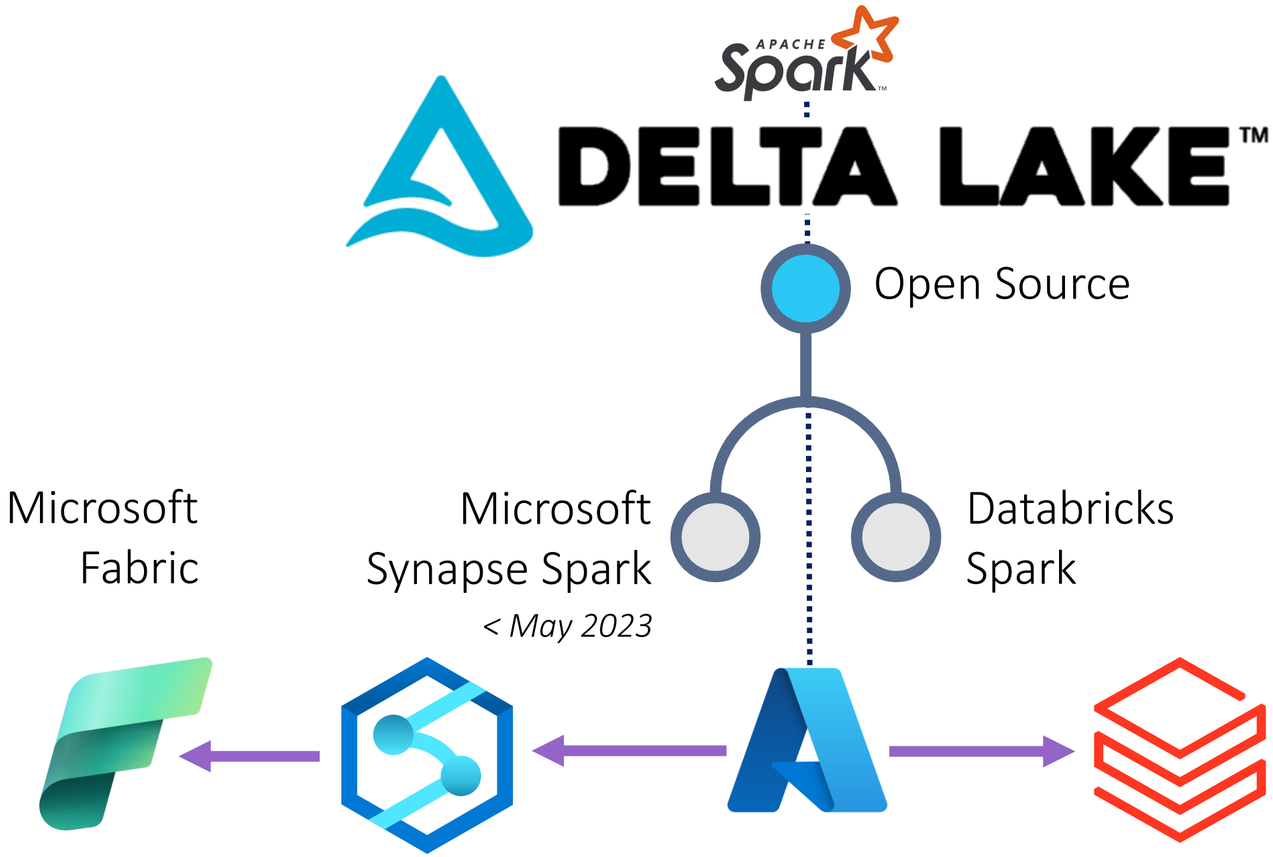


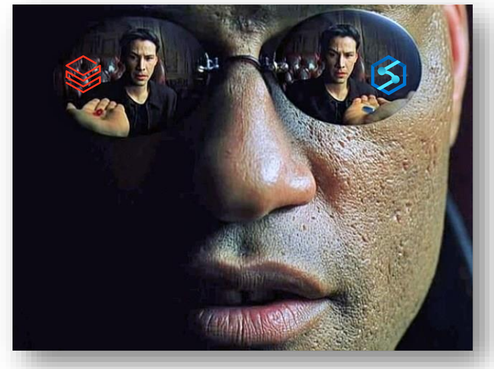
What is Delta Lake?



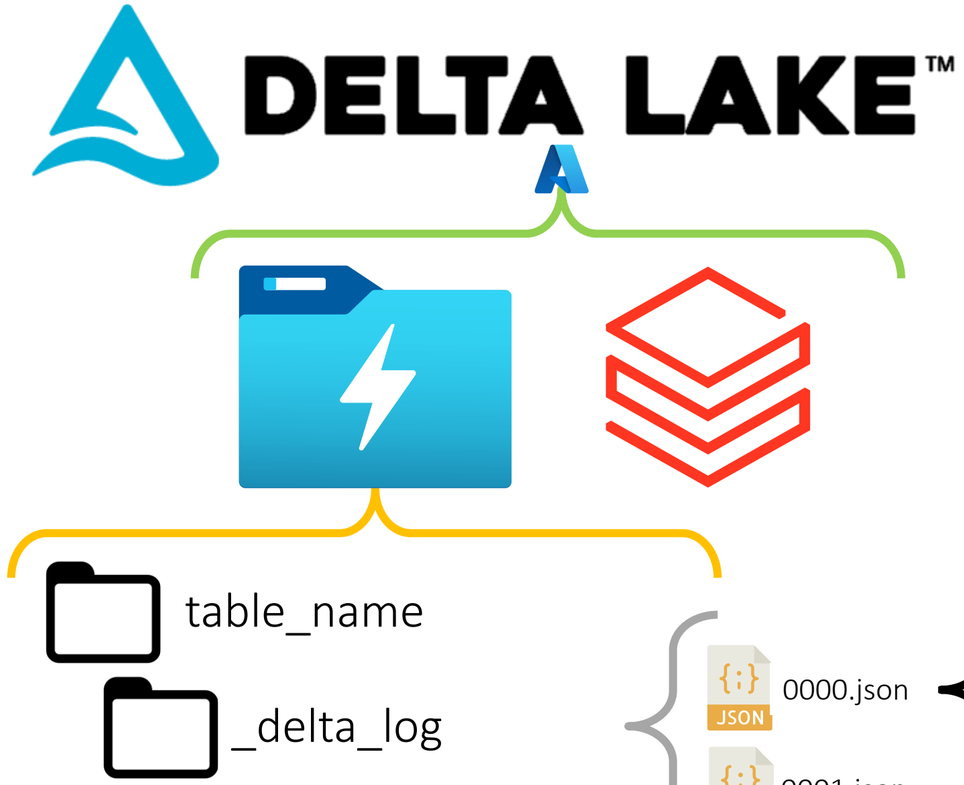
"Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark™ and big data workloads."

Which Spark Implementation is Better?



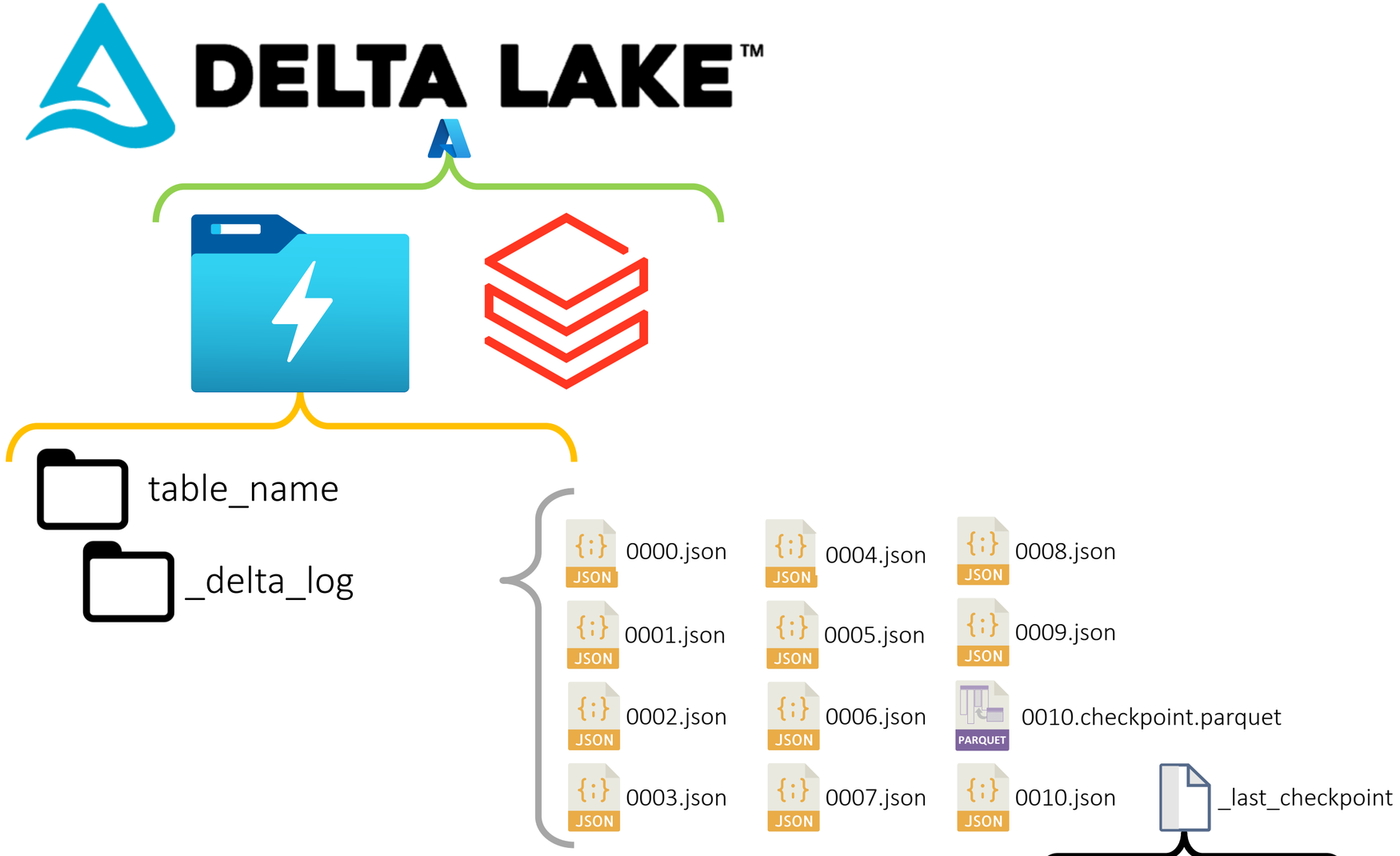


Delta Table - Transaction Log

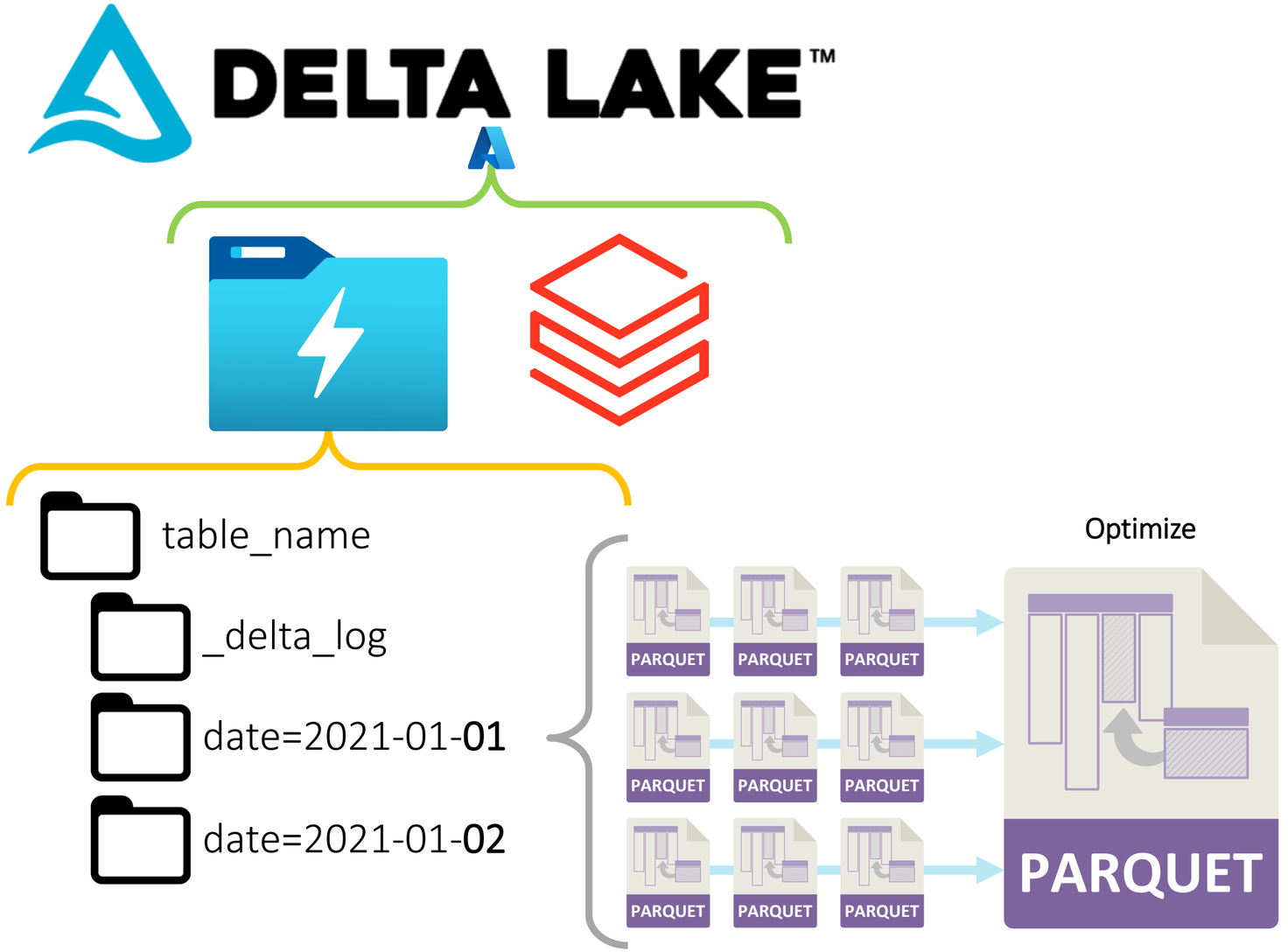



```
{
    "commitInfo":
    {
        "timestamp":1628596034417,
        "operation":"WRITE",
        "operationParameters":
        {
            "mode":"ErrorIfExists",
            "partitionBy":"[]"
        },
        "isBlindAppend":true,
        "operationMetrics":
            {
                "numFiles":"6",
                "numOutputBytes":"2407",
                "numOutputRows":"5"
            }
    }
}
{"protocol":{"minReaderVersion":1,"minWriterVersion":2}}
{
    "metaData":
    {
    "id":"58e5de01-de72-4d5b-a208-d0b4ae919efe",
    "format":
    {
        "provider":"parquet",
        "options":{}
    },
    "schemaString":
                "{\"type\":\"struct\",\"fields\":[{\"name\":\"id\",\"type\":\"long\",
                  \"nullable\":true,\"metadata\":{}}]}",
    "partitionColumns":[],
    "configuration":{},
    "createdTime":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}}
```

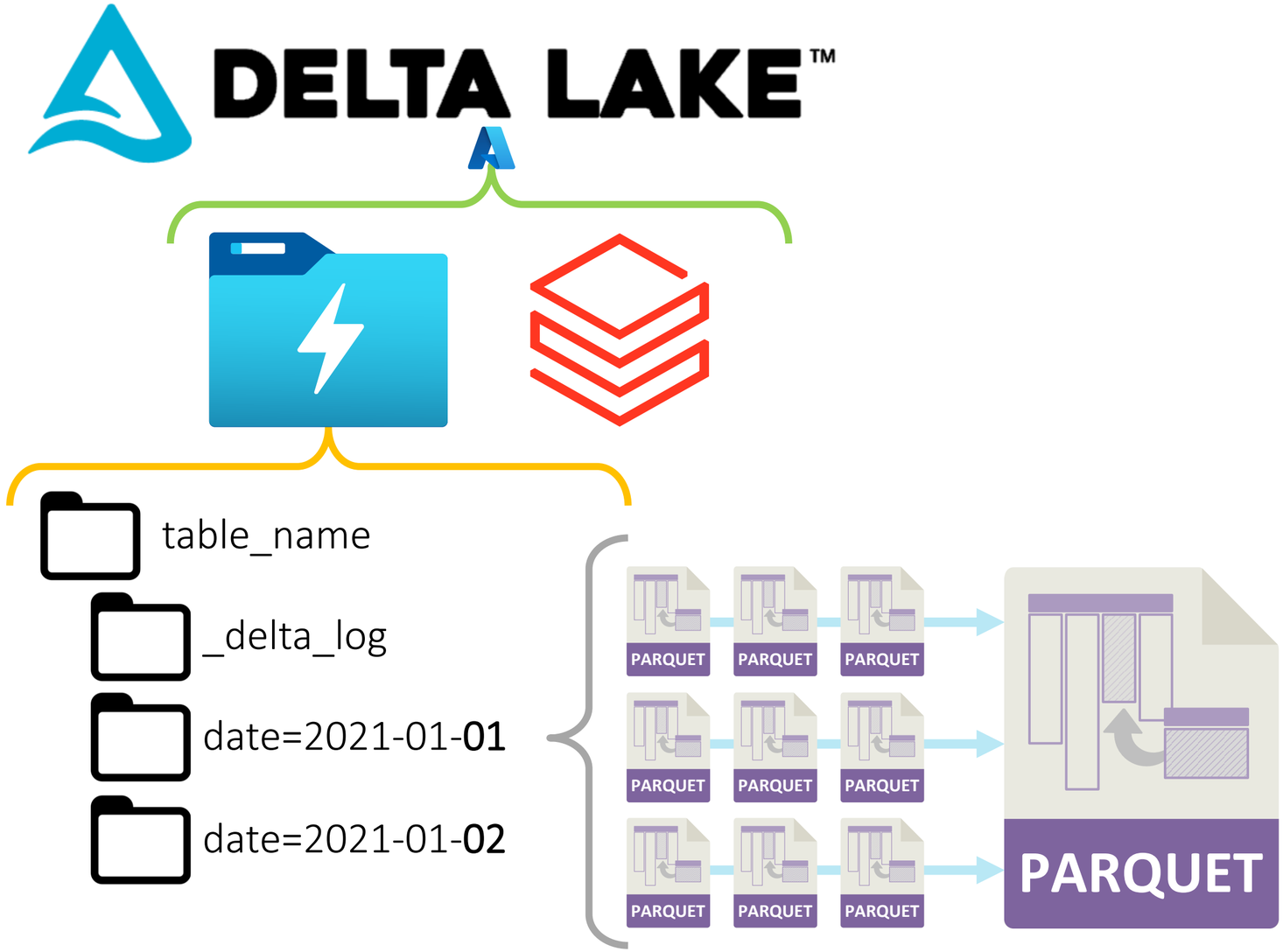
Delta Table - Transaction Log



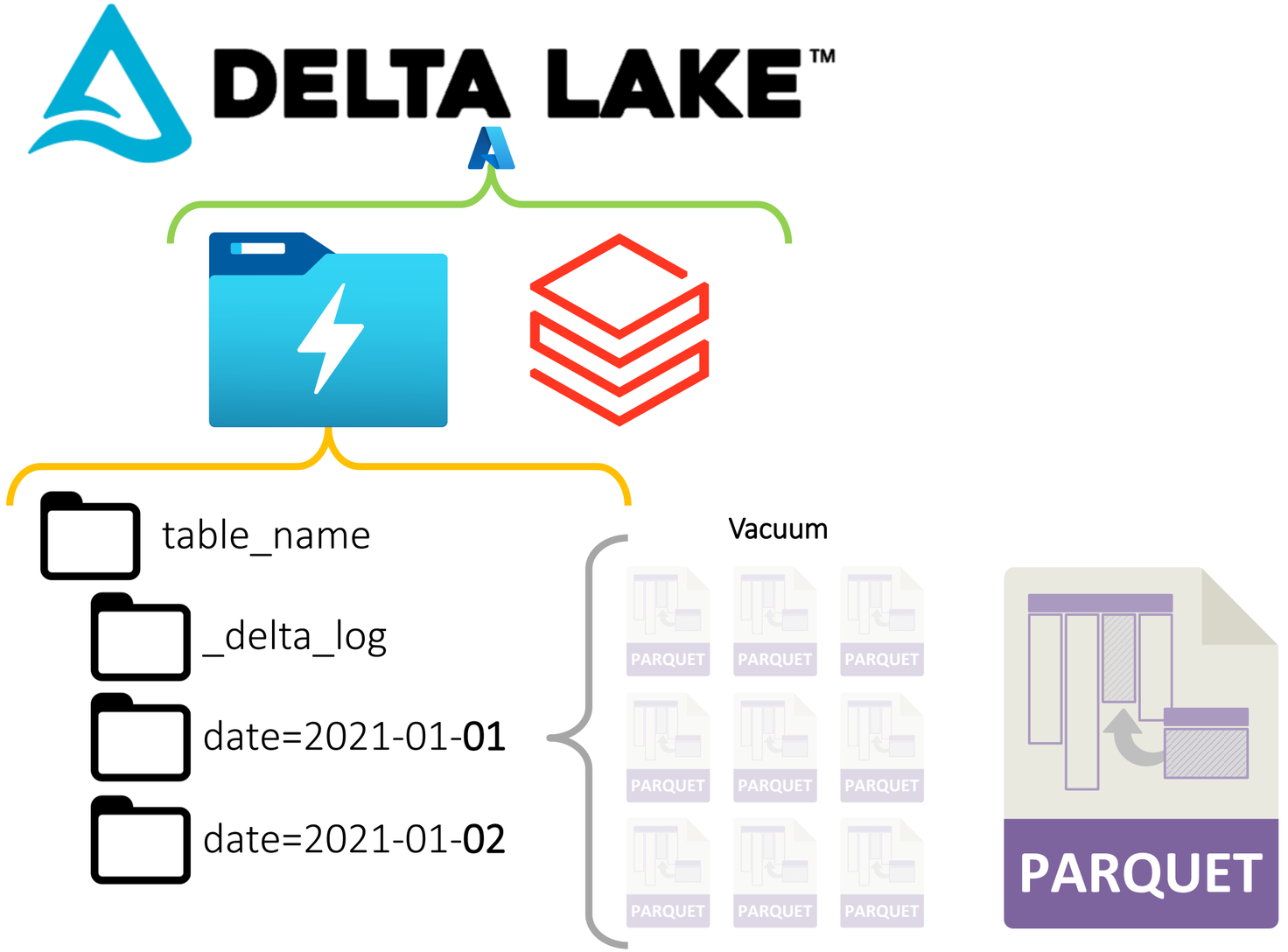
Delta Table - Transaction Log



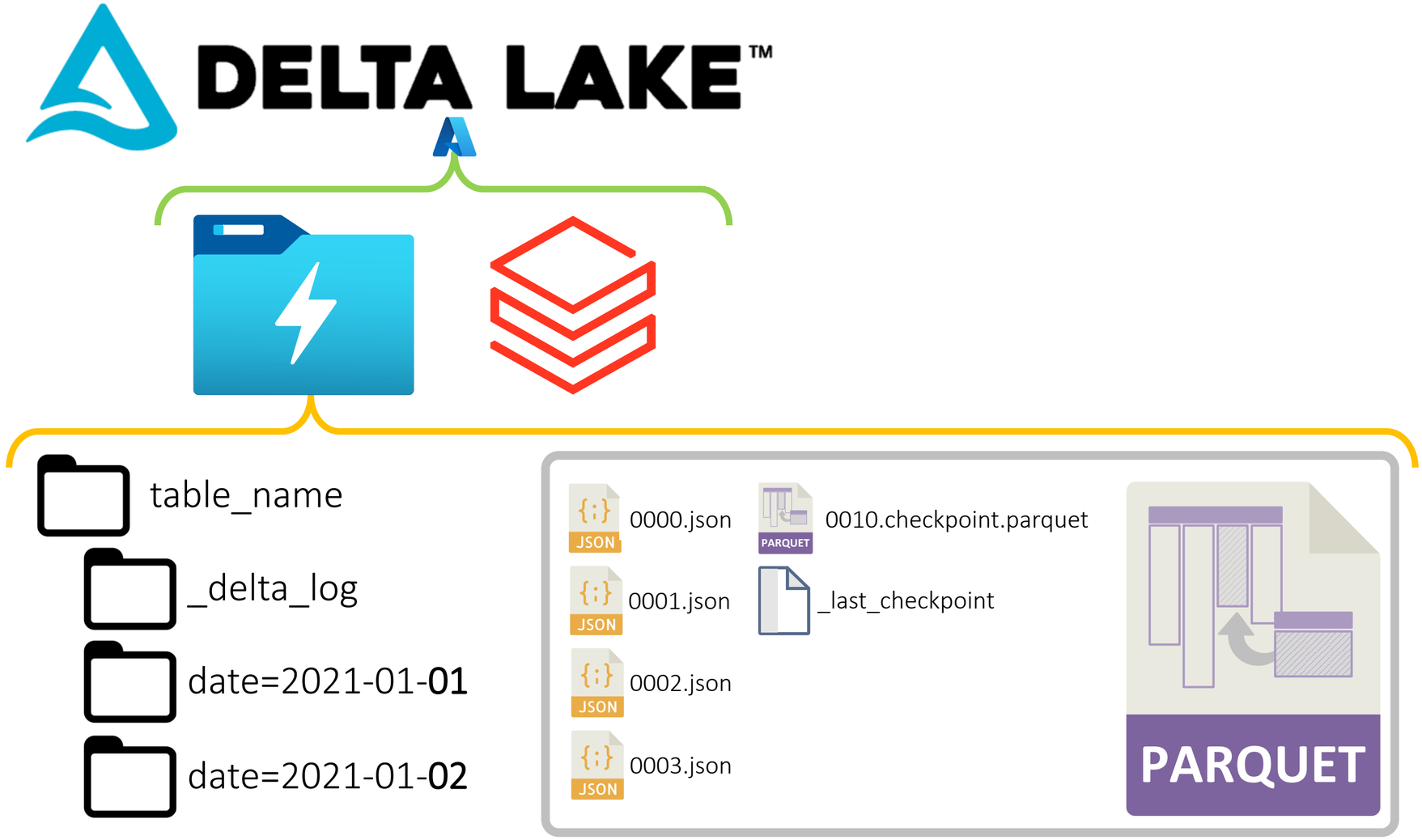
Delta Table - Transaction Log



Delta Table - Transaction Log



Delta Table – On Disk



DELTA LAKE WITH PYTHON

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

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")
```

databricks

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
[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.
```


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 VACCUM.
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

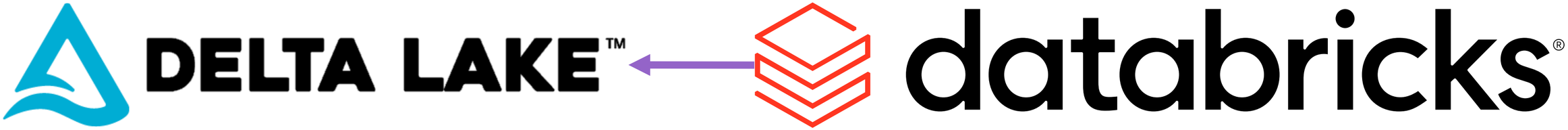
```
*Databricks Delta Lake feature
ALTER TABLE [table_name | delta.`path/to/delta_table`]
SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true)
```

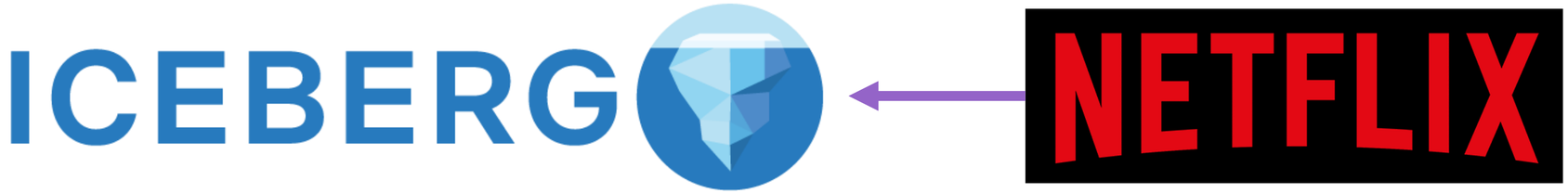
Cache frequently queried data in Delta Cache

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

databricks

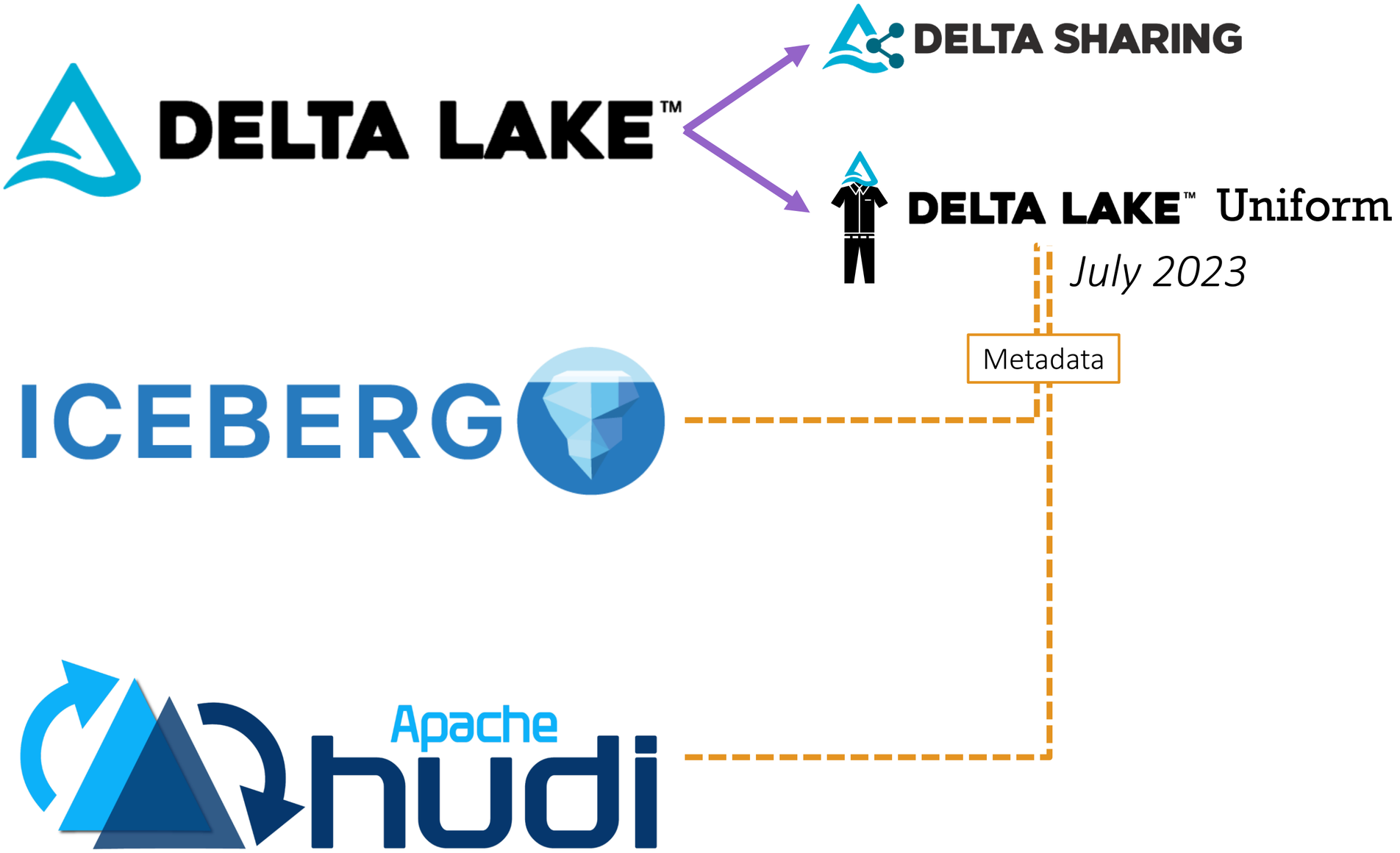
ACID Data Frameworks for Data Lakes





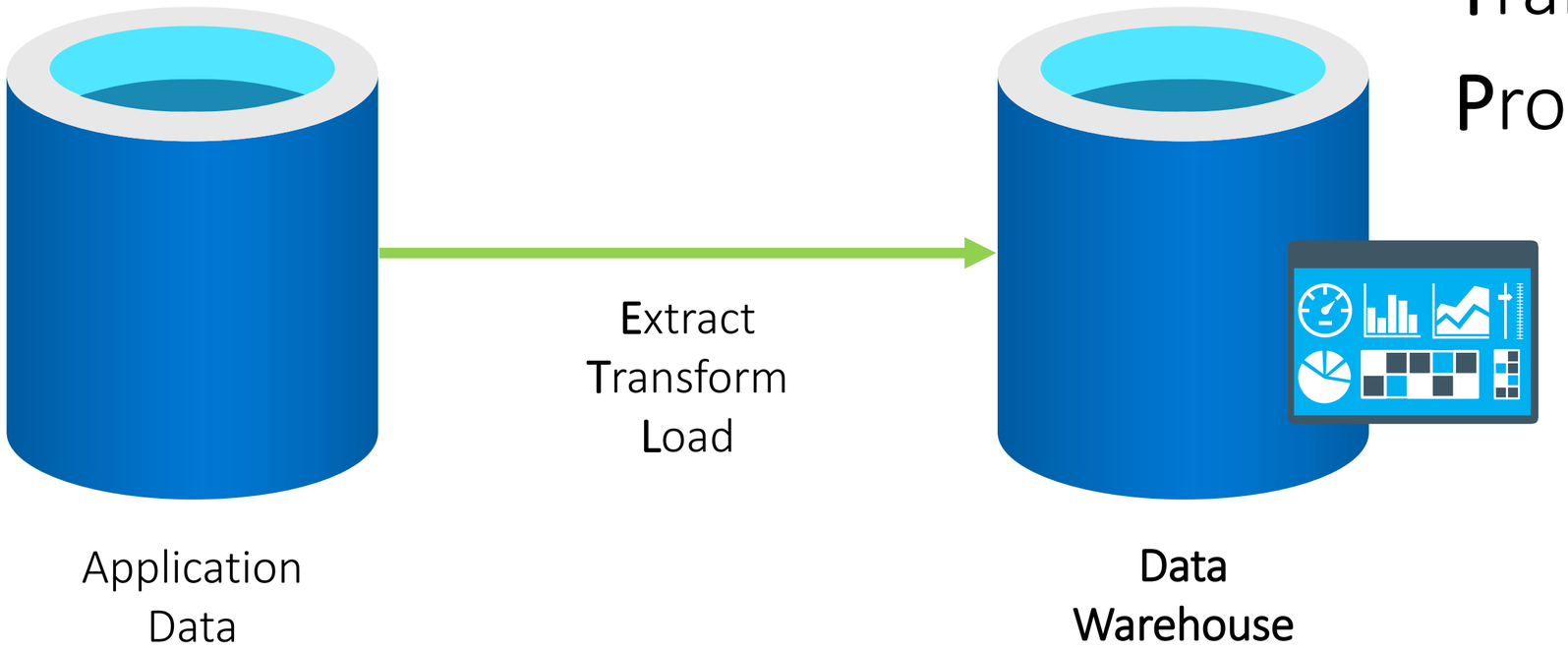
ACID Data Frameworks for Data Lakes

YouTube https://www.youtube.com/watch?v=CU8krhcNI2M



Data Warehouse

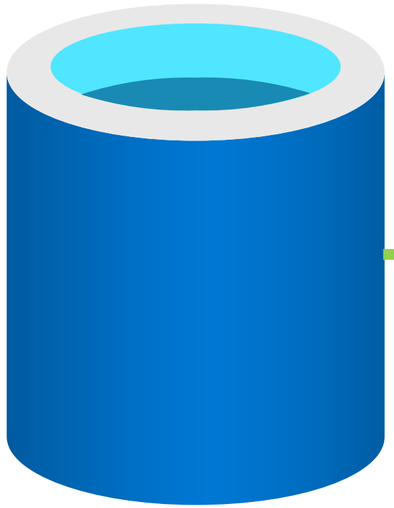
Online
Line
Transactional
Processing



Offline
Analytical
Transactional
Processing

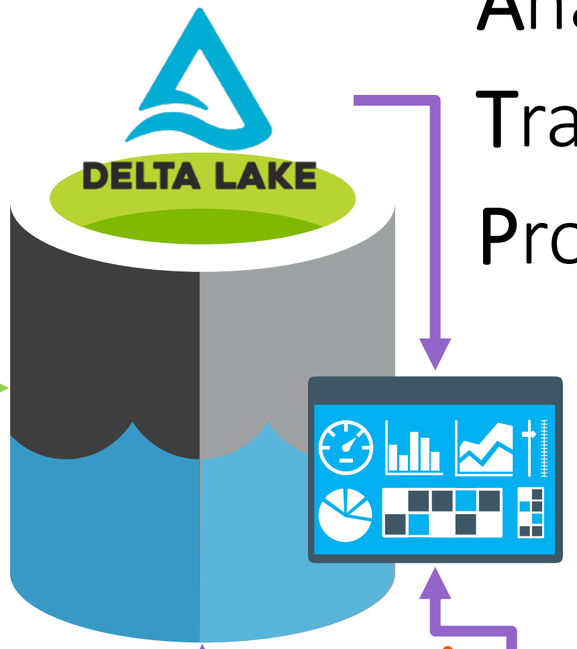
Lake House

Online
Line
Transactional
Processing

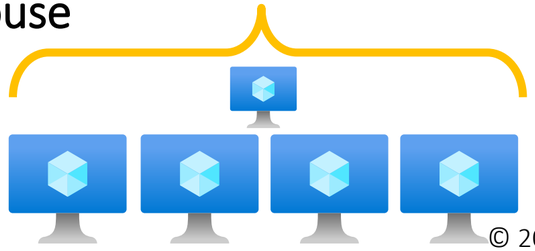


Application
Data

Extract
Transform
Load



Lake
House



Offline
Analytical
Transactional
Processing

https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html

Lake House

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

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
Page information
Cite this page
Wikidata item

Print/export
Download as PDF
Printable version

Languages
العربية
Deutsch
Español
Français

The Lake House (film)

From Wikipedia, the free encyclopedia

This article includes a list of general references, but it remains largely unverified because **it lacks sufficient corresponding inline citations**. Please help to improve this article by introducing more precise citations. *(October 2017) (Learn how and when to remove this template message)*

The Lake House is a 2006 American fantasy romantic drama film directed by Alejandro Agresti, starring Keanu Reeves and Sandra Bullock (who had previously appeared together in the box office hit *Speed*). It was written by David Auburn.[2] A remake of the South Korean motion picture *Il Mare* (2000), it centers on an architect living in 2004 and a doctor living in 2006 who meet via letters left in a mailbox at the lake house where they have lived at separate points in time. They carry on correspondence over two years, remaining separated by their original difference of two years.[3]

Contents [hide]
1 Plot
2 Cast
3 Production
4 Music
5 Reception
5.1 Box office
5.2 Critical response
5.3 Home media
5.4 Awards
6 References
7 External links

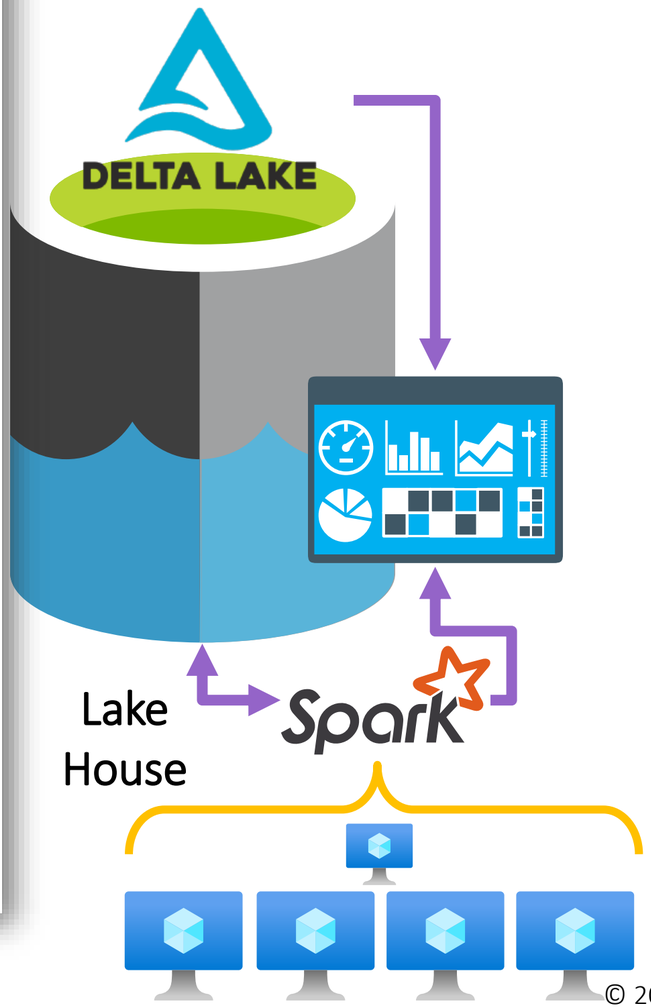
Plot [edit]

In 2006, Dr. Kate Forster (Sandra Bullock) is leaving a lake house that she has been renting in Chicago. Kate leaves a note in the mailbox for the next tenant to forward her mail, adding that the paint-embedded pawprints on the path leading to the house were already there when she arrived.

The Lake House

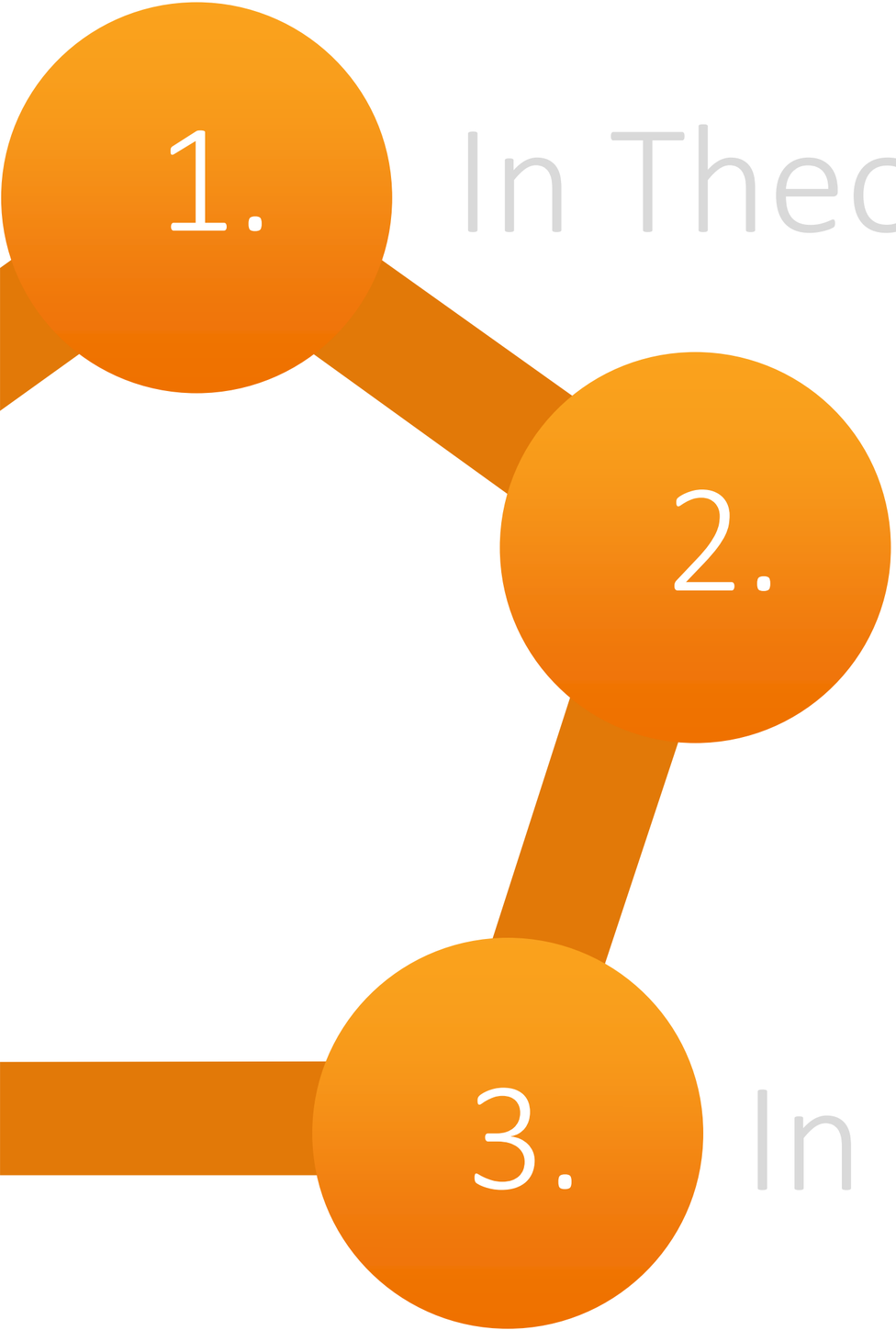
Theatrical release poster

Directed by	Alejandro Agresti
Written by	David Auburn
Based on	*Il Mare* by Kim Eun-jeong Kim Mi-yeong
Produced by	Doug Davison Roy Lee
Starring	Keanu Reeves



Cloud Formations - Knowledge Transfer & Training

Agenda



In Theory

In Practice

In Reality

Create a resource
Home
Dashboard
All services
FAVORITES
Application Insights
SQL servers
Microsoft Purview accounts
SQL databases
Dedicated SQL pools (formerly SQL DW)
Analysis Services
Azure Cosmos DB
Azure Synapse Analytics
Azure Data Explorer Clusters
Data factories
Batch accounts
Azure Databricks
Data Lake Analytics
Data Lake Storage Gen1
Storage accounts
IoT Hub
Event Hubs
Event Grid Subscriptions
Stream Analytics jobs

Delta Lakes

Private dashboard

Create | Upload | Refresh | Full screen | Edit | Share | Export | Clone | Assign tags | Delete | Feedback

Auto refresh : **Off**

Last updated: an hour ago

learningsynap...
Synapse workspace

Kind

Location
North Europe

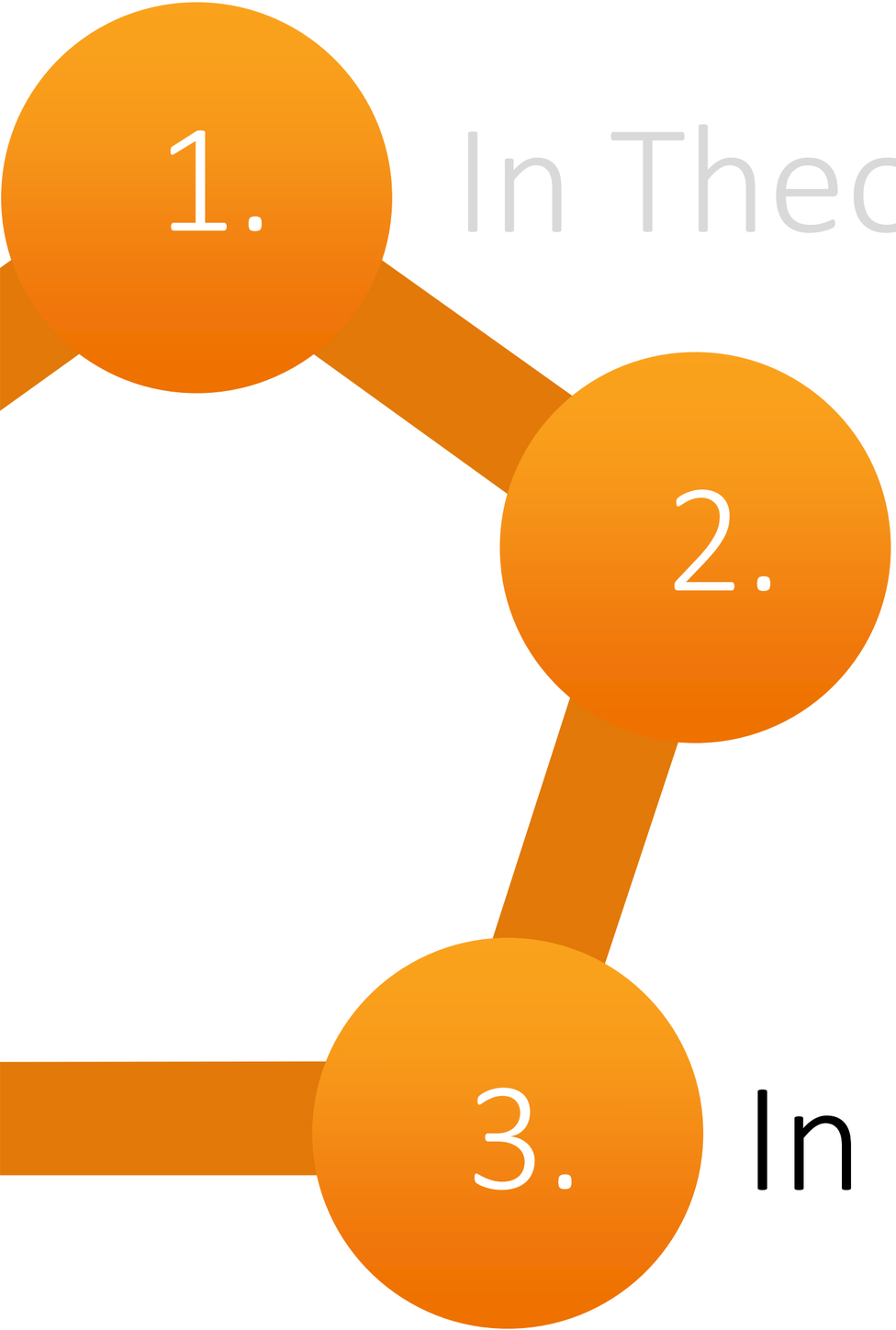
BricksOfData01
Workspace

learningsynap...
Storage account

Kind
StorageV2

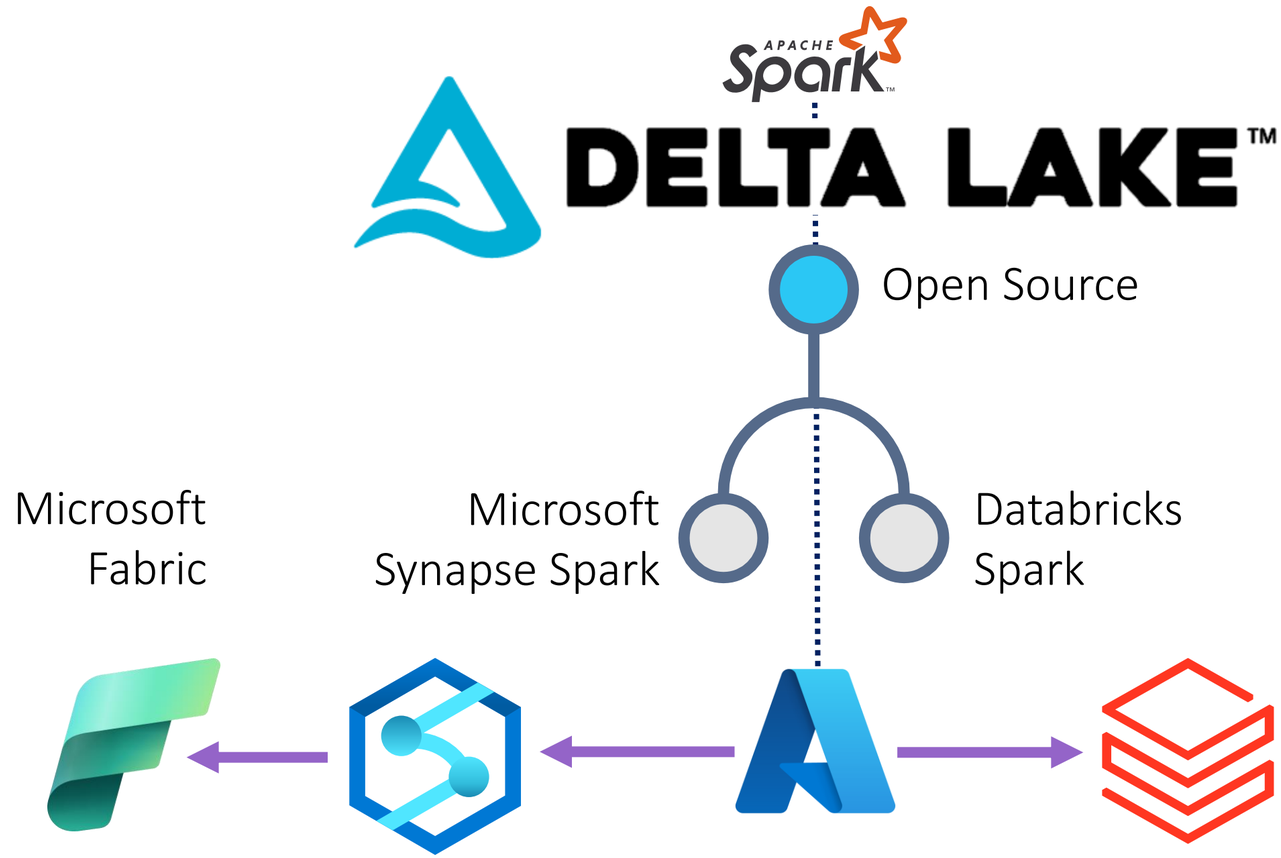
Location
North Europe

Agenda



1. In Theory
2. In Practice
3. In Reality

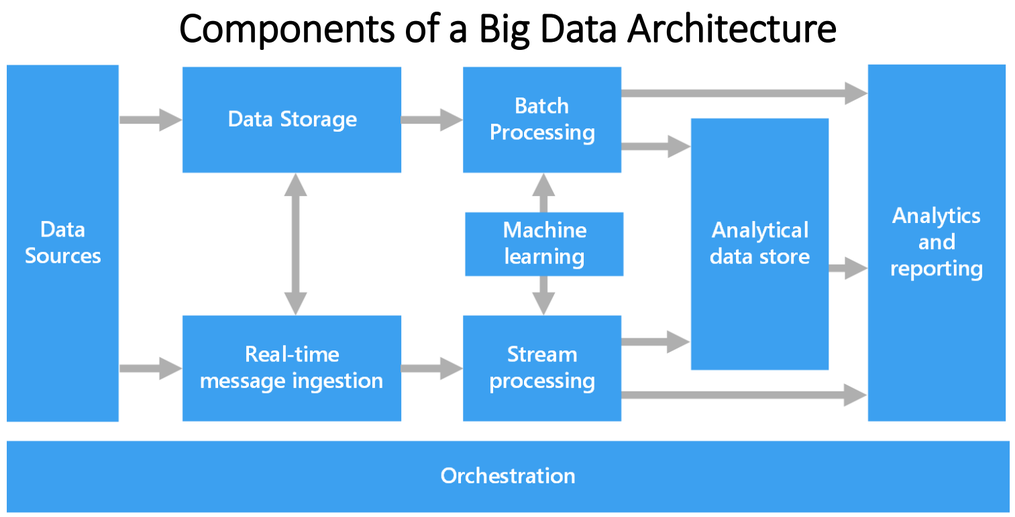
The Problem Is Choice



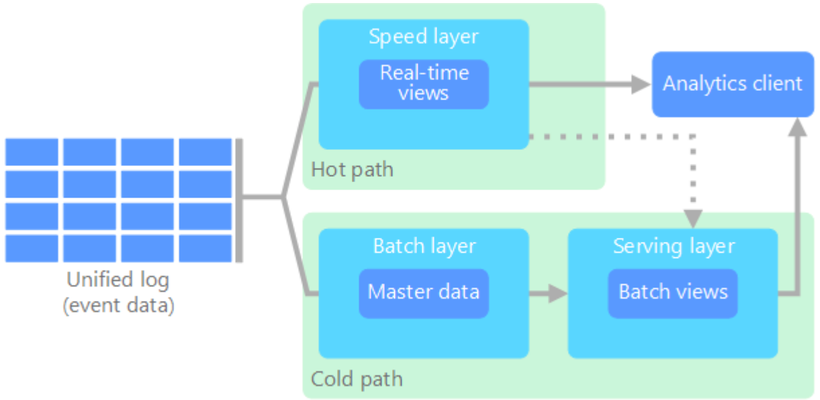
The Problem Is Choice

The Problem Is Choice

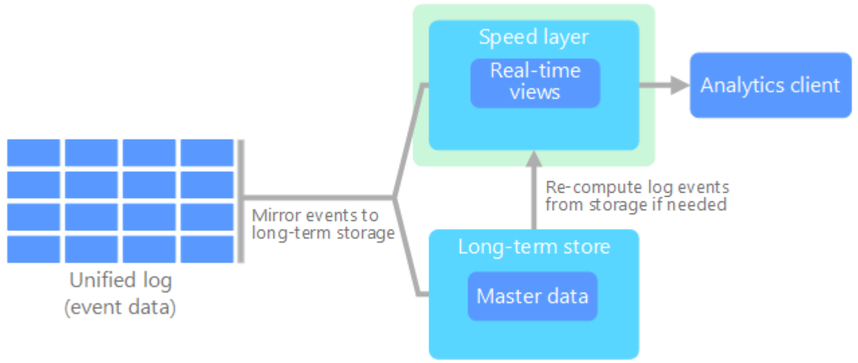
Lambda & Kappa Architectures



Lambda



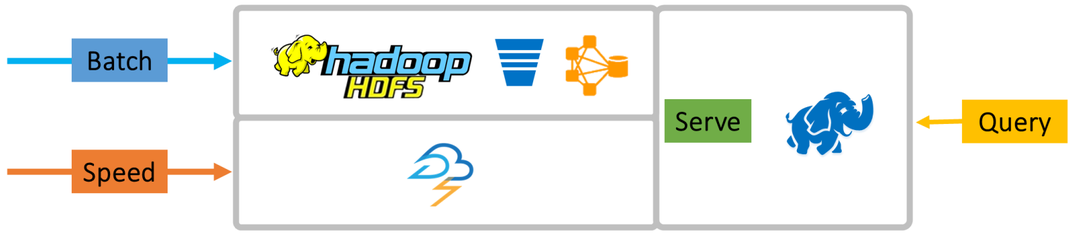
Kappa

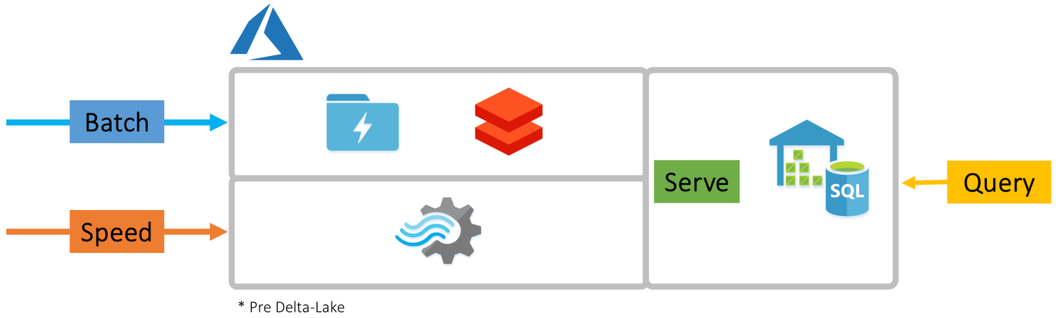


https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/

Delta Lake in the Context of Lambda & Kappa

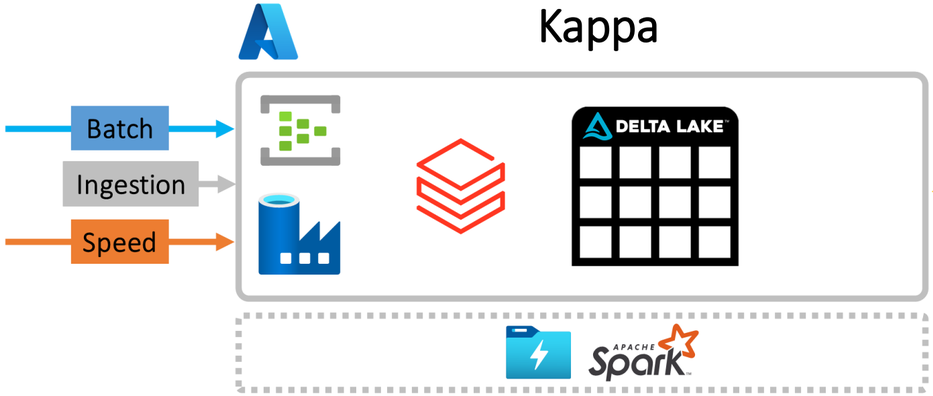
Lambda

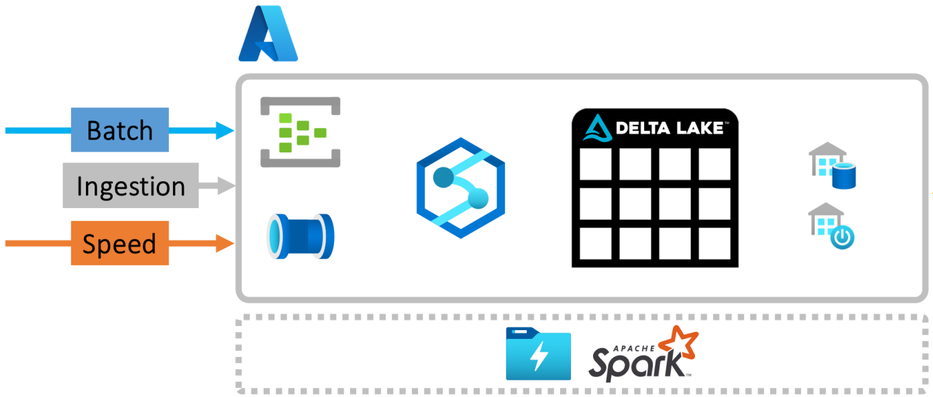


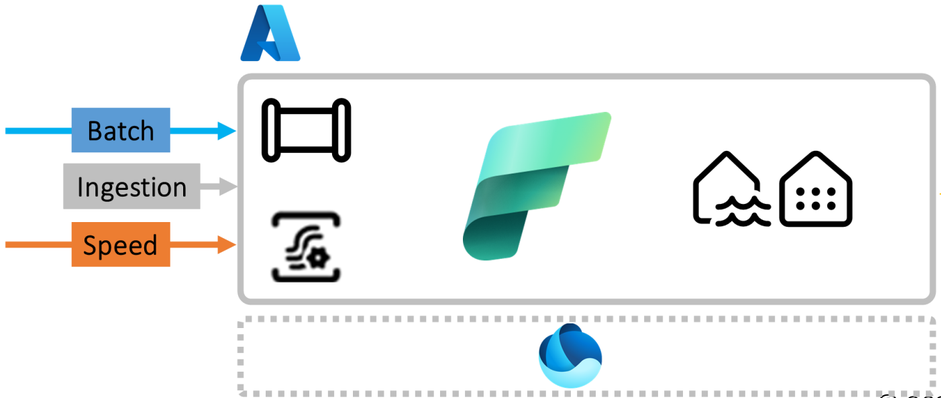


* Pre Delta-Lake

Kappa



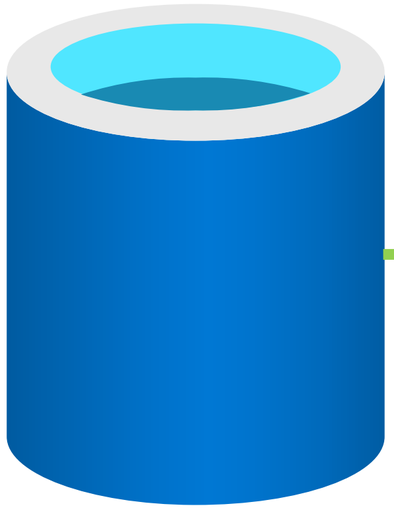




Cloud Formations - Knowledge Transfer & Training

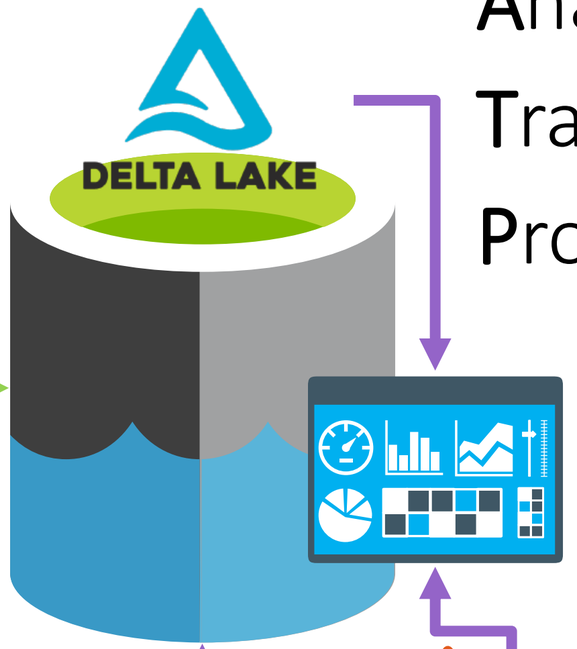
Lake House

Online
Line
Transactional
Processing

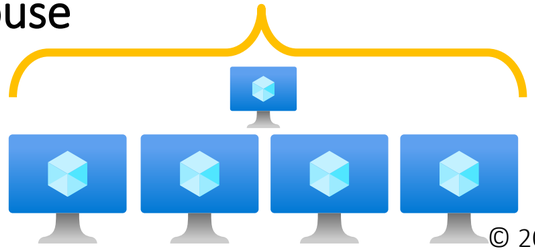


Application
Data

Extract
Transform
Load



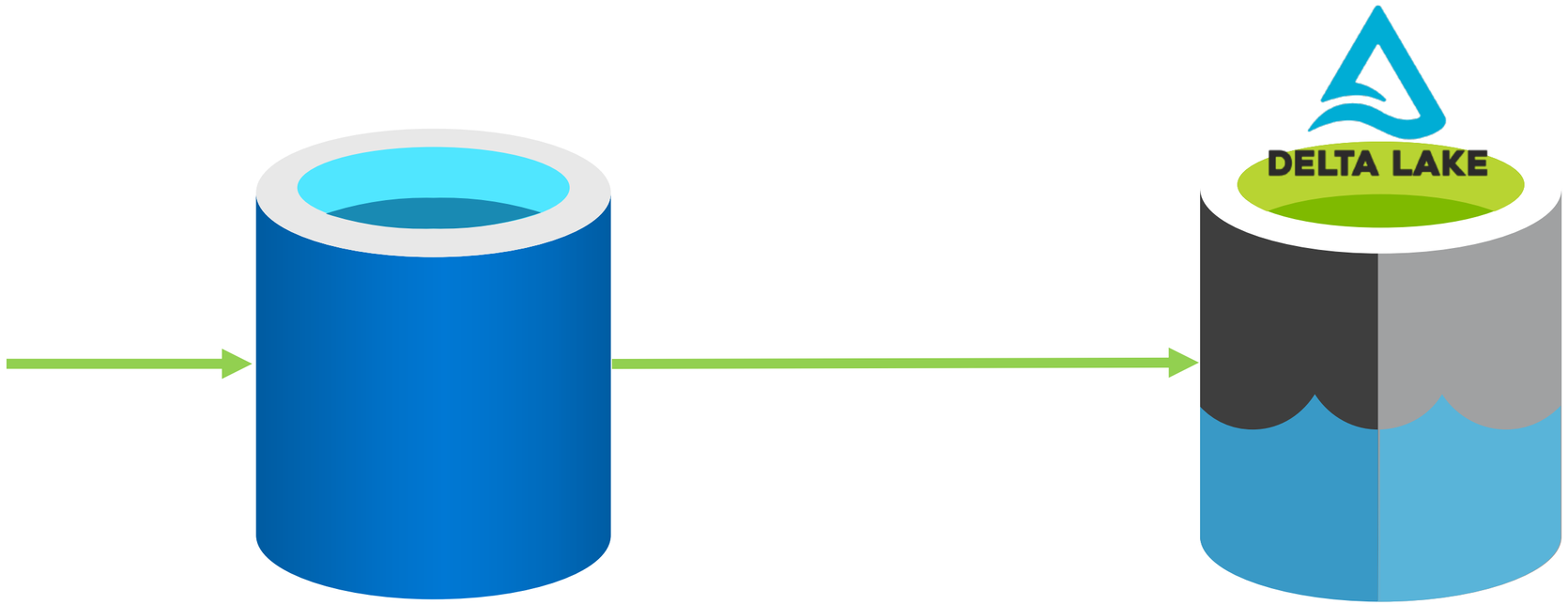
Lake
House



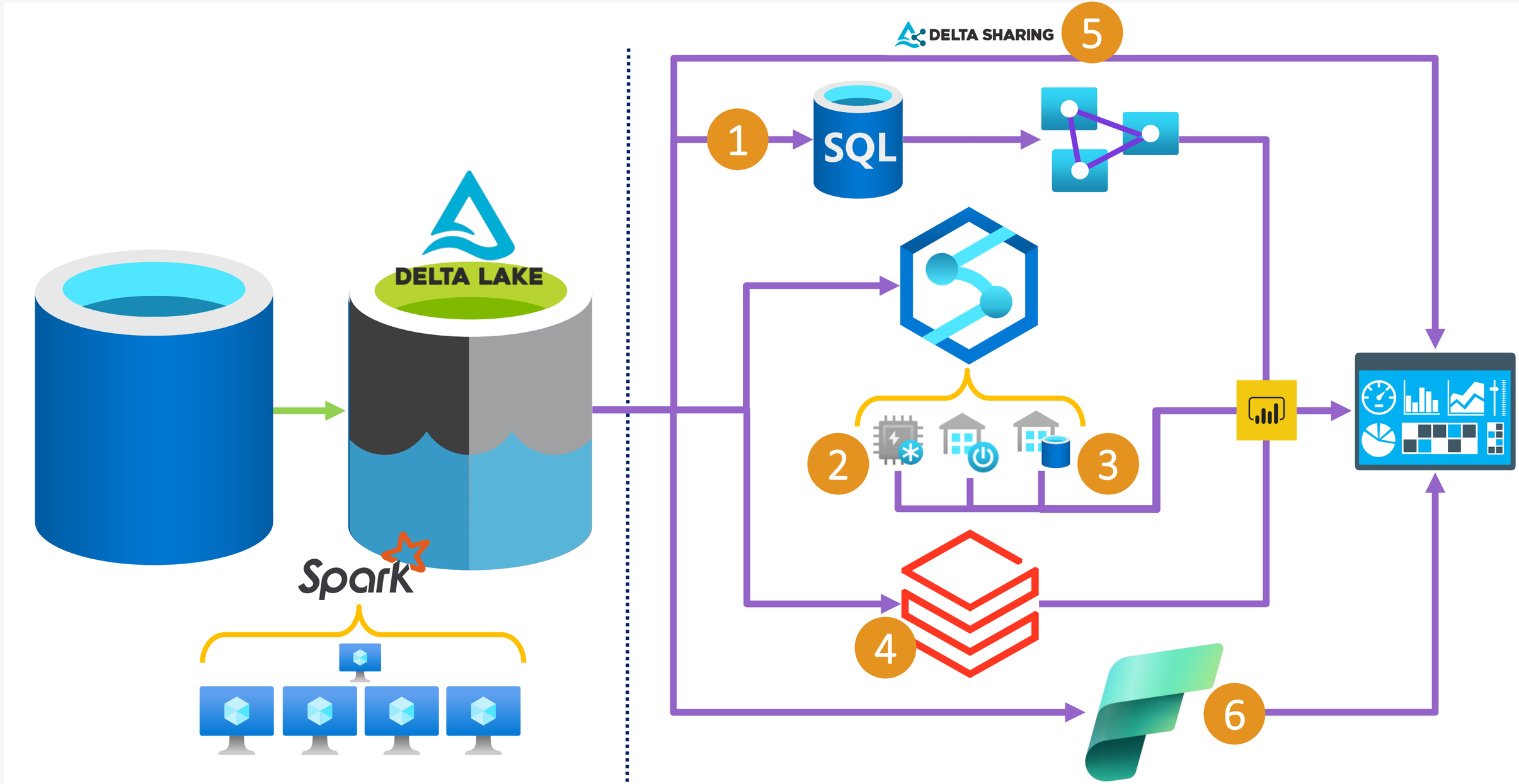
Offline
Analytical
Transactional
Processing

https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html

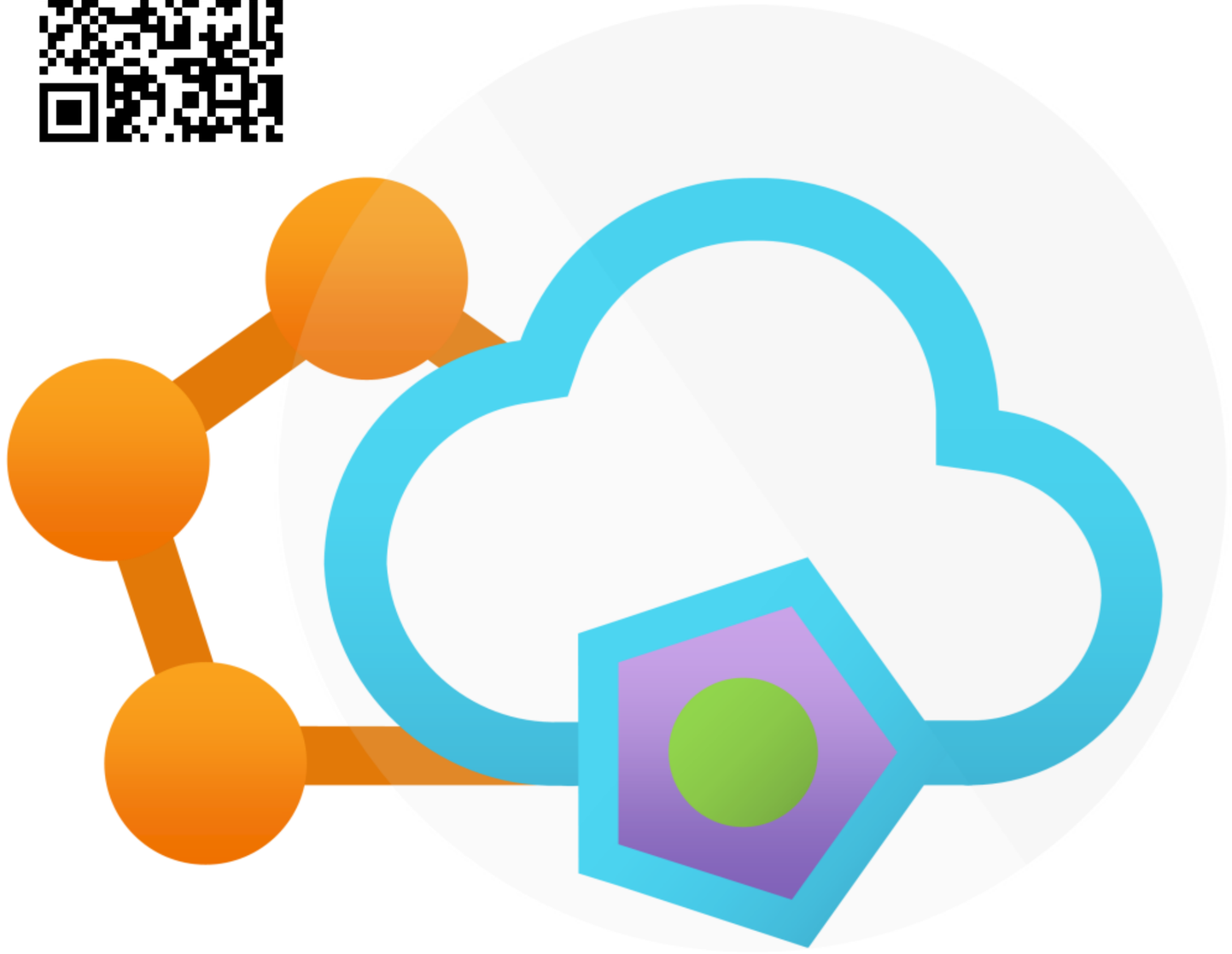
Consuming Our Lake House



Consuming Our Lake House in Azure



<< Session Feedback



Thank You

mrpaulandrew.com
paul@mrpaulandrew.com
In/mrpaulandrew
@mrpaulandrew

https://cloudformations.org
contactus@cloudformations.org
In/CloudFormations
@CloudFormsLtd
CloudFormationsLtd

Cloud Formations