CSCI E-103: Data Engineering for Analytics
Lecture 04: Data Transformations, Design Patterns, and Compliance

Harvard Extension School

Fall 2024

- **Course:** CSCI E-103: Data Engineering for Analytics
- **Lecture:** Lecture 04: Transformations & Patterns
- **Instructor:** Anindita Mahapatra & Eric Gieseke
- **Objective:** Master data engineering design patterns, understand compliance (GDPR/CCPA), Spark internals (partitions, joins), and CDC/SCD concepts

Key Summary

This lecture covers essential data transformation concepts. We explore **design patterns** for big data (Lambda, Kappa, CQRS, Data Mesh), understand **compliance requirements** (GDPR, CCPA) and their engineering implications, dive into **Spark internals** (jobs/stages/tasks, table vs Spark partitions, Z-ordering, join strategies), and learn about **CDC (Change Data Capture)** and **SCD (Slowly Changing Dimensions)** for tracking data changes over time.

Contents

1 What Are Design Patterns?

Definition:

Design Pattern A design pattern is a **reusable, proven solution template** for common problems in software or data engineering. Patterns provide tested approaches that you can adapt to your specific context.

1.1 Why Use Design Patterns?

1. **Embody Good Design Principles**: Patterns naturally incorporate principles like abstraction, separation of concerns, and divide-and-conquer
2. **Common Vocabulary**: Instead of explaining complex designs every time, say "let's use Lambda architecture" and everyone understands
3. **Proven Solutions**: These patterns have been battle-tested in production systems
4. **Faster Development**: Don't reinvent the wheel—use established templates

Warning

Apply Judiciously

"When you have a hammer, everything looks like a nail." Don't force-fit patterns. Always evaluate whether a pattern actually fits your problem context (data volume, latency requirements, cost constraints).

2 Big Data Design Patterns

Big data patterns differ from traditional software patterns (Gang of Four). They focus on scale, velocity, volume, and distributed processing challenges.

2.1 Patterns by Pipeline Stage

Table 1: *Design Patterns by Pipeline Stage*

Stage	Pattern	Purpose
Modeling	Star/Snowflake Schema	Analytical data warehouse structure
	Vault Modeling	Preserve all change history
Ingestion	Connector Pattern	Uniform interface to diverse sources
	Lambda/Kappa	Handle batch + streaming
	Compute/Storage Separation	Scale independently
Transform	Schema on Read/Write	When to define schema
	ACID Transactions	Data integrity at scale
	Multi-Hop Pipeline	Progressive data refinement
Storage	Columnar Storage	Fast analytical queries
	Denormalized Tables	Avoid expensive joins
Analytics	In-Stream Analytics	Real-time processing before storage

2.2 Architecture Patterns

2.2.1 Lambda Architecture

Splits data flow into two paths:

- **Batch Layer**: Slow but accurate—processes all historical data
- **Speed Layer**: Fast but approximate—processes real-time data
- **Serving Layer**: Merges results from both layers

Use case: When you need both real-time insights AND historically accurate analysis.

2.2.2 Kappa Architecture

Single streaming pipeline handles everything:

- All data treated as a stream
- For ”batch” needs, replay the stream from beginning
- Simpler than Lambda (one codebase)

Use case: Real-time processing, event-driven systems, IoT.

2.2.3 Event-Driven Architecture (EDA)

System reacts to events (state changes):

- **Event Producers**: Generate events
- **Event Streaming**: Kafka, Event Hubs
- **Event Consumers**: React to events

Use case: Microservices, e-commerce transaction processing.

2.3 Storage Patterns

2.3.1 CQRS (Command Query Responsibility Segregation)

Definition:

CQRS Separate the system's **write operations** (commands) from **read operations** (queries) into different models that can be scaled independently.

Why? Write-heavy workloads and read-heavy workloads have different requirements. Separating them allows optimal scaling for each.

2.3.2 Polyglot Persistence

Definition:

Polyglot Persistence "Use the right database for the job." Instead of forcing all data into one database type, use multiple specialized databases:

- Relational DB for transactional data
- Document store for unstructured data
- Graph DB for relationship-centric data

2.3.3 Data Lake Pattern

Store all data (structured, semi-structured, unstructured) in its **raw form**:

- "Store now, figure out how to use later"
- Preserves original data for future unknown use cases
- Uses Schema-on-Read (define schema when reading)

2.4 Processing Patterns

2.4.1 Micro-batch Processing

Process streaming data in small, regular intervals:

- Not true streaming, but "near real-time"
- Batch interval defines processing frequency

- Spark Structured Streaming default mode

2.4.2 Multi-Hop (Medallion) Architecture

Progressive data refinement through stages:

- **Bronze (Landing)**: Raw data, minimal transformation
- **Silver (Refined)**: Cleaned, validated, business logic applied
- **Gold (Aggregated)**: Analytics-ready, pre-computed aggregations

Key insight: As you move toward Gold, data quality increases but data availability decreases (Bronze is available immediately).

2.4.3 Minimize Data Movement

Moving petabytes is expensive. Use:

- **Time Travel**: Version data—access historical states without copying
- **Zero-Copy Clone**: Clone tables by copying only metadata, not data
- **Delta Share**: Share data access without physical transfer

2.5 Other Notable Patterns

- **CAP Theorem Selection**: Choose CP (consistency) or AP (availability) based on needs
- **Data Mesh**: Decentralized data ownership—each domain team owns their data as a ”product”
- **Connector/Bridge Pattern**: Uniform API across diverse data sources (e.g., JDBC)

3 Data Compliance: GDPR and CCPA

Data engineers have legal and ethical responsibilities when handling personal data.

3.1 Key Regulations

Definition:

GDPR (General Data Protection Regulation) EU regulation protecting personal data of EU citizens. Applies to ANY company processing EU citizen data, regardless of location. Heavy fines for violations (up to 4% of global revenue).

Definition:

CCPA (California Consumer Privacy Act) California law providing similar protections for California residents.

3.2 Engineering Implications

Two key rights create engineering challenges:

3.2.1 1. Right of Erasure (Right to be Forgotten)

Users can request deletion of their personal data.

Warning

The Challenge
In a data lake with petabytes of data across millions of files, how do you find and delete one user's records? Traditional formats like Parquet are immutable—you can't just delete a row.
Solution: Delta Lake, Hudi, Iceberg support fine-grained deletes.

3.2.2 2. Right of Portability

Users can request their data be exported and transferred to another service.

3.3 Technical Solutions

- **Fine-grained Updates/Deletes**: Use Delta Lake for row-level operations
- **Pseudonymization**: Replace identifiers with tokens
  - Store [User ID ↔ Token] mapping separately
  - All analytics use tokens only
  - For erasure: just delete the mapping—data becomes unlinkable

4 Spark Execution: Jobs, Stages, Tasks

Understanding how Spark executes your code helps you write more efficient pipelines.

4.1 Execution Hierarchy

Key Summary

Job → Stages → Tasks

1. **Job**: Created for each **action** (e.g., `save()`, `collect()`)
2. **Stage**: Jobs split at **shuffle** boundaries (data redistribution points)
3. **Task**: Smallest unit—one task per partition, runs on executor cores in parallel

4.2 Shuffle Operations

Definition:

Shuffle The process of redistributing data across cluster nodes. Required for operations like `groupBy()`, `join()`, `reduceByKey()` where data from different partitions must be combined.

Why expensive?

- Data transferred over network between executors
- Disk I/O for intermediate data
- Synchronization overhead

Minimize shuffles when possible!

5 Table Partitions vs Spark Partitions

These are two completely different concepts that beginners often confuse.

Important:

Library Analogy

- **Table Partitions** = Physical bookshelves organized by category (reduces disk I/O)
- **Spark Partitions** = Number of librarians working in parallel (increases CPU parallelism)

5.1 Comparison Table

Table 2: *Table Partitions vs Spark Partitions*

Aspect	Table Partitioning	Spark Partitioning
Level	Database/Table	Processing/Runtime
What it is	Physical organization on disk by column values	Logical distribution across cluster nodes
Purpose	Minimize data scans (disk I/O)	Maximize parallelism (CPU)
Controlled by	User (explicit DDL)	Spark (dynamic, based on data/config)
Optimization	Partition Pruning	Shuffle optimization

5.2 Table Partitioning Best Practices

- Choose columns frequently used in `WHERE` clauses (date, country, region)
- Avoid high-cardinality columns (user_id)—creates millions of tiny partitions
- Target at least 1GB per partition
- **Modern advice**: For data < 1TB, don't partition—let Delta Lake optimize

```sql
CREATE TABLE sales (
    id INT,
    amount DECIMAL,
    sale_date DATE,
    country STRING
)
PARTITIONED BY (country, sale_date);
```

Listing 1: Creating a Partitioned Table

5.3 Z-Ordering

Definition:

Z-Ordering A Delta Lake optimization that co-locates related data on disk based on multiple columns. Improves query performance through data skipping.

When to use: Columns frequently filtered together but not suitable as partition keys.

```
-- Compact files and organize by ip_address, port
OPTIMIZE network_logs
ZORDER BY (ip_address, port);
```

Listing 2: OPTIMIZE with Z-Ordering

6 Spark Join Strategies

Spark automatically selects the most efficient join strategy based on data characteristics.

6.1 Broadcast Hash Join

- **Condition**: One table is small (fits in memory)
- **Mechanism**: Broadcast small table to all executors
- **Pros**: No shuffle, no sort—very fast
- **Cons**: Only works with small tables

```
from pyspark.sql.functions import broadcast

# Explicitly broadcast the small table
df_result = df_large.join(broadcast(df_small), "key")
```

Listing 3: Forcing Broadcast Join

6.2 Shuffle Hash Join

- **Condition**: One side is 3x+ smaller, partition fits in memory
- **Mechanism**: Shuffle data, build hash table on smaller side
- **Pros**: Handles larger tables than broadcast
- **Cons**: Requires shuffle (no sort)

6.3 Sort Merge Join

- **Condition**: Default for large tables
- **Mechanism**: Shuffle both sides, sort, then merge
- **Pros**: Handles any data size, very robust
- **Cons**: Requires shuffle AND sort—slowest

Key Summary

Join Strategy Selection

1. **Small table?** → Broadcast Hash Join (fastest)
2. **Medium asymmetry?** → Shuffle Hash Join
3. **Both large?** → Sort Merge Join (default)

7 CDC and SCD: Tracking Data Changes

7.1 CDC: Change Data Capture

Definition:

CDC (Change Data Capture) A technique for identifying and capturing changes (inserts, updates, deletes) in source data so they can be replicated to target systems incrementally.

Why CDC?

- Full table reloads are expensive and slow
- Only process what changed since last sync
- Near real-time data synchronization

Common CDC approaches:

- **Log-based**: Read database transaction logs (most accurate)
- **Timestamp-based**: Query records modified after last sync
- **Trigger-based**: Database triggers capture changes

7.2 SCD: Slowly Changing Dimensions

Definition:

SCD (Slowly Changing Dimension) A dimension table attribute that changes over time (e.g., customer address). SCD defines how to handle historical values when changes occur.

7.2.1 SCD Type 1: Overwrite

- Simply update the record—no history kept
- **Use case**: Corrections, when history doesn't matter

```sql
UPDATE customers
SET address = 'New Address'
WHERE customer_id = 123;
```

Listing 4: SCD Type 1: Simple Update

7.2.2 SCD Type 2: Add New Row

- Keep all historical versions as separate rows
- Add columns: `effective_date`, `end_date`, `is_current`
- **Use case**: Full audit trail, historical analysis

```
-- Old record
| customer_id | address     | effective | end_date   | current |
|-------------|-------------|-----------|------------|---------|
| 123         | Old Address | 2020-01-01| 2024-01-15 | false   |
```

```
| 123          | New Address | 2024-01-15| 9999-12-31 | true     |
```

Listing 5: SCD Type 2: Historical Record

7.3 MERGE Statement for CDC/SCD

Delta Lake's MERGE handles upserts (update or insert) efficiently:

```sql
MERGE INTO target_table t
USING source_table s
ON t.id = s.id
WHEN MATCHED THEN
    UPDATE SET t.value = s.value
WHEN NOT MATCHED THEN
    INSERT (id, value) VALUES (s.id, s.value)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

Listing 6: MERGE for Upsert Operations

8 Delta Lake Operations

8.1 OPTIMIZE: File Compaction

```sql
-- Compact small files into larger ones
OPTIMIZE my_table;

-- Compact and Z-order
OPTIMIZE my_table ZORDER BY (column1, column2);
```

Listing 7: OPTIMIZE Command

Why? Small files hurt query performance due to file listing overhead.

8.2 VACUUM: Clean Old Versions

```sql
-- Remove files older than retention period (default 7 days)
VACUUM my_table;

-- Custom retention (requires safety flag)
VACUUM my_table RETAIN 168 HOURS;  -- 7 days
```

Listing 8: VACUUM Command

Warning: After VACUUM, time travel to cleaned versions is impossible.

8.3 Time Travel

```sql
-- Query specific version
SELECT * FROM my_table VERSION AS OF 10;

-- Query by timestamp
SELECT * FROM my_table TIMESTAMP AS OF '2024-01-01';

-- Restore to previous version
RESTORE TABLE my_table TO VERSION AS OF 5;
```

Listing 9: Time Travel Queries

8.4 Clone: Zero-Copy Duplication

```sql
-- Shallow clone (metadata only)
CREATE TABLE my_table_clone SHALLOW CLONE my_table;

-- Deep clone (full copy)
CREATE TABLE my_table_copy DEEP CLONE my_table;
```

Listing 10: Clone Operations

9 Summary and Key Takeaways

Key Summary

Key Takeaways

1. **Design Patterns** provide proven solutions for common data engineering problems. Use them judiciously—match pattern to problem.
2. **Key Architecture Patterns**:
   - Lambda: Separate batch + streaming layers
   - Kappa: Unified streaming layer
   - CQRS: Separate read and write models
   - Data Mesh: Decentralized data ownership
3. **Compliance (GDPR/CCPA)**: Right to erasure and portability require fine-grained data operations. Use Delta Lake + pseudonymization.
4. **Spark Execution**: Job → Stages (shuffle boundary) → Tasks (partition)
5. **Two Types of Partitions**:
   - Table Partitions: Physical organization, minimize disk I/O
   - Spark Partitions: Logical distribution, maximize parallelism
6. **Z-Ordering**: Co-locate related data for multi-column filtering
7. **Join Strategies**: Broadcast (small table) > Shuffle Hash > Sort Merge (default)
8. **CDC**: Capture incremental changes from source systems
9. **SCD**: Handle dimension changes—Type 1 (overwrite) vs Type 2 (history)
10. **Delta Lake Operations**: OPTIMIZE (compact), VACUUM (clean), Time Travel, Clone

Warning

Practical Advice

☐ For data < 1TB, skip table partitioning—let Delta optimize

☐ Minimize shuffles in Spark—they're expensive

☐ Use MERGE for CDC/SCD operations

☐ Set up regular OPTIMIZE jobs for frequently updated tables

☐ Always consider compliance early—retrofitting is painful