Chapter 17: Data Processing at Scale

Introduction: Big Data Challenges

When you have massive amounts of data, traditional processing doesn't work.

```
Small Data (Traditional):
┌─────────────────────────┐
│ 1 GB of data            │
│ Process on single machine │
│ Time: 10 minutes        │
│ ✓ Simple                │
└─────────────────────────┘

Big Data (Scale):
┌─────────────────────────┐
│ 1 PB (1,000,000 GB)     │
│ Single machine:         │
│ • Would take 10M minutes │
│ • = 19 years!           │
│ • Won't fit in memory   │
│ ✗ Impossible            │
└─────────────────────────┘

Solution: Distributed Processing
┌─────────────────────────┐
│ 1 PB of data            │
│ 1,000 machines in parallel│
│ Time: 10 minutes        │
│ ✓ Practical             │
└─────────────────────────┘
```

---

1. Batch Processing vs Stream Processing

Batch Processing

Concept: Process large volumes of data at once, periodically.

```
Batch Processing Timeline:

00:00 ───────────────────── 24:00 ───────────────────── 48:00
  │                │                │
  │ Data accumulates  │ Data accumulates  │
  │ during the day   │ during the day   │
  │                │                │
  ↓                ↓                ↓
Process         Process         Process
(at night)      (at night)      (at night)

Characteristics:
- Processes data in large chunks
- Run periodically (hourly, daily, weekly)
- High latency (hours to days)
- High throughput (TB to PB per job)
- Complex analytics possible
```

Example: E-commerce Daily Sales Report

```
Daily Batch Job (runs at midnight):

Input:
- 10 million orders from past 24 hours
- 50 GB of transaction data

Process:
1. Load all orders (6 PM - 12 AM)
2. Calculate metrics:
   • Total revenue
   • Revenue by category
   • Top products
   • Customer segments
   • Inventory needs
3. Generate reports

Output:
- Sales dashboard
- Email to management
- Inventory recommendations

Duration: 2 hours
Freshness: Up to 26 hours old (24h + 2h processing)
```

Batch Processing Code:

```python
from pyspark.sql import SparkSession

# Initialize Spark
spark = SparkSession.builder \
    .appName("DailySalesReport") \
    .getOrCreate()

# Read data (24 hours of orders)
orders = spark.read.parquet("s3://data/orders/2024-01-20/*.parquet")

# Batch processing
daily_report = orders \
    .filter(orders.status == 'completed') \
    .groupBy('category') \
    .agg(
        sum('total').alias('revenue'),
        count('order_id').alias('order_count'),
        avg('total').alias('avg_order_value')
    ) \
    .orderBy('revenue', ascending=False)

# Save results
daily_report.write.parquet("s3://reports/daily-sales/2024-01-20/")

# Show top categories
daily_report.show(10)

# Output:
# +-------------+----------+-----------+---------------+
# |  category   | revenue  |order_count | avg_order_value|
# +-------------+----------+-----------+---------------+
# | Electronics | 1250000 |    5234    |     238.79     |
# | Clothing    | 890000  |   12456    |      71.45     |
# | Books       | 456000  |   23123    |      19.72     |
# +-------------+----------+-----------+---------------+
```

Stream Processing

Concept: Process data continuously as it arrives.

```
Stream Processing Timeline:

Event 1 → Process → Result (0.1s)
Event 2 → Process → Result (0.1s)
Event 3 → Process → Result (0.1s)
Event 4 → Process → Result (0.1s)
...continuous...

Characteristics:
- Processes data in real-time
- Event-by-event or micro-batches
- Low latency (milliseconds to seconds)
- Lower throughput per instance
- Simpler per-event operations
```

Example: Real-time Fraud Detection

```
Stream Processing (processes every transaction):

Event arrives → Process → Alert (within 100ms)

Transaction 1: $50 at Starbucks NYC
 ↓ Process in 50ms
 ✓ Normal (no alert)

Transaction 2: $5000 at Electronics Store LA
 ↓ Process in 80ms
 ⚠️  Unusual amount + unusual location
 → Alert sent immediately!

Transaction 3: $10 at Gas Station LA
 ↓ Process in 45ms
 ⚠️  After flagged transaction, same location
 → Confirm fraud pattern
 → Block card immediately!

Latency: 50-100ms per transaction
Freshness: Real-time (immediate)
```

Stream Processing Code:

```javascript
```

```
const { Kafka } = require('kafkajs');

const kafka = new Kafka({ brokers: ['kafka:9092'] });
const consumer = kafka.consumer({ groupId: 'fraud-detector' });

// Real-time fraud detection
async function streamProcessing() {
  await consumer.connect();
  await consumer.subscribe({ topic: 'transactions' });

  const userHistory = new Map();

  await consumer.run({
    eachMessage: async ({ message }) => {
      const transaction = JSON.parse(message.value);
      const startTime = Date.now();

      // Get user's recent transactions
      if (!userHistory.has(transaction.userId)) {
        userHistory.set(transaction.userId, []);
      }

      const history = userHistory.get(transaction.userId);

      // Detect fraud patterns (real-time!)
      const fraudScore = calculateFraudScore(transaction, history);

      if (fraudScore > 0.8) {
        // High fraud risk - alert immediately!
        await alertFraud({
          transactionId: transaction.id,
          userId: transaction.userId,
          amount: transaction.amount,
          fraudScore,
          processingTime: Date.now() - startTime
        });

        // Block card
        await blockCard(transaction.userId);

        console.log(`🚨 FRAUD DETECTED (${Date.now() - startTime}ms)`);
      }

      // Update history (keep last 20)
      history.push(transaction);
      if (history.length > 20) {
```

```
      history.shift();
    }

    console.log(`Processed transaction in ${Date.now() - startTime}ms`);
  }
 });
}

function calculateFraudScore(transaction, history) {
 let score = 0;

 // Pattern 1: Large amount
 const avgAmount = history.reduce((sum, t) => sum + t.amount, 0) / history.length;
 if (transaction.amount > avgAmount * 5) {
  score += 0.4;
 }

 // Pattern 2: Geographic anomaly
 if (history.length > 0) {
  const lastLoc = history[history.length - 1].location;
  const distance = calculateDistance(lastLoc, transaction.location);
  const timeDiff = transaction.timestamp - history[history.length - 1].timestamp;

  if (distance > 1000 && timeDiff < 3600000) { // 1000km in 1 hour
   score += 0.5;
  }
 }

 // Pattern 3: Multiple transactions quickly
 const recent = history.filter(t =>
  transaction.timestamp - t.timestamp < 300000 // Last 5 minutes
 );
 if (recent.length >= 5) {
  score += 0.3;
 }

 return Math.min(score, 1.0);
}

streamProcessing();
```

Batch vs Stream Comparison

```
┌──────────────────────────┬────────────────────────┬──────────────────────────┐
│ Feature       │ Batch       │ Stream      │
```

```
├────────────────────┼──────────────────┼────────────────────┤
│ Data Volume      │ Large (TB-PB)  │ Continuous      │
│ Latency        │ Hours/Days     │ Milliseconds/Sec │
│ Processing      │ Periodic       │ Continuous      │
│ Complexity      │ Complex queries │ Simple per-event │
│ Throughput      │ Very high      │ Medium         │
│ Use Case       │ Reports,ETL     │ Monitoring,fraud │
├────────────────────┼──────────────────┼────────────────────┤
│ Examples       │              │              │
│             │ Daily sales     │ Click tracking   │
│             │ ML training     │ Real-time alerts │
│             │ Data warehouse  │ Live dashboards  │
│             │ Backup/Archive  │ Fraud detection  │
└────────────────────┴──────────────────┴────────────────────┘

Latency vs Throughput Trade-off:
Batch: Low latency ✗, High throughput ✓
Stream: Low latency ✓, Lower throughput ✗
```

---

2. MapReduce Paradigm

What is MapReduce?

Concept: Process massive datasets by splitting into map and reduce phases.

```
MapReduce Workflow:

Input Data (Large)
├── Split 1 (1 GB)
├── Split 2 (1 GB)
├── Split 3 (1 GB)
└── Split 4 (1 GB)
    ↓
MAP Phase (Parallel)
├── Mapper 1: Process Split 1 → [key-value pairs]
├── Mapper 2: Process Split 2 → [key-value pairs]
├── Mapper 3: Process Split 3 → [key-value pairs]
└── Mapper 4: Process Split 4 → [key-value pairs]
    ↓
SHUFFLE & SORT (Group by key)
Group all values for same key together
    ↓
REDUCE Phase (Parallel)
├── Reducer 1: Process key group 1 → Result
├── Reducer 2: Process key group 2 → Result
└── Reducer 3: Process key group 3 → Result
    ↓
Output (Aggregated Results)
```

MapReduce Example: Word Count

Problem: Count word frequency in 1 TB of text files.

```
Input Files (4 files):
File 1: "hello world"
File 2: "hello hadoop"
File 3: "world of hadoop"
File 4: "hello world hadoop"

MAP Phase (4 mappers in parallel):
────────────────────────────────────
Mapper 1 (File 1):
  Input: "hello world"
  Output: [("hello", 1), ("world", 1)]

Mapper 2 (File 2):
  Input: "hello hadoop"
  Output: [("hello", 1), ("hadoop", 1)]

Mapper 3 (File 3):
  Input: "world of hadoop"
  Output: [("world", 1), ("of", 1), ("hadoop", 1)]

Mapper 4 (File 4):
  Input: "hello world hadoop"
  Output: [("hello", 1), ("world", 1), ("hadoop", 1)]

SHUFFLE & SORT:
────────────────────────────────────
Group by key:
  "hello": [1, 1, 1]      (from mappers 1, 2, 4)
  "world": [1, 1, 1]      (from mappers 1, 3, 4)
  "hadoop": [1, 1, 1]     (from mappers 2, 3, 4)
  "of": [1]               (from mapper 3)

REDUCE Phase (parallel):
────────────────────────────────────
Reducer 1:
  Input: ("hello", [1, 1, 1])
  Output: ("hello", 3)

Reducer 2:
  Input: ("world", [1, 1, 1])
  Output: ("world", 3)

Reducer 3:
  Input: ("hadoop", [1, 1, 1])
  Output: ("hadoop", 3)
```

```
Reducer 4:
  Input: ("of", [1])
  Output: ("of", 1)

Final Output:
---------------------------------
hello: 3
world: 3
hadoop: 3
of: 1
```

MapReduce Code (Hadoop)

```java
```

```java
// Mapper
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {

        // Input: Line of text
        String line = value.toString();

        // Split into words
        String[] words = line.split("\\s+");

        // Emit (word, 1) for each word
        for (String w : words) {
            word.set(w.toLowerCase());
            context.write(word, one);
        }
    }
}

// Reducer
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {

        // Input: (word, [1, 1, 1, ...])
        int sum = 0;

        // Sum all occurrences
        for (IntWritable val : values) {
            sum += val.get();
        }

        // Emit (word, count)
        context.write(key, new IntWritable(sum));
    }
}

// Driver (Main)
public class WordCount {
```

```java
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");

        job.setJarByClass(WordCount.class);
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

// Run:
// hadoop jar wordcount.jar WordCount /input /output
```

MapReduce: More Complex Example

Problem: Calculate average order value per customer

```python
```

```python
from mrjob.job import MRJob
from mrjob.step import MRStep
import json

class CustomerAvgOrder(MRJob):

    def mapper(self, _, line):
        """
        Input: JSON line with order data
        Output: (customer_id, order_total)
        """
        order = json.loads(line)
        yield order['customer_id'], order['total']

    def reducer(self, customer_id, order_totals):
        """
        Input: (customer_id, [total1, total2, total3, ...])
        Output: (customer_id, average_order_value)
        """
        totals = list(order_totals)
        avg = sum(totals) / len(totals)

        yield customer_id, {
            'avg_order_value': avg,
            'total_orders': len(totals),
            'total_spent': sum(totals)
        }

if __name__ == '__main__':
    CustomerAvgOrder.run()

# Input data:
# {"customer_id": "C1", "total": 100}
# {"customer_id": "C1", "total": 200}
# {"customer_id": "C2", "total": 50}
# {"customer_id": "C1", "total": 150}

# Map output:
# ("C1", 100), ("C1", 200), ("C2", 50), ("C1", 150)

# Shuffle & sort:
# ("C1", [100, 200, 150])
# ("C2", [50])

# Reduce output:
```

```
# C1: {"avg_order_value": 150, "total_orders": 3, "total_spent": 450}
# C2: {"avg_order_value": 50, "total_orders": 1, "total_spent": 50}
```

---

3. Apache Hadoop Ecosystem

Hadoop Architecture

```
┌─────────────────────────────────────────────────────┐
│          Hadoop Ecosystem                          │
├─────────────────────────────────────────────────────┤
│                                                     │
│  ┌──────────────────────────────────────────────┐  │
│  │       YARN (Resource Manager)                 │  │
│  │ Manages cluster resources                     │  │
│  └──────────────┬───────────────────────────────┘  │
│                 │                                   │
│  ┌──────────────┴───────────────────────────────┐  │
│  │     HDFS (Storage Layer)                      │  │
│  │ Distributed file system                       │  │
│  └──────────────────────────────────────────────┘  │
│                                                     │
│ Processing Frameworks (on top):                     │
│  ┌──────────┐  ┌──────────┐  ┌──────────┐          │
│  │MapReduce │  │  Spark   │  │  Flink   │          │
│  └──────────┘  └──────────┘  └──────────┘          │
│                                                     │
│ Data Access Tools:                                  │
│  ┌──────────┐  ┌──────────┐  ┌──────────┐          │
│  │  Hive    │  │   Pig    │  │  HBase   │          │
│  │  (SQL)   │  │ (Script) │  │(NoSQL DB)│          │
│  └──────────┘  └──────────┘  └──────────┘          │
└─────────────────────────────────────────────────────┘
```

HDFS (Hadoop Distributed File System)

Already covered in Chapter 8, quick recap:

```
HDFS Architecture:

NameNode (Master):
├─ Manages metadata
├─ Knows which blocks are where
└─ Single point (but has backup)

DataNodes (Workers):
├─ Store actual data blocks
├─ 128 MB blocks
├─ 3 replicas per block
└─ Report health to NameNode

Example: Store 1 GB file
1 GB / 128 MB = 8 blocks
8 blocks × 3 replicas = 24 total blocks
Distributed across DataNodes
```

Hive (SQL on Hadoop)

Concept: Write SQL, executes as MapReduce jobs.

```
sql
```

```sql
-- Create table
CREATE TABLE orders (
    order_id INT,
    customer_id INT,
    total DECIMAL(10,2),
    order_date DATE
)
STORED AS PARQUET
LOCATION '/data/orders';

-- Query (compiles to MapReduce)
SELECT
    customer_id,
    COUNT(*) as order_count,
    SUM(total) as total_spent,
    AVG(total) as avg_order
FROM orders
WHERE order_date >= '2024-01-01'
GROUP BY customer_id
HAVING SUM(total) > 1000
ORDER BY total_spent DESC
LIMIT 100;

-- Behind the scenes:
-- 1. Hive compiles SQL to MapReduce jobs
-- 2. Launches map tasks to scan data
-- 3. Shuffle groups by customer_id
-- 4. Reduce tasks aggregate
-- 5. Returns results

-- Can process petabytes!
-- Just write SQL!
```

---

4. Apache Spark

Why Spark? (Faster than MapReduce)

```
MapReduce Problem:
Every operation writes to disk!

Job 1: Read HDFS → Map → Write to HDFS
Job 2: Read HDFS → Reduce → Write to HDFS
Job 3: Read HDFS → Map → Write to HDFS
       ↑ Slow disk I/O between every step!
```

```
Spark Solution:
Keep data in memory!

Job: Read HDFS → Map → Reduce → Map → Output
          ↑ All in memory (RAM) ↑
          100x faster for iterative jobs!

Performance Comparison:
MapReduce: 100 GB in 10 minutes (disk-bound)
Spark: 100 GB in 1 minute (memory-bound)
```

Spark RDD (Resilient Distributed Dataset)

```
RDD: Immutable distributed collection

┌─────────────────────────────────────────┐
│          RDD Partitioning               │
├─────────────────────────────────────────┤
│                                         │
│  Original Data (10 GB)                  │
│  ↓                                      │
│  Split into partitions:                 │
│  ┌──────────┐ ┌──────────┐ ┌──────────┐ │
│  │Partition │ │Partition │ │Partition │ │
│  │    1     │ │    2     │ │    3     │ │
│  │  (2 GB)  │ │  (2 GB)  │ │  (2 GB)  │ │
│  └──────────┘ └──────────┘ └──────────┘ │
│     ↓            ↓            ↓         │
│  ┌──────────┐ ┌──────────┐ ┌──────────┐ │
│  │Executor 1│ │Executor 2│ │Executor 3│ │
│  │(Worker)  │ │(Worker)  │ │(Worker)  │ │
│  └──────────┘ └──────────┘ └──────────┘ │
│                                         │
│  Processed in parallel across cluster   │
└─────────────────────────────────────────┘
```

Spark Code Examples

```python
```

```python
from pyspark.sql import SparkSession

# Initialize Spark
spark = SparkSession.builder \
    .appName("SparkExample") \
    .config("spark.executor.memory", "4g") \
    .getOrCreate()

# Example 1: Word Count (much simpler than Hadoop!)
text_file = spark.read.text("hdfs://data/input.txt")

word_counts = text_file \
    .rdd \
    .flatMap(lambda line: line.value.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)

word_counts.saveAsTextFile("hdfs://data/output")

# Example 2: DataFrame API (SQL-like)
df = spark.read.parquet("s3://data/orders/")

# Filter
high_value = df.filter(df.total > 1000)

# Aggregation
revenue_by_category = df \
    .groupBy("category") \
    .agg(
        sum("total").alias("revenue"),
        count("order_id").alias("order_count"),
        avg("total").alias("avg_order")
    )

revenue_by_category.show()

# Example 3: Join (across multiple datasets)
orders = spark.read.parquet("s3://data/orders/")
customers = spark.read.parquet("s3://data/customers/")

# Join orders with customer info
result = orders.join(
    customers,
    orders.customer_id == customers.id,
    "inner"
)
```

```python
# Group by customer tier
tier_analysis = result \
    .groupBy(customers.tier) \
    .agg(
        sum(orders.total).alias("revenue"),
        count(orders.order_id).alias("orders")
    )

tier_analysis.show()

# Example 4: Window functions (complex analytics)
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number, rank

# Rank products by revenue within each category
window_spec = Window.partitionBy("category").orderBy(desc("revenue"))

ranked = df \
    .withColumn("rank", rank().over(window_spec)) \
    .filter(col("rank") <= 10)  # Top 10 per category

ranked.show()

# Example 5: Machine Learning (MLlib)
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import VectorAssembler

# Prepare features
assembler = VectorAssembler(
    inputCols=["age", "income", "purchases"],
    outputCol="features"
)

data = assembler.transform(df)

# Train model
lr = LogisticRegression(
    featuresCol="features",
    labelCol="will_churn"
)

model = lr.fit(data)

# Predict
predictions = model.transform(test_data)
```

Spark Streaming (Micro-Batch)

```python
```

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import window, col, count, avg

spark = SparkSession.builder \
    .appName("SparkStreaming") \
    .getOrCreate()

# Read from Kafka (streaming source)
stream = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "transactions") \
    .load()

# Parse JSON
from pyspark.sql.functions import from_json, col
from pyspark.sql.types import StructType, StringType, DoubleType

schema = StructType() \
    .add("transaction_id", StringType()) \
    .add("user_id", StringType()) \
    .add("amount", DoubleType()) \
    .add("timestamp", StringType())

transactions = stream \
    .selectExpr("CAST(value AS STRING)") \
    .select(from_json(col("value"), schema).alias("data")) \
    .select("data.*")

# Windowed aggregation (5-minute windows)
windowed_metrics = transactions \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window(col("timestamp"), "5 minutes"),
        col("user_id")
    ) \
    .agg(
        count("transaction_id").alias("tx_count"),
        sum("amount").alias("total_amount"),
        avg("amount").alias("avg_amount")
    )

# Write to console (for demo)
query = windowed_metrics \
    .writeStream \
```

```
    .outputMode("update") \
    .format("console") \
    .start()

query.awaitTermination()

# Output every 5 minutes:
# +--------------------+-------+--------+-----------+----------+
# |window              |user_id|tx_count|total_amount|avg_amount|
# +--------------------+-------+--------+-----------+----------+
# |{2024-01-20 10:00,...|user123|   45   |   4532.50  |  100.72  |
# |{2024-01-20 10:00,...|user456|   23   |   1234.00  |   53.65  |
# +--------------------+-------+--------+-----------+----------+
```

---

5. Lambda Architecture

Concept: Batch + Stream = Best of Both Worlds

```
┌─────────────────────────────────────────────────────────┐
│           Lambda Architecture                           │
├─────────────────────────────────────────────────────────┤
│                                                         │
│  Data Sources (All Data)                                │
│  ┌──────────────────────────────────────┐               │
│  │ Events, Logs, Transactions           │               │
│  └────────────┬─────────────────────────┘               │
│               │                                         │
│        ┌──────┴───────┐                                 │
│        ↓              ↓                                 │
│  ┌───────────┐  ┌───────────┐                           │
│  │  Batch    │  │  Speed    │                           │
│  │  Layer    │  │  Layer    │                           │
│  │           │  │           │                           │
│  │ Hadoop/   │  │  Storm/   │                           │
│  │  Spark    │  │  Flink    │                           │
│  │           │  │           │                           │
│  │Complete   │  │Real-time  │                           │
│  │Accurate   │  │Approx.    │                           │
│  │ Slow      │  │  Fast     │                           │
│  │(hours)    │  │  (sec)    │                           │
│  └─────┬─────┘  └─────┬─────┘                           │
│        │      │                                         │
│        │  ┌───┴────────────────────┐                    │
│        │  │              │                              │
│        ↓  ↓              ↓                              │
```

```
│  ┌─────────────┐      ┌──────────┐      │
│  │Batch View   │      │Speed View│      │
│  │(Complete)   │      │(Recent)  │      │
│  └──────┬──────┘      └────┬─────┘      │
│         │                  │            │
│         └────────┬─────────┘            │
│                  ↓                      │
│         ┌────────────────┐              │
│         │Serving Layer   │              │
│         │(Merge Views)   │              │
│         └────────────────┘              │
│                  ↓                      │
│            User Queries                 │
└─────────────────────────────────────────┘

Example: Page View Analytics

Batch Layer:
- Processes all historical data
- Runs daily at midnight
- Complete, accurate counts
- Available at 2 AM

Speed Layer:
- Processes events in real-time
- Updates every second
- Approximate counts
- Available immediately

Serving Layer:
- Query at 10 AM:
  • Batch view: Accurate until midnight (10 hours old)
  • Speed view: Last 10 hours (approximate)
  • Combined: Complete picture!
```

Lambda Architecture Implementation

python

```python
# BATCH LAYER (Spark)
from pyspark.sql import SparkSession

def batch_processing():
    spark = SparkSession.builder.appName("BatchLayer").getOrCreate()

    # Process all historical data
    all_page_views = spark.read.parquet("s3://data/page-views/")

    # Calculate complete statistics
    page_stats = all_page_views \
        .groupBy("page_id", "date") \
        .agg(
            count("user_id").alias("total_views"),
            countDistinct("user_id").alias("unique_visitors")
        )

    # Save to serving layer
    page_stats.write \
        .mode("overwrite") \
        .parquet("s3://serving/batch-views/")

    print("Batch layer updated (complete and accurate)")

# SPEED LAYER (Kafka Streams)
from kafka import KafkaConsumer
import json

def speed_layer():
    consumer = KafkaConsumer(
        'page-views',
        bootstrap_servers=['kafka:9092']
    )

    # Real-time aggregation
    recent_views = {}  # page_id -> count

    for message in consumer:
        event = json.loads(message.value)
        page_id = event['page_id']

        # Update count
        recent_views[page_id] = recent_views.get(page_id, 0) + 1

        # Store in fast database (Redis)
        redis_client.hincrby('page_views_realtime', page_id, 1)
```

```python
        # Publish to serving layer
        if recent_views[page_id] % 100 == 0:
            print(f"Speed layer: Page {page_id} has {recent_views[page_id]} recent views")

# SERVING LAYER (API)
from flask import Flask, jsonify
import redis

app = Flask(__name__)
redis_client = redis.Redis(host='localhost', port=6379)
spark = SparkSession.builder.appName("ServingLayer").getOrCreate()

@app.route('/api/page-stats/<page_id>')
def get_page_stats(page_id):
    # Get batch view (complete historical data)
    batch_df = spark.read.parquet("s3://serving/batch-views/")
    batch_stats = batch_df \
        .filter(batch_df.page_id == page_id) \
        .agg(sum("total_views").alias("historical_views")) \
        .collect()

    historical_views = batch_stats[0]['historical_views'] if batch_stats else 0

    # Get speed view (recent data)
    recent_views = int(redis_client.hget('page_views_realtime', page_id) or 0)

    # Combine both
    total_views = historical_views + recent_views

    return jsonify({
        'page_id': page_id,
        'total_views': total_views,
        'batch_views': historical_views,  # Complete, accurate
        'speed_views': recent_views,       # Recent, approximate
        'batch_updated': '2024-01-20 02:00:00',
        'speed_updated': 'real-time'
    })

app.run()

# Example response:
# {
#   "page_id": "page-123",
#   "total_views": 15234,
#   "batch_views": 15000,  (until midnight)
#   "speed_views": 234,     (since midnight)
```

```
#   "batch_updated": "2024-01-20 02:00:00",
#   "speed_updated": "real-time"
# }
```

Lambda Architecture Pros and Cons

```
✓ ADVANTAGES:
  - Complete and accurate (batch layer)
  - Real-time updates (speed layer)
  - Fault tolerant (can recompute from batch)
  - Handle both historical and recent data

✗ DISADVANTAGES:
  - Complex (two processing pipelines!)
  - Code duplication (same logic in batch and speed)
  - Operational overhead (maintain two systems)
  - Eventual consistency (batch lags behind speed)

When to use:
✓ Need both accuracy and real-time
✓ Historical analytics + live dashboards
✓ Can handle complexity
✓ Have resources for two systems

Examples:
- LinkedIn (uses Lambda for analytics)
- Twitter (for trending topics, analytics)
```

---

6. Kappa Architecture

Simplified Alternative to Lambda

Concept: Everything is a stream! No batch layer.

```
┌─────────────────────────────────────────────────────────────┐
│                    Kappa Architecture                        │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  Data Sources                                               │
│  ┌─────────────────────────────────────────┐                │
│  │ Events, Logs, Transactions              │                │
│  └──────────────────┬──────────────────────┘                │
│                     │                                       │
│                     ↓                                       │
│      ┌─────────────────┐                                    │
│      │ Event Stream    │                                    │
│      │ (Kafka)         │                                    │
│      │                 │                                    │
│      │ Retained        │                                    │
│      │ (30 days)       │                                    │
│      └────────┬────────┘                                    │
│               │                                             │
│    ┌──────────┴──────────────┬──────────────┐               │
│    ↓          ↓              ↓                              │
│  ┌───────────┐ ┌───────────┐ ┌───────────┐                  │
│  │Stream     │ │Stream     │ │Stream     │                  │
│  │Proc 1     │ │Proc 2     │ │Proc 3     │                  │
│  │           │ │           │ │           │                  │
│  │Real-time  │ │Real-time  │ │Real-time  │                  │
│  └─────┬─────┘ └─────┬─────┘ └─────┬─────┘                  │
│        │             │             │                        │
│    └───────────────┬────────────────┘                       │
│                    ↓                                        │
│        ┌────────────────────┐                               │
│        │ Serving Layer      │                               │
│        └────────────────────┘                               │
│                                                             │
│  To reprocess (e.g., bug fix):                              │
│  → Replay stream from offset 0                              │
│  → Recompute all views                                      │
│  → No separate batch system needed!                         │
└─────────────────────────────────────────────────────────────┘

Key Difference from Lambda:
- Only ONE processing code path (stream)
- Reprocess by replaying stream
- Simpler!
```

Kappa Implementation

javascript

```
const { Kafka } = require('kafkajs');

class KappaProcessor {
  constructor() {
    this.kafka = new Kafka({ brokers: ['kafka:9092'] });
    this.consumer = this.kafka.consumer({ groupId: 'kappa-processor' });
    this.producer = this.kafka.producer();
  }

  async processStream() {
    await this.consumer.connect();
    await this.producer.connect();

    await this.consumer.subscribe({
      topic: 'page-views',
      fromBeginning: false  // Or true to reprocess all
    });

    // State (in-memory or database)
    const state = {
      pageViews: new Map(),      // page_id -> count
      userSessions: new Map(),   // user_id -> session
      hourlyMetrics: new Map()   // hour -> metrics
    };

    await this.consumer.run({
      eachMessage: async ({ message }) => {
        const event = JSON.parse(message.value);

        // Process event (update all views)
        this.updatePageViews(state, event);
        this.updateSessions(state, event);
        this.updateHourlyMetrics(state, event);

        // Publish derived events
        await this.publishDerivedEvents(event, state);
      }
    });
  }

  updatePageViews(state, event) {
    const pageId = event.page_id;
    const current = state.pageViews.get(pageId) || 0;
    state.pageViews.set(pageId, current + 1);

    // Persist to database
```

```
  db.update('page_views', { page_id: pageId }, { count: current + 1 });
}

updateSessions(state, event) {
  const userId = event.user_id;
  const now = new Date(event.timestamp);

  let session = state.userSessions.get(userId);

  if (!session || now - session.lastActivity > 30 * 60 * 1000) {
    // New session
    session = {
      sessionId: generateId(),
      startTime: now,
      pageViews: 0
    };
    state.userSessions.set(userId, session);
  }

  session.pageViews++;
  session.lastActivity = now;
}

updateHourlyMetrics(state, event) {
  const hour = new Date(event.timestamp).getHours();
  const key = `${event.date}_${hour}`;

  if (!state.hourlyMetrics.has(key)) {
    state.hourlyMetrics.set(key, {
      views: 0,
      uniqueUsers: new Set()
    });
  }

  const metrics = state.hourlyMetrics.get(key);
  metrics.views++;
  metrics.uniqueUsers.add(event.user_id);
}

async publishDerivedEvents(event, state) {
  // Publish aggregated metrics to output topic
  const pageViews = state.pageViews.get(event.page_id);

  if (pageViews % 1000 === 0) {
    await this.producer.send({
      topic: 'page-view-milestones',
      messages: [{
```

```
          value: JSON.stringify({
            page_id: event.page_id,
            milestone: pageViews,
            timestamp: new Date().toISOString()
          })
        }]
      });
    }
  }

  // Reprocess from beginning (e.g., after bug fix)
  async reprocess() {
    console.log('Starting reprocessing from beginning...');

    // Create new consumer group
    const reprocessConsumer = this.kafka.consumer({
      groupId: `kappa-reprocess-${Date.now()}`
    });

    await reprocessConsumer.connect();
    await reprocessConsumer.subscribe({
      topic: 'page-views',
      fromBeginning: true  // Replay entire stream!
    });

    // Process all events from scratch
    // Rebuild all views
    await reprocessConsumer.run({
      eachMessage: async ({ message }) => {
        // Same processing logic
        // Rebuilds state from scratch
      }
    });

    console.log('Reprocessing complete');
  }
}

const processor = new KappaProcessor();
processor.processStream();
```

Lambda vs Kappa

```
┌─────────────┬─────────────┬─────────────┐
│ Feature     │ Lambda      │ Kappa       │
```

Layers	Batch + Speed	Stream only
Code	Duplicate	Single codebase
Complexity	High	Lower
Latency	Mixed	Consistent
Accuracy	Batch=perfect	Stream=good
Reprocessing	Batch layer	Replay stream
Operational	Complex	Simpler

When to use Lambda:
✓ Need perfect accuracy (batch)
✓ Complex batch analytics
✓ Different processing for batch vs real-time

When to use Kappa:
✓ All processing is stream-based
✓ Can replay stream (Kafka retention)
✓ Want simpler architecture
✓ Real-time is good enough

Modern Trend: Moving toward Kappa
Reason: Spark Streaming bridges the gap
  Can do both batch and stream with same code!

---

7. Real-Time Analytics

Use Cases

1. FRAUD DETECTION
  Process: Transaction → Analyze → Alert (< 100ms)
  Pattern: Compare to user history, geo-location

2. RECOMMENDATION
  Process: Click → Update model → Show recommendations
  Pattern: Collaborative filtering in real-time

3. MONITORING
  Process: Metric → Aggregate → Alert (< 1 second)
  Pattern: Window aggregation, anomaly detection

4. REAL-TIME BIDDING (Ad Tech)
  Process: Page load → Auction → Ad served (< 100ms)
  Pattern: Complex matching in real-time

```
5. LIVE DASHBOARDS
   Process: Event → Aggregate → Update UI (< 5 seconds)
   Pattern: Windowed aggregation, streaming joins
```

Real-Time Analytics Pipeline

```javascript
```

```
// Complete real-time analytics system

class RealTimeAnalytics {
  constructor() {
    this.kafka = new Kafka({ brokers: ['kafka:9092'] });
    this.redis = new Redis({ host: 'redis' });
    this.websockets = new Map();  // Active dashboard connections
  }

  // Stage 1: Collect events
  async collectEvents() {
    const producer = this.kafka.producer();
    await producer.connect();

    // Simulate events
    setInterval(async () => {
      const event = {
        type: 'page_view',
        userId: `user-${Math.floor(Math.random() * 1000)}`,
        pageId: `page-${Math.floor(Math.random() * 100)}`,
        timestamp: new Date().toISOString(),
        duration: Math.floor(Math.random() * 60000)
      };

      await producer.send({
        topic: 'events',
        messages: [{ value: JSON.stringify(event) }]
      });
    }, 10);  // 100 events/second
  }

  // Stage 2: Real-time aggregation
  async aggregateRealTime() {
    const consumer = this.kafka.consumer({ groupId: 'aggregator' });
    await consumer.connect();
    await consumer.subscribe({ topic: 'events' });

    // Tumbling windows (1 minute)
    const windows = new Map();
    const WINDOW_SIZE = 60000;  // 1 minute

    await consumer.run({
      eachMessage: async ({ message }) => {
        const event = JSON.parse(message.value);
        const timestamp = new Date(event.timestamp).getTime();
```

```
      // Determine window
      const windowStart = Math.floor(timestamp / WINDOW_SIZE) * WINDOW_SIZE;
      const windowKey = `${event.pageId}:${windowStart}`;

      // Update window
      if (!windows.has(windowKey)) {
        windows.set(windowKey, {
          pageId: event.pageId,
          windowStart: new Date(windowStart).toISOString(),
          viewCount: 0,
          uniqueUsers: new Set(),
          totalDuration: 0
        });
      }

      const window = windows.get(windowKey);
      window.viewCount++;
      window.uniqueUsers.add(event.userId);
      window.totalDuration += event.duration;

      // Publish window results to Redis
      await this.redis.setex(
        `window:${windowKey}`,
        300, // 5 minute TTL
        JSON.stringify({
          pageId: window.pageId,
          windowStart: window.windowStart,
          viewCount: window.viewCount,
          uniqueUsers: window.uniqueUsers.size,
          avgDuration: window.totalDuration / window.viewCount
        })
      );

      // Cleanup old windows
      const now = Date.now();
      for (const [key, win] of windows) {
        const winStart = new Date(win.windowStart).getTime();
        if (now - winStart > 3600000) { // 1 hour old
          windows.delete(key);
        }
      }
    }
  });
}

// Stage 3: Real-time dashboards
async streamToDashboard() {
```

```
  const consumer = this.kafka.consumer({ groupId: 'dashboard' });
  await consumer.connect();
  await consumer.subscribe({ topic: 'events' });

  // Track metrics
  let eventCount = 0;
  let lastUpdate = Date.now();

  await consumer.run({
    eachMessage: async ({ message }) => {
      eventCount++;

      // Update dashboard every second
      const now = Date.now();
      if (now - lastUpdate > 1000) {
        const metrics = {
          eventsPerSecond: eventCount,
          timestamp: new Date().toISOString()
        };

        // Send to all connected dashboards via WebSocket
        this.broadcastToDashboards(metrics);

        eventCount = 0;
        lastUpdate = now;
      }
    }
  });
}

// Stage 4: Anomaly detection
async detectAnomalies() {
  const consumer = this.kafka.consumer({ groupId: 'anomaly-detector' });
  await consumer.connect();
  await consumer.subscribe({ topic: 'events' });

  // Baseline metrics
  const baselines = new Map();

  await consumer.run({
    eachMessage: async ({ message }) => {
      const event = JSON.parse(message.value);

      // Get baseline for this page
      if (!baselines.has(event.pageId)) {
        // Initialize from historical data
        const historical = await this.getHistoricalAverage(event.pageId);
```

```
      baselines.set(event.pageId, historical);
    }

    const baseline = baselines.get(event.pageId);

    // Get current rate (from Redis)
    const currentRate = await this.getCurrentRate(event.pageId);

    // Detect spike (3x baseline)
    if (currentRate > baseline * 3) {
      console.log(`🚨 Anomaly: Page ${event.pageId} has ${currentRate} views/min (baseline: ${baseline})`);

      await this.alertOps({
        type: 'traffic_spike',
        pageId: event.pageId,
        expected: baseline,
        actual: currentRate,
        severity: currentRate > baseline * 10 ? 'critical' : 'warning'
      });
    }

    // Update baseline (moving average)
    baselines.set(
      event.pageId,
      baseline * 0.95 + currentRate * 0.05
    );
  }
 });
}

async getCurrentRate(pageId) {
  // Get recent window from Redis
  const keys = await this.redis.keys(`window:${pageId}:*`);

  if (keys.length === 0) return 0;

  let totalViews = 0;
  for (const key of keys) {
    const data = JSON.parse(await this.redis.get(key));
    totalViews += data.viewCount;
  }

  return totalViews / keys.length;  // Average per window
}

async getHistoricalAverage(pageId) {
  // Query batch data for baseline
```

```
    // Simplified: return reasonable default
    return 100; // 100 views/minute baseline
  }

  broadcastToDashboards(metrics) {
    // Send to all WebSocket clients
    this.websockets.forEach(ws => {
      ws.send(JSON.stringify(metrics));
    });
  }

  async alertOps(alert) {
    console.log('Sending alert:', alert);
    // Send to PagerDuty, Slack, etc.
  }
}

// Start all stages
const analytics = new RealTimeAnalytics();
analytics.collectEvents();
analytics.aggregateRealTime();
analytics.streamToDashboard();
analytics.detectAnomalies();
```

Real-Time Analytics Patterns

Pattern 1: Windowed Aggregation

```
Time-based windows:

Tumbling Window (non-overlapping):
┌────┬────┬────┬────┬────┐
│ 0-5 │ 5-10│10-15│15-20│20-25│ minutes
└────┴────┴────┴────┴────┘

Each event in exactly one window

Sliding Window (overlapping):
┌────────┐
│ 0-5    │
└────────┘
 ┌────────┐
 │ 1-6    │
 └────────┘
  ┌────────┐
  │ 2-7    │
  └────────┘

Windows slide by 1 minute
Event can be in multiple windows

Session Window (activity-based):
┌──────┐   ┌──────┐
│Active│ Gap  │Active│
└──────┘ (idle) └──────┘

Window ends after 30 min inactivity
```

Implementation:

```javascript
```

```
class WindowedAggregation {
  constructor(windowSizeMs, slideMs = null) {
    this.windowSize = windowSizeMs;
    this.slide = slideMs || windowSizeMs;  // Default: tumbling
    this.windows = new Map();
  }

  processEvent(event) {
    const timestamp = new Date(event.timestamp).getTime();

    // Find all windows this event belongs to
    const windowStarts = this.getWindowStarts(timestamp);

    windowStarts.forEach(windowStart => {
      const windowKey = `${event.key}:${windowStart}`;

      if (!this.windows.has(windowKey)) {
        this.windows.set(windowKey, {
          key: event.key,
          windowStart,
          windowEnd: windowStart + this.windowSize,
          count: 0,
          sum: 0,
          events: []
        });
      }

      const window = this.windows.get(windowKey);
      window.count++;
      window.sum += event.value;
      window.events.push(event);
    });

    // Emit completed windows
    this.emitCompletedWindows(timestamp);
  }

  getWindowStarts(timestamp) {
    if (this.slide === this.windowSize) {
      // Tumbling window
      const windowStart = Math.floor(timestamp / this.windowSize) * this.windowSize;
      return [windowStart];
    } else {
      // Sliding window
      const starts = [];
      let start = Math.floor(timestamp / this.slide) * this.slide;
```

```
      while (start + this.windowSize > timestamp) {
        starts.push(start);
        start -= this.slide;
      }

      return starts;
    }
  }

  emitCompletedWindows(currentTimestamp) {
    const completed = [];

    for (const [windowKey, window] of this.windows) {
      // Window is complete if current time > window end
      if (currentTimestamp > window.windowEnd) {
        completed.push(windowKey);

        // Emit result
        console.log('Window completed:', {
          key: window.key,
          start: new Date(window.windowStart).toISOString(),
          end: new Date(window.windowEnd).toISOString(),
          count: window.count,
          average: window.sum / window.count
        });
      }
    }

    // Cleanup completed windows
    completed.forEach(key => this.windows.delete(key));
  }
}

// Usage
const aggregator = new WindowedAggregation(
  5 * 60 * 1000,  // 5-minute windows
  1 * 60 * 1000   // Slide by 1 minute (overlapping)
);

// Process events
aggregator.processEvent({
  key: 'page-123',
  value: 1,
  timestamp: '2024-01-20T10:00:00Z'
});
```

```
aggregator.processEvent({
  key: 'page-123',
  value: 1,
  timestamp: '2024-01-20T10:01:00Z'
});

// Event at 10:00:00 is in windows:
// - [09:56:00 - 10:01:00]
// - [09:57:00 - 10:02:00]
// - [09:58:00 - 10:03:00]
// - [09:59:00 - 10:04:00]
// - [10:00:00 - 10:05:00]
```

Pattern 2: Stream Joins

```
Joining two event streams:

Stream A (Orders):        Stream B (Payments):
Order 1 (10:00:00)       Payment 1 (10:00:05)
Order 2 (10:00:10)       Payment 2 (10:00:15)
Order 3 (10:00:20)       Payment 3 (10:00:25)

Goal: Match orders with payments

Challenges:
- Events arrive out of order
- Time difference between order and payment
- Need to buffer events

Solution: Time-windowed join
Wait up to 1 minute for matching event
```

Implementation:

```javascript
```

```
class StreamJoiner {
  constructor() {
    this.orderBuffer = new Map();
    this.paymentBuffer = new Map();
    this.maxWaitTime = 60000; // 1 minute
  }

  async processOrder(order) {
    const orderId = order.orderId;

    // Check if payment already arrived
    if (this.paymentBuffer.has(orderId)) {
      const payment = this.paymentBuffer.get(orderId);
      this.paymentBuffer.delete(orderId);

      // Emit joined event
      this.emitJoined(order, payment);
      return;
    }

    // Buffer order, wait for payment
    this.orderBuffer.set(orderId, {
      order,
      arrivedAt: Date.now()
    });

    // Set timeout to cleanup
    setTimeout(() => {
      if (this.orderBuffer.has(orderId)) {
        console.log(`Order ${orderId} timeout (no payment received)`);
        this.orderBuffer.delete(orderId);
      }
    }, this.maxWaitTime);
  }

  async processPayment(payment) {
    const orderId = payment.orderId;

    // Check if order already arrived
    if (this.orderBuffer.has(orderId)) {
      const { order } = this.orderBuffer.get(orderId);
      this.orderBuffer.delete(orderId);

      // Emit joined event
      this.emitJoined(order, payment);
      return;
```

```
    }

    // Buffer payment, wait for order
    this.paymentBuffer.set(orderId, {
      payment,
      arrivedAt: Date.now()
    });

    setTimeout(() => {
      if (this.paymentBuffer.has(orderId)) {
        console.log(`Payment ${orderId} timeout (no order received)`);
        this.paymentBuffer.delete(orderId);
      }
    }, this.maxWaitTime);
  }

  emitJoined(order, payment) {
    const joined = {
      orderId: order.orderId,
      userId: order.userId,
      orderTotal: order.total,
      paymentAmount: payment.amount,
      paymentStatus: payment.status,
      orderTimestamp: order.timestamp,
      paymentTimestamp: payment.timestamp,
      timeDiff: new Date(payment.timestamp) - new Date(order.timestamp)
    };

    console.log('Joined event:', joined);

    // Publish to output stream
    kafka.publish('order-payment-joined', joined);
  }
}

const joiner = new StreamJoiner();

// Process streams
kafka.subscribe('orders', (order) => joiner.processOrder(order));
kafka.subscribe('payments', (payment) => joiner.processPayment(payment));
```

Pattern 3: Real-Time Dashboard

```javascript
```

```
// Live dashboard with WebSocket updates

const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

class LiveDashboard {
  constructor() {
    this.kafka = new Kafka({ brokers: ['kafka:9092'] });
    this.currentMetrics = {
      eventsPerSecond: 0,
      activeUsers: new Set(),
      topPages: [],
      errorRate: 0
    };

    this.clients = new Set();
  }

  async start() {
    // Accept WebSocket connections
    wss.on('connection', (ws) => {
      console.log('Dashboard client connected');
      this.clients.add(ws);

      // Send current metrics immediately
      ws.send(JSON.stringify(this.currentMetrics));

      ws.on('close', () => {
        this.clients.delete(ws);
      });
    });

    // Process events
    const consumer = this.kafka.consumer({ groupId: 'dashboard' });
    await consumer.connect();
    await consumer.subscribe({ topic: 'events' });

    let eventCount = 0;
    let lastUpdate = Date.now();

    await consumer.run({
      eachMessage: async ({ message }) => {
        const event = JSON.parse(message.value);
        eventCount++;

        // Update metrics
```

```
        this.currentMetrics.activeUsers.add(event.userId);

        // Every second, calculate and broadcast
        const now = Date.now();
        if (now - lastUpdate > 1000) {
          this.currentMetrics.eventsPerSecond = eventCount;
          this.currentMetrics.activeUsersCount = this.currentMetrics.activeUsers.size;

          // Broadcast to all connected clients
          this.broadcast(this.currentMetrics);

          // Reset
          eventCount = 0;
          this.currentMetrics.activeUsers.clear();
          lastUpdate = now;
        }
      }
    });
  }

  broadcast(data) {
    const message = JSON.stringify(data);

    this.clients.forEach(client => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(message);
      }
    });
  }
}

const dashboard = new LiveDashboard();
dashboard.start();

// Browser client:
// const ws = new WebSocket('ws://localhost:8080');
// ws.onmessage = (event) => {
//   const metrics = JSON.parse(event.data);
//   updateDashboard(metrics);
// };
```

Key Takeaways

1. **Batch vs Stream:**
   - Batch: Large volumes, periodic, high latency

- Stream: Continuous, real-time, low latency
- Use both in Lambda architecture

2. **MapReduce:**
   - Map: Transform data in parallel
   - Reduce: Aggregate results
   - Shuffle: Group by key
   - Foundation of big data processing
3. **Hadoop Ecosystem:**
   - HDFS: Distributed storage
   - YARN: Resource management
   - MapReduce: Processing framework
   - Hive: SQL on Hadoop
4. **Apache Spark:**
   - In-memory processing (100x faster)
   - Unified API (batch + stream)
   - RDD, DataFrame, SQL
   - Machine learning (MLlib)
5. **Lambda Architecture:**
   - Batch layer: Complete, accurate
   - Speed layer: Real-time, approximate
   - Serving layer: Merge both
   - Complex but comprehensive
6. **Kappa Architecture:**
   - Stream processing only
   - Simpler than Lambda
   - Replay for reprocessing
   - Modern approach
7. **Real-Time Analytics:**
   - Windowed aggregation
   - Stream joins

- Anomaly detection
- Live dashboards

Practice Problems

1. Design a real-time analytics system for Twitter (trending topics, user engagement)
2. Calculate: 1 PB of data, 1000 machines, process at 100 GB/sec. How long to process?
3. Compare Lambda vs Kappa for Netflix viewing analytics. Which would you choose?
4. Design a fraud detection system processing 100,000 transactions/second

Next Chapter Preview

Chapter 18: **Search Systems**

- Full-text search
- Inverted indexes
- Elasticsearch
- Ranking and relevance
- Search optimization

Ready to continue with more chapters or practice system design problems?