Here's the complete textbook content in a clean, printable format:

MASTERING DATA ENGINEERING IN PALANTIR FOUNDRY
From Novice to Certified Expert
Comprehensive Edition 2.0

TABLE OF CONTENTS

PART 0: BEFORE WE BEGIN - UNDERSTANDING FOUNDRY'S MINDSET

CHAPTER 0.1: WHAT MAKES FOUNDRY DIFFERENT?

Traditional Data Platforms:
```
Source → ETL Pipeline → Data Warehouse → BI Tools
```

Foundry's Approach:
```
Sources → Foundry (Raw Data → Cleaned Data → **Ontology**) → EVERYTHING
```

Key Insight: In Foundry, everything connects back to central **business meaning**. You're not just moving data; you're building a **digital twin** of your organization's operations.

CHAPTER 0.2: FOUNDRY'S CORE METAPHORS

1. **The Repository is a Time Machine**
   - Every change saved forever with timestamp and version
   - Can revert to any point in time
   - Complete audit trail

2. **The Ontology is a Dictionary**
   - Defines business terms, not just technical terms
   - Creates common language across organization
   - Enables semantic queries

3. **Transforms are Recipes**
   - Take ingredients (input data)
   - Produce dishes (output data)
   - Leave perfect records of what they did

4. **Workflows are Assembly Lines**
   - Orchestrate recipes in right order
   - Schedule at right time
   - Manage dependencies automatically

PART I: FOUNDATIONAL CONCEPTS (DEEP DIVE)

CHAPTER 1: YOUR FIRST DAY IN FOUNDRY - UNDERSTANDING THE INTERFACE

1.1 THE MAIN APPLICATIONS

| Application | Purpose | Key Tools | Metaphor |

|-------------|---------|-----------|----------|
Data Integration	Bring data in	Connectors, Loaders, Contour	Loading dock
Transform	Clean and shape data	Code Repos, SQL Transforms, Prepare	Kitchen
Orchestration	Schedule pipelines	Workflows, Jobs	Project manager
Ontology	Define business meaning	Object Types, Properties	Dictionary department
Monitor	Observe health	Alerts, Metrics, Lineage	Control room

1.2 NAVIGATION BASICS

Resource Identifier (RID):
```
ri.foundry.main.dataset.3f4b5c6d-1234-5678-9abc-def012345678
```
- **ri** = Resource Identifier
- **foundry.main.dataset** = Type of resource
- **UUID** = Unique identifier

Paths vs RIDs:
- **Paths:** Human-readable (`/Global Sales/Customers/USA`)
- **RIDs:** Permanent identifiers (never change)
- **Best Practice:** Use RIDs in code, paths in configuration

1.3 THE FOUNDRY FILE SYSTEM

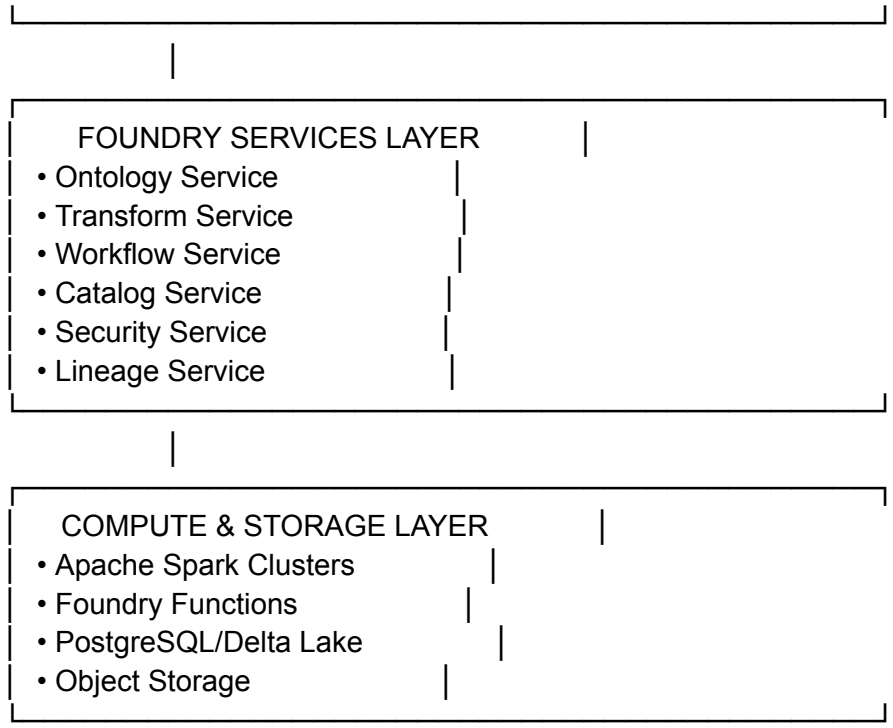
```
/ (Root)
├── Global (Shared with everyone)
│   ├── Sales
│   ├── Marketing
│   └── Operations
├── Team (Shared with your team)
└── Private (Only you)
```

Permission Inheritance:
- Folders control visibility
- `/Global/Sales` = Everyone in Sales can access
- Child resources inherit parent permissions

CHAPTER 2: FOUNDRY'S CORE ARCHITECTURE

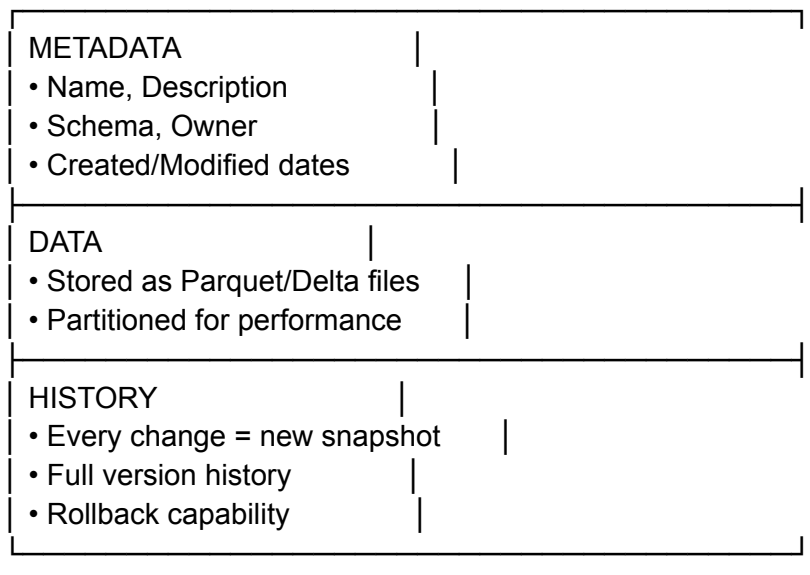
2.1 THREE-LAYER ARCHITECTURE

```
┌─────────────────────────────────────────────────────┐
│          USER INTERFACES                            │
│  • Web UI    • APIs                                 │
│  • SDKs      • SQL Queries                          │
```



2.2 DATA STORAGE - DATASETS

Dataset Structure:



Dataset Types:

1. **Spark Datasets**
   - Built on Apache Spark
   - Terabyte-scale processing
   - Parquet/Delta storage

2. **SQL Datasets**
   - Materialized SQL views
   - Auto-refresh on source change
   - Great for business logic

3. **Ontology Datasets**
   - Directly tied to Object Types
   - Auto-populated on sync
   - Enable semantic queries

2.3 VERSIONING IN ACTION

Monday:
- Dataset RID: `ri.dataset.aaa`
- Contains: Customer 123 = "John Smith"

Tuesday (after update):
- New RID: `ri.dataset.bbb`
- Contains: Customer 123 = "John Q. Smith"
- `ri.dataset.aaa` still exists unchanged
- Lineage: `aaa → bbb`

Key Benefit: Reproducibility. Pipelines using specific RIDs always get same data.

CHAPTER 3: DATA MODELING FUNDAMENTALS

3.1 DATA HIERARCHY

```
Level 0: Files (CSV, JSON, PDF)
    ↓ Loaders/Contour
Level 1: Raw Datasets (As-is from source)
    ↓ Transform
Level 2: Cleaned Datasets (Business-ready)
    ↓ Apply Links
Level 3: Ontology Objects (Business meaning)
    ↓ Ontology Sync
Level 4: Connected Knowledge Graph
```

3.2 SCHEMA MANAGEMENT

Schema-on-Read (Flexible):
```python
# Foundry infers schema automatically
df = spark.read.csv("/path/to/file")
# Can handle new columns
```

Schema Enforcement (Strict):
```python
from pyspark.sql.types import StructType, StringType, IntegerType

schema = StructType([
    StructField("customer_id", IntegerType(), True),
    StructField("name", StringType(), False)  # Required field
])
```

Best Practice Progression:
1. Exploration: Schema-on-read
2. Development: Add validation
3. Production: Enforce strict schema

PART II: HANDS-ON DATA ENGINEERING WORKFLOWS

CHAPTER 4: DATA INGESTION

4.1 THREE INGESTION METHODS

Method	When to Use	Frequency	Complexity
Connectors & Loaders	Scheduled imports	Scheduled	Medium
Contour	One-time, exploratory	Manual	Low
API Upload	Programmatic needs	Event-driven	High

4.2 LOADER CONFIGURATION EXAMPLE

```yaml
# salesforce_loader.yaml
source:
  type: salesforce
  connection: salesforce-prod
  object: Opportunity
  query: |
    SELECT Id, Name, Amount, CloseDate
    FROM Opportunity
    WHERE LastModifiedDate >= YESTERDAY

destination:
  path: /Global/Sales/bronze/opportunities
  format: parquet
  mode: append
```

```
schedule:
  cron: "0 3 * * *"  # 3 AM daily
  timezone: UTC

notifications:
  on_failure:
    - email: data-team@company.com
    - slack: "#data-alerts"
```

4.3 BRONZE-SILVER-GOLD PATTERN

Bronze Layer (Raw):
- Path: `/project/raw/`
- Strategy: Append-only
- No transformations
- Preserve source fidelity

Silver Layer (Cleaned):
- Path: `/project/cleaned/`
- Actions: Deduplicate, validate, standardize
- Add: Surrogate keys, business rules

Gold Layer (Business):
- Path: `/project/gold/`
- Contains: Aggregates, joined views
- Ready for: Analytics, reporting, ML

CHAPTER 5: TRANSFORMATION

5.1 TOOL SELECTION MATRIX

Tool	Best For	Scale	Skill Required
Prepare	Simple cleansing	Small	Low (no-code)
SQL Transform	Business logic	Medium	Medium (SQL)
PySpark Transform	Complex processing	Large	High (Python)
Foundry Functions	Event-driven tasks	Micro	Medium (Python/TS)

5.2 CODE REPOSITORY STRUCTURE

```
/customer-pipeline/
├── transforms-python/
│   ├── __init__.py
│   ├── clean_customers.py
│   └── enrich_orders.py
├── transforms-spark/
```

```
│   └── process_large_data.scala
├── tests/
│   ├── test_clean_customers.py
│   └── test_data_quality.py
├── .pre-commit-config.yaml
├── .synthea-build.yaml
└── README.md
```

5.3 PRODUCTION PYSPARK TRANSFORM

```python
"""
CLEAN CUSTOMER DATA TRANSFORM
Input: Raw customer data
Output: Cleaned customer data + Quality report
"""

from transforms.api import transform, Input, Output
import pyspark.sql.functions as F
from pyspark.sql.window import Window

@transform(
    raw_customers=Input("/Global/Sales/bronze/customers"),
    cleaned_customers=Output("/Global/Sales/silver/customers_clean"),
    quality_report=Output("/Global/Sales/reports/quality_daily")
)
def clean_customer_data(raw_customers, cleaned_customers, quality_report):

    # 1. READ RAW DATA
    df = raw_customers.dataframe()

    # 2. BASIC CLEANING
    # Trim whitespace
    for col_name, col_type in df.dtypes:
        if col_type == "string":
            df = df.withColumn(col_name, F.trim(F.col(col_name)))

    # 3. STANDARDIZE COUNTRY CODES
    country_map = {
        "United States": "USA",
        "US": "USA",
        "United Kingdom": "UK",
        "GB": "UK"
    }

    mapping_expr = F.create_map([F.lit(x) for pair in country_map.items() for x in pair])
    df = df.withColumn("country_std",
```

```
                    F.coalesce(mapping_expr[F.col("country")],
                               F.col("country")))

    # 4. DEDUPLICATE (KEEP MOST RECENT)
    window = Window.partitionBy("customer_id").orderBy(F.col("updated_at").desc())
    df = (df.withColumn("row_num", F.row_number().over(window))
          .filter(F.col("row_num") == 1)
          .drop("row_num"))

    # 5. QUALITY CHECKS
    checks = []

    # Uniqueness check
    unique_ratio = df.select("customer_id").distinct().count() / df.count()
    checks.append({
        "check": "customer_id_uniqueness",
        "passed": unique_ratio >= 0.99,
        "value": unique_ratio
    })

    # Null check
    for field in ["customer_id", "email"]:
        null_pct = df.filter(F.col(field).isNull()).count() / df.count()
        checks.append({
            "check": f"{field}_not_null",
            "passed": null_pct <= 0.01,
            "value": null_pct
        })

    # 6. CREATE QUALITY REPORT
    df_report = spark.createDataFrame(checks)

    # 7. WRITE OUTPUTS
    cleaned_customers.write_dataframe(
        df,
        partition_cols=["country_std", "load_date"]
    )

    quality_report.write_dataframe(df_report)

    # 8. LOG METRICS
    cleaned_customers.set_stat("row_count", df.count())
    cleaned_customers.set_stat("quality_passed", all(c["passed"] for c in checks))
```

5.4 SQL TRANSFORM EXAMPLE

```sql
```

```sql
-- customer_360_view.sql
-- Business-ready customer 360 view

WITH customer_orders AS (
    SELECT
        customer_id,
        COUNT(*) as total_orders,
        SUM(amount) as lifetime_value,
        MAX(order_date) as last_order_date
    FROM `/Global/Sales/silver/orders`
    WHERE status = 'COMPLETED'
    GROUP BY customer_id
),

customer_support AS (
    SELECT
        customer_id,
        COUNT(*) as ticket_count,
        SUM(CASE WHEN status = 'OPEN' THEN 1 ELSE 0 END) as open_tickets
    FROM `/Global/Support/tickets`
    GROUP BY customer_id
)

SELECT
    c.*,
    COALESCE(co.total_orders, 0) as total_orders,
    COALESCE(co.lifetime_value, 0) as lifetime_value,
    co.last_order_date,
    COALESCE(cs.ticket_count, 0) as ticket_count,
    COALESCE(cs.open_tickets, 0) as open_tickets,

    -- Customer health score
    CASE
        WHEN co.lifetime_value > 10000 AND cs.open_tickets = 0 THEN 'HEALTHY'
        WHEN co.lifetime_value < 1000 OR cs.open_tickets > 3 THEN 'RISK'
        ELSE 'NEUTRAL'
    END as health_score,

    CURRENT_TIMESTAMP() as snapshot_time

FROM `/Global/Sales/silver/customers_clean` c
LEFT JOIN customer_orders co ON c.customer_id = co.customer_id
LEFT JOIN customer_support cs ON c.customer_id = cs.customer_id
```

CHAPTER 6: ORCHESTRATION

6.1 WORKFLOW DAG EXAMPLE

```
                    ┌─────────────────────┐
                    │   Start (4 AM)      │
                    └──────────┬──────────┘
                               │
                    ┌──────────┴──────────┐
                    ▼                     ▼
          ┌─────────────────┐   ┌─────────────────┐
          │ Load Customers  │   │ Load Orders     │
          │   (Job A)       │   │   (Job B)       │
          └────────┬────────┘   └────────┬────────┘
                   │                     │
                   └──────────┬──────────┘
                              │
                              ▼
                    ┌─────────────────┐
                    │ Create 360 View │
                    │    (Job C)      │
                    └────────┬────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Apply Ontology  │
                    │    (Job D)      │
                    └────────┬────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Send Daily      │
                    │   Report        │
                    │    (Job E)      │
                    └─────────────────┘
```

6.2 WORKFLOW DEFINITION

```yaml
# daily-customer-pipeline.yaml
name: "daily-customer-pipeline"
description: "Process customer data daily"

schedule:
  trigger: "cron"
  expression: "0 4 * * *"  # 4 AM daily
  timezone: "America/New_York"

jobs:
  load_raw_customers:
    type: spark
    transform: "/transforms/load_customers"
```

```
    resources:
      executor_instances: 4
      executor_memory: "8g"

  clean_customer_data:
    type: spark
    transform: "/transforms/clean_customers"
    depends_on: ["load_raw_customers"]

  create_customer_360:
    type: sql
    query: "/queries/customer_360.sql"
    depends_on: ["clean_customer_data"]

  send_daily_report:
    type: email
    to: "sales-team@company.com"
    subject: "Daily Customer Update"
    body: "Processed {{clean_customer_data.output_row_count}} customers"
    depends_on: ["create_customer_360"]

notifications:
  on_failure:
    - type: slack
      channel: "#data-alerts"
      message: "Pipeline failed: {{workflow.error_message}}"
```

6.3 ADVANCED PATTERNS

Pattern 1: Fan-Out, Fan-In
```yaml
# Process regions in parallel
process_usa:
  type: spark
  transform: "/transforms/process_region"
  parameters: {"region": "USA"}

process_europe:
  type: spark
  transform: "/transforms/process_region"
  parameters: {"region": "EUROPE"}

process_asia:
  type: spark
  transform: "/transforms/process_region"
  parameters: {"region": "ASIA"}
```

```
combine_results:
  type: spark
  transform: "/transforms/combine_regions"
  depends_on: ["process_usa", "process_europe", "process_asia"]
```

Pattern 2: Conditional Execution
```yaml
check_data_quality:
  type: spark
  transform: "/transforms/quality_check"

process_data:
  type: spark
  transform: "/transforms/process"
  condition: "{{check_data_quality.result.passed}}"
  depends_on: ["check_data_quality"]

alert_on_failure:
  type: email
  condition: "not {{check_data_quality.result.passed}}"
  depends_on: ["check_data_quality"]
```

CHAPTER 7: ONTOLOGY

7.1 OBJECT TYPE DEFINITION

```yaml
# customer_object.yaml
object_type:
  name: "Customer"
  description: "Company purchasing our products"

  primary_key:
    - "customer_id"

  properties:
    - name: "customer_id"
      type: "string"
      required: true

    - name: "customer_name"
      type: "string"

    - name: "industry"
      type: "enum"
      allowed_values: ["TECH", "FINANCE", "HEALTHCARE", "RETAIL"]
```

```
    - name: "annual_revenue"
      type: "decimal"
      unit: "USD"

    - name: "relationship_manager"
      type: "link"
      linked_object_type: "Employee"

    - name: "contracts"
      type: "link"
      linked_object_type: "Contract"
      cardinality: "many"
```

7.2 APPLYING LINKS

```python
from transforms.api import transform, Input, Output
from transforms.ontology import apply_links

@transform(
    customers=Input("/Global/Sales/silver/customers_clean"),
    customers_linked=Output("/Global/Sales/gold/customers_linked")
)
def link_customers(customers, customers_linked):
    df = customers.dataframe()

    linked_df = apply_links(
        dataframe=df,
        links={
            "customer_id": "Customer.customer_id",
            "company_name": "Customer.customer_name",
            "sales_rep_id": "Customer.relationship_manager",
            "industry_code": "Customer.industry"
        }
    )

    customers_linked.write_dataframe(linked_df)
```

7.3 ONTOLOGY QUERYING

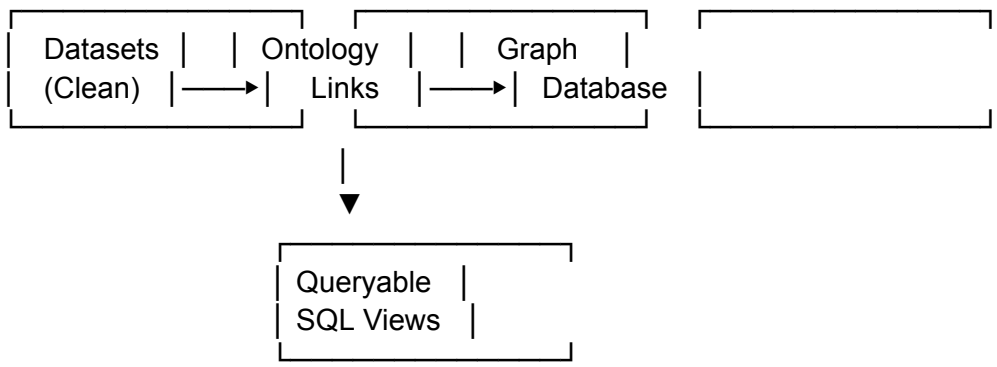
Traditional SQL (Joins Required):
```sql
SELECT
    c.customer_name,
    o.order_date,
```

```
    o.amount
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE c.country = 'USA'
```

Ontology SQL (No Explicit Joins):
```sql
SELECT
    customer.name AS customer_name,
    customer.orders.order_date,
    customer.orders.amount,
    customer.contracts.start_date,
    customer.relationship_manager.email
FROM Customer
WHERE customer.country = 'USA'
  AND customer.industry = 'TECH'
  AND customer.contracts.status = 'ACTIVE'
```

How It Works:
1. Foundry understands relationships from ontology
2. Automatically generates optimal joins
3. You write business logic, not database logic

7.4 ONTOLOGY SYNC PROCESS



Sync Steps:
1. Apply links in transforms
2. Go to Ontology Manager
3. Select Object Types
4. Click "Sync"
5. Wait for completion (minutes to hours)
6. Query using ontology syntax

PART III: PRODUCTION READINESS & BEST PRACTICES

CHAPTER 8: DATAOPS IN FOUNDRY

8.1 BRANCHING STRATEGY

Development Flow:
```
Main Branch (production)
   ↑ Cut
Staging Branch (UAT)
   ↑ Merge
Feature Branch (development)
```

Branch Creation:
```bash
# Create feature branch
foundry branch create --name "feature/add-segmentation"

# Work in branch
/path/in/branch/feature/add-segmentation/

# Merge to staging
foundry merge --source feature/add-segmentation --target staging

# Cut to production
foundry cut create --source staging --target main --message "Release v1.2.0"
```

8.2 CI/CD PIPELINE

```yaml
# .synthea-build.yaml
name: Data Pipeline CI/CD

on:
  push:
    branches: [main, staging]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: foundry-spark
```

```
    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Run unit tests
      run: python -m pytest tests/ -v

    - name: Validate schemas
      run: python scripts/validate_schemas.py

    - name: Check data quality
      run: python tests/quality/test_suite.py

  deploy:
    needs: test
    if: github.ref == 'refs/heads/main'

    steps:
    - name: Cut to production
      run: foundry cut create --source staging --target main

    - name: Deploy workflow
      run: foundry workflow deploy pipeline.yaml

    - name: Run smoke tests
      run: python tests/smoke/test_production.py
```

8.3 TESTING FRAMEWORK

Unit Test Example:
```python
# tests/test_customer_transform.py
import pytest
from transforms.api import Input, Output
from pyspark.sql import SparkSession
import clean_customers

def test_customer_cleaning(spark):
    """Test customer data cleaning logic."""

    # Create test data
    test_data = [
        (1, "  JOHN  ", "US", "john@email.com"),
        (2, "Jane", "United States", "jane@email.com")
    ]

    test_df = spark.createDataFrame(
```

```
        test_data,
        ["customer_id", "name", "country", "email"]
    )

    # Mock Foundry objects
    class MockInput:
        def dataframe(self):
            return test_df

    class MockOutput:
        def write_dataframe(self, df):
            self.result = df

    # Execute transform
    input_mock = MockInput()
    output_mock = MockOutput()

    clean_customers.clean_customer_data(input_mock, output_mock)

    # Assertions
    result = output_mock.result
    assert result.count() == 2
    assert result.filter("name = '  JOHN  '").count() == 0
    assert result.filter("country = 'USA'").count() == 2
```

CHAPTER 9: MONITORING & ALERTING

9.1 MONITORING DASHBOARD ELEMENTS

Pipeline Health:
- ✅ Success rate (target: >99%)
- ⏱️ Average duration (track trends)
- 📈 Resource utilization
- ⏳ Queue wait times

Data Quality:
- 📊 Row count changes
- ❌ Null value percentages
- 🔄 Schema drift detection
- 🕒 Freshness (time since update)

9.2 ALERT CONFIGURATION

```yaml
# dataset_monitors.yaml
monitors:
  - type: row_count
```

```
    dataset: "/Global/Sales/gold/customer_360"
    condition: "change_percentage > 30"
    action: "alert"

  - type: freshness
    dataset: "/Global/Sales/bronze/customers"
    expected_interval: "24h"
    action: "email:data-team@company.com"

  - type: schema_change
    dataset: "/Global/Sales/silver/orders"
    action: "slack:#data-schema-changes"

  - type: data_quality
    dataset: "/Global/Sales/gold/revenue"
    checks:
      - column: "revenue_amount"
        rule: "> 0"
      - column: "customer_id"
        rule: "not_null"
    action: "pagerduty:data-engineers"
```

9.3 LINEAGE ANALYSIS

Questions Lineage Answers:
1. **Upstream:** "Where did this number come from?"
2. **Downstream:** "Who's using this dataset?"
3. **Impact:** "What breaks if I change this?"
4. **Timeline:** "How long does data flow take?"

Lineage Visualization:
```
Dataset A (v1.0) → Transform X → Dataset B (v1.0)
      ↓                  ↓
Dataset A (v2.0) → Transform X → Dataset B (v2.0)
      ↓                  ↓
   Source              Dashboard
                         ↓
                    Business User
```

CHAPTER 10: PERFORMANCE OPTIMIZATION

10.1 SPARK OPTIMIZATION

Common Issue 1: Data Skew
```python
```

```
# PROBLEM: Some keys have millions of records
df.join(large_df, "customer_id")  # Causes skew

# SOLUTION 1: Salting technique
df.withColumn("salt", (F.rand() * 100).cast("int"))
  .join(large_df.withColumn("salt", (F.rand() * 100).cast("int")),
        ["customer_id", "salt"])

# SOLUTION 2: Broadcast for small tables
from pyspark.sql.functions import broadcast
df.join(broadcast(small_df), "key")
```

Common Issue 2: Too Many Small Files
```python
# Write with optimal file size
(df.repartition(100)  # Aim for ~100MB files
   .write
   .parquet("/output"))

# Enable adaptive query execution
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")
spark.conf.set("spark.sql.adaptive.advisoryPartitionSizeInBytes", "128MB")
```

Common Issue 3: Memory Issues
```python
# Increase memory for heavy operations
spark.conf.set("spark.executor.memory", "16g")
spark.conf.set("spark.driver.memory", "8g")
spark.conf.set("spark.memory.fraction", "0.8")
```

10.2 FOUNDRY-SPECIFIC OPTIMIZATIONS

1. Partition Strategy:
```python
# Time-based partitioning
df.write.partitionBy("year", "month", "day").parquet("/output")

# Business key partitioning
df.write.partitionBy("region", "department").parquet("/output")
```

2. Incremental Processing:
```python
# Process only new/changed data
```

```
last_run = get_last_successful_run()
new_data = df.filter(F.col("updated_at") > last_run)

# Watermark technique
df.withWatermark("event_time", "1 hour")
  .groupBy("customer_id", window("event_time", "1 hour"))
  .count()
```

3. Caching Strategy:
```python
# Cache frequently used datasets
df.cache().count()  # Materialize cache

# Check if caching helps
if df.storageLevel.useMemory:
    print("Dataset is cached in memory")
```

4. Query Optimization:
```sql
-- Use predicate pushdown
SELECT * FROM orders
WHERE order_date >= '2024-01-01'
  AND region = 'USA'  -- Foundry pushes to storage layer

-- Avoid SELECT *
SELECT customer_id, order_date, amount  -- Only needed columns
FROM orders

-- Use appropriate join types
-- Foundry optimizes based on statistics
```

10.3 PERFORMANCE CHECKLIST

Before Production:
- [ ] Partition columns defined
- [ ] File sizes optimized (~100MB each)
- [ ] Data skew addressed
- [ ] Appropriate join strategies
- [ ] Memory settings tuned
- [ ] Caching strategy defined

Monitoring in Production:
- [ ] Spark UI metrics tracked
- [ ] Duration baselines established
- [ ] Alert thresholds set

- [ ] Resource utilization monitored
- [ ] Query plans reviewed regularly

PART IV: EXAM PREPARATION & CERTIFICATION

CHAPTER 11: EXAM STRUCTURE

Exam Details:
- **Questions:** 60-80 multiple choice
- **Time:** 120 minutes
- **Format:** Online proctored
- **Passing Score:** ~70%

Topic Weights:
1. **Data Ingestion (20%):** Connectors, Loaders, Contour
2. **Transformation (25%):** Code Repos, SQL Transforms, PySpark
3. **Orchestration (20%):** Workflows, Jobs, Scheduling
4. **Ontology (20%):** Object Types, Links, Querying
5. **Operations (15%):** Monitoring, Security, Best Practices

CHAPTER 12: PRACTICE QUESTIONS

QUESTION 1: INGESTION
Scenario: You need to ingest 50 GB of CSV data from S3 daily. Files arrive at random times. What's the MOST efficient approach?

Options:
A) Use Contour to manually upload each file
B) Create a Loader with daily schedule
C) Use Foundry Function triggered by S3 events
D) Write Python script using Foundry API

Answer: B (Loader with schedule)
Explanation: C would work but over-engineered. Loaders are designed for scheduled bulk ingestion.

QUESTION 2: TRANSFORMATION
Scenario: PySpark job failing with "Out of Memory" on driver. First action?

Options:
A) Increase spark.driver.memory
B) Add more partitions
C) Check for data skew using Spark UI
D) Switch to SQL transforms

Answer: C (Check for skew)

Explanation: Always diagnose before treating. Skew is common cause of OOM.

QUESTION 3: ONTOLOGY
Scenario: Dataset has employee_id, manager_id, department_id. Want to query employees and navigate to managers. First step?

Options:
A) Create SQL views
B) Apply links to Object Types
C) Write transform to join tables
D) Use Prepare to merge datasets

Answer: B (Apply links)
Explanation: Links connect data to ontology enabling navigation queries.

QUESTION 4: ORCHESTRATION
Scenario: Job B depends on Job A. Job A fails. What happens?

Options:
A) Job B runs anyway
B) Job B waits for manual restart
C) Job B is skipped
D) Workflow fails immediately

Answer: C (Job B is skipped)
Explanation: Dependent jobs skip when dependencies fail (configurable).

QUESTION 5: BEST PRACTICES
Scenario: Team of 5 engineers working on same pipeline. Best approach?

Options:
A) All work in main branch
B) Use feature branches
C) Create separate projects
D) Work in different folders

Answer: B (Feature branches)
Explanation: Branches enable parallel development with isolation.

CHAPTER 13: 6-WEEK STUDY PLAN

Week 1-2: Foundation Building
- [ ] Complete Foundry tutorials
- [ ] Learn basic PySpark
- [ ] Practice SQL transforms
- [ ] Create simple pipeline

Week 3-4: Hands-On Practice

- [ ] Build bronze-silver-gold pipeline
- [ ] Create Object Type and links
- [ ] Set up scheduled workflow
- [ ] Implement data quality checks

Week 5: Advanced Topics
- [ ] Study performance optimization
- [ ] Practice monitoring setup
- [ ] Review security models
- [ ] Understand CI/CD in Foundry

Week 6: Exam Preparation
- [ ] Take practice exams
- [ ] Review official documentation
- [ ] Join community forums
- [ ] Schedule exam

Daily Study Routine:
- Morning (30 min): Review concepts
- Afternoon (60 min): Hands-on practice
- Evening (30 min): Practice questions

CHAPTER 14: EXAM DAY STRATEGY

Before Exam:
1. **Technical Check:**
   - Test computer and internet
   - Close all unnecessary applications
   - Clear workspace (proctoring requirements)

2. **Materials Ready:**
   - Government ID
   - Water bottle
   - Scratch paper and pen (if allowed)

3. **Mental Preparation:**
   - Review key concepts
   - Practice breathing exercises
   - Set positive mindset

During Exam:
1. **Time Management:**
   - First pass: Answer known questions (60 minutes)
   - Second pass: Review flagged questions (40 minutes)
   - Final check: Review all answers (20 minutes)

2. **Question Strategy:**
   - Read each question twice

   - Eliminate obviously wrong answers
   - Flag uncertain questions
   - Watch for "MOST" and "BEST" keywords

3. **Technical Questions:**
   - Think about scalability
   - Consider Foundry best practices
   - Remember specific Foundry terminology

Common Pitfalls to Avoid:
- ❌ Overthinking simple questions
- ❌ Changing answers without reason
- ❌ Spending too long on one question
- ❌ Forgetting about business context
- ❌ Ignoring Foundry-specific features

After Exam:
- Take notes on difficult questions
- Celebrate completion
- Plan next steps regardless of outcome
- Request detailed feedback if available

APPENDICES

APPENDIX A: FOUNDRY CLI QUICK REFERENCE

```bash
# DATASET COMMANDS
foundry dataset list --path "/Global/Sales"
foundry dataset read --rid ri.dataset.abc --limit 10
foundry dataset write --path "/my/dataset" --file data.csv
foundry dataset delete --rid ri.dataset.abc

# TRANSFORM COMMANDS
foundry transform build --path "/my/transform"
foundry transform test --path "/my/transform"
foundry transform deploy --path "/my/transform"

# BRANCH COMMANDS
foundry branch list
foundry branch create --name "feature/new-transform"
foundry branch delete --name "old-branch"
foundry cut create --source staging --target main --message "Release v1.0"

# WORKFLOW COMMANDS
foundry workflow list
```

```
foundry workflow run --name "daily-pipeline"
foundry workflow logs --run-id run-123
foundry workflow status --name "daily-pipeline"

# AUTHENTICATION
foundry login
foundry logout
foundry whoami

# PROJECT MANAGEMENT
foundry project create --name "Sales-Analytics"
foundry project list
foundry project info --name "Sales-Analytics"
```

APPENDIX B: COMMON PYSPARK PATTERNS

```python
# 1. READING DATA
df = spark.read.parquet("/path/to/dataset")
df = spark.read.csv("/path/to/csv", header=True, inferSchema=True)
df = spark.read.json("/path/to/json")

# 2. WRITING DATA
(df.write
  .mode("overwrite")  # or "append", "ignore", "error"
  .partitionBy("date")
  .parquet("/output/path"))

# 3. COMMON TRANSFORMATIONS
# Filtering
df = df.filter(F.col("status") == "ACTIVE")

# Adding columns
df = df.withColumn("full_name",
                F.concat(F.col("first_name"),
                      F.lit(" "),
                      F.col("last_name")))

# Aggregations
df_agg = (df.groupBy("department")
        .agg(F.count("*").alias("employee_count"),
            F.avg("salary").alias("avg_salary"))
        .orderBy(F.desc("employee_count")))

# Window functions
from pyspark.sql.window import Window
window_spec = Window.partitionBy("department").orderBy("salary")
```

```
df = df.withColumn("salary_rank", F.row_number().over(window_spec))

# Handling nulls
df = df.fillna({"department": "Unknown", "salary": 0})
df = df.dropna(subset=["employee_id", "email"])

# Type casting
df = df.withColumn("salary", F.col("salary").cast("decimal(10,2)"))
```

APPENDIX C: ERROR MESSAGES & SOLUTIONS

Error Message	Likely Cause	Solution
`Resource not found`	Incorrect RID or path	Verify spelling, check permissions
`Permission denied`	Insufficient folder access	Request access, check parent folder permissions
`Transform build failed`	Syntax error in code	Check Python/Scala syntax, dependencies
`Job timeout`	Job running too long	Increase timeout, optimize code, check for infinite loops
`Out of memory`	Data skew or insufficient memory	Check Spark UI for skew, increase executor memory
`Connection refused`	Network or service issue	Check Foundry status page, verify network connectivity
`Invalid credentials`	Authentication expired	Run `foundry login` to refresh
`Dataset schema mismatch`	Schema changed unexpectedly	Check upstream changes, enforce schema validation
`Partition column not found`	Wrong column name in partitionBy	Verify column exists, check case sensitivity
`Duplicate output dataset`	Multiple writes to same path	Ensure unique output paths, check for race conditions

APPENDIX D: GLOSSARY OF FOUNDRY TERMS

Term	Definition
RID	Resource Identifier - unique ID for everything in Foundry
Dataset	Table-like structure storing data in Foundry
Transform	Code that processes data from inputs to outputs
Workflow	DAG of jobs that run on schedule or trigger
Job	Single unit of work in a workflow
Ontology	Graph-based model of business concepts and relationships
Object Type	Blueprint for business entities in ontology
Link	Connection between dataset column and ontology property
Branch	Isolated workspace for development
Cut	Process of promoting changes between branches
Loader	Configuration for scheduled data ingestion
Connector	Pre-built adapter for external data sources

Contour	UI tool for manual data upload and exploration
Lineage	Tracking of data flow and dependencies
Monitor	Automated check for data quality or pipeline health
Function	Serverless compute for lightweight tasks
Prepare	No-code tool for data transformation
Sync	Process of materializing ontology to queryable datasets

APPENDIX E: CERTIFICATION CHECKLIST

Before Taking Exam:
- [ ] Completed at least 3 full Foundry projects
- [ ] Built production pipeline with error handling
- [ ] Implemented data quality framework
- [ ] Set up monitoring and alerts
- [ ] Practiced ontology modeling
- [ ] Taken 2+ practice exams
- [ ] Reviewed all official documentation
- [ ] Scheduled exam at optimal time

Exam Day Checklist:
- [ ] Government-issued ID ready
- [ ] Workspace cleared (proctoring requirements)
- [ ] Computer fully charged + charger available
- [ ] Internet connection stable
- [ ] Water bottle nearby
- [ ] 15 minutes early for check-in
- [ ] Positive mindset established

FINAL WORDS OF WISDOM

1. Think Like a Product Builder
You're not just building pipelines; you're creating data products. Consider:
- Who are your users?
- What problems do you solve?
- How do you ensure reliability?
- How do you measure success?

2. Embrace Foundry's Philosophy
- **Version everything** - reproducibility is power
- **Model relationships** - data in context is valuable
- **Automate quality** - trust enables speed
- **Collaborate widely** - break down silos

3. Continuous Learning Path
1. **Foundational:** Master the basics (complete)
2. **Advanced:** Deep dive into performance and scale

3. **Expert:** Lead complex implementations
4. **Architect:** Design organization-wide solutions

4. Certification is a Milestone, Not Destination
The exam validates knowledge, but real expertise comes from:
- Building and breaking things
- Learning from failures
- Teaching others
- Staying curious

5. Remember Why This Matters
Every pipeline you build, every dataset you clean, every ontology you design helps someone make better decisions. You're enabling:
- Faster business insights
- More accurate predictions
- Better customer experiences
- Smarter strategic choices

You're not just a data engineer. You're a translator between raw data and business value.

CONTACT & COMMUNITY

Official Resources:
- Palantir Foundry Documentation
- Foundry Community Forums
- Official Training Programs
- Certification Study Guide

Practice Environments:
- Foundry Training Instances
- Community Sandboxes
- Open Datasets for Practice
- Sample Projects Repository

Stay Updated:
- Release Notes (quarterly updates)
- Best Practices Guides
- Case Studies
- User Group Meetings

GOOD LUCK ON YOUR CERTIFICATION JOURNEY!

May your pipelines always run green, your data always be clean, and your ontology always be meaningful.

- The Foundry Architect