The Case for PySpark: Why Architectural Consistency Over Tool Specialization

Executive Summary

This document provides a comprehensive architectural and practical comparison between DBT and Python/PySpark for financial regulatory platforms. While DBT excels as a SQL transformation tool, it fundamentally cannot handle the mathematical requirements of regulatory calculations. When Python/PySpark is already required for complex calculations, using it for all data processing—including simple ETL—provides superior architectural consistency, debugging capabilities, and operational efficiency.

Table of Contents

Architectural Fundamentals

What DBT Really Is

DBT (Data Build Tool) is architecturally a **templating and orchestration layer** that:

```
DBT Architecture:
┌─────────────────┐
│   DBT Project   │
│  (SQL + Jinja)  │
└─────────────────┘
         │
         │ Compile
         ▼
┌─────────────────┐
│   Pure SQL      │
└─────────────────┘
         │
         │ Execute
         ▼
```

```
┌─────────────────┐
│ Data Warehouse  │
│  (SQL Engine)   │
└─────────────────┘
```

Key Limitations: - Constrained by SQL’s computational capabilities - Cannot perform iterative algorithms - No complex mathematical functions beyond basic arithmetic - Cannot implement statistical models or simulations - Limited to what the underlying data warehouse can execute

What Python/PySpark Really Is

Python with PySpark is architecturally a **complete programming environment**:

```
Python/PySpark Architecture:

┌─────────────────────┐
│  Python Application │
│  - Full language    │
│  - Math libraries   │
│  - ML algorithms    │
└─────────────────────┘
           │
           │ Submit
           ▼
┌─────────────────────┐
│   Spark Context     │
│  (Orchestration)    │
└─────────────────────┘
           │
           │ Distribute
           ▼
┌─────────────────────┐
│  Execution Engine   │
│  - Spark Cluster    │
│  - Local Mode       │
│  - Cloud (Glue/EMR)│
└─────────────────────┘
```

Key Capabilities: - Complete programming language with control flow - Access to NumPy, SciPy, and thousands of specialized libraries - Distributed computing across clusters - Multiple execution backends (Spark, Glue, EMR, local) - Used by every major financial institution for risk and regulatory calculations

Industry Reality Check

Major financial institutions’ technology choices tell the story: - **JPMorgan**: Athena platform built on Python - **Goldman Sachs**: GS Quant quantitative platform in Python - **Bank of America**: Quartz risk management in Python - **CitiGroup**: Risk and regulatory calculations in Python

DBT usage in these institutions? Limited to data warehouse transformations and basic reporting—never for regulatory calculations.

The Myth of DBT Simplicity

Simple ETL: LCR Calculation Comparison

Let's examine a "simple" ETL task—calculating the Liquidity Coverage Ratio (LCR). This Basel III requirement involves straightforward transformations but reveals the true complexity of both approaches.

DBT Implementation Structure

```
lcr_calculation/
├── models/
│   ├── staging/
│   │   ├── stg_assets.sql
│   │   └── stg_transactions.sql
│   ├── intermediate/
│   │   ├── int_categorized_assets.sql
│   │   ├── int_cash_flows.sql
│   │   └── int_hqla_aggregated.sql
│   └── marts/
│       └── lcr_final.sql
└── macros/
    ├── categorize_assets.sql
    └── apply_haircuts.sql
```

DBT Asset Categorization (Simplified)

```
-- models/intermediate/int_categorized_assets.sql
{{ config(materialized='incremental') }}

WITH categorized AS (
    SELECT
        *,
        CASE
            WHEN asset_type IN ('CASH', 'CENTRAL_BANK_RESERVES') THEN
'LEVEL_1'
            WHEN asset_type = 'SOVEREIGN_BOND' AND rating >= 'AA-'
THEN 'LEVEL_1'
            WHEN asset_type = 'COVERED_BOND' AND rating >= 'AA-' THEN
'LEVEL_2A'
            WHEN asset_type = 'CORPORATE_BOND' AND rating >= 'BBB-'
THEN 'LEVEL_2B'
            ELSE 'NON_HQLA'
        END AS liquidity_category,

        CASE
            WHEN asset_type IN ('CASH', 'CENTRAL_BANK_RESERVES') THEN
0.0
            WHEN asset_type = 'SOVEREIGN_BOND' AND rating >= 'AA-'
THEN 0.0
```

```
            WHEN asset_type = 'COVERED_BOND' AND rating >= 'AA-' THEN
        0.15
            WHEN asset_type = 'CORPORATE_BOND' AND rating >= 'BBB-'
        THEN 0.50
            ELSE 1.0
        END AS haircut
    FROM {{ ref('stg_assets') }}
)

SELECT
    *,
    market_value * (1 - haircut) AS post_haircut_value,
    CURRENT_TIMESTAMP() AS categorization_timestamp
FROM categorized

-- No error handling possible
-- No debugging capability
-- No dynamic rule loading
```

PySpark Implementation (Same Logic, More Capability)

```
def categorize_assets(self, assets_df):
    """Categorize assets with full debugging and error handling"""

    # Runtime inspection
    self.logger.info(f"Processing {assets_df.count()} assets")

    # Check data quality
    invalid_assets = assets_df.filter(col("rating").isNull() |
        (col("market_value") < 0))
    if invalid_assets.count() > 0:
        self.logger.warning(f"Found {invalid_assets.count()} invalid
        assets")
        if self.debug_mode:
            invalid_assets.show(20)

    # Apply categorization with error handling
    categorized = assets_df \
        .withColumn("liquidity_category",
            when((col("asset_type") == "CASH") |
                 (col("asset_type") == "CENTRAL_BANK_RESERVES"),
        "LEVEL_1")
            .when((col("asset_type") == "SOVEREIGN_BOND") &
                  (col("rating") >= "AA-"), "LEVEL_1")
            .when((col("asset_type") == "COVERED_BOND") &
                  (col("rating") >= "AA-"), "LEVEL_2A")
            .when((col("asset_type") == "CORPORATE_BOND") &
                  (col("rating") >= "BBB-"), "LEVEL_2B")
            .otherwise("NON_HQLA")
        ) \
        .withColumn("haircut",
            when(col("liquidity_category") == "LEVEL_1", 0.0)
            .when(col("liquidity_category") == "LEVEL_2A", 0.15)
            .when(col("liquidity_category") == "LEVEL_2B", 0.50)
            .otherwise(1.0)
        ) \
        .withColumn("post_haircut_value",
            col("market_value") * (1 - col("haircut"))
        )

    # Add comprehensive tracing
    return categorized \
        .withColumn("categorization_timestamp", current_timestamp()) \
        .withColumn("categorization_rule_version",
        lit("BASEL_III_LCR_v1.2")) \
        .withColumn("debug_info",
            struct(
                col("asset_type").alias("original_type"),
                col("rating").alias("original_rating"),
                col("market_value").alias("original_value")
            )
        )
```

The code complexity is similar, but PySpark provides: - Runtime debugging and inspection - Error handling without pipeline failure - Data quality checks - Comprehensive tracing for audit - Ability to load rules dynamically

When Simple Becomes Complex: Production Debugging

Scenario: LCR Suddenly Drops Below Regulatory Threshold

DBT Debugging Approach:

```
-- Must create new model and deploy to production
-- models/debug/lcr_investigation.sql

SELECT
    calculation_date,
    COUNT(*) as asset_count,
    AVG(CASE WHEN liquidity_category = 'LEVEL_1' THEN 1 ELSE 0 END) as
        level1_ratio
FROM {{ ref('int_categorized_assets') }}
WHERE calculation_date >= DATEADD(day, -7, CURRENT_DATE)
GROUP BY calculation_date

-- Deploy, wait, check results
-- Still don't know which specific assets caused the issue
-- Need another deployment cycle to dig deeper
```

PySpark Debugging Approach:

```
def debug_lcr_drop_interactive(self, calculation_date):
    """Interactively debug production issue without deployment"""
```

```
        # Load production data immediately
        assets = self.spark.read.parquet(
            f"regulatory_calculations/lcr/asset_details/calculation_date=
            {calculation_date}"
        )

        # Interactive investigation
        print(f"Total assets: {assets.count()}")

        # Check for rating downgrades
        yesterday = calculation_date - timedelta(days=1)
        assets_yesterday = self.spark.read.parquet(
            f"regulatory_calculations/lcr/asset_details/calculation_date=
            {yesterday}"
        )

        # Find specific changes
        downgrades = assets.alias("today").join(
            assets_yesterday.alias("yesterday"),
            "asset_id"
        ).filter(
            col("today.liquidity_category") !=
            col("yesterday.liquidity_category")
        )

        print(f"Found {downgrades.count()} assets with category changes:")
        downgrades.select(
            "asset_id",
            "yesterday.rating",
            "today.rating",
            "yesterday.liquidity_category",
            "today.liquidity_category",
            "today.market_value"
        ).show()

        # Trace specific asset through entire calculation
        problem_asset = downgrades.first()
        self.trace_complete_calculation(problem_asset.asset_id,
            calculation_date)
```

Result: PySpark identifies root cause in minutes; DBT requires multiple deployment cycles over hours or days.

Real-World Implementation Comparison

Complex Calculations: Credit Valuation Adjustment (CVA)

CVA is a Basel III requirement that demonstrates why Python/PySpark is mandatory for regulatory calculations.

Mathematical Requirements

- Monte Carlo simulation with 10,000+ paths
- Geometric Brownian Motion for interest rate modeling
- Complex derivative pricing at each time step
- Probability of default integration
- Discounting and aggregation

Python/PySpark Implementation

```
class CVACalculator:
    def simulate_interest_rates(self, initial_rate, volatility,
        drift):
        """Monte Carlo simulation using Geometric Brownian Motion"""
        rates = np.zeros((self.num_simulations, self.time_steps))
        rates[:, 0] = initial_rate

        # Generate random shocks
        shocks = np.random.normal(0, 1, (self.num_simulations,
        self.time_steps - 1))

        # Simulate paths - IMPOSSIBLE in SQL
        for t in range(1, self.time_steps):
            rates[:, t] = rates[:, t-1] * np.exp(
                (drift - 0.5 * volatility**2) * self.dt +
                volatility * np.sqrt(self.dt) * shocks[:, t-1]
            )

        return rates

    def calculate_cva(self, swap_portfolio):
        """Calculate CVA with full traceability"""
        results = []

        for swap in swap_portfolio:
            # Simulate rates
            rates = self.simulate_interest_rates(
                swap.initial_rate, swap.volatility, swap.drift
            )

            # Calculate exposures
            exposures = self.calculate_swap_exposures(rates, swap)

            # Apply PD and LGD
            cva = self.integrate_credit_risk(exposures, swap.pd_curve,
        swap.lgd)

            # Full trace for regulators
            results.append({
                'trade_id': swap.trade_id,
                'cva': cva,
                'simulation_paths': rates,  # Complete audit trail
                'exposures': exposures,
                'calculation_timestamp': datetime.now()
            })

        return results
```

DBT "Implementation"

```
-- This is literally impossible in DBT/SQL
-- DBT cannot:
-- 1. Generate random numbers for Monte Carlo
-- 2. Implement iterative algorithms
-- 3. Calculate complex mathematical functions
-- 4. Maintain state across simulations

-- The best DBT can do:
SELECT
    trade_id,
    cva_value,
    calculation_date
FROM pre_calculated_cva_results  -- Calculated outside DBT
WHERE calculation_date = '{{ var("calculation_date") }}'

-- This completely defeats the purpose
```

This isn't a limitation that can be worked around—it's a fundamental architectural constraint.

The Case for Architectural Consistency

When You Already Need Python/PySpark

Since regulatory calculations **require** Python/PySpark, using it for everything provides:

1. Unified Development Practices

```
class UnifiedRegulatoryPlatform:
    """Single platform for all calculations - simple to complex"""

    def __init__(self):
        self.calculation_registry = {
            # Simple ETL
            "LCR": LCRPipeline,
            "NSFR": NSFRPipeline,

            # Medium complexity
            "ECL": ECLCalculator,
            "MARKET_RISK_VAR": VARCalculator,
            # High complexity
            "CVA": CVACalculator,
            "XVA": XVACalculator,
            "ECONOMIC_CAPITAL": EconomicCapitalModel
        }

    def run_any_calculation(self, calc_type, params):
        """Unified interface for all calculations"""
        calculator = self.calculation_registry[calc_type](**params)

        # Common patterns for all calculations
        with self.monitoring() as monitor:
            result = calculator.calculate()
            result = self.add_tracing(result)
            result = self.validate_quality(result)
            self.save_with_lineage(result)

        return result
```

2. Consistent Debugging Across All Calculations

```
class UnifiedDebugging:
    """Same debugging tools for simple and complex calculations"""

    def debug_any_calculation(self, calc_type, issue_date):
        # Load results
        results = self.load_calculation_results(calc_type, issue_date)

        # Common debugging pattern
        print(f"Debugging {calc_type} for {issue_date}")
        print(f"Total records: {results.count()}")

        # Show distribution
        results.groupBy(self.get_key_field(calc_type)).count().show()

        # Find anomalies
        anomalies = self.detect_anomalies(results, calc_type)
        if anomalies.count() > 0:
            print(f"Found {anomalies.count()} anomalies")
            anomalies.show()

        # Trace specific records
        self.trace_calculation_path(anomalies.first(), calc_type)
```

3. Single Testing Framework

```
class UnifiedTesting:
    """Consistent testing for all calculations"""

    def test_calculation(self, calc_type):
        # Same test structure for all
```

```
            test_data = self.generate_test_data(calc_type)

            # Unit tests
            self.run_unit_tests(calc_type, test_data)

            # Integration tests
            self.run_integration_tests(calc_type, test_data)

            # Regression tests
            self.run_regression_tests(calc_type, test_data)

            # Performance tests
            self.run_performance_tests(calc_type, test_data)
```

4. Unified Operations

```
# Single deployment configuration
regulatory_platform:
  image: pyspark-regulatory:latest
  calculations:
    # Simple ETL
    - LCR:
        schedule: "0 6 * * *"
        complexity: "simple"

    # Complex calculations
    - CVA:
        schedule: "0 20 * * *"
        complexity: "complex"

  # Same monitoring for all
  monitoring:
    alerts: enabled
    sla_checks: enabled
    quality_checks: enabled
```

The Hidden Costs of Mixed Architecture

Using DBT for “simple” ETL while using Python/PySpark for complex calculations creates:

1. **Skill Fragmentation**
   - Need both SQL and Python expertise
   - Context switching between paradigms
   - Inconsistent development practices
2. **Operational Complexity**
   - Two deployment pipelines
   - Different monitoring systems
   - Separate error handling patterns
3. **Debugging Nightmares**
   - Different tools for different calculations

- Cannot trace data flow seamlessly
- Inconsistent logging and monitoring

4. **Integration Overhead**
    - Data hand-offs between systems
    - Format conversions
    - Coordination complexities

Practical Implications

For Regulatory Compliance

Row-Level Tracing: A Competitive Advantage - CCAR requires attribute-level lineage - Row-level tracing exceeds requirements - Positions for future regulatory evolution - Essential for AI/LLM validation

PySpark Enables:

```
# Complete audit trail for any calculation
traced_result = calculation_result \
    .withColumn("trace_id", monotonically_increasing_id()) \
    .withColumn("calculation_steps",
        array(
            struct(lit("input").alias("step"),
                   col("input_data").alias("value")),
            struct(lit("categorization").alias("step"),
                   col("category_data").alias("value")),
            struct(lit("aggregation").alias("step"),
                   col("aggregated_data").alias("value")),
            struct(lit("final").alias("step"),
                   col("final_result").alias("value"))
        )
    )
```

For Team Productivity

Single Technology Stack Benefits: - Developers work across all components - Knowledge sharing is natural - Reduced training requirements - Career growth within one technology

For Future AI/LLM Integration

Why PySpark is LLM-Ready:

```
def llm_generated_calculation(self, prompt):
    """Execute LLM-generated calculations safely"""

    # LLM generates Python code
    generated_code = self.llm.generate_calculation(prompt)

    # Validate before execution
```

```
        validation_result = self.validate_generated_code(generated_code)

        if validation_result.is_safe:
            # Execute with monitoring
            result = self.execute_with_sandbox(generated_code)

            # Validate results
            self.validate_calculation_results(result)

            # Provide row-level proof
            return self.add_calculation_proof(result, generated_code)
        else:
            raise ValueError(f"Generated code failed validation:
            {validation_result.issues}")
```

DBT Cannot: - Execute dynamically generated code - Provide row-level validation - Handle complex mathematical operations - Integrate with Python-based LLMs

Conclusion

The choice between DBT and Python/PySpark for regulatory platforms is not about comparing equivalent tools—they serve fundamentally different purposes:

- **DBT**: A SQL templating engine suitable for data warehouse transformations
- **Python/PySpark**: A complete computational platform essential for regulatory calculations

When Python/PySpark is already required for complex calculations (CVA, VaR, ECL, etc.), using it for all data processing provides:

1. **Architectural Consistency**: One technology, one platform, unified practices
2. **Superior Debugging**: Full visibility and control at every step
3. **Operational Efficiency**: Single deployment, monitoring, and maintenance
4. **Future Readiness**: Prepared for AI/LLM integration with row-level validation
5. **Regulatory Compliance**: Exceeds requirements with row-level tracing

The apparent "simplicity" of DBT is actually inflexibility in disguise. In regulated environments where you must: - Explain every calculation - Handle edge cases gracefully - Debug production issues quickly - Adapt to changing regulations - Integrate with AI tools

…Python/PySpark's capabilities aren't optional—they're essential.

The Bottom Line: When you need Python/PySpark for regulatory calculations, using it for everything isn't just convenient—it's architecturally superior. The consistency, debugging capabilities, and operational benefits far outweigh any perceived simplicity from using different tools for different tasks.

In the world of financial regulation, where accuracy, auditability, and agility are paramount, architectural consistency with Python/PySpark provides the robust foundation necessary for current and future requirements.