Parameterization, Implementation & Optimization

Parameterization

1. Centralized Configuration - `PipelineConfig` Class

All pipeline parameters are declared in a Python `dataclass`, enabling consistent and clean access throughout the application.

```
@dataclass
class PipelineConfig:
    aws_access_key: str = None
    aws_secret_key: str = None
    bucket_name: str = "default-bucket"
    s3_directory: str = "sensor_data/"
    local_path: str = "/tmp/data"
    db_credentials_path: str = "/etc/db_creds.json"
    input_files: List[str] = None
    sensor_patterns: List[str] = None
    default_start_date: str = "2024-01-01"
    lookback_days: int = 30
    jdbc_fetch_size: int = 10000
    jdbc_num_partitions: int = 8
    write_mode: str = "overwrite"
    use_aqe: bool = True
    enable_skew_handling: bool = True
```

2. Parameter Sources

Source	Use Case
Airflow Variables	Runtime overrides (e.g., file paths, S3 dirs)
AWS Secrets Manager	Credentials (secure)
Local JSON Config	Developer/test config

3. Parameter Usage

- **DataLoader**: `bucket_name`, `s3_directory`, `input_files`, `local_path`
- **DataProcessor**: `sensor_patterns`
- **DatabaseManager**: `jdbc_fetch_size`, `jdbc_num_partitions`
- **S3Writer**: `write_mode`
- **Spark Session**: `use_aqe`, `enable_skew_handling`

4. Parameter Hierarchy

1. **Secrets Manager** → Highest priority
2. **Airflow Variables** → Mid-level overrides
3. **Local JSON** → Default fallback

5. Validation Checks

Before execution:

```
assert self.config.jdbc_fetch_size > 0
assert self.config.write_mode in ["overwrite", "append"]
assert all(f.endswith('.parquet') for f in self.config.input_files)
```

Section 2: Implementation Guide –

1. Infrastructure Setup

- **EC2 Instance**:
  - Dev: `t3.xlarge`
  - Prod: `r5.2xlarge`

- **IAM Role**: Attach S3 + SecretsManager + Glue permissions

2. Software Installation

```
# Java + Python
sudo yum install java-11-amazon-corretto python3-pip git awscli -y
pip3 install --upgrade pip

# Spark
wget https://dlcdn.apache.org/spark/spark-3.4.1/spark-3.4.1-bin-hadoop3.tgz
tar -xvf spark-3.4.1-bin-hadoop3.tgz
sudo mv spark-3.4.1-bin-hadoop3 /opt/spark
echo 'export SPARK_HOME=/opt/spark' >> ~/.bashrc
echo 'export PATH=$PATH:$SPARK_HOME/bin' >> ~/.bashrc
source ~/.bashrc

# Python packages
pip install pyspark==3.4.1 boto3 pyarrow findspark
```

---

3. Configuration Files

`/etc/db_creds.json`

```
{
  "YourDB": {
    "host": "your-rds-endpoint",
    "dbname": "sensor_db",
    "user": "spark_user",
    "password": "your_password"
  }
}
```

---

4. Deploy and Run

A. Clone & Navigate

```
git clone https://github.com/your-org/sensor-pipeline.git
cd sensor-pipeline
```

B. Execute Locally

```
spark-submit --master local[4] main.py --config-source file --config-path
config/pipeline_config.json
```

C. Execute on EC2

```
nohup spark-submit \
  --master local[*] \
  --executor-memory 4G \
  --driver-memory 2G \
  main.py --config-source aws --secret-name prod/sensor_pipeline \
  > logs/run.log 2>&1 &
```

---

6. Monitoring & Logs

- **Spark UI**: http://<EC2_PUBLIC_IP>:4040

Log tail:

```
 tail -f logs/run.log
```

Enable Spark event logs:

```
 --conf spark.eventLog.enabled=true \
--conf spark.eventLog.dir=s3://your-bucket/spark-logs/
```

---

7. Security Considerations

- Use IAM roles over keys wherever possible
- Restrict security group to known IPs
- Encrypt S3 + Secrets Manager
- Rotate secrets regularly

Recommended and Applied Optimizations

---

1. Adaptive Query Execution (AQE)

Parameter: `use_aqe = True`
Spark Config: .config("spark.sql.adaptive.enabled", self.config.use_aqe)

- Dynamically adjusts the number of shuffle partitions.
- Converts sort-merge joins to broadcast joins where applicable.
- Significantly improves performance on large and skewed datasets.

---

2. Skew Join Handling

Parameter: `enable_skew_handling = True`
Spark Config: .config("spark.sql.adaptive.skewJoin.enabled", self.config.enable_skew_handling)

- Detects and mitigates data skew during shuffle-intensive joins.
- Helps avoid long-running or failed stages due to uneven partition sizes.

---

3. Broadcast Join for Small Tables

Code: df.join(broadcast(tags_df), df.tagid == tags_df.id, "left")

- Broadcasts the smaller `tags_df` to all worker nodes.
- Prevents shuffle joins, reducing network I/O and execution time.

---

4. Repartitioning for File Size Optimization

Code: df.repartition(max(1, df.count() // 100000))

- Dynamically repartitions DataFrame before writing to S3.

- Ensures approximately 100,000 records per file, balancing performance and read efficiency.

---

5. Use of `.persist(StorageLevel.MEMORY_AND_DISK)` for Caching

Code:data[filename] = df.persist(StorageLevel.MEMORY_AND_DISK)

- Caches frequently accessed DataFrames in memory with disk fallback.
- Reduces redundant reads from S3 or local storage.

---

6. Data Coalescing During Reads

Code: if df.rdd.getNumPartitions() > 1:

df = df.coalesce(1)

- Reduces the number of partitions for small input files.
- Avoids unnecessary parallelism that could degrade performance.

---

7. JDBC Parallelism

Parameters:

- `jdbc_fetch_size = 10000`
- `jdbc_num_partitions = 8`

Code:

"fetchSize": str(self.config.jdbc_fetch_size),

"numPartitions": str(self.config.jdbc_num_partitions),

"partitionColumn": "tagid"

- Enables **parallel reading** from PostgreSQL using the `tagid` column.

- Improves throughput and reduces bottlenecks during data ingestion.

---

8. File Size Capping via `maxRecordsPerFile`

Code: .option("maxRecordsPerFile", 100000)

- Prevents oversized Parquet files.
- Optimizes read performance on S3 and improves downstream processing.

---

9. Duplicate Prevention with `dropDuplicates()`

Code: sensor_df.dropDuplicates(["datetime"])

- Ensures idempotency by removing duplicate entries before writing.
- Prevents redundancy in historical sensor records.

---

10. Dynamic Output Partitioning and Unioning

Logic:

- Appends to existing data when `write_mode = "append"`.
- Performs union with existing S3 files to maintain continuity.

---

11. Minimal Selective Projection and Column Casting

Code: df.select([field.name for field in expected_schema])

- Applies schema enforcement and column pruning.
- Improves memory efficiency by avoiding wide transformations.

---

12. Clean Resource Unpersisting & Cache Management

Code: df.unpersist()

```
self.spark.catalog.clearCache()
```

- Ensures memory is released post-pipeline execution.
- Prevents memory leaks during long-running or repeated jobs.
