Contents

1 Spark Task Size Instrumentation

This document explains how to use the instrumentation added to diagnose large Spark task sizes and potential OOM issues.

1.1 Problem

You may see warnings like:

```
WARN TaskSetManager: Stage 76 contains a task of very large size (1159 KiB). The
WARN DAGScheduler: Broadcasting large task binary with size 4.6 MiB
```

These warnings indicate that Spark is serializing large objects to send to workers, which can cause: - Slow task startup times - Network bottlenecks - Out of Memory (OOM) errors - Poor cluster performance

1.2 Common Causes

1. **Large Closures**: Functions passed to `map`, `mapPartitions`, etc. capture large objects from the driver

2. **Deep DataFrame Lineage**: Long chains of transformations create large execution plans
3. **Large Broadcast Variables**: Broadcasting large objects to all workers
4. **Inefficient Aggregations**: Complex aggregation logic captured in closures

1.3 How to Use Instrumentation

1.3.1 Option 1: Use Verbosity Flag (Recommended)

The easiest way is to increase the verbosity level when running predictions:

```
# Run with verbosity=2 to see instrumentation output
python bin/predict_classifier.py --verbosity 2
```

This will automatically: - Measure closure sizes for mapPartitions operations - Track DataFrame lineage depth - Checkpoint DataFrames when lineage gets too deep (>30 stages) - Report metrics at the end of each operation

1.3.2 Option 2: Set in Environment Config

Add to your `env_config`:

```
VERBOSITY=2  # or 3 for even more detail
```

1.3.3 Option 3: Programmatic Use

If using the classifier programmatically:

```
classifier = SkolClassifierV2(
    spark=spark,
    verbosity=2,  # Enable instrumentation
    ...
)
```

1.4 Interpreting the Output

1.4.1 Verbosity Levels

- **0**: Silent (no instrumentation output)
- **1**: Warnings only (closures >1000 KiB, lineage >50 stages, broadcasts >4 MiB)
- **2**: Info (all operations with summary metrics)
- **3**: Debug (detailed step-by-step information)

1.4.2 Example Output

```
================================================================================
CouchDBOutputWriter.save_annotated: Starting
================================================================================
  DataFrame 'predictions_input':
    Columns: 6
    Partitions: 24
    Lineage depth: 42 stages

  Checkpointing 'predictions_before_format' due to deep lineage (42 stages)
  ✓ Cached 'predictions_before_format'

================================================================================
save_distributed: Starting CouchDB save operation
================================================================================
  DataFrame 'save_input':
    Columns: 3
    Partitions: 24
    Lineage depth: 15 stages

  Closure 'save_partition': 2.3 KiB

================================================================================
Instrumentation Metrics Summary
================================================================================

Closure Sizes:
    save_partition: 2.3 KiB

DataFrame Lineage Depths:
  ⚠️ predictions_input: 42 stages, 24 partitions
    predictions_before_format: 5 stages, 24 partitions (after checkpoint)
    save_input: 15 stages, 24 partitions
================================================================================
```

1.4.3 What to Look For

1. **Large Closures (>1000 KiB)**:
   - Indicates captured objects in the function
   - Solution: Use broadcast variables or pass minimal data
2. **Deep Lineage (>30 stages)**:
   - Creates large execution plans
   - Solution: Checkpoint/cache intermediate results
   - *Note: Instrumentation automatically checkpoints at >30 stages when verbosity >= 1*

3. **Many Partitions with Small Data**:
   - Creates overhead
   - Solution: Coalesce partitions

1.5 Automatic Optimizations

When verbosity >= 1, the instrumentation automatically:

1. **Checkpoints deep lineages**: If a DataFrame has >30 stages, it's automatically cached to break the lineage
2. **Reports warnings**: Large closures, broadcasts, and deep lineages are flagged

1.6 Manual Instrumentation

You can also use the instrumentation API directly in your code:

```
from skol_classifier.instrumentation import SparkInstrumentation

# Initialize
instr = SparkInstrumentation(verbosity=2)

# Measure closure size
def my_function(partition):
    # ... your logic ...
    pass

instr.measure_closure_size(my_function, "my_function")

# Analyze DataFrame
instr.analyze_dataframe(df, "my_dataframe", count=False)

# Checkpoint if needed
df = instr.checkpoint_if_needed(df, "my_dataframe", lineage_threshold=30)

# Get summary
print(instr.get_metrics_summary())
```

1.7 Best Practices

1. **Run with verbosity=2 periodically** to check for issues
2. **Watch for warnings** about large closures or deep lineages
3. **Use broadcast variables** for large read-only data shared across tasks
4. **Checkpoint intermediate results** in long pipelines
5. **Keep closures minimal** - only capture what's needed

6. **Coalesce partitions** after filtering to reduce task count

1.8 Troubleshooting Large Task Sizes

1.8.1 If you see large closure warnings:

1. Check what objects are being captured in the closure
2. Move large objects outside the closure
3. Use broadcast variables for shared data
4. Pass only necessary parameters

Example of fixing a large closure:

```
# BAD: Captures entire self object
def save_partition(partition):
    for row in partition:
        self.conn.save(row)  # Captures self!

# GOOD: Only capture what's needed
conn_params = (url, db, user, password)
def save_partition(partition):
    conn = Connection(*conn_params)  # Minimal capture
    for row in partition:
        conn.save(row)
```

1.8.2 If you see deep lineage warnings:

1. The instrumentation will auto-checkpoint at verbosity >= 1
2. Manually add `.cache()` or `.checkpoint()` at logical points
3. Consider materializing intermediate results

1.8.3 If you see OOM errors:

1. Increase verbosity to see what's large
2. Reduce partition count if you have many small partitions
3. Increase executor memory
4. Break pipeline into smaller stages with checkpoints

1.9 Performance Impact

The instrumentation has minimal overhead: - Verbosity 0: No overhead - Verbosity 1: <1% overhead (warnings only) - Verbosity 2: ~2-5% overhead (closure measurement + lineage checking) - Verbosity 3: ~5-10% overhead (detailed logging)

The automatic checkpointing at verbosity >= 1 can actually *improve* performance by breaking deep lineages.

1.10 Related Spark Settings

You may also want to tune these Spark settings:

```
spark = SparkSession.builder \
    .config("spark.driver.maxResultSize", "4g") \
    .config("spark.executor.memory", "8g") \
    .config("spark.driver.memory", "8g") \
    .config("spark.sql.shuffle.partitions", "200") \
    .getOrCreate()
```

1.11 Further Reading

- Spark Programming Guide - Broadcasting
- Spark SQL Guide - Performance Tuning
- DataFrame Checkpointing