# **Advanced Spark Programming [Expert]: Advanced Pillars**

## **📚 Sumário**

1. [Visão Geral](https://claude.ai/chat/5d63c15e-e1ad-4bee-a1a2-5df3aba0a8fd#vis%C3%A3o-geral)
2. [Arquitetura do Spark](https://claude.ai/chat/5d63c15e-e1ad-4bee-a1a2-5df3aba0a8fd#arquitetura-do-spark)
3. [Componentes Internos](https://claude.ai/chat/5d63c15e-e1ad-4bee-a1a2-5df3aba0a8fd#componentes-internos)
4. [Partições e Paralelismo](https://claude.ai/chat/5d63c15e-e1ad-4bee-a1a2-5df3aba0a8fd#parti%C3%A7%C3%B5es-e-paralelismo)
5. [Alocação de Recursos](https://claude.ai/chat/5d63c15e-e1ad-4bee-a1a2-5df3aba0a8fd#aloca%C3%A7%C3%A3o-de-recursos)
6. [Modelo de Execução](https://claude.ai/chat/5d63c15e-e1ad-4bee-a1a2-5df3aba0a8fd#modelo-de-execu%C3%A7%C3%A3o)
7. [Otimizações Avançadas](https://claude.ai/chat/5d63c15e-e1ad-4bee-a1a2-5df3aba0a8fd#otimiza%C3%A7%C3%B5es-avan%C3%A7adas)
8. [Casos Práticos](https://claude.ai/chat/5d63c15e-e1ad-4bee-a1a2-5df3aba0a8fd#casos-pr%C3%A1ticos)

## **🎯 Visão Geral**

### **Objetivos da Aula**

* Compreender os fundamentos internos do Apache Spark
* Dominar conceitos de particionamento e paralelismo
* Aprender técnicas avançadas de otimização
* Entender o funcionamento do Catalyst Optimizer
* Aplicar conhecimentos em casos práticos

### **Metodologia**

O treinamento utiliza **storytelling** com conteúdo progressivo, conectando teoria e prática através de demonstrações e casos reais.

## 

## 

## **🏗️ Arquitetura do Spark**

### **Componentes Principais**

#### **1. Driver**

* Recebe e analisa aplicações submetidas
* Cria planos de execução lógicos e físicos
* Gerencia o ciclo de vida da aplicação
* Comunica-se com o Cluster Manager
* Mantém a SparkSession

#### **2. Executores**

* Executam tarefas atribuídas pelo Driver
* Armazenam dados em memória/disco
* Reportam status ao Driver
* Cada executor possui sua própria JVM

#### **3. Cluster Manager**

* Gerencia recursos do cluster
* Aloca executores conforme necessário
* Monitora saúde dos nós
* Tipos disponíveis:
  + **Standalone**: Gerenciador nativo do Spark
  + **YARN**: Integração com ecossistema Hadoop
  + **Mesos**: Gerenciador de recursos genérico
  + **Kubernetes**: Orquestração de containers

### **Fluxo de Execução**

Cliente → Driver → Cluster Manager → Executores

1. Cliente submete aplicação ao Driver
2. Driver analisa e cria plano de execução
3. Driver solicita recursos ao Cluster Manager
4. Cluster Manager aloca executores
5. Driver envia tarefas aos executores
6. Executores processam e retornam resultados

## **⚙️ Componentes Internos**

### **DAG Scheduler**

* Converte operações em Directed Acyclic Graph (DAG)
* Divide trabalho em **stages**
* Identifica dependências entre stages
* Otimiza pipeline de execução

### **Task Scheduler**

* Agenda tarefas para execução
* Distribui tarefas entre executores
* Monitora progresso das tarefas
* Gerencia reexecução em caso de falha

### **Conceitos Fundamentais**

#### **Jobs**

* Unidade de trabalho criada por **ações** (count, save, collect)
* Contém múltiplos stages

#### **Stages**

* Conjunto de tarefas que podem executar em paralelo
* Delimitados por operações de **shuffle**
* Executam transformações narrow

#### **Tasks**

* Menor unidade de execução
* Processa uma partição por vez
* Executa em slots (cores) dos executores

### **Hierarquia de Execução**

Job → Stages → Tasks → Partições

## **📊 Partições e Paralelismo**

### **Conceito Fundamental**

**Partição é a unidade principal de paralelismo no Spark**

### **Regras Importantes**

1. **Número de Partições = Número de Tasks**
2. **Uma task processa uma partição por vez**
3. **Partições ideais: 3-4x número de cores**

### **Configuração de Partições**

#### **Max Partition Bytes**

spark.conf.set("spark.sql.files.maxPartitionBytes", "134217728") # 128MB padrão

### **Cálculo de Partições**

Para um arquivo de 374.6 MB com max partition de 128 MB:

374.6 MB ÷ 128 MB = ~3 partições (teórico)

**Importante**: O Spark pode criar mais partições baseado em:

* Metadados do arquivo Parquet
* Estrutura de row groups
* Estatísticas disponíveis

### **Estratégias de Otimização**

#### **1. Ajuste do Max Partition Bytes**

✅ **Quando usar:**

* Tamanho do dataset é conhecido
* Processamento de backfill
* Camada Silver com arquivos padronizados

# Exemplo: Para 24 partições desejadas com arquivo de 374.6 MB

ideal\_partition\_size = 374.6 / 24 # ~15.6 MB

spark.conf.set("spark.sql.files.maxPartitionBytes", str(int(15.6 \* 1024 \* 1024)))

#### **2. Repartition**

✅ **Quando usar:**

* Tamanho do dataset é variável
* Necessário redistribuir dados
* Após operações que alteram distribuição

df = df.repartition(24) # Force 24 partições

⚠️ **Importante**: Repartition sempre causa shuffle (operação cara)

### **Thread Pool**

* Gerencia tarefas aguardando execução
* Mantém fila de 3-4x tarefas por core
* Por executor (não global)

## **💻 Alocação de Recursos**

### **Configurações Recomendadas**

#### **Configuração Inicial Padrão**

* **3 nós** com **4 cores** e **16 GB RAM** cada
* Total: 12 cores, 48 GB RAM
* Suporta ~48 tarefas simultâneas (4x cores)

### **Modos de Escalonamento**

#### **Horizontal (Scale-out)**

✅ **Vantagens:**

* Maior paralelismo
* Melhor distribuição de dados
* Tolerância a falhas

❌ **Desvantagens:**

* Maior overhead de rede
* Mais complexidade de gerenciamento

#### **Vertical (Scale-up)**

✅ **Vantagens:**

* Menos overhead de rede
* Operações locais mais rápidas

❌ **Desvantagens:**

* Menor paralelismo
* Limite físico de recursos

### **Cálculo de Recursos - Caso Real**

**Cenário**: 100 GB/dia, 30 dias retention, SLA 1 hora

# Dados

total\_size = 100 GB \* 30 = 3 TB

daily\_size = 100 GB

partition\_size = 128 MB

# Cálculo

partitions\_per\_day = 100 GB / 128 MB = 781 partições

cores\_needed = 781 tasks / 60 minutes = 13 cores

# Configuração Final

4 executores × 4 cores = 16 cores (margem de segurança)

4 executores × 16 GB RAM = 64 GB RAM total

### **Dynamic Allocation**

spark.conf.set("spark.dynamicAllocation.enabled", "true")

spark.conf.set("spark.dynamicAllocation.minExecutors", "3")

spark.conf.set("spark.dynamicAllocation.maxExecutors", "5")

## **🔄 Modelo de Execução**

### **Modos de Execução**

#### **Client Mode**

* Driver executa no cliente
* Executores no cluster
* Uso: desenvolvimento e testes

#### **Cluster Mode**

* Driver e executores no cluster
* Tolerância a falhas
* Uso: produção

### **Scheduling Modes**

#### **FIFO (First In, First Out)**

* Execução sequencial de jobs
* Simples e previsível
* Pode causar gargalos com jobs grandes

#### **FAIR**

* Compartilhamento de recursos
* Usa pools de execução
* Melhor para workloads mistos

### **Cluster Managers em Produção**

| **Serviço** | **Cluster Manager** | **Scheduling Mode** |
| --- | --- | --- |
| Databricks | Standalone | FAIR |
| Azure HDInsight | YARN | FIFO |
| Google Dataproc | YARN/K8s | FIFO/FAIR |
| Amazon EMR | YARN | FAIR |

## **🚀 Otimizações Avançadas**

### **Catalyst Optimizer**

#### **Pipeline de Otimização**

1. **Parsed Logical Plan**: Análise sintática
2. **Analyzed Logical Plan**: Validação e binding
3. **Optimized Logical Plan**: Aplicação de regras
4. **Physical Plans**: Múltiplos planos físicos
5. **Selected Physical Plan**: Melhor plano baseado em custo
6. **RDD Execution**: Código executável

### **Whole Stage Code Generation**

Agrupa múltiplas operações em único bytecode JVM:

Sem Code Gen: Filter → Map → Aggregate (3 chamadas JVM)

Com Code Gen: WholeStageCodegen(Filter+Map+Aggregate) (1 chamada JVM)

**Benefícios:**

* Reduz overhead de chamadas JVM
* Melhora cache locality
* Otimiza uso de CPU

### **Adaptive Query Execution (AQE)**

Otimizações em tempo de execução:

1. **Dynamic Partition Coalescing**
   * Reduz partições pequenas automaticamente
2. **Dynamic Join Strategy**
   * Muda de SortMergeJoin para BroadcastHashJoin
3. **Dynamic Skew Join Optimization**
   * Detecta e corrige data skew
4. **Dynamic Filter Pushdown**
   * Otimiza filtros baseado em estatísticas runtime

### **Projeto Tungsten**

Otimizações de CPU e memória:

* Gerenciamento manual de memória
* Cache-aware computation
* Code generation
* Eliminação de overhead da JVM

## **📝 Casos Práticos**

### **Caso 1: Otimização com Max Partition Bytes**

**Problema**: Arquivo de 374.6 MB processando lentamente

**Solução Original (com repartition)**:

df = spark.read.parquet("file.parquet") # 6 partições

df = df.repartition(24) # Força 24 partições

# Tempo: 1.4 minutos (com shuffle)

**Solução Otimizada**:

spark.conf.set("spark.sql.files.maxPartitionBytes", str(17 \* 1024 \* 1024))

df = spark.read.parquet("file.parquet") # 22 partições automáticas

# Tempo: 3 segundos (sem shuffle)

**Ganho**: 93% de redução no tempo!

### **Caso 2: Broadcast Join Optimization**

**Problema**: Join entre tabelas de 3.5 GB e 100 MB

**Sem Broadcast**:

result = df\_large.join(df\_small, "id")

# Usa SortMergeJoin (lento, com shuffle)

**Com Broadcast**:

from pyspark.sql.functions import broadcast

result = df\_large.join(broadcast(df\_small), "id")

# Usa BroadcastHashJoin (rápido, sem shuffle)

### **Caso 3: Processamento em Larga Escala**

**Cenário**: 2.7 TB de dados para processar

**Estratégia**:

1. Análise dos arquivos: média de 50 MB cada
2. Ajuste do max partition bytes para 50 MB
3. Cálculo de recursos necessários
4. Configuração de cluster adequada

**Resultado**: Redução de 45 minutos no tempo total (de 2h para 1h15min)

## **🎓 Principais Aprendizados**

### **Conceitos-Chave**

1. **Partições são fundamentais** - Dominar particionamento é essencial para performance
2. **Shuffle é caro** - Sempre ocorre entre stages, evite quando possível
3. **Tasks = Partições** - Relação 1:1 sempre
4. **3-4x regra** - Partições ideais = 3-4x número de cores
5. **RDD vs DataFrame** - DataFrame sempre mais rápido devido ao Catalyst
6. **Broadcast quando possível** - Para tabelas pequenas (<100MB em produção)
7. **AQE ajuda mas não resolve tudo** - Otimizações manuais ainda necessárias

### **Dicas de Troubleshooting**

1. **Identificar Stragglers**: Tasks que demoram muito mais que a média
2. **Verificar Shuffle**: Procurar por Exchange no plano de execução
3. **Monitorar Spill**: Dados indo para disco indica falta de memória
4. **Analisar Skew**: Distribuição desigual de dados entre partições
5. **Revisar Storage**: Formato e particionamento dos dados fonte

### **Ferramentas de Análise**

* **Spark UI**: Jobs, Stages, Tasks, Executors
* **Event Timeline**: Visualização temporal de execução
* **DAG Visualization**: Fluxo de transformações
* **SQL Tab**: Planos de execução detalhados
* **Explain Plan**: Análise de query optimization

## **📚 Referências e Próximos Passos**

### **Tópicos do Próximo Módulo**

* Antipatterns e Patterns
* Técnicas avançadas de debugging
* Otimizações específicas por tipo de operação
* Casos complexos de produção

### **Comandos Úteis**

# Verificar partições

df.rdd.getNumPartitions()

# Ver plano de execução

df.explain(True)

# Configurações importantes

spark.conf.set("spark.sql.adaptive.enabled", "true")

spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")

spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")

*Documento gerado com base na aula ministrada por Luan Moreno - Engenharia de Dados Academy*