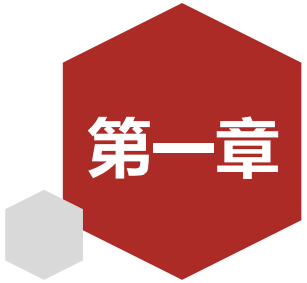
Spark新特性+核心回顾

学习目标
Learning Objectives

1. 掌握Spark的Shuffle流程

2. 掌握Spark3.0新特性

3. 理解并复习Spark的核心概念



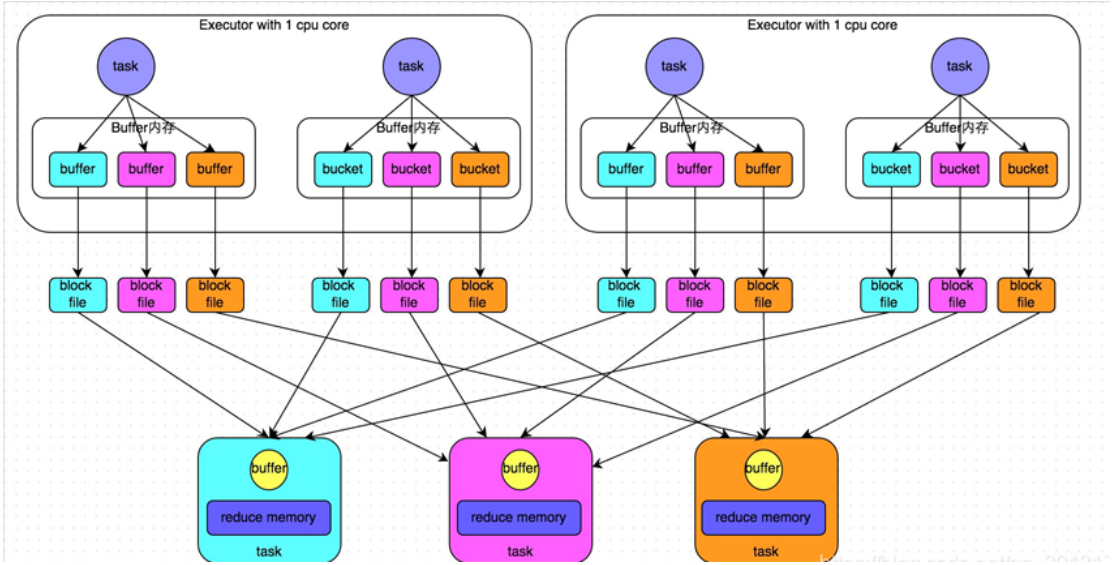
Spark Shuffle

1.1 Spark Shuffle

Map 和 Reduce

在Shuffle过程中. 提供数据的称之为Map端(Shuffle Write) 接收数据的 称之为 Reduce端(Shuffle Read)

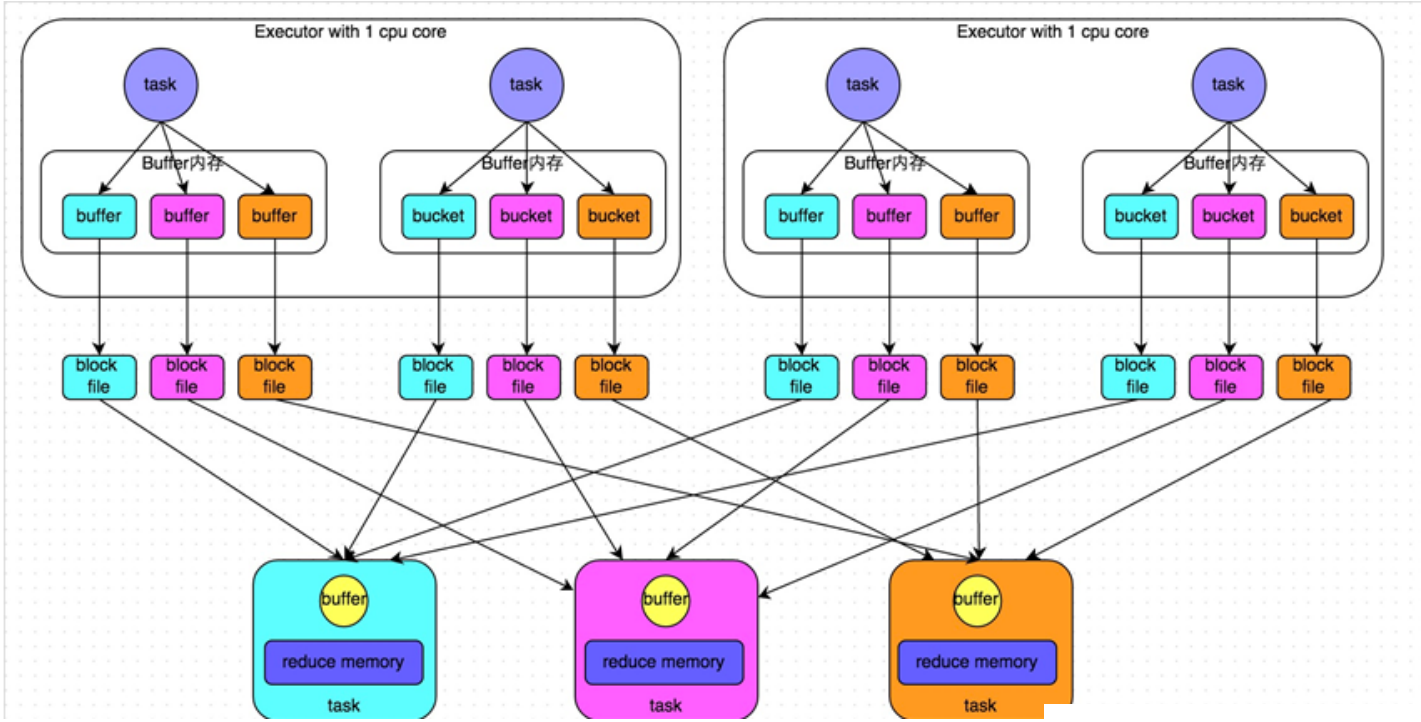
在Spark的两个阶段中, 总是前一个阶段产生 一批Map提供数据, 下一阶段产生一批Reduce接收数据



1.2 HashShuffleManager

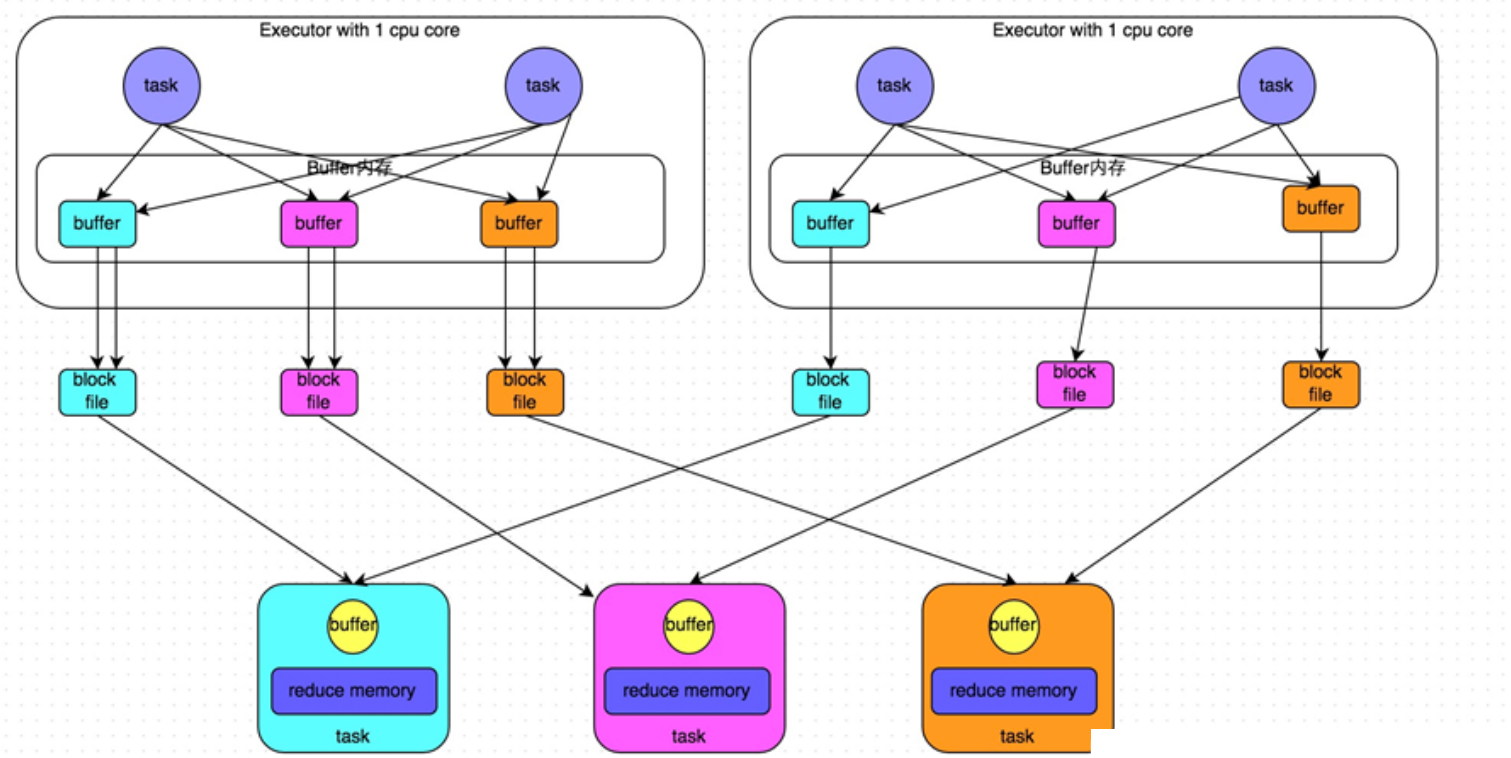
Spark 提供2种Shuffle管理器:

- HashShuffleManager
- SortShuffleManager



未经优化的HashShuffleManager

1.2 HashShuffleManager

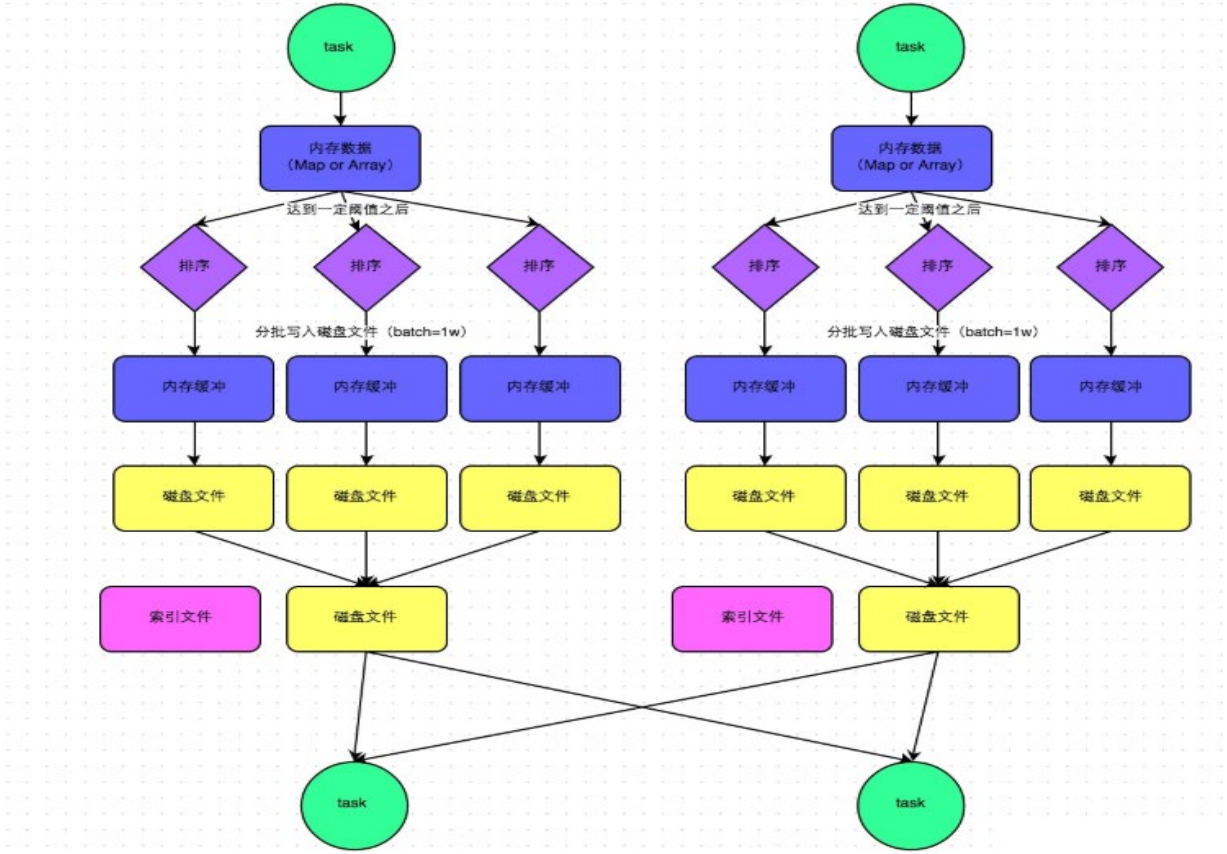


优化后HashShuffleManager

基本和未优化的一致,不同点在于
1. 在一个Executor内, 不同Task是共享Buffer缓冲区
2. 这样减少了缓冲区乃至写入磁盘文件的数量, 提高性能

1.3 SortShuffleManager

SortShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是bypass运行机制。

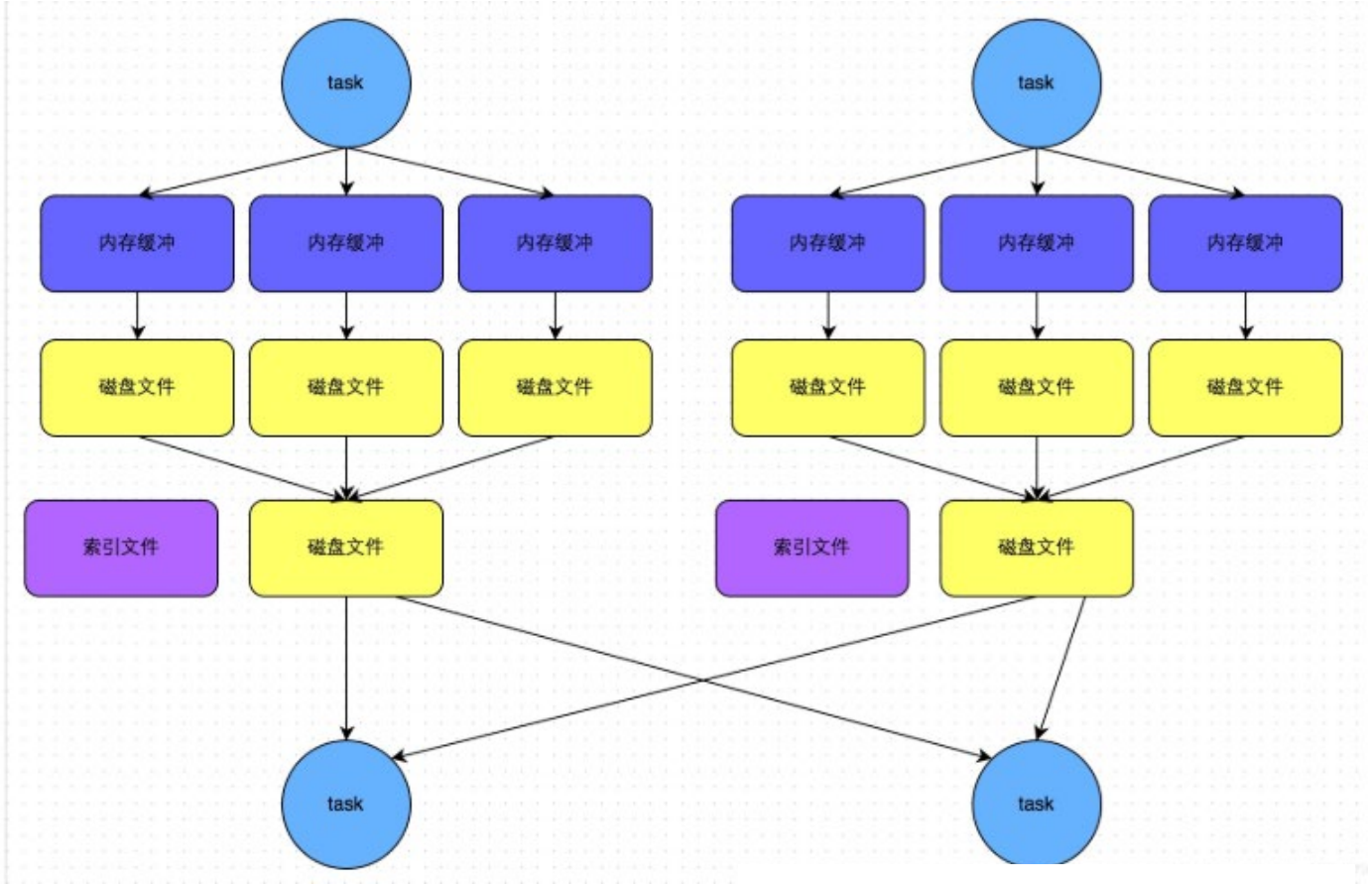


普通机制的SortShuffleManager

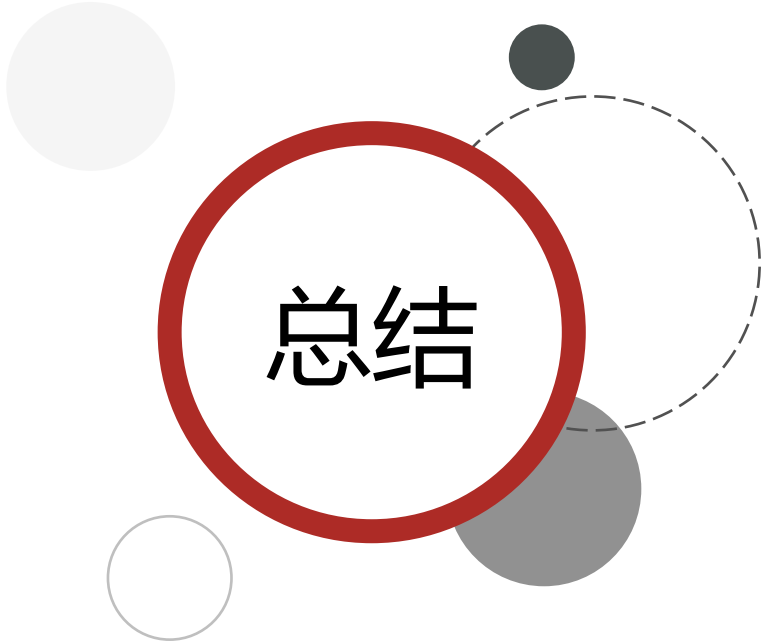
1.3 SortShuffleManager

bypass运行机制的触发条件如下：
1)shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold=200参数的值。
2)不是聚合类的shuffle算子(比如reduceByKey)。

同普通机制基本类同，区别在于，写入磁盘临时文件的时候不会在内存中进行排序
而是直接写，最终合并为一个task一个最终文件
所以和普通模式IDE区别在于:
第一，磁盘写机制不同;
第二，不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销



ByPass机制的SortShuffleManager



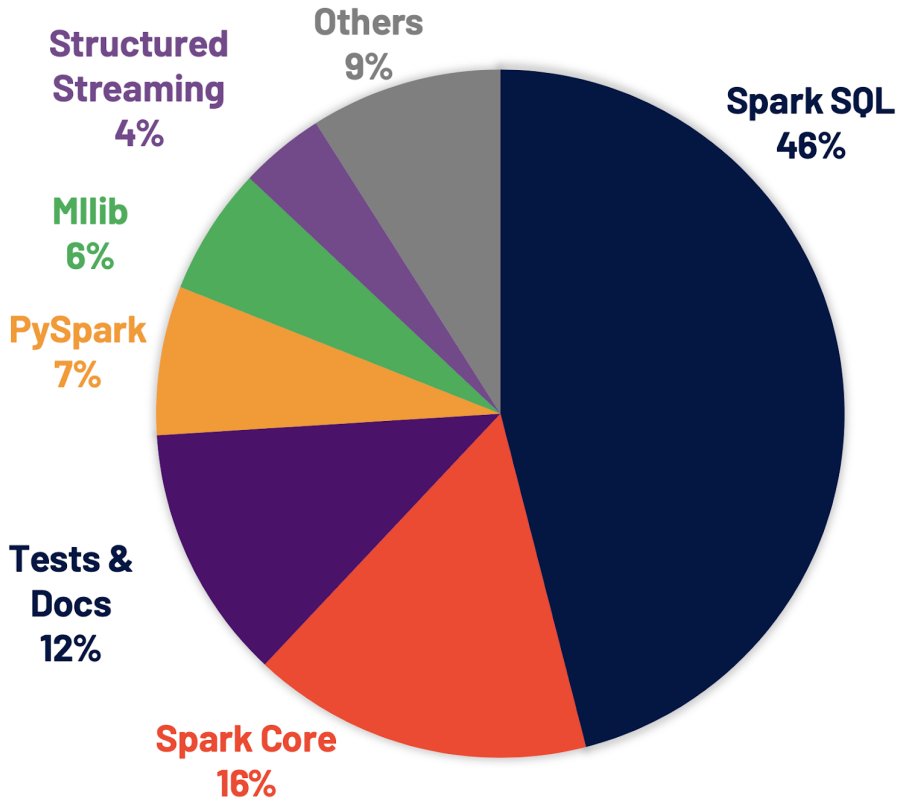
1. SortShuffle对比HashShuffle可以减少很多的磁盘文件,以节省网络IO的开销
2. SortShuffle主要是对磁盘文件进行合并来进行文件数量的减少, 同时两类Shuffle都需要经过内存缓冲区溢写磁盘的场景. 所以可以得知, 尽管Spark是内存迭代计算框架, 但是内存迭代主要在窄依赖中. 在宽依赖(Shuffle)中磁盘交互还是一个无可避免的情况. 所以, 我们要尽量减少Shuffle的出现, 不要进行无意义的Shuffle计算.

Spark 3.0 新特性

2.1 3.0新特性概览

Here are the biggest new features in Spark 3.0:

- 2x performance improvement on TPC-DS over Spark 2.4, enabled by **adaptive query execution**, **dynamic partition pruning** and other optimizations
- ANSI SQL compliance
- Significant **improvements in pandas APIs**, including Python type hints and additional pandas UDFs
- Better Python error handling, simplifying PySpark exceptions
- **New UI** for structured streaming
- **Up to 40x speedups** for calling R user-defined functions
- Over 3,400 Jira tickets resolved

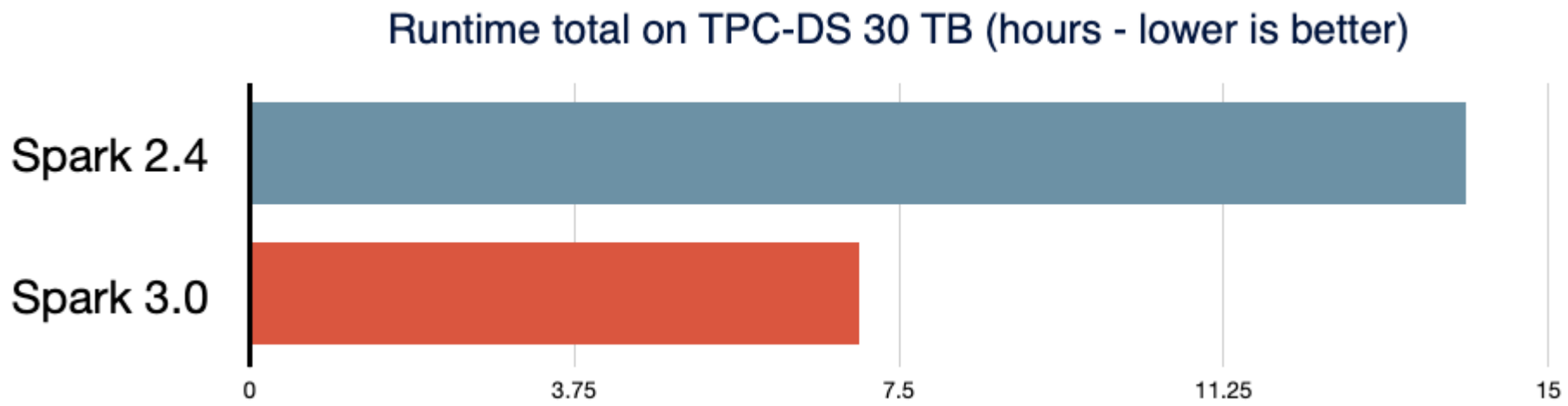


3.0新特性更新的模块占比

2.1 3.0新特性概览

对比2.4版本, 3.0在TPC-DS基准测试中

性能超过2.4版本, 达到了2倍的提升



30TB基准数据量计算测试

2.2 Adaptive Query Execution 自适应查询(SparkSQL)

由于缺乏或者不准确的数据统计信息(元数据)和对成本的错误估算(执行计划调度)导致生成的初始执行计划不理想

在Spark3.x版本提供Adaptive Query Execution自适应查询技术

通过在”运行时”对查询执行计划进行优化，允许Planner在运行时执行可选计划,这些可选计划将会基于运行时数据统计进行动态优化, 从而提高性能.

Adaptive Query Execution AQE主要提供了三个自适应优化:

- 动态合并 Shuffle Partitions
- 动态调整Join策略
- 动态优化倾斜Join(Skew Joins)

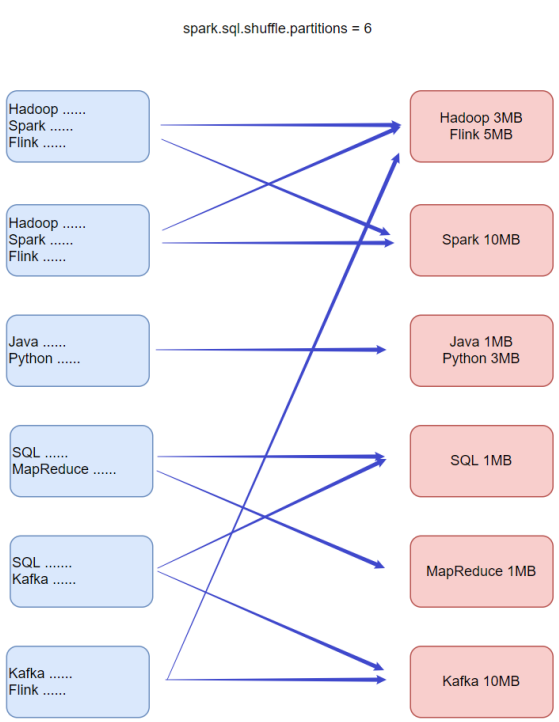
开启AQE方式

```
set spark.sql.adaptive.enabled = true;
```

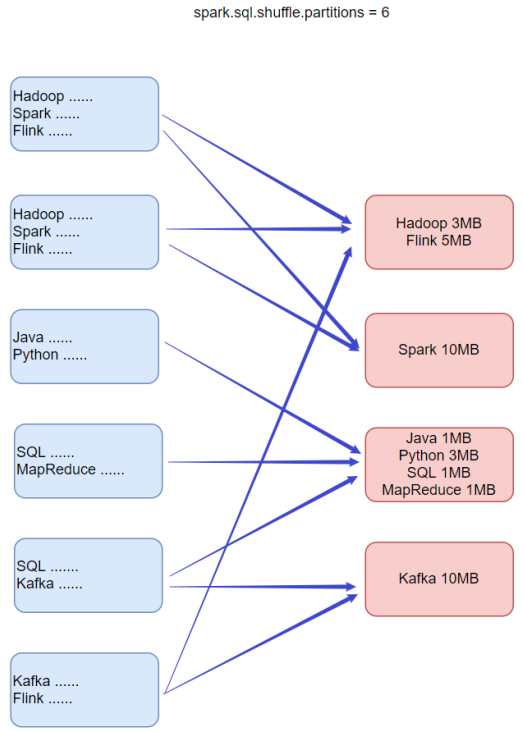
2.2 Adaptive Query Execution 自适应查询(SparkSQL)

- 动态合并 Dynamically coalescing shuffle partitions

可以动态调整shuffle分区的数量。用户可以在开始时设置相对较多的shuffle分区数，AQE会在运行时将相邻的小分区合并为较大的分区。



AQE OFF

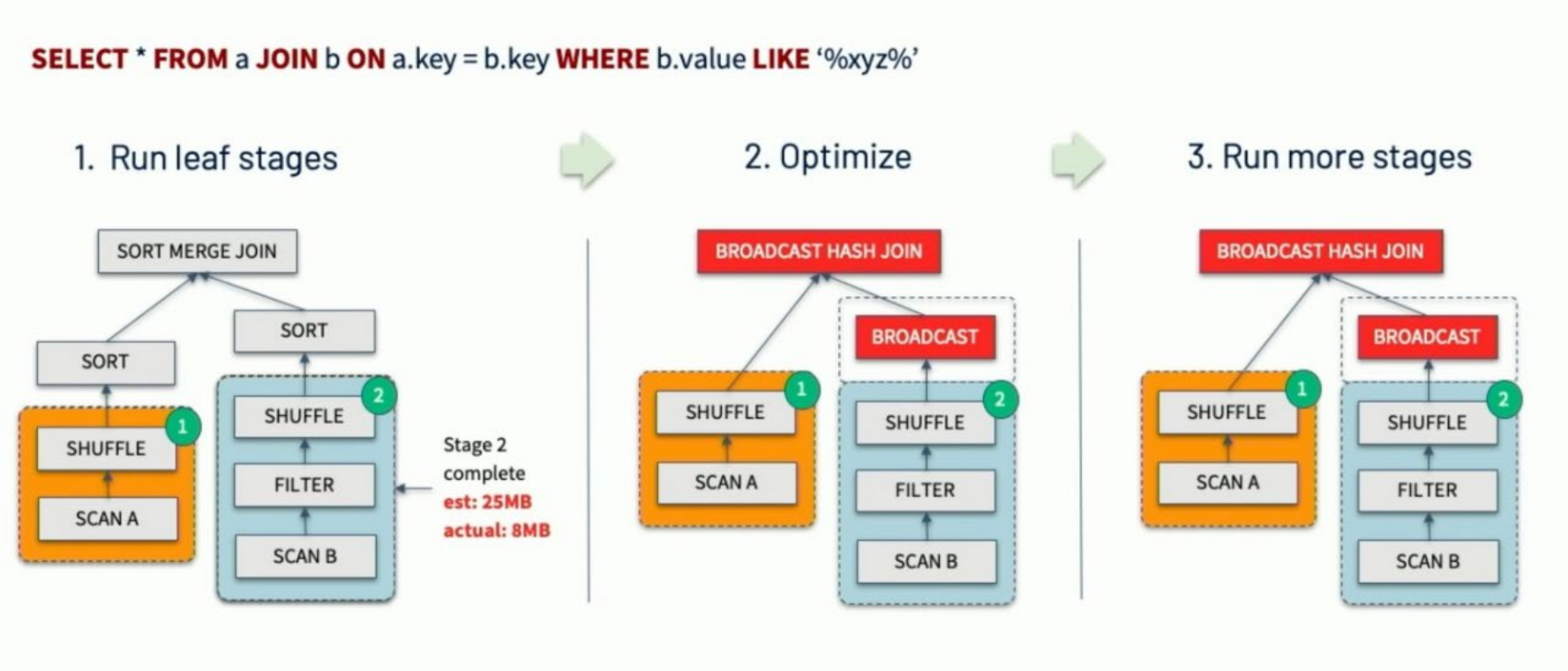


AQE ON

2.2 Adaptive Query Execution 自适应查询(SparkSQL)

- 动态调整Join策略 Dynamically switching join strategies

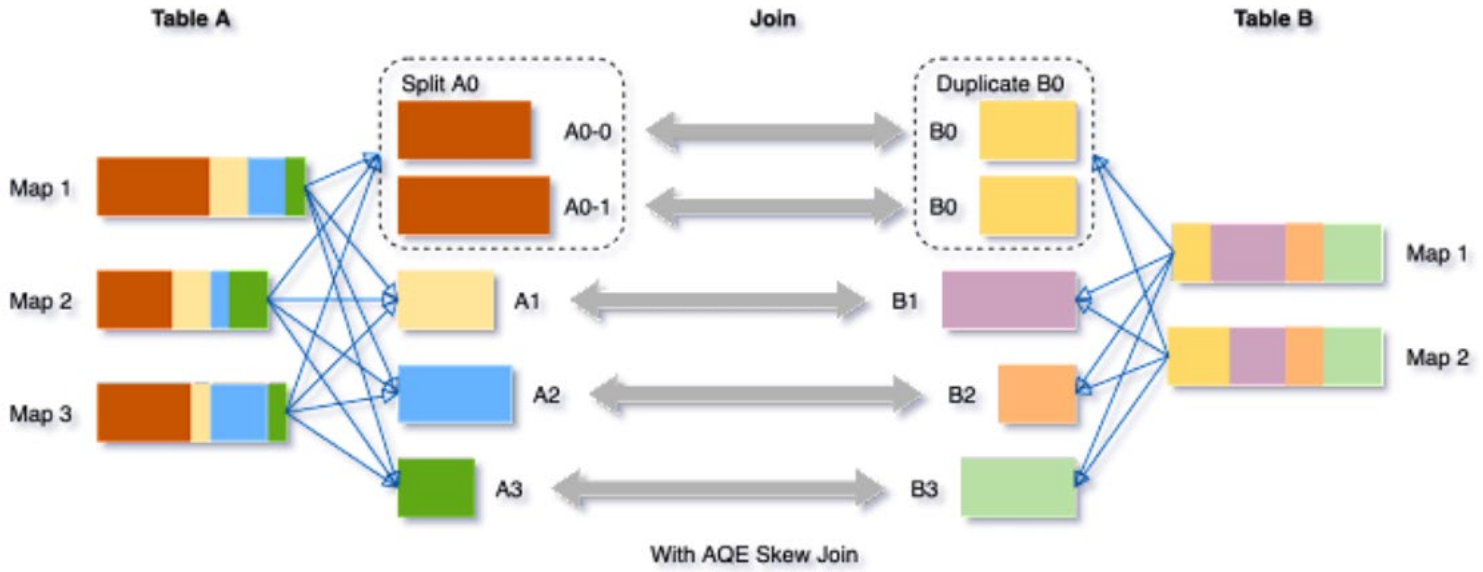
此优化可以在一定程度上避免由于缺少统计信息或着错误估计大小（当然也可能两种情况同时存在），而导致执行计划性能不佳的情况。这种自适应优化可以在运行时sort merge join转换成broadcast hash join，从而进一步提升性能



2.2 Adaptive Query Execution 自适应查询(SparkSQL)

- 动态优化倾斜Join

skew joins可能导致负载的极端不平衡，并严重降低性能。在AQE从shuffle文件统计信息中检测到任何倾斜后，它可以将倾斜的分区分割成更小的分区，并将它们与另一侧的相应分区连接起来。这种优化可以并行化倾斜处理，获得更好的整体性能。

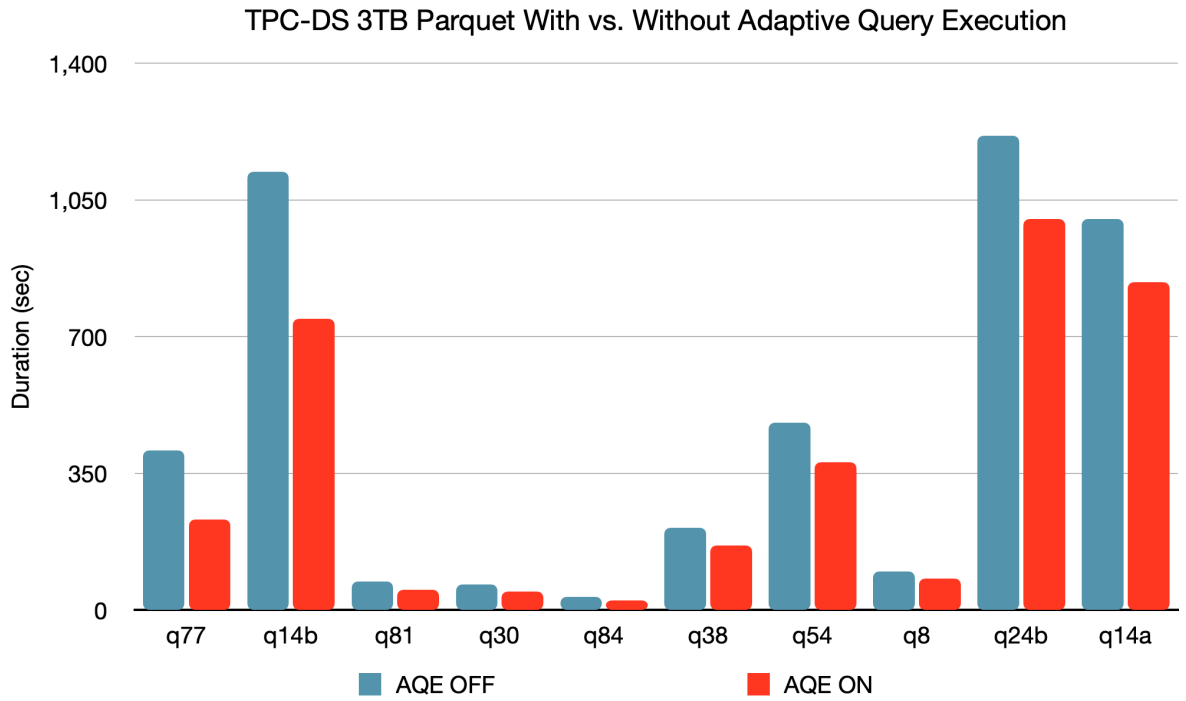


触发条件:
1. 分区大小 > spark.sql.adaptive.skewJoin.skewedPartitionFactor (default=10) * "median partition size(中位数分区大小)"
2. 分区大小 > spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes (default = 256MB)

2.2 Adaptive Query Execution 自适应查询(SparkSQL)

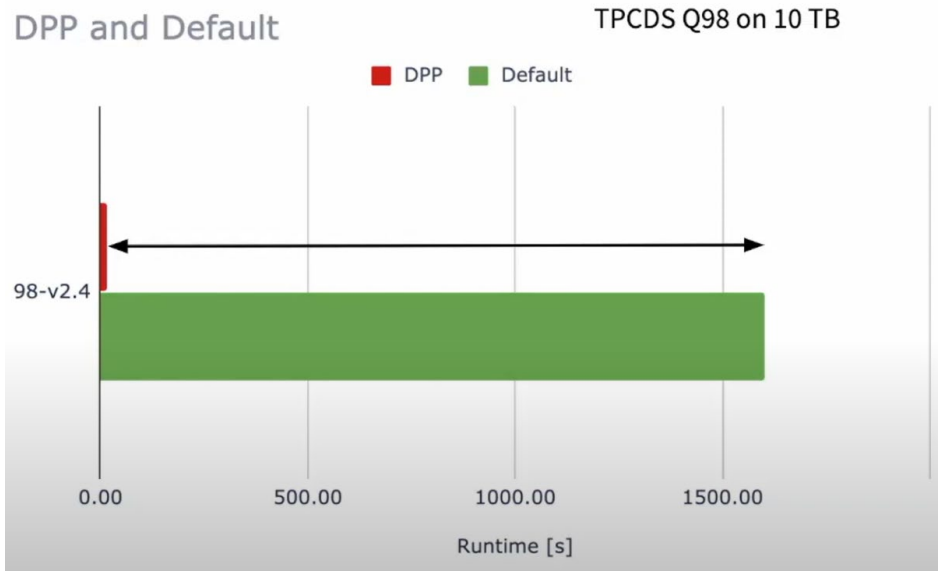
AQE 总结:

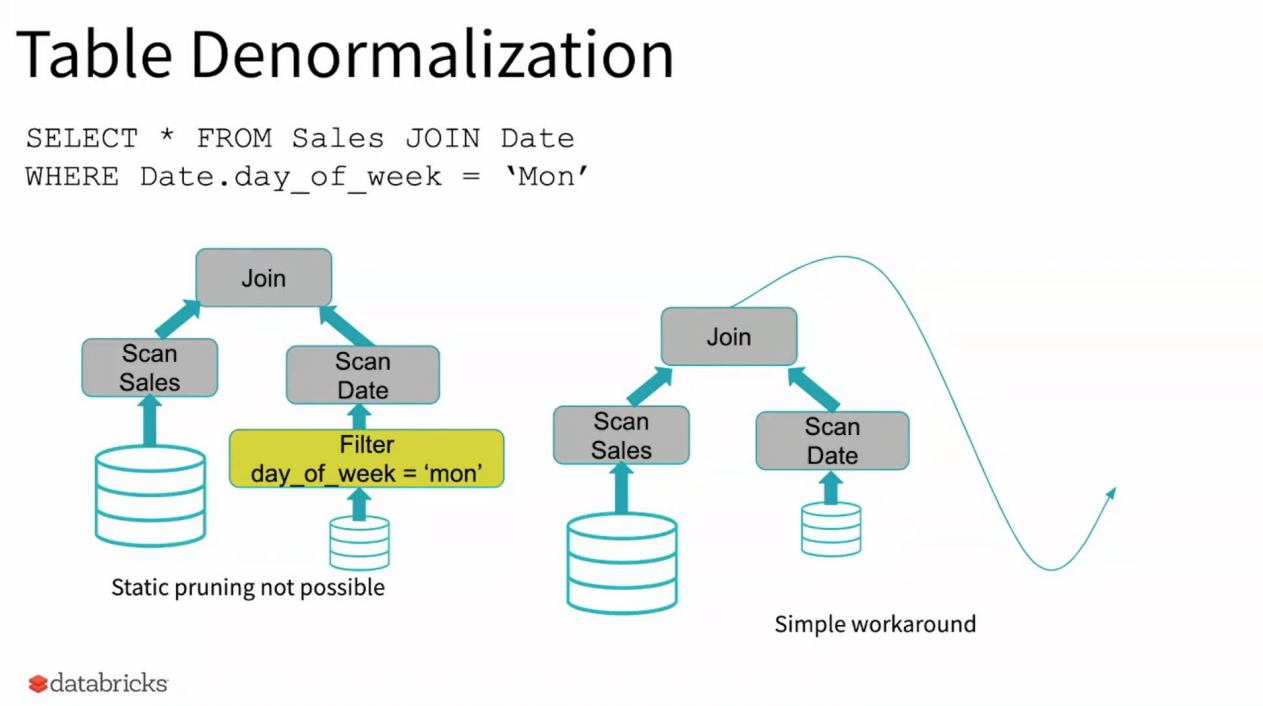
1. AQE的开启通过: spark.sql.adaptive.enabled 设置为true开启
2. AQE是自动化优化机制, 无需我们设置复杂的参数调整, 开启AQE符合条件即可自动化应用AQE优化
3. AQE带来了极大的SparkSQL性能提升



2.3 Dynamic Partition Pruning 动态分区裁剪(SparkSQL)

当优化器在编译时无法识别可跳过的分区时，可以使用"动态分区裁剪"，即基于运行时推断的信息来进一步进行分区裁剪。这在星型模型中很常见，星型模型是由一个或多个并且引用了任意数量的维度表的事实表组成。在这种连接操作中，我们可以通过识别维度表过滤之后的分区来裁剪从事实表中读取的分区。在一个TPC-DS基准测试中，102个查询中有60个查询获得2到18倍的速度提升。



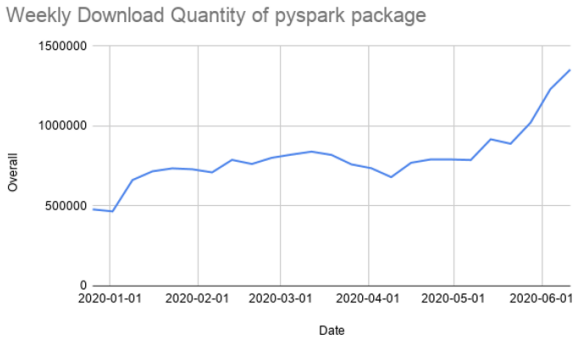


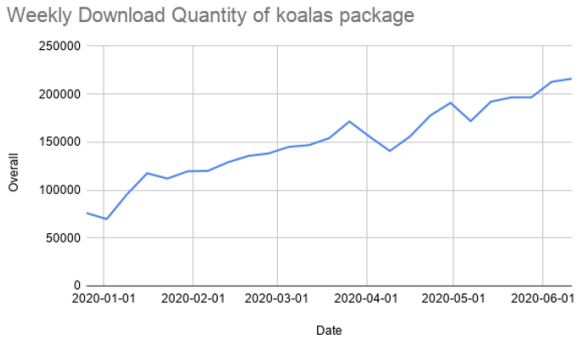
2.4 增强的Python API: PySpark和Koalas

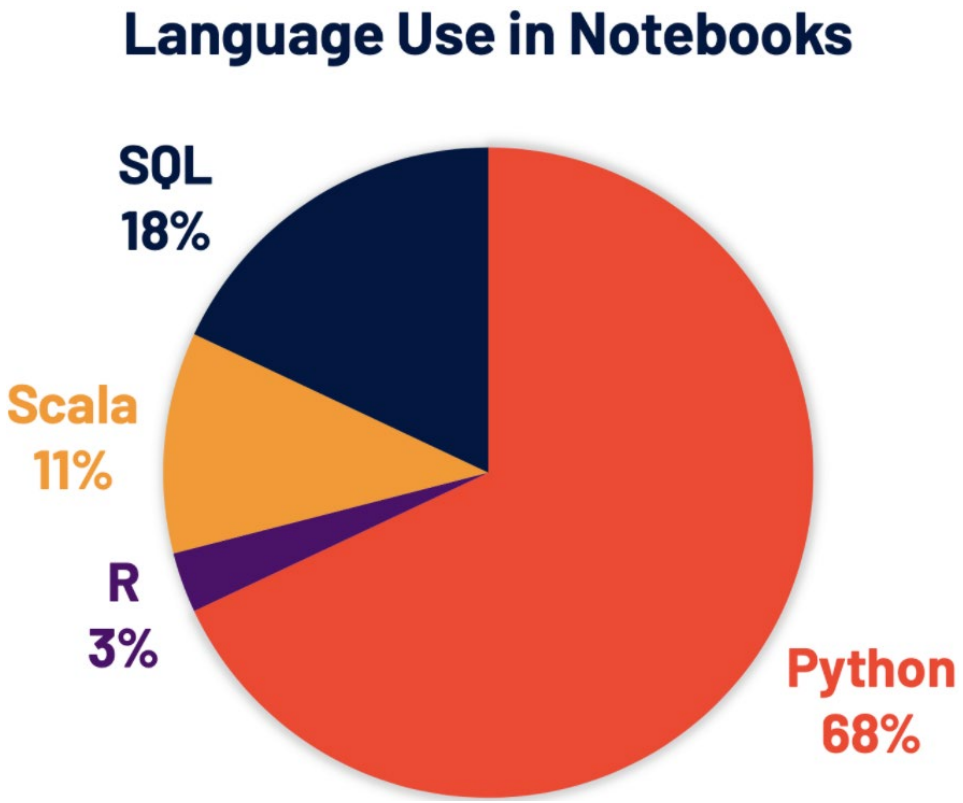
Python现在是Spark中使用较为广泛的编程语言，因此也是Spark 3.0的重点关注领域。Databricks有68%的notebook命令是用Python写的。PySpark在 Python Package Index上的月下载量超过 500 万。

很多Python开发人员在数据结构和数据分析方面使用pandas API，但仅限于**单节点**处理。Databricks会持续开发**Koalas——基于Apache Spark的pandas API实现**，让数据科学家能够在分布式环境中更高效地处理大数据。

经过一年多的开发，Koalas实现对pandas API将近80%的覆盖率。Koalas每月PyPI下载量已迅速增长到85万，并以每两周一次的发布节奏快速演进。虽然Koalas可能是从单节点pandas代码迁移的最简单方法，但很多人仍在使用PySpark API，也意味着PySpark API也越来越受欢迎。







2.5 Koalas入门演示 - Koalas DataFrame构建

pip install koalas 安装koalas类库

```
import pandas as pd
import numpy as np
import databricks.koalas as ks
from pyspark.sql import SparkSession
```

- 构建Pandas的DatetimeIndex

```
dates = pd.date_range('20130101', periods=6)
```

- 构建Pandas的DataFrame

```
pdf = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))
```

- 基于PDF构建Koalas DataFrame

```
kdf = ks.from_pandas(pdf);     type(kdf)
```

或者基于SparkSession构建

```
sdf = spark.createDataFrame(pdf)                    # 先转换PandasDataFrame成SparkDataFrame
kdf = sdf.to_koalas()                  # 转换SparkDataFrame到KoalasDataFrame
```

或者直接创建kdf也可以

```
kdf = ks.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
                          'foo', 'bar', 'foo', 'foo'],
                    'B': ['one', 'one', 'two', 'three',
                          'two', 'two', 'one', 'three'],
                    'C': np.random.randn(8),
                    'D': np.random.randn(8)})
```

2.5 Koalas入门演示 - Koalas DataFrame 查看数据

```
kdf.head()
```

	A	B	C	D
0	-0.621429	1.515041	-1.735483	-1.235009
1	0.844961	-0.999771	0.108356	0.109456
2	1.343862	-1.257980	0.099766	-0.137677
3	3.001767	-0.208167	-1.059449	0.312599
4	-0.035864	0.312126	0.252281	0.627551

```
kdf.index
```

```
Int64Index([0, 1, 2, 3, 4, 5], dtype='int64')
```

```
kdf.columns
```

```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

```
kdf.to_numpy()
```

```
array([[-0.62142908,  1.51504106, -1.73548271, -1.23500912],
       [ 0.84496072, -0.99977056,  0.10835608,  0.10945554],
       [ 1.34386224, -1.25797981,  0.09976648, -0.13767659],
       [ 3.0017674 , -0.20816676, -1.05944851,  0.31259853],
       [-0.03586387,  0.31212594,  0.2522808 ,  0.62755129],
       [-1.20040429,  0.27613401, -0.34430818, -0.3679344 ]])
```

```
kdf.describe()
```

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.555482	-0.060436	-0.446473	-0.115169
std	1.517076	1.007223	0.792741	0.648616
min	-1.200404	-1.257980	-1.735483	-1.235009
25%	-0.621429	-0.999771	-1.059449	-0.367934
50%	-0.035864	-0.208167	-0.344308	-0.137677
75%	1.343862	0.312126	0.108356	0.312599
max	3.001767	1.515041	0.252281	0.627551

posing your data

```
kdf.T
```

	0	1	2	3	4	5
A	-0.621429	0.844961	1.343862	3.001767	-0.035864	-1.200404
B	1.515041	-0.999771	-1.257980	-0.208167	0.312126	0.276134
C	-1.735483	0.108356	0.099766	-1.059449	0.252281	-0.344308
D	-1.235009	0.109456	-0.137677	0.312599	0.627551	-0.367934

```
kdf.sort_index(ascending=False)
```

	A	B	C	D
5	-1.200404	0.276134	-0.344308	-0.367934
4	-0.035864	0.312126	0.252281	0.627551
3	3.001767	-0.208167	-1.059449	0.312599
2	1.343862	-1.257980	0.099766	-0.137677
1	0.844961	-0.999771	0.108356	0.109456
0	-0.621429	1.515041	-1.735483	-1.235009

ng by value

```
kdf.sort_values(by='B')
```

	A	B	C	D
2	1.343862	-1.257980	0.099766	-0.137677
1	0.844961	-0.999771	0.108356	0.109456
3	3.001767	-0.208167	-1.059449	0.312599
5	-1.200404	0.276134	-0.344308	-0.367934
4	-0.035864	0.312126	0.252281	0.627551
0	-0.621429	1.515041	-1.735483	-1.235009

2.5 Koalas入门演示 - Koalas DataFrame 缺失值处理

```
pdf1 = pdf.reindex(index=dates[0:4], columns=list(pdf.columns) + ['E'])
```

```
pdf1.loc[dates[0]:dates[1], 'E'] = 1
```

```
kdf1 = ks.from_pandas(pdf1)
```

```
kdf1
```

	A	B	C	D	E
2013-01-01	-0.621429	1.515041	-1.735483	-1.235009	1.0
2013-01-02	0.844961	-0.999771	0.108356	0.109456	1.0
2013-01-03	1.343862	-1.257980	0.099766	-0.137677	NaN
2013-01-04	3.001767	-0.208167	-1.059449	0.312599	NaN

op any rows that have missing data.

```
kdf1.dropna(how='any')
```

	A	B	C	D	E
2013-01-01	-0.621429	1.515041	-1.735483	-1.235009	1.0
2013-01-02	0.844961	-0.999771	0.108356	0.109456	1.0

g missing data.

```
kdf1.fillna(value=5)
```

	A	B	C	D	E
2013-01-01	-0.621429	1.515041	-1.735483	-1.235009	1.0
2013-01-02	0.844961	-0.999771	0.108356	0.109456	1.0
2013-01-03	1.343862	-1.257980	0.099766	-0.137677	5.0
2013-01-04	3.001767	-0.208167	-1.059449	0.312599	5.0

2.5 Koalas入门演示 - Koalas DataFrame 分组计算

```
kdf3 = ks.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
                           'foo', 'bar', 'foo', 'foo'],
                     'B': ['one', 'one', 'two', 'three',
                           'two', 'two', 'one', 'three'],
                     'C': np.random.randn(8),
                     'D': np.random.randn(8)})
```

```
kdf.groupby('A').sum()
```

A	C	D
bar	-0.992825	2.671637
foo	-2.110918	2.842644

```
kdf.groupby(['A', 'B']).sum()
```

A	B	C	D
foo	one	-0.122637	0.844931
	two	-2.182467	0.252681
bar	three	-1.776986	-0.092022
foo	three	0.194186	1.745033
bar	two	0.386921	1.995358
	one	0.397240	0.768301

2.5 Koalas入门演示 - Koalas DataFrame 数据导入导出

CSV

CSV is straightforward and easy to use. See here to write a CSV file and here to read a CSV file.

[50]:
```
kdf.to_csv('foo.csv')
ks.read_csv('foo.csv').head(10)
```

[50]:

	A	B	C	D
0	-0.821342	-0.325142	0.904636	-0.925984
1	1.498758	0.045747	0.904636	0.726606
2	1.498758	0.045747	0.904636	0.726606
3	1.498758	1.534086	0.904636	0.726606
4	1.498758	1.534086	0.904636	0.726606
5	1.498758	1.534086	0.904636	0.726606
6	1.498758	1.534086	0.904636	0.726606
7	1.498758	1.534086	0.904636	0.856176
8	1.498758	1.534086	0.904636	0.856176
9	1.498758	1.534086	0.904636	1.532448

Parquet

Parquet is an efficient and compact file format to read and write faster. See here to write a Parquet file and here to read a Parquet file.

[51]:
```
kdf.to_parquet('bar.parquet')
ks.read_parquet('bar.parquet').head(10)
```

[51]:

	A	B	C	D
0	-0.821342	-0.325142	0.904636	-0.925984
1	1.498758	0.045747	0.904636	0.726606
2	1.498758	0.045747	0.904636	0.726606
3	1.498758	1.534086	0.904636	0.726606
4	1.498758	1.534086	0.904636	0.726606

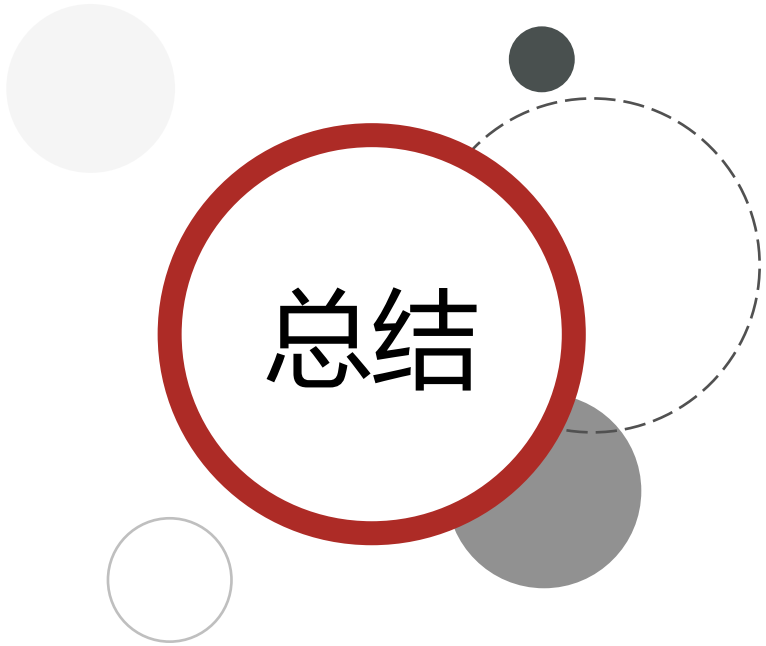
Spark IO

In addition, Koalas fully support Spark's various datasources such as ORC and an external datasource. See here to write it to the specified datasource and here to read it from the datasource.

[52]:
```
kdf.to_spark_io('zoo.orc', format="orc")
ks.read_spark_io('zoo.orc', format="orc").head(10)
```

[52]:

	A	B	C	D
0	-0.821342	-0.325142	0.904636	-0.925984
1	1.498758	0.045747	0.904636	0.726606
2	1.498758	0.045747	0.904636	0.726606
3	1.498758	1.534086	0.904636	0.726606
4	1.498758	1.534086	0.904636	0.726606
5	1.498758	1.534086	0.904636	0.726606
6	1.498758	1.534086	0.904636	0.726606
7	1.498758	1.534086	0.904636	0.856176
8	1.498758	1.534086	0.904636	0.856176
9	1.498758	1.534086	0.904636	1.532448



1. AQE的开启通过: spark.sql.adaptive.enabled 设置为true开启,触发后极大提升SparkSQL计算性能
2. 动态分区裁剪可以让我们更好的优化运行时分区内数据的量级. 通过动态的谓词下推来获取传统静态谓词下推无法获得的更高过滤属性, 减少操作的分区数据量以提高性能.
3. 新版Koalas类库可以让我们写Pandas API(Koalas提供)然后将它们运行在分布式的Spark环境上, Pandas开发者也能快速上手Spark

第三章 概念回顾总结

3 课程概念大总结