一、HDFS (分布式文件系统)

1. 架构核心

NameNode (NN): 管理者。存储元数据（文件名、目录结构、Block 映射），处理客户端请求。

DataNode (DN): 工作者。存储实际数据块(Block)，执行读写操作，定期向 NN 汇报心跳。

SecondaryNN: 辅助。定期合并 Fsimage 和 Edits，辅助 NN 恢复，**不是** NN 的热备。

2. 副本机制 (核心考点)

默认副本: 3 份。

存放策略: ① 本机/同机架 ② 异地机架（防机架断电） ③ 异地机架的另一节点。

作用: 数据冗余保证可靠性；多副本提高读取效率。

容错推导:

原则: 副本数≤ 数据节点数。

宕机分析: 若副本数=3，允许同时宕机 2 个节点而不丢数据。

推导: 如果业务要求"允许 N 台机器同时坏"，则副本数至少为 $N+1$

3. HDFS Shell 常用命令

codeBash

```
hadoop fs -ls /path          # 查看目录
hadoop fs -mkdir -p /dir     # 级联创建目录
hadoop fs -put local_f hdfs_f   # 上传
hadoop fs -get hdfs_f local_f   # 下载
hadoop fs -cat /file         # 查看内容
hadoop fs -rm -r /dir        # 递归删除
hadoop fs -du -h /dir        # 查看大小
```

数据安全性判断

判断公式：存活副本数=原副本数−宕机节点中包含该 Block 的节点数

若存活副本数≥1：数据安全且可读。

若存活副本数= 0：数据丢失。

副本数与节点数的推导

场景题 1：我有 3 个 DataNode，副本数设置为 4，合理吗？

答案：不合理。

原因：HDFS 规定同一个节点不能存储同一个 Block 的两个副本。如果不合理设置，最多只能存 3 个副本，第 4 个副本无法分配，文件会一直处于 "Under-Replicated"副本不足状态。

结论：最大副本数≤DataNode 节点总数。

场景题 2：推导合理副本数(容错需求)

集群有 100 个节点，要求允许同时挂掉 3 个节点而不丢失数据，副本数至少设为多少？

逻辑：副本数>最大允许挂掉的节点数

或者说：副本数=最大允许挂掉的节点数+1

计算：允许挂 3 个，则至少要有 3+1=4 个副本。

场景题 3：小文件合并：为什么实验中要合并小文件上传？

逻辑：大量小文件会产生大量元数据，耗尽 NameNode 的内存 HDFS 适合存大文件。

二、MapReduce (离线计算-核心考点）

1. 核心流程与伪代码

Map 阶段: 读取分片→解析→map(k1, v1)→输出 (k2, v2)。

Reduce 阶段: 拉取数据→合并/排序→reduce(k2, List[v2])→聚合→输出 (k3, v3)。

伪代码(WordCount):

Map: foreach word in line: emit(word, 1)

Reduce: sum = 0; foreach val in values: sum += val; emit(key, sum)

2. Shuffle 过程 (连接 Map 和 Reduce 的桥梁)

Map 端: 写入环形缓冲区→溢写→**分区→排序**→合并→落盘。

Buffer：写入环形缓冲区。

Partition（分区）：标记数据去哪个 Reduce（Hash 取模）。

Sort（排序）：（考点）缓冲区内按 Key 排序。

Spill（溢写）：落盘生成小文件。

Merge：小文件合并成大文件。

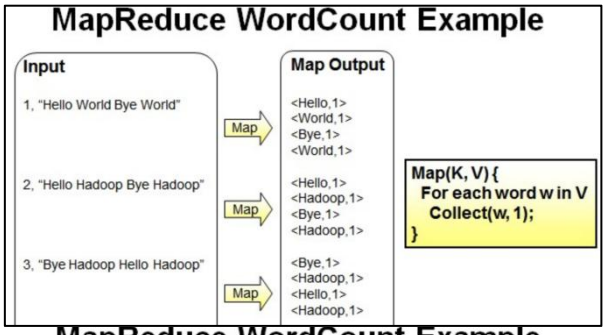
Reduce 端: 拉取→ 归并排序→分组。

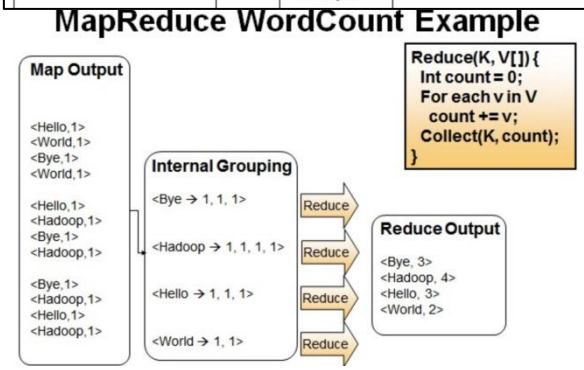
Fetch：拉取各 Map 节点的数据。

Merge Sort：归并排序，合并文件。

Grouping：构造 <Key，ValueList> 传给 Reduce 函数。

作用: 保证数据按 Key 汇聚，是 MR 最耗时的阶段（IO 密集）。





1. Map/Reduce 工作原理与伪代码

Map 阶段：″拆分″。读取分片，解析为 <k1, v1>，处理后输出 <k2, v2>。

伪代码：for word in line.split(): emit(word, 1)

Reduce 阶段：″聚合″。接收 <k2, List[v2]>，对 Values 进行统计，输出 <k3, v3>。

伪代码：sum=0; for v in values: sum+=v; emit(key, sum)

2. Shuffle 过程 (Map 与 Reduce 的桥梁)

定义：**将 Map 输出的数据按 Key 排序、分组传送到 Reduce 的过程。**

核心作用：保证相同的 Key 汇聚到同一个 Reduce 节点。

三、Hive (数据仓库)

1. 分区表 (Partition)

定义: 将大表按字段（如日期）拆分为子目录。

场景: 避免全表扫描，极大提升查询效率。

2. **分区表物理形态** 本质: HDFS 上的子目录。

3. **命名规范:** /表名/分区列名=分区值/数据文件。

例: /logs/day=20260101/file.txt

特点: 数据文件中不存储分区列的值，值由目录名决定。

```
/user/hive/warehouse/t_log/
|
├── day=20260101/          <-- 这就是一个分区（目录）
|   └── data_part1.txt   <-- 这一天的实际数据在这里
|
├── day=20260102/          <-- 这是另一个分区
|   └── data_part2.txt
|
└── day=20260103/
    └── data_part3.txt
```

HQL:

创建：

CREATE TABLE t_log (...) PARTITIONED BY (day string);

加载：LOAD DATA LOCAL INPATH '...' INTO TABLE t_log PARTITION(day='20230101');

查询 (自动触发分区裁剪):

SELECT * FROM t_log WHERE day='20230101';

2. 分桶表 (Bucket)

定义: 将文件按哈希值拆分为固定数量的文件(hash(key) % N)。

优势: 分桶 Join 。当两表均按 Join 键分桶且桶数倍数关系时，可避免 Shuffle，直接在 Map 端 Join，效率极高。

HQL:

创建：CREATE TABLE t_user (...) CLUSTERED BY (id) INTO 4 BUCKETS;

开启优化：set hive.optimize.bucketmapjoin = true;

set hive.enforce.bucketing = true;

3. 常用 HQL

聚合: SELECT city, count(*) FROM t GROUP BY city;

关联: SELECT a.*, b.* FROM a JOIN b ON a.id = b.id;

四、HBase (NoSQL 数据库)

1. 核心组件

HMaster: 管理表结构(DDL)，分配 Region，负载均衡，不处理数据读写。

RegionServer: 处理数据的读写(DML)，维护 Region，管理 HLog 和 HFile。

2. 核心特性与场景

特性: 列存储、稀疏性（Null 不占空）、多版本、海量存储。

场景: 写密集、实时随机读写、结构不固定（画像标签）、时序数据。

VS 关系型: HBase 不支持复杂 Join/事务，适合非结构化海量数据；RDBMS 适合事务性强的小规模数据。

3. RowKey 设计 (必考)　　作用: 唯一索引，决定数据分布。

设计原则:

唯一性: 必须唯一。

长度: 越短越好（建议 16 字节内），节省存储。

散列性: 避免热点问题。**方法:** 倒序、加盐、哈希。

Schema 设计: 表名简短；列族通常 **1 个**，最多不超过 3 个。

4. HBase Shell

启动:hbase shell

创建表 (表名: Student, 列族: Info, 版本数: 3):

```
create 'Student', {NAME=>'Info', VERSIONS=>3}, 'Grades'
```

查看表描述:describe 'Student'

插入/更新数据 (Put: 表, RowKey, 列族:列, 值):

```
put 'Student', '0001', 'Info:Name', 'Elon'
```

获取单行:get 'Student', '0001'　　**统计行数**:count 'Student'

删除操作:deleteall 'Student', '0001'

删整行:disable 'Student' drop 'Student'　　**扫描全表**:scan 'Student'

限制返回版本数:scan 'Student', {VERSIONS=>3}

ValueFilter:查询值等于 19 的数据:scan 'Student', {FILTER=>"ValueFilter(=, 'binary:19')"}

ColumnPrefixFilter:查询列名前缀为'B'的数据:scan 'Student', {FILTER=>"ColumnPrefixFilter('B')"}

组合过滤:列名前缀为 B 且值包含 90:scan 'Student', {FILTER=>"ColumnPrefixFilter('B') AND ValueFilter(=, 'substring:90')"}

```
disable 'tb'; drop 'tb'        # 删除表
```

RowKey	ColumnFamily : CF1		ColumnFamily : CF2	
	Column: C11	Column: C12	Column: C21	Column: C22
"com.google.www"	t1:"value" t2:"value1" t3:"value2"	t1:"value" t2:"value1" t3:"value2" t4:"value3" t5:"value4"	t1:"value" t2:"value1" t3:"value2"	t1:"value"
"com.facebook.www"	t1:"value" t2:"value1"	t1:"value" t2:"value1" t3:"value2"	t1:"value" t2:"value1"	t1:"value"

五、Spark (内存计算)

1. DataFrame vs RDD

RDD: 底层抽象，只读，无 Schema，开发繁琐，手动优化。

DataFrame: 类似 Table，有 Schema，使用 Spark SQL，**Catalyst 优化器**自动优化执行计划，比 MR 快（基于内存）。

2. RDD 特性

惰性求值: Transformation 算子，只记逻辑不执行；遇到 Action 算子才触发计算。**优势:** 允许 Spark 优化 DAG 图，合并步骤。

宽窄依赖: 窄依赖（无 Shuffle，快），宽依赖（有 Shuffle，如 reduceByKey）。

Spark vs MR: Spark 基于内存计算(快 10-100 倍)、代码简洁、技术栈统一(SQL/ML 等)；MR 基于磁盘、代码冗长。

4. **RDD 高效计算与 IO**

聚合优化: reduceByKey 含 Map 端预聚合，效率远高于 groupByKey。

求平均值技巧：先 map 转为(v, 1)再聚合。

格式化保存: saveAsTextFile 前需用 map 拼接字符串，否则会保留元组括号。

RDD 操作

```
rdd = sc.parallelize([1, 2, 3, 4])
res = rdd.filter(lambda x: x > 2) \  # 过滤
      .map(lambda x: (x, 1)) \    # 转换
      .reduceByKey(lambda a,b: a+b)# 分组聚合
res.saveAsTextFile("/out")         # 保存
```

Spark SQL 操作

```
df = spark.read.json("path")
df.select("name", "age").filter("age > 18").show()
df.groupBy("dept").count().sort("count").show()  # 分组统计排序
df.write.parquet("/out")          # 保存
```

1. 过滤操作

filter 保留 True 元素。

```
rdd = sc.parallelize([(1, "Alice", 85), (2, "Bob", 92)])
pass_std = rdd.filter(lambda x: x[2] > 90) # 筛选分>90
```

2. 分组与聚合

groupByKey 效率低；reduceByKey 含 Map 端预聚合，效率高。

求和:

```
scores = sc.parallelize([(1, 90), (1, 80), (2, 95)])
total = scores.reduceByKey(lambda a, b: a + b)
```

求平均 (转(v,1)累加后除):

```
avg = scores.map(lambda x: (x[0], (x[1], 1))) \
      .reduceByKey(lambda a, b: (a[0]+b[0], a[1]+b[1])) \
      .map(lambda x: (x[0], x[1][0] / x[1][1]))
```

3. 格式转换与保存

saveAsTextFile 存元组会带括号，需先拼接字符串。

```
fmt = total.map(lambda x: str(x[0]) + "::" + str(x[1]))
fmt.saveAsTextFile("hdfs://master:9000/output/result")
```

六、Kafka (消息队列)

1. 核心概念

Topic: 逻辑分类。

Partition (分区): 物理拆分，实现**负载均衡**和**高吞吐**。一个 Topic 可以有多个分区。

Replica (副本): 保证数据**不丢失**。Leader 负责读写，Follower 只同步。

数据不丢失机制: ACK 机制（0, 1, all/-1），ISR（同步副本集合）。

2. 设计规范

命名: 见名知意（如 order_create_topic）。

分区数设计:

1.根据吞吐量需求　　2.分区越多,并发度越高,但打开的文件句柄也越多.通常设置为消费者组内消费者数量的倍数。

副本数设计:建议 2-3 个（太多影响写入性能，太少不安全）

1.根据可靠性需求　　2.允许同时挂掉 2 个节点而不丢数据 3.限制: 副本数≤集群 Broker 节点数(同个分区的多个副本不能在同一台机器上)。

3. 控制台命令

```
# 生产者
kafka-console-producer.sh --bootstrap-server node1:9092 --topic test
# 消费者
kafka-console-consumer.sh --bootstrap-server node1:9092 --topic test --from-beginning
# 查看 Topic 详情
kafka-topics.sh --describe --bootstrap-server node1:9092 --topic test
```

Producer (生产者)　　**作用:** 向 Topic 推(Push) 数据。

分区策略如下:

有 Key: Hash(Key) 决定，保证同 Key 有序。

无 Key: 轮询，负载均衡。

可靠性 (ACKs):　　0: 不等待确认 (可能丢)。

1: Leader 落盘即可。

all: 所有 ISR 副本落盘 (数据强一致)。

Consumer (消费者)　　**作用:** 从 Topic 拉(Pull) 数据

Consumer Group (CG): 实现高吞吐的核心

组内: 消息通过 Partition 分摊处理 (负载均衡)。

组间: 消息互不影响 (发布/订阅模式)。

Offset: 记录消费进度，保存在 Kafka 的 __consumer_offsets 主题中。