目录

大厂数仓实习面试题整理

一 大数据计算框架

1 大数据生态

层级	作用	常见工具
数据采集层	把日志、数据库数据采集进系统	Flume、Logstash、Sqoop、Maxwell
数据传输层	把采集的数据可靠传输到存储/计算引擎	Kafka、Pulsar、RabbitMQ
数据存储层	保存原始/中间/结果数据	HDFS、Hive、HBase、MySQL、Doris、ClickHouse
数据计算层	对数据进行 ETL、分析、聚合	MapReduce、Spark、Flink、Presto、Hive SQL
数据服务层	将结果开放给可视化/接口	Elasticsearch、Doris、API 网关、Superset
管理与运维层	调度、监控、权限、安全	Airflow、DolphinScheduler、Yarn、Ranger、Atlas
可视化分析层	面向业务人员展示数据结果	Superset、Tableau、FineBI、Quick BI

2 大数据基础（Hadoop/Spark 原理简答）

大数据技术是为了解决超大规模数据的存储与计算问题，其基础架构主要包括：

（1）Hadoop（以存为主）：

- 核心组件:
  - HDFS：分布式文件系统，负责大数据的可靠存储（副本机制）

    HDFS 是 Hadoop 的分布式文件系统，设计原则是一致性优于低延迟，采用块存储、主从架构。NameNode 管理元数据，DataNode 存实际数据。支持副本机制、容错性强，适合海量离线数据存储与读取。
  - MapReduce：分布式计算模型，适合批处理，执行慢但稳定
  - YARN：资源调度系统，管理内存/CPU 资源，协调任务分配

（2）Spark（以算为主）：

- 是一个基于内存的分布式计算引擎
- 相比 MapReduce 更快，主要优点:
  - 内存计算：减少磁盘 I/O
  - DAG 执行图：优化任务流程

- 支持批处理、流处理、SQL、机器学习等
- 支持容错机制（Lineage 血缘追踪）、多语言 API（Scala/Java/Python）

3 hadoop 的文件存储格式

格式	特点	使用场景
Text	普通文本，易读，体积大	开发初期测试
CSV/TSV	结构化文本，字段分隔	常用于日志分析、导出
SequenceFile	Hadoop 专用二进制 KV 存储格式	MapReduce 中间数据存储
Avro	二进制格式，自带 schema	跨语言兼容、Kafka 传输
Parquet	列式存储，压缩率高，查询快	推荐用于 Spark SQL / Hive
ORC	Hive 优化格式，列式，压缩更强	Hive/Presto 推荐使用

为什么 Spark 的 DAG 能优化任务？

- 可以提前合并多个 map 操作为一个 Stage（如合并多个窄依赖），~~减少中间 Shuffle~~，减少中间结果写磁盘、反序列化、task 切分、调度压力，从而减少资源浪费和 IO 压力
- 只在必要时才触发 Shuffle（宽依赖边界）
- 失败时只重算失败的分支，而不是整个任务

4 HDFS 存储的特殊性

- 分布式、冗余、高容错、顺序读写优化
- 存储格式为块（block），默认每块 128MB
- 不适合小文件和频繁随机写操作
- 设计原则是“一次写入，多次读取”，用于离线批处理场景

5 Hdfs 为什么是 3 个副本，而不是 2 个或 4 个

3 个副本 = 数据可靠性与存储开销的平衡 Hadoop 默认 3 个副本：一个本地机架、一个远程机架、一个随机

- 1 个副本容易丢
- 2 个副本容错低（主备同时挂掉仍会丢）
- 3 个副本可容忍任意一个节点或机架宕机

6 Hdfs 读写流程

读流程：

- Client 向 NameNode 查询文件 block 所在 DataNode
- 直接从 DataNode 拉取数据（优先就近副本）

写流程：

- Client 向 NameNode 申请 block 空间
- NameNode 分配副本 DataNode 列表（如 A→B→C）
- Client 写入 A，A 再写 B，B 再写 C
- 所有 DataNode 确认完成后返回成功

7 项目上有数据丢失了怎么办？

分情况处理：

- **源数据阶段**：找上游系统或日志恢复
- **中间表**：回溯血缘图重新跑
- **下游报表异常**：定位依赖链，使用校验、对账、报警机制

建议日常加：

- 数据校验、数据质量平台
- 数据快照或 checkpoint
- checkpoint + lineage 机制（如 Spark）

8 大数据架构

典型 Lambda 架构三层：

层级	说明	工具
数据采集层	日志采集、传输	Kafka, Flume
计算层	离线 + 实时	Spark, Flink, Hive
存储层	存储计算结果	HDFS, HBase, Doris

服务层	输出结果	可视化、接口调用

9 secondary nn 起到什么作用

- NameNode 每次元数据更新都会写 edits log
- Secondary NameNode 定期拉取 fsimage + edits → 合并 → 生成新快照，减少 NameNode 重启加载压力

10 多分区如何保证有序

- 对数据进行全局排序（必须用 shuffle）
- 使用 sortWithinPartitions()：保证**每个分区内部有序**
- 使用 repartitionAndSortWithinPartitions()：同时分区和排序，**用于写入表时保持桶有序**

11 Spark 和 mapreduce 的区别

对比维度	MAPREDUCE	SPARK
架构类型	基于磁盘的两阶段模型	基于内存的 DAG 执行模型
计算模型	只能 map → reduce，无法链式操作	支持丰富算子、链式执行
执行效率	每个阶段落盘，慢	内存计算，快很多
容错方式	数据副本（靠 HDFS）	Lineage 血缘追踪，失败可局部重算
SHUFFLE 机制	每轮 Job 必定 shuffle	智能判断宽/窄依赖，减少 shuffle 次数
编程难度	Java 编写复杂逻辑	支持 Python/Scala，开发体验更好
使用场景	传统离线批处理	批处理、流处理、机器学习、SQL 全能型
易用性	Job 嵌套、调优麻烦	UI 调试、DAG 可视化强
生命周期	较早期框架，逐步被替代	当前主流引擎之一，广泛部署

- MapReduce：慢、硬、两阶段，适合简单批处理
- Spark：快、灵活、全场景，内存计算 + DAG 优化

✓ “Spark 用的是 RDD 作为弹性分布式数据集”

✓ “Shuffle 是性能瓶颈，Spark 更智能地控制 Shuffle 触发”

✓ “Spark DAG 会自动分析宽依赖并提前切 Stage，提高并行度”

岗位导向	面试时可以这样说
数仓开发	Spark SQL 替代 Hive SQL，执行速度更快，能支撑实时数仓建设
数据平台	Spark 支持多语言（Python/Scala），可与 Airflow/DolphinScheduler 集成
实时计算	Spark Structured Streaming 支持准实时指标计算，可接 Kafka / HBase
AI/ML	Spark MLlib 提供机器学习库，可直接跑分布式训练流程

维度	Spark 优势
性能	内存计算 + DAG 优化，远快于 MapReduce
开发效率	支持链式 API（map、filter、reduceByKey），支持 SQL、DataFrame，开发体验优于 MR 的 Java 编码
多场景支持	除了批处理，还支持流处理（Structured Streaming）、SQL 查询、机器学习（MLlib）、图计算
容错机制	血缘追踪替代副本机制，更节省资源
生态兼容	可读取 Hive、HDFS、Kafka 等常见大数据组件
调试可视化	Spark UI 可查看 DAG 图、Stage、Shuffle，便于性能调优

12 怎么理解 RDD 的弹性？

Spark 引入了 RDD（Resilient Distributed Dataset）弹性分布式数据集，这是对 MapReduce 关键突破：

什么是 RDD 的弹性？

“弹性”是指：当部分计算节点失败时，Spark 可以通过记录的血缘（lineage）关系，自动恢复丢失的数据分片

RDD 的核心特性：

- 不可变：每次转换返回新的 RDD
- 懒执行：只有触发 action 才执行
- 可并行操作：适合分布式处理
- 支持 checkpoint、缓存、容错机制

小结：

RDD 像一个“有血缘关系的分布式表格”，可追溯、可重算，不怕单点失败。

13 MapReduce 的工作原理

MapReduce = 分布式两阶段计算框架

执行流程（牢记 5 步）：

- Map 阶段：切分数据 → 对每行做处理，输出 KV（如：单词 → 1）
- Shuffle 阶段：相同 key 的数据聚集到一起（分区 + 排序）
- Reduce 阶段：聚合 value，如同一单词求和
- 输出阶段：写入 HDFS 或其他目标
- 任务调度：YARN 负责调度每个 Task 的执行

关键机制：

- 多个 Map → 多个 Reduce，分布式并行
- Shuffle 落盘（I/O 成本高）
- 容错靠数据副本，不靠过程血缘

缺点：每轮只能执行一次 map → reduce，无法链式操作。

14 Hive 和 spark 区别

比较维度	HIVE（传统）	SPARK SQL
执行引擎	MapReduce / Tez（慢）	Spark DAG（快）
性能	中间多次落盘，慢	内存计算，优化器支持多级优化
编程方式	SQL 脚本	支持 SQL、DataFrame、RDD 多种
实时性	批处理为主（慢）	支持 Structured Streaming，准实时
调试能力	调试困难	有 Spark UI，DAG 可视化
适用场景	离线日/小时级报表	离线 + 准实时任务 + 多维分析
兼容性	Hive 元数据兼容 Spark	Spark 可接 Hive Metastore，互通数据

Spark SQL 是 Hive SQL 的高性能替代方案，几乎是现代数据开发的主流工具。

15 spark 相关的八股

- spark join

Join 类型	是否触发 Shuffle	优化方式
shuffle join	是	大表 join 小表，推荐 **broadcast join**
broadcast join	否	小表广播到每个节点，避免 shuffle

- spark 任务的提交流程
  - 解析代码，生成 DAG
  - 提交 Job 给 SparkContext
  - SparkContext 交给 YARN/Mesos/K8s 调度资源
  - 分发 Task 到 Executor 执行
- spark 算子

类型	示例
转换（懒执行）	map, filter, groupByKey, join
行动（触发执行）	count(), collect(), saveAsTextFile()

driver 在什么时候运行?

Driver 在任务开始时就运行，负责：

- DAG 图构建
- Job 划分
- Stage 提交
- 汇总 Executor 返回结果

16 Spark.shuffle.partitions 的调参，spark 小文件参数

spark.sql.shuffle.partitions：

- 默认是 200
- 控制 Spark SQL 或 DataFrame 中 shuffle 后的并行度

什么时候调?

- 如果数据量小 → 降低 partition 数（避免生成很多小文件）
- 数据量大 → 增加 partition（加快并发）
- spark.conf.set("spark.sql.shuffle.partitions", 50)

参数	作用
spark.sql.files.maxPartitionBytes	控制每个 partition 最大字节数
spark.sql.files.openCostInBytes	控制 Spark 每次读取文件的开销估值
spark.sql.parquet.mergeSchema	是否自动合并 schema，可能影响性能
合并小文件写出	用 coalesce() or repartition() 优化写出前的 partition 数

- spark.sql.files.openCostInBytes 是 Spark 在读取多个小文件时，用来估计每打开一个文件的“成本”大小（开销）
- 默认值是 4MB
- Spark 在估算 partition 划分时，会用这个开销 + 文件实际大小一起决定一个 partition 是否合并多个小文件
- 作用：避免小文件太多，每个都单独读一个 partition，造成“任务多，数据少”的低效执行

spark.sql.parquet.mergeSchema 是做什么的？

这个参数只影响 Parquet 表的字段结构合并行为

- 如果你往同一个表路径多次写入不同 schema 的 Parquet 文件
- 比如第一次是字段 A/B，第二次是 A/B/C
- 如果这个参数为 true，读取时会自动把多个 schema 合并（形成 A/B/C）
- 如果为 false，则报错或只识别部分字段

实际建议：大多数场景下关掉 mergeSchema 会更快更稳定

方法	作用	是否 SHUFFLE
COALESCE(N)	合并 partition 数量（**只减少**）	✕ 无 Shuffle（更快）
REPARTITION(N)	精确设置 partition 数量（可增/减）	会 Shuffle

用法：

python

```
df = df.coalesce(10)       # 合并为 10 个分区，适合写出前优化小文件
df = df.repartition(100)   # 重新打散划分成 100 个分区，适合大表 Join 前预分区
```

17 怎么快速根据 spark stage 找到对应的代码

找代码位置的方法：

- 打开 Spark UI
- 进入 Job → 选中某个 Stage
- 看 RDD Lineage → 找 call site（代码路径）
- 对应代码会标明是哪个算子触发的（如 .groupBy()）

常见触发 Stage 的位置：

- groupBy, join, distinct, repartition, sort → Shuffle 算子
- count(), collect() → Action 算子（触发执行）

18 Spark 如何划分 stage

Spark 会基于 DAG 执行图来自动划分 Stage。

Stage 划分规则：

- 遇到宽依赖（shuffle）边界 → 新建一个 Stage
- 每个 Stage 包含多个 Task（按 partition 划分）

19 Spark 宽依赖怎么理解？举个例子？

宽依赖定义：

当前 RDD 的一个分区依赖于多个上游分区的数据

会触发 Shuffle + 重新分区 + 跨节点拉数据

举例（含 Shuffle）：

Python：

```
rdd1 = sc.textFile("a.txt")
rdd2 = rdd1.groupBy(lambda x: x.split(",")[0])
```

- 同一个 key 可能来自不同 rdd1 的分区
- Spark 会对数据分区、Hash 拉取、排序处理

对比窄依赖（map、filter） → 一对一，不触发 Shuffle

项目	含义
STAGE	是 Spark 执行计划的最小调度单元，一组可以并行运行的 Task
划分依据	宽依赖（需要 Shuffle）处会被拆分为不同 Stage
用途	提高并发度、便于失败时部分重试

20 Spark 的 shuffle 和 hadoop 的 shuffle 的区别是什么？各自的原理？什么情况下会触发 shuffle？

Spark 支持 Map-side combine、Bypass Merge、Sort-based Shuffle 等策略

Shuffle 是 MapReduce 中的“灵魂阶段”，用来做数据重分布 + 排序 + 聚合前准备：

Shuffle 作用流程：

- Map Task 输出 KV 对
- 对输出进行 Hash 分区（决定发给哪个 Reduce）
- 每个 Reduce 拉取自己负责的 key 的全部值
- Reduce 执行聚合逻辑（如求和）

举例说明：

WordCount → Map 输出 (word, 1)

Shuffle → 按 word 聚合所有 1 到一个 Reduce

Reduce 输出 (word, count)

Shuffle 是性能瓶颈的原因：

- 多次磁盘落盘 + 跨节点网络传输
- 每个 Reduce 要拉很多 Map 输出，容易拖慢速度
- 不支持优化（如 Broadcast Join、Map-side 聚合）

21 Hadoop 底层机制如何判断哪个副本错了？

HDFS 中副本一致性判断机制如下：

什么是“心跳（Heartbeat）”？

Hadoop 中的 DataNode → NameNode 的定期信号报告机制，作用是让 NameNode 知道 DataNode 还“活着”。

- NameNode 会检查：
  - 是否有 DataNode 宕机（心跳超时）
  - 是否有副本 Block 损坏（DataNode 上报的校验和）
- 每个 Block 都有对应的 Checksum，如果：
  - 某个副本返回的数据校验失败 → NameNode 会标记该副本为损坏
  - 然后从其他健康副本中复制一个新的副本进行替换

所以判断机制是基于：心跳 + Checksum + 异常上报

在哪看？

- Hadoop Web UI（NameNode 页面）
- 默认端口：http://namenode_host:9870
- 点击：DataNodes，就能看到每个节点的心跳状态、最后活跃时间、是否丢失副本

什么是 Checksum？

校验和，用于判断数据文件块是否损坏。每个 Block 都自动带有一个对应的 checksum 文件。

- DataNode 在读写 block 的过程中，都会自动生成并校验 checksum。

- 如果发现某个副本 block 的 checksum 不对 → 判定该副本损坏 → NameNode 自动发起副本恢复操作

在哪看？

- Checksum 是系统层透明实现的，不直接暴露在 UI
- 你可以通过命令行间接验证：`hdfs fsck /路径 -files -blocks -locations`
- 它会列出哪些 block 损坏、哪些副本丢失

22 如果一个 spark 任务跑得慢，怎么去排查分析？

步骤	内容	工具
① 检查 SPARK UI	查看 Job / Stage 是否卡住，是否某个 Stage 异常慢	Spark Web UI
② 看是否数据倾斜	某些 Task 明显慢，执行时间不均	Spark UI - Stage - Task 列表
③ 检查 SHUFFLE	Shuffle read/write 特别大，是否反复 spill	Spark UI - Stage - Shuffle Read
④ 看 GC 情况	Driver 或 Executor 是否频繁 GC	Spark UI - Executor 标签页
⑤ 看资源分配	Executor 内存/核数是否足够，是否 OOM	Spark 配置 + YARN Resource

GC: garbage collection – 垃圾回收，释放内存，在进行 GC 时其他工作会停止

如果是 SQL 查询慢，也可以：

- 查看 Physical Plan 是否有 Cartesian Join / Cross Join
- 查看是否使用了 broadcast join

Physical Plan（物理执行计划）是什么？

是 Spark SQL 编译执行的 最终底层执行逻辑 DAG，包括每个算子、Shuffle、Join 类型等信息。

你可以在 Spark UI → SQL 页面看到：

- Parsed Logical Plan
- Analyzed Logical Plan
- Optimized Logical Plan
- Physical Plan ← 就是实际执行的计划（可看到是否用了 Broadcast）

Broadcast Join 是什么？

小表广播 join，是 Spark 中优化 Join 的一大利器。

场景：一张小表（<10M） join 一张大表

sql

```
SELECT /*+ BROADCAST(dim_table) */ *
```

```
FROM fact_table
JOIN dim_table
ON fact_table.id = dim_table.id
```

优势：

- 小表直接广播到每个节点
- 避免 Shuffle（大表数据无需重分区）

触发方式：

- Spark 会自动判断小表（默认 < 10MB）自动使用
- 或手动指定 broadcast() 函数

23 如何甄别是否发生数据倾斜？从 spark ui 的哪些指标去看？

数据倾斜：在分布式计算中，某些 Task 处理的数据量显著大于其他 Task，导致执行时间严重

数据倾斜的迹象：

- 某些 Task 耗时远远超过其他 Task（如多数 Task < 1s，但某一个 > 60s）
- Shuffle Read/Write 字节数极度不均衡
- 某个 Stage 卡很久，且 Task 数量少

页面	指标	判断方法
Stages	Duration（耗时）	看每个 Stage 是否异常慢
Tasks（某 Stage 内）	Shuffle Read Size / Duration	是否某个 Task Read 特别大、耗时远超其他
SQL 页面	Physical Plan	是否有 Skew warning、Shuffle spill 等信息提示
Executors	GC Time	是否因为 GC 导致执行慢或卡死

24 Spark ui 有哪些菜单信息？每个信息代表什么意思？

默认端口 4040

菜单	含义
Jobs	查看每个 Spark Job 的执行情况（DAG 图、状态）
Stages	每个 Job 被拆分的 Stage，宽依赖会分多个 Stage
Tasks	每个 Stage 中具体执行的任务（分区维度）
SQL	展示 SQL 查询的 Logical Plan / Physical Plan / 优化建议
Executors	每个 Executor 的内存/CPU 使用情况、GC 时间

Environment	Spark 配置项（包括内存、shuffle 参数等）
Storage	查看缓存的 RDD、DataFrame
Streaming（如果是流任务）	观察每批次数据处理详情、延迟等

倾斜常见出现位置：

- groupByKey, reduceByKey, join, distinct 等 Shuffle 宽依赖算子
- 特别是在某些 key 频次极高（如 111111、NULL、空字符串）时

25 Spark ui 调优

步骤	调优点	方式
① 看 SQL 执行计划	是否使用了笛卡尔积、全表 join？	尽量用 broadcast、限制 join
② 分析任务倾斜	是否存在大 key（如用户 ID）？	采用 salting 或 map-side join
③ Shuffle 任务耗时长	是否 spark.sql.shuffle.partitions 太多/太少？	合理调整并行度
④ GC 频繁 / OOM	内存不够？代码有缓存？	加 executor memory 或优化缓存逻辑
⑤ 小文件过多	导致 stage 多、shuffle 扰动	用 coalesce 或 merge 控制文件数
⑥ 窄依赖链过长	避免中间 Dataset 过长 lineage	checkpoint 或写出中间结果

Spark UI 是调优的“体检报告”，重点关注慢 stage、大 task、GC 时间、Shuffle 数据量。

26 Spark 的内存管理

从 Spark 1.6+ 之后，内存分两块:

内存区域	用途	举例
Execution Memory	用于 shuffle、join、aggregation 过程的中间缓存	sort,hash,join,aggregateByKey
Storage Memory	用于缓存 RDD、DataFrame	.cache() .persist()

动态分配：

- Execution 和 Storage 内存之间是弹性的
- 优先满足 Execution，如果缓存不够，Storage 会被挤占

参数	描述
spark.executor.memory	每个 executor 的总内存
spark.memory.fraction	用于 Execution+Storage 的比例（默认 0.6）
spark.memory.storageFraction	其中分给 Storage 的最大比例（默认 0.5）

常见问题：

- 缓存太多 → Storage OOM
- join 中 shuffle 数据太大 → Execution OOM
- 使用 persist() 后忘记 unpersist()，内存压力升高

27 在 mr 的 shuffle 中都有哪些排序

阶段	排序类型	描述
Map 输出前	**Map-side Sort**	Map 阶段输出前对 key 做排序（全局有序）
Reduce 拉取前	**Partition 排序**	同一个 reduce 接收的 key 排好顺序
Reduce 输入前	**Merge Sort**	多个 Map 输出合并成一条大流，归并排序

排序作用：

- 保证 Reduce 能有序地接收数据，支持 sortByKey、窗口类聚合等操作
- 代价高，尤其涉及磁盘 spill 和网络 IO，常为性能瓶颈

28 Hadoop 和 hive 的关系

项目	Hadoop	Hive
本质	分布式计算平台（HDFS + MapReduce + YARN）	数据仓库工具，基于 Hadoop 之上
作用	提供底层存储、计算资源	提供 SQL 接口 + 元数据管理
引擎	MapReduce	底层用 MapReduce、Tez、Spark
依赖	独立运行	依赖 Hadoop 提供存储和执行环境

Hive 是 Hadoop 上的 SQL 层封装，本身不能存储或执行，需要调用 Hadoop 提供的能力。

29 Hadoop 能支持数据更新吗

不支持直接“更新/修改”文件内容！

原因：

- HDFS 是 追加写 + 一次写多读 设计，不支持随机写入/修改
- 类似“只能追加日志，不能改过去的日志”

替代方案：

方案	描述
用 HBase	HBase 是 Hadoop 上的“随机读写型数据库”
用拉链表（数仓方法）	新增一行标记“新版本”，旧数据保留
用 DeltaLake / Hudi（高级方案）	支持 ACID + 数据更新（已开始广泛使用）

HBase 是否直接修改 HDFS 上内容?

不是“修改” HDFS 文件，而是用 HDFS 存储 WAL 和 HFile 来支持“伪修改”行为。

工作机制如下：

- 所有写入数据首先进内存 MemStore
- 同时写入 WAL（Write-Ahead Log）日志文件，存在 HDFS
- 达到一定量后，flush 落盘为 HFile
- 查询时将 MemStore + HFile 合并，得到最终值
- 删除只是打上“删除标记”，定期合并清除（compaction）

所以：HBase 是基于 HDFS 做“可更新、随机读写”能力封装的 KV 存储引擎，和 Hive 不同。

Hive 是纯 SQL 层封装，底层就是一堆不可修改的文件（ORC、Parquet）

DeltaLake / Apache Hudi 是什么?

它们是构建在 Spark / HDFS / 对象存储上的 支持 ACID 的数据湖系统：

系统	全称	主要功能
ltaLake（Databricks	增删改查 + ACID + 版本控制 + 并发	Spark 原生支持
di（Uber）	Hadoop Upserts Deletes and Incrementals	支持增量写、拉链表更新
berg（Netflix）	基于快照的版本控制	分区优化强大（被阿里接入支持）

它们是弥补 Hive 无更新能力 + HDFS 无事务能力的替代方案，也是未来趋势。

30 Spark 的 dataframe 和 dataset

特性	DataFrame	Dataset
类型系统	弱类型（类比 Pandas）	强类型（类比 Java Bean）
编程语言支持	Java, Scala, Python	Scala, Java
编译期检查	无	有（支持编译时类型校验）
性能	快，优化好	略慢，适合复杂对象

使用场景	SQL 操作，ETL 工程师常用	Java/Scala 工程师偏好，适合函数式编程

DataFrame 是什么？

类似于 Pandas 的表格结构，是 Spark SQL 的核心抽象。

- 结构化数据（带 schema）
- 提供 SQL 风格的操作（select、where、groupBy）
- 弱类型，只检查字段名，不检查字段类型
- 适合 Python/Scala 开发，代码简洁，优化器可自动优化

Dataset 是什么？

是强类型、带编译期检查的结构化数据集，仅限 Java / Scala 使用

- 比 DataFrame 更类型安全
- 支持函数式操作（如 map、flatMap）
- 语法上类似于 RDD + schema

Spark dataframe(python) + spark SQL 更容易与现代数仓生态整合（DeltaLake / Hudi / 数据湖）

- Spark 原生支持 Hudi、DeltaLake 等新型数据湖格式
- 这些数据湖技术都支持 Python DataFrame API，但不直接支持 HiveQL
- 如果你未来考虑流批一体、数据湖架构，Spark + Python 是更好的选择

31 Hive 和 spark 的理解和区别

维度	Hive	Spark
类型	数据仓库工具	通用分布式计算引擎
适用场景	离线数据仓库	批处理 + 流处理 + SQL + ML
执行引擎	MapReduce/Tez/Spark	原生 DAG 执行
执行性能	多次落盘、慢	内存计算、快
SQL 支持	HiveQL	SparkSQL，支持 DataFrame 操作
资源调度	YARN	YARN / Standalone / K8s
容错机制	副本机制（HDFS）	血缘机制（lineage）+ checkpoint
数据源	HDFS 为主	HDFS、Hive、JDBC、Kafka、S3、ES 等

Hive：传统批处理 SQL 工具

Spark：现代高性能计算平台，能做 SQL、流处理、机器学习等

32 Yarn 任务调度队列

YARN 全称：Yet Another Resource Negotiator，是 Hadoop 生态的资源管理 & 调度框架。

- 负责统一管理所有节点的内存/CPU 等资源
- 接收上层应用提交的 Job 请求（如 Spark/Flink/Hive）
- 分配 Container 给具体任务执行

YARN 调度逻辑 & 队列机制：

- 资源池按队列划分（比如：开发队列、线上队列、测试队列）
- 每个队列有最大资源额度 / 最小资源保障 / 权重优先级
- 用户提交 Spark Job 时需要指定队列名称

--queue dev

调度策略有三种（面试必背）：

调度器	特点	适用场景
FIFO（默认）	先进先出，不考虑资源类型	简单环境
Capacity Scheduler	多队列资源隔离 + 保障最低资源	多部门共享
Fair Scheduler	各个 Job 尽量公平共享资源	大数据批处理集群

Spark on YARN 的部署模式：

模式	描述
ent *模式*	Driver 在本地机器上运行（适合调试）
ster *模式*	Driver 和 Executor 都由 YARN 调度

结论：YARN 是 Spark/Flink 的调度“大脑”，调度规则由队列 + 优先级 + Scheduler 共同决定。

33 Spark 是实时还是准实时

标准定义：

Spark 本身是“准实时”计算引擎，不是纯实时

- 但 Spark 通过模块划分，可以实现不同程度的实时性：

模块	实时程度	描述
Spark Core / Spark SQL	批处理（离线）	T+1 日志清洗、DWD 建模
Spark Streaming	准实时（微批）	每 1~10 秒处理一次数据
Structured Streaming	准实时 + 精确一次语义	支持事件时间、延迟容忍
Spark + Kafka	实时日志采集 + 延迟 1s 级	

所以面试答题推荐：

“Spark 本质是准实时框架，尤其是 Structured Streaming 在处理流式数据上可以达到秒级延迟，支持 Exactly Once 保证，满足大多数实时业务需求。”

首先明确：Spark 是一个通用的分布式计算平台，不是像 HDFS 那样的存储系统。

它本质是 一个计算引擎，包括：

组件	用途
Spark Core	基础运行时，支持 RDD
Spark SQL	结构化查询语言（支持 DataFrame）
Spark Streaming	老版流计算框架（基于微批）
uctured Streaming	新一代流处理框架（更强大，推荐使用）
MLlib	机器学习模块
GraphX	图计算模块

Spark Streaming 是什么？

Spark 最早期的流处理模块，采用 Micro-Batch（微批）机制。

- 每隔 1~10 秒，Spark 会从数据源（如 Kafka）中拉一批数据，生成一个 mini-RDD 批处理
- 处理逻辑和普通 RDD 类似
- 缺点是：不支持复杂事件时间、状态处理、语义保障弱

→ 这个模块现在已逐渐被替代，推荐使用 Structured Streaming。

Structured Streaming 是什么？

Spark 2.x 引入的新一代流处理引擎，语法与 Spark SQL 完全统一，支持强大的状态管理和容错机制。

核心特性：

- “看起来像静态表，实际上是流式数据”，统一 batch 和 streaming 开发模型

- 支持 Watermark、水位线机制、状态 TTL、Exactly Once 保证
- 同样支持 Python、SQL、Scala 写法

举例（Structured Streaming 写法）：

python

```
df = spark.readStream.format("kafka").option(...).load()
df.selectExpr("CAST(value AS STRING)").writeStream.format("console").start()
```

Spark + Kafka 是什么？

Kafka 是 Spark 流计算的主流数据源，搭配 Structured Streaming 构成 “准实时处理方案”。

Kafka 提供实时日志流 / 用户行为数据

Spark 通过 Structured Streaming 处理这些数据，并支持：

- 实时指标聚合
- 实时清洗 + 落库
- 与维表 join，写入 Hudi / Hive / ClickHouse 等

所以回答你的问题：

是的，Spark 就是一个统一计算平台，你可以通过配置和代码选择：

- 使用批处理 or 流处理（Streaming）
- 使用 SQL or Python/Scala/Java API
- 使用什么数据源（Kafka、Hive、HDFS、JDBC、S3）

你不需要进入一个“平台界面”，而是在代码层通过 SparkSession 来控制：

python

```
spark = SparkSession.builder.enableHiveSupport().getOrCreate()
```

34 spark v.s. flink

维度	Spark	Flink
处理模型	微批处理（Structured Streaming）	真正的事件驱动流处理
延迟	秒级延迟	毫秒级延迟
时间语义	支持事件时间，但处理复杂	事件时间 + 水位线处理很强
State 管理	中间状态管理不如 Flink 灵活	支持有状态操作 + Checkpoint 恢复
窗口计算	简单支持	非常强，支持滚动/滑动/会话等多种窗口

复杂算子	API 全，但调优麻烦	API 更细粒度、状态可控
主语言	Scala / Python	Java / Scala（Python 不够成熟）
典型场景	批处理 + 准实时	实时数仓、金融风控、监控报警等高实时要求场景

Spark：通用计算平台，适合“批 + 准实时”；

Flink：专注流处理，适合“实时强一致”。

Flink 是更偏向强实时的流计算引擎，适合对延迟特别敏感的业务（如风控秒级响应）；而 Spark 在批处理和准实时领域生态更成熟，性能也足够，特别适合离线数仓、指标建模、ETL 处理这些场景。

在我当前的项目中，指标处理和建模逻辑以离线 + 准实时为主，Spark 更贴合我们的需求。

对比点	Spark	Flink
定位	通用计算平台（批为主）	强实时流计算引擎
延迟	秒级（Structured Streaming）	毫秒级（事件驱动）
容错机制	基于 RDD lineage / checkpoint	精确一次语义（基于状态一致性）
API 易用性	SQL/DataFrame 更成熟	更依赖 Java/Scala
社区生态	大数据、AI、数据湖广泛集成	在金融风控/IoT 等实时场景强势
与数据湖结合	Spark → DeltaLake/Hudi 原生支持	Flink 支持但相对新起步

什么时候选 Spark?

- 离线数仓
- 批流一体的准实时指标
- SQL 友好型开发（Python、BI 接入）

什么时候选 Flink?

- 秒级反应的风控 / 实时监控系统
- IoT / 电商推荐系统
- 需要极致低延迟和精准状态处理的系统

总结口播版：

“不是 Spark 不能做实时，而是 Flink 在实时场景更细致精细；而 Spark 的广泛生态、低门槛、批流一体的能力让它仍然是主流数仓开发和实时系统建设的优选平台之一。”

语义等级	描述	举例
At most once（至多一次）	数据只处理一次，失败不重试，可能丢数据	快但不安全
At least once（至少一次）	数据至少处理一次，失败会重试，但可能重复	重复数据需手动去重

Exactly once（精确一次）	不多不漏，**每条数据仅处理一次**	最难实现，但最理想

Exactly Once 如何实现？

这需要：

- 数据源支持 offset 管理（如 Kafka）
- 处理过程具备幂等或事务保证（如不重复更新状态）
- 数据写出端支持幂等 / 事务写入（如 idempotent sink / Hudi 事务落盘）

举个例子：

- Kafka 有 offset
- Spark Structured Streaming 自动管理 offset 提交 + Checkpoint
- Sink 是 Hudi，支持写入幂等

→ 整个链路就能做到："每条数据只被完整处理一次"

为什么重要？

在实时系统里，如果语义控制不好，就会：

- 处理漏数：不安全（At Most Once）
- 重复计数：业务出错（At Least Once）
- 延迟高：吞吐降低（Exactly Once 成本更高）

"精确一次语义指每条数据在处理过程中不重不漏。Structured Streaming 支持自动 checkpoint 和 offset 管理，在输入（如 Kafka）和输出（如 Hudi）都具备幂等性的前提下，可以保障完整的 Exactly Once 语义。"

35 Kafka 和 Flink 在实时计算中的作用？

Kafka 是一个分布式消息队列，常用于**实时数据采集与传输**，它负责：

- 接收上游日志/埋点/设备数据
- 将数据以 Topic 的形式存储并分发
- 保证数据的顺序性、持久性、可重复消费

Flink 是一个分布式流处理引擎，常用于**实时数据计算与分析**，它负责：

- 从 Kafka 中消费数据流
- 对流数据进行 ETL、聚合、窗口计算、维表关联
- 实时写入下游存储如 Doris、HBase、ClickHouse

类比：

Kafka = “高速公路”，Flink = “行驶在路上的货车处理厂”

二 数据调优

1 数据倾斜的解决方案？

一、Join 过程中的数据倾斜

方法	原理	适用情况
Broadcast Join	小表广播到各个 Executor，避免 shuffle	小表（<10MB）与大表 join
Salting 加盐	将热点 key 拆分为多个“伪 key”，打散后 join，再还原	某几个 key 数据量特别大
Repartition + Filter	将热点 key 单独提取出来做分开 join，其余正常 join	热点 key 可识别、比例不高
Map-side join（提前关联）	在数据预处理阶段就先完成 join，避免后期聚合	数据流程允许提前聚合
调整 partition 数量	提高 spark.sql.shuffle.partitions 让更多 task 分担压力	总数据量大，节点资源足够时

二、Aggregation 聚合类操作倾斜（如 groupByKey）

方法	原理	适用情况
使用 reduceByKey 替代 groupByKey	reduceByKey 支持 map-side combine，减少 shuffle 量	尽可能使用 reduceByKey
使用 combineByKey / aggregateByKey	自定义聚合逻辑，控制内存和聚合效率	自定义聚合场景
过滤热点 key 单独聚合处理	把热点 key 提前拆出来单独聚合	热点 key 占比不高
加盐打散 key → 聚合后再去盐	拆 key → 分组聚合 → 再合并结果	key 分布高度倾斜

三、Input 输入倾斜

输入数据本身不均匀，如某个文件特别大、某些文件内容复杂，导致 map 阶段倾斜

方法	原理	适用情况

手动拆分大文件	把大文件切成多个小文件，提升并发度	单个文件 > 分区容量时
合并小文件 / 文件预处理	减少碎片化文件，提高 IO 效率	小文件过多场景
调整 Hadoop split 策略	提高最小 split size，避免分区不均	Hadoop 源头可控场景

四、逻辑复杂 / 算子倾斜

某些分支逻辑复杂，或者某个 key 对应的业务逻辑特别重（如大量外部资源访问）

方法	原理	适用情况
热点 key 分支提前处理	把热点 key 单独走分支处理	可识别热点 key 时
异步缓存（如 Redis）预处理	热点数据放 Redis 减轻 Spark 负担	热点数据不变场景
UDF/UDAF 优化逻辑	减少复杂函数中 IO / 数据结构开销	算子执行耗时不均场景

五、调参优化（辅助策略）

- **调整并行度**：`spark.sql.shuffle.partitions = 200+`
- **启用动态分区合并**：`spark.sql.adaptive.enabled = true`
- **内存缓存**：`rdd.persist(StorageLevel.MEMORY_AND_DISK)`

2 如果在 shuffle 过程中发生倾斜怎么办

常见应对策略如下（记住分类）：

一、针对 Join 倾斜（如大表 + 小表）：

方法	描述
oadcast Join	小表广播到每个节点，避免 shuffle
lting（加盐）	将热点 key 拆成多个 key，打散后再 join，然后聚合回去
partition 前打散大表	用 repartition(n) 或自定义 hash 算子将大 key 分摊

举个例子：你有一个大表 fact + 一个维表 dim，发生了 join 倾斜

方法 1：Broadcast Join

- 小表广播到每个节点，不需要根据 Key 进行 shuffle → 不存在倾斜
- 原理：跳过 Shuffle 步骤，直接 map-side join，彻底绕过问题

方法 2：加盐（Salting）

Step 1：为大 Key 添加一个随机数（如 0~9）→ 拆分为多个不同 key

Step 2：为维表或另一张表也添加相同规则的随机数副本

Step 3：Join 完之后再聚合回原来的 Key（去盐）

- 原理：将一个热点 key 拆成多个 → 分散到多个分区去处理
- 效果：让单个 Task 的压力被分摊 → 执行时间均衡

方法 3：热点 key 单独处理

- 把 topN 大 key 用 where 语句单独挑出来 → 单独处理 join 或聚合
- 其余数据走正常逻辑
- 最后 union 结果

总结：所有方案本质上都是让胖的 task 变瘦、瘦的 task 多干点事

二、针对 Aggregation 倾斜（如 groupByKey）：

方法	描述
用 reduceByKey 替代 groupByK	reduceByKey 支持 map-side 聚合
定义 combineByKey	提前做部分聚合
隔离热点 key 单独处理	将 top N 热 key 抽出单独处理，避免影响整体 stage

总结口诀：

“Join 用广播 / 加盐，聚合用 reduce / 分治”。

典型 Shuffle 流程（以 groupByKey 为例）：

Map 阶段

- 每个 Task 对自己的数据做预处理（如 map、flatMap）
- 将输出根据 key 做 hash 分区，形成多个临时文件（每个文件对一个下游 reduce 分区）

Spill 阶段（可能）

- 内存不足时会将数据 spill 到磁盘

Reduce 拉取阶段

- Reduce Task 会去拉取各个 Map 任务对应自己分区的数据块
- 拉取过程中还会 排序、合并、聚合

这个过程是 网络 I/O + 磁盘 I/O + 内存排序 + 数据聚集的组合，是整个 Spark / MR 中最复杂 & 最耗时的阶段。

*为什么 **Shuffle** 会引发数据倾斜？*

*倾斜成因：本质是 **Key** 与 **Partition** 分配严重不均衡。*

3 Repartition 和 coloase 区别

项	Repartition	Coalesce
是否 Shuffle	✅ 有 Shuffle	❌ 尽量无 Shuffle
适用场景	增加 / 重新划分分区	减少分区（合并）
性能	慢（重新打散）	快（合并就近）
用法	rdd.repartition(n)	rdd.coalesce(n)

4 .persist()和.cache()的区别

方法	存储级别	是否等价
.cache()	MEMORY_AND_DISK	等价于 .persist(StorageLevel.MEMORY_AND_DISK)
.persist()	可自定义存储级别	更灵活

5 数据量大怎么处理

1. 存储优化

方法	原理
分区（Partition）	按时间、地区等字段分区，避免全表扫描
分桶（Bucketing）	配合 join 时优化 shuffle
压缩格式	使用 Parquet、ORC 减少 I/O
小文件合并	减少文件系统压力（如 spark.sql.files.maxPartitionBytes）

2. 计算优化

方法	原理
减少 shuffle	尽量避免 groupByKey，使用 reduceByKey、broadcast join
控制数据倾斜	hotspot key 单独处理，加盐、分阶段聚合
合理并行度	调整 spark.sql.shuffle.partitions

缓存热点数据	使用 .persist() 减少重复计算

3. 资源优化

方法	原理
分布式执行	利用 Spark、Flink 分布式能力切分任务
调大 executor 内存	防止 OOM、减少 GC 时间
合理配置 core/instance	防止资源浪费或竞争

4. 调度优化

方法	原理
分批调度	分阶段处理大表，减少一次性压力
拆表处理	分主题域、分业务线并行计算
使用 DAG 血缘控制依赖	避免重复跑全链路

5. 架构选型优化

情况	建议
数据分析类	考虑 ClickHouse / Doris 做汇总层
明细记录类	用 HBase 或数据湖（Hudi/Delta）
实时类	Flink + Kafka + OLAP 数据库
批处理类	Spark/Hive + HDFS 更适合 T+1 报表

6 解析 json 的函数

7 小文件治理怎么做？

方法	原理
合并写入	控制 spark.sql.files.maxPartitionBytes，合理设置分区大小
合并输出	使用 coalesce() 合并分区再写入
合并已有文件	使用 INSERT OVERWRITE 合并旧数据
压缩存储	使用 Parquet/ORC 减少写入文件数量
控制小文件产生	预先设置分区策略，减少过细颗粒度

8 上游系统崩溃了怎么办？

9 没有主键关联的两个表如何保鲜有序性

严格来说：

- 没有主键，数据就天然无序
- 分布式处理（如 Spark）本身就不保证顺序

若必须保序，可用：

- 增加时间戳字段，按时间排序
- 加入业务 ID（如 session_id、user_id）建立伪主键
- 用 window + orderBy 控制局部顺序
- 对输入数据使用 repartitionAndSortWithinPartitions() 控制分区内顺序

10 数据幂等性问题如何解决

幂等性 = **多次处理同一份数据，不会导致结果重复或异常**

方法	说明
写入使用 UPSERT（幂等写）	通过主键覆盖旧记录
设置幂等 ID	使用业务唯一标识符（如 request_id）做幂等判断
写入去重	在写入前做 distinct 或使用窗口取 latest
合并策略	使用 Hudi / Delta Lake，自动识别变更合并写入

11 下游系统如何避免数据重复问题

- 写入下游前先做去重（基于主键或业务字段）
- 保证幂等机制（如 UPSERT / MERGE）
- 增加 checkpoint、watermark 控制窗口重复计算（Flink 场景）
- 使用输出日志 + 状态标记表防止误写
- 引入输出唯一标识（如任务批次 ID）

三 数据库的处理与调优（各种 SQL）

1 最新的数据处理和存储技术

存储层（数据湖技术）：

技术	特点
Delta Lake	ACID + 历史版本追溯 + UPSERT
Apache Hudi	写少读多场景，支持实时数据更新
Apache Iceberg	弹性强，支持大宽表，兼容 Presto/Trino

分析引擎：

技术	特点
Doris	MPP 引擎，支持高并发实时 OLAP 查询
ClickHouse	高速分析数据库，列存
StarRocks	Doris 的进化版，支持更多计算场景

实时处理：

技术	特点
Flink	实时计算领导者，精准一次处理
Kafka + Flink + Doris	实时数仓组合经典架构
Streaming LakeHouse	实时流写入数据湖（Delta、Hudi、Iceberg）

2 持久化的数据库有哪些

类别	常见产品	特点
关系型	MySQL, PostgreSQL	行式存储，支持事务
列式	Doris, ClickHouse	OLAP 查询快，适合大数据分析
KV 存储	HBase, Redis（AOF/RDB 模式）	非结构化或半结构化，高并发读写
文档型	MongoDB	存储 JSON 格式，灵活
数据湖	Delta Lake, Hudi, Iceberg	支持 ACID 的 HDFS 层数据管理

3 索引的类型

索引类型	描述
普通索引（Normal Index）	最基本的索引，没有唯一性要求。
唯一索引（Unique Index）	索引列的值必须唯一，可包含 NULL。

主键索引（Primary Key）	特殊的唯一索引，不能为空且唯一。
组合索引（联合索引）	多列组合成的一个索引，常用于多条件查询。
全文索引（Fulltext）	用于全文搜索，MyISAM / InnoDB 支持。
空间索引（Spatial）	用于地理数据类型（GIS）。

4 索引的前缀原则

联合索引会优先使用从左到右连续的字段进行匹配，中间跳过某一列后，后续字段即使出现在查询条件中也无法命中该联合索引。

例：联合索引 (a, b, e)

- 能命中索引:
  - ✓ WHERE a = 1 ✅
  - ✓ WHERE a = 1 AND b = 2 ✅
  - ✓ WHERE a = 1 AND b = 2 AND e = 3 ✅
  - ✓ WHERE a = 1 AND e = 3 ✅（范围内用到了 a，所以可部分使用索引）
- **不能命中索引：**
  - ✓ WHERE b = 2 AND c = 3 ❌（a 没有出现，违反最左前缀）
  - ✓ WHERE e = 3 ❌（a、b 都缺失）

5 数据报表存储这块用过哪些产品，用过哪些存储引擎？

数据报表常用产品：

- **Hive**：基于 HDFS 的数据仓库，适合大数据离线报表。
- **ClickHouse**：OLAP 引擎，支持秒级多维分析，适合报表系统。
- **MySQL / PostgreSQL**：中小型系统中使用较多，适合明细查询和小型聚合。

存储引擎：

- ClickHouse 自带列式存储（MergeTree 等）

6 Mysql 和 hive 的区别，各自优势

对比项	MySQL	Hive
类型	关系型数据库（OLTP）	大数据仓库工具（OLAP）

处理场景	实时事务处理	海量数据分析
存储结构	行存储	列存储（Parquet/ORC）
执行方式	基于索引快速查询	将 SQL 转换为 MapReduce/Spark 任务执行
延迟	低延迟，毫秒级	高延迟，秒~分钟级
优势	实时、事务支持强、语法丰富	支持超大数据量分析、扩展性强、可与 Hadoop 生态无缝集成

7 Parquet / ORC 是什么格式？

列式存储格式，常用于 Hive、Spark、ClickHouse 等数据仓库中：

ORC（Optimized Row Columnar）：

- Hadoop 社区推出，Hive 官方推荐格式
- **压缩率高**，支持轻量索引，读取快
- 支持复杂数据类型（struct、map、array）

Parquet：

- Apache 提出的通用列式格式（由 Twitter + Cloudera 推动）
- 被 Spark、Presto、Flink 等广泛支持
- 更通用，跨平台兼容性更好

8 Mysql 两种存储引擎

InnoDB（默认）

- 支持事务、行级锁、外键、崩溃恢复
- 实现 ACID 原子性，适合高并发业务场景

MyISAM

- 不支持事务，只有表级锁
- 查询快，占用资源少，适合读多写少的场景（现在较少用）

9 项目中主从复制的缺点

主从复制用于读写分离，但也存在以下问题：

问题	说明
主从延迟	写入主库后，数据同步到从库存在延迟，可能导致读到旧数据
一致性风险	强一致性要求高的业务不能单纯依赖主从
故障转移复杂	主挂了需要手动或借助工具（如 MHA）切换角色
写入瓶颈未解决	从库不能写，写操作压力都在主库上

10 MySQL 索引原理，B+树 vs Hash 索引

索引的本质：

索引是 **加速查询的“数据结构”**，本质是 **“字段值到物理数据页地址的映射”**。

在 MySQL InnoDB 中，最常用的是：

- **B+ 树索引（默认）**
- Hash 索引仅适用于 **Memory 引擎** 或 **特定场景下的二级索引加速（如哈希辅助索引）**

B+树索引

特点：

- 多路搜索树（高扇出，常为上百~上千）
- 所有数据存储在**叶子节点**
- 叶子节点之间**有指针相连**，方便**范围查询**

优点：

优势	描述
高扇出 + 多 key 存储	节点高度低，查询快（O(logN)）
顺序存储 + 范围查询友好	叶子节点链表可顺序遍历
支持主键聚簇存储	主键与整行数据同在

适用场景：

- 大部分 InnoDB 表查询（默认索引结构）
- 范围查找、排序、分页、聚合等场景都非常适合

Hash 索引

特点：

- 通过 **Hash 函数将字段值映射为位置**
- 本质是 **key -> hash(key) -> 地址**
- 查询时直接定位，非常快

优点：

- 查询效率极高（O(1)）——等值查询非常快！

缺点：

缺陷	描述
只支持等值查询	不支持 LIKE、>、BETWEEN
无法排序	Hash 后失去顺序信息
有哈希冲突风险	多个 key 可能映射同一槽位
不适合范围查询	查询 age > 30 会全表扫

适用场景：

- **Memory 引擎默认使用 Hash 索引**
- 特定业务中对高频等值查找的字段使用哈希索引优化

B+树 vs Hash 索引 对比表

对比项	**B+树索引（InnoDB 默认）**	**Hash 索引**
查询效率	O(logN)	O(1)（等值）
支持范围查询	✅	❌
支持排序操作	✅	❌
是否支持主键聚簇存储	✅	❌
是否受哈希冲突影响	❌	✅ 有冲突
使用引擎	InnoDB	Memory
典型应用场景	普通表查询、范围查询	高速等值查询缓存表

11 为什么 B+树存储千万级数据只需要 3-4 层

背后原理：高扇出 + 每页存储多条记录

假设条件：

- B+ 树每个节点存放 1000 个 key（扇出为 1000）
- 数据总量为 10,000,000（千万级）

计算层数：

```
yaml
复制编辑
根（第 0 层） -> 1000 个 key，可指向 1000 个页
第 1 层：1000 * 1000 = 1,000,000 个 key
第 2 层：1,000,000 * 1000 = 1,000,000,000
```

- 所以最多只需 3 层就能覆盖 10^7 条数据
- 查找复杂度为 $O(\log_{1000}N)$，非常快

面试术语：

因为 B+ 树的每个非叶子节点可以存储大量 key，导致树的**高度非常低**，实际中千万级数据只需要 3~4 层，磁盘 I/O 次数极少，因此查询性能非常高。

12 Mysql 索引为什么能提高查询效率

- B+ 树结构减少了磁盘 IO 次数，从 O(n) 降为 O(logN)
- 查找数据只需几次磁盘页定位，而非全表扫描
- 使用联合索引 + 最左前缀原则可以高效定位复杂条件
- 覆盖索引可避免回表，进一步提升性能

13 什么事实务操作？如何保证一次提交的是事务操作？

事务是数据库中用于保证 **数据一致性和安全性** 的基本单位。

事务（Transaction）的定义：

一组 SQL 操作，要么全部成功（提交 COMMIT），要么全部失败（回滚 ROLLBACK）。

事务的四大特性（ACID）：

ACID 就是事务的核心属性：

缩写	含义	举例
A - 原子性	要么全做，要么全不做	转账操作不能只扣钱不加钱
C - 一致性	执行前后数据都合法	外键/约束必须满足
I - 隔离性	并发操作彼此隔离	张三查账不受李四转账影响
D - 持久性	提交后数据永久保存	宕机重启数据不丢失

如何执行事务操作（以 MySQL 为例）：

```
Sql
START TRANSACTION;  -- 显式开启事务
UPDATE accounts SET balance = balance - 100 WHERE user_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE user_id = 2;
COMMIT;  -- 提交事务

-- 如果出错：
ROLLBACK;  -- 回滚事务
```

- 事务功能 **只有 InnoDB 引擎支持**，MyISAM 不支持事务。
- 一般在金融、电商类系统中大量使用事务来确保数据一致性。

14 MVCC

定义：

MVCC（Multi-Version Concurrency Control）是 **InnoDB 实现并发控制的一种机制**，用来实现**读写不阻塞（即：读不加锁）**，提高并发性能。

核心机制：

- 每行记录都维护两个隐藏字段：**trx_id（创建版本）+ roll_pointer（回滚指针）**
- 查询时会根据当前事务的 Read View 判断是否能“看到”该记录

适用隔离级别：

- **Repeatable Read（可重复读）**：InnoDB 默认，依赖 MVCC 实现快照读
- 不适用于 Serializable 或 Read Uncommitted（前者加锁，后者不隔离）

15 什么是聚簇索引和非聚簇索引

聚簇索引（Clustered Index）：

- **数据和主键索引存储在一起**（数据就是索引）
- 一张表只有一个聚簇索引（通常是主键）

非聚簇索引（Secondary Index）：

- 索引和数据 **分开存储**
- 索引页存储的是 key + 主键值（而不是行地址）

举例：

```
sql
复制编辑
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  age INT,
  INDEX idx_name (name)
);
```

- id 是聚簇索引（数据按 id 顺序存储）
- idx_name 是非聚簇索引，需要通过 **回表** 找到完整数据

16 主键索引和二级索引的查询过程

主键索引查询流程（聚簇索引）：

- 直接在 B+ 树中查找主键
- 找到后，B+ 树叶子节点上**就存储了整行数据**

二级索引查询流程：

- 首先查找二级索引（如 name 索引）
- 拿到匹配项的 **主键 id**
- 再根据主键去主键索引的 B+ 树上“**回表**”找完整数据

17 主键索引如何构建 B+树

构建原则：

- **主键作为 key，整行数据作为 value**
- 数据页是按照主键顺序排序的（物理上排序）
- 每个 B+ 树节点最多可存放多个 key（页大小一般为 16KB）

18 什么是跳表？为什么 mysql 不适用跳表

19 索引失效有哪些？

失效原因	示例
使用函数	WHERE LEFT(name, 3) = 'Tom'（失效）
隐式转换	WHERE id = '123'（id 是 int，'123' 是字符串）
使用 or 连接多个字段	WHERE a = 1 OR b = 2（没有联合索引）
模糊查询前缀	LIKE '%abc'（前面有通配符）
索引字段参与运算	WHERE age + 1 = 20
联合索引不满足最左前缀原则	WHERE b = 2（索引 a,b）时失效

使用 EXPLAIN 可以检查是否命中索引。

20 回表查询

定义：

当我们用 **二级索引查询**时，只能拿到主键值，再通过主键**回到聚簇索引中获取整行数据**，这个过程叫“回表”。

21 非聚簇索引中的字段只存储了主键值吗？

是的，InnoDB 的 **二级索引（非聚簇索引）** 的叶子节点只存储：【索引字段的值】+【对应主键值】

22 对数据治理的理解

1) 数据标准治理
    a) 字段命名规范（如 user_id, order_id）

b) 指标口径定义（如 GMV、UV）

2) 数据质量治理

a) 空值校验、数据类型校验、业务逻辑校验

b) 报表准确率监控、定时比对、DQC 平台

3) 数据血缘与可追溯

a) 建立数据血缘图（上游 → 下游）

b) 支持 lineage 回溯、定位问题源头

4) 权限与安全管理

a) 数据脱敏、分级授权

b) 敏感字段加密（如身份证、手机号）

5) 数据资产与指标管理

a) 建立指标平台，统一定义指标

b) 建数据资产平台/数据地图（Data Catalog）

23 说一下为什么使用 LookUpJoin

24 什么情况下可以使用 map join

Map Join 又叫 Broadcast Join（广播小表到所有 Mapper）

使用条件：

- 一张表非常小（几百 KB ~ 几 MB）
- 另一张表很大（大表不广播）
- 在 Hive 中通过 /*+ MAPJOIN(b) */ 或配置 hive.auto.convert.join=true 启用

优点	说明
不需要 Shuffle	避免大表传输，减少网络开销
查询效率高	小表直接加载进内存，在 Map 阶段就完成 Join
适合星型模型	常用于事实表 + 多个维度小表关联

25 如何解决大表 join 大表的数据倾斜

方法	说明
✅ **加随机前缀**	给热点 key 加前缀打散，Join 后再去除
✅ **拆分 Join**	将热点 key 单独处理，其他 key 正常 Join
✅ **改为 Map Join + 缓存**	将热点数据缓存广播避免大表 Join
✅ **使用 Skew Join（Hive 支持）**	Hive 自动检测倾斜并优化执行计划

✅ **扩大并发数**	提高 reduce task 数量分散计算压力

26 大表和小表 join 时候的优化方法？

优化核心目标：利用 小表广播、大表避免 Shuffle。

方法	描述
✅ Map Join / Broadcast Join	小表广播到所有任务节点，减少 Shuffle
✅ Join 顺序调整	确保小表在前（Hive 中为右侧）才能广播
✅ 使用 Hint 提示	Hive 中使用 /*+ MAPJOIN(small_table) */
✅ 小表提前过滤	避免无效数据参与广播
✅ 小表缓存到内存	Spark/Flink 支持 cache 小表，减少 IO

27 Bucket join 优化原理

Bucket Join 是 Hive 提供的一种 Join 优化方式，用于**提前对数据分桶，减少 Join 阶段数据 Shuffle**。

原理：

- 将大表和小表按照 **相同的字段进行分桶（bucketed by）**
- Join 时，只需要 Join **相同桶号的数据文件**
- 因为数据预分区了，**避免了 runtime Shuffle**，执行更快

使用要求：

- 两表必须按同样的字段 bucketed by 且 sorted by
- 两表 bucket 数需一致（或整倍数）

优点：

优势	描述
避免 Shuffle	提前分桶让 Join 变成本地文件匹配
加快 Join	减少数据传输和排序
适合大表间稳定字段 Join	如 user_id、order_id

28 Sql 数据倾斜怎么办？

方法	描述

✅ **加随机前缀**	将热点 key 如 user_id = 123 拆为 123_1, 123_2 多份处理后汇总
✅ **拆分热点 key 处理**	将数据倾斜的 key 单独用特例逻辑处理，其他正常处理
✅ **使用 Skew Join / Hive skewjoin 优化**	自动处理倾斜 key（Hive/Spark 支持）
✅ **过滤无关数据**	提前过滤减少参与计算的无用数据
✅ **提高并发**	增加 reducer 数量，分散计算压力

29 SparkSQL 执行过程

SQL -> Logical Plan -> Optimized Logical Plan -> Physical Plan -> DAG -> Task -> 执行

详细阶段：

1. **解析器（Parser）**：SQL 语句转换为初始 Logical Plan
2. **分析器（Analyzer）**：绑定字段、类型校验
3. **优化器（Catalyst Optimizer）**：规则优化（如谓词下推、常量合并、子查询优化）
4. **物理计划生成（Planner）**：生成多种物理执行方案，选成本最低者
5. **DAG Scheduler**：将物理计划转为 RDD DAG
6. **执行器（Executor）**：分布式执行

30 如何设计数据报表的存储，MYSQL 已经不能用了，查询效率太低，这时候如何存储？

替代方式	说明
✅ **引入 OLAP 引擎**	ClickHouse / Doris / StarRocks：列式存储，秒级多维分析
✅ **引入缓存层**	Redis / Presto / Materialized View 提前缓存报表数据
✅ **构建数据仓库分层（ODS/DWD/DWB/ADS）**	拆分预聚合任务，避免查询时计算
✅ **存储格式优化**	Hive 表使用 ORC/Parquet 压缩格式，适配 Spark/Presto 查询
✅ **维度建模**	设计星型/雪花模型，优化字段冗余、表连接复杂度

31 Sql 查询性能中慢查询、快查询、数据倾斜的问题

类型	特征	原因	解决思路

慢查询	执行 > 秒级	未命中索引、全表扫描、Join 多表	建索引、优化 SQL、拆分子查询
快查询	秒内响应	索引命中、预聚合、少 Join	查询小表或命中缓存
数据倾斜	查询时间不均	key 分布不均、大量 NULL 或 0	加前缀、拆分逻辑、使用 skew join

32 sql 中的 LATERAL VIEW

LATERAL VIEW（Hive / Spark 专属）

作用：

用于处理复杂字段结构如数组、map、struct 的展开操作，特别是配合 explode() 函数，将嵌套结构扁平化为多行。

33 Mysql 中的锁

34 mysql 乐观锁

35 Mysql 如何避免重复插入数据

36 什么是跳表？为什么 mysql 不适用跳表

37 Mysql 怎么解决并发问题

38 元数据锁

39 X 类型的锁是什么

40 Hivesql 查询时间长，怎么优化

类型	说明
全表扫描	缺乏分区，查询无法过滤数据
数据倾斜	某些 key 过于集中，导致 reduce 压力不均
大量 Join 操作	Join 未调优，未使用 map join 等优化
任务并发低	Map/Reduce 数量不足，资源使用率低
文件过小或过多	小文件过多造成 NameNode 负担和 I/O 频繁
没有使用 ORC/Parquet	文本格式处理开销大

优化方向	操作建议
✅ **加分区过滤**	使用 WHERE dt='2024-01-01' 启用分区裁剪
✅ **使用列式存储格式**	优先选择 ORC、Parquet，支持压缩 + 向量化

✅ **Map Join 优化**	小表广播，避免 reduce shuffle
✅ **控制文件数量**	合并小文件，提高 HDFS 读写效率
✅ **数据倾斜处理**	使用 skew join 或加随机前缀打散热点 key
✅ **并发参数调优**	提高 mapreduce.reduce.tasks, tez.grouping.max-size

41 Hive 上大量数据流转到 sql 上，有个字段是 json 结构体，怎么操作

方法一：get_json_object（常用）

```
sql
SELECT
  get_json_object(json_col, '$.user.name') AS user_name
FROM your_table;
```

- 适用于提取 JSON 中某个字段的值
- 返回值为 string

方法二：json_tuple（提取多个字段）

```
sql
SELECT
  json_tuple(json_col, 'user_id', 'age') AS user_id, age
FROM your_table;
```

- 一次提取多个字段
- 更高效，返回多个列，字段名顺序对应

方法三（结构体处理）：使用 lateral view + explode(json_array) 处理嵌套数组

```
sql
复制编辑
SELECT user_id, tag
FROM your_table
LATERAL VIEW explode(split(get_json_object(json_col, '$.tags'), ',')) AS tag;
```

42 Mysql 千万级数据慢查询怎么分析

1）定位慢 SQL

- 查看慢查询日志：

```
bash
SHOW VARIABLES LIKE '%slow_query%';
```

- 启用慢日志：long_query_time=1

2）使用 EXPLAIN 分析执行计划

sql

```
EXPLAIN SELECT * FROM orders WHERE user_id = 123;
```

重点关注字段：

字段	说明
type	联接类型（all=全表，ref=使用索引）
key	使用的索引
rows	扫描行数
extra	是否使用临时表 / filesort 等不利提示

3）优化建议：

场景	优化方式
没有走索引	创建合适索引（单列或联合索引）
联合索引未命中	调整 WHERE 顺序满足最左前缀原则
使用函数	避免在索引列上使用函数/运算
分页慢	避免深分页，使用延迟游标方式优化 LIMIT
表太大	垂直拆分/归档老数据

43 Group by 和 partition by 的区别、场景

对比项	GROUP BY	PARTITION BY
功能	聚合整组数据为一行	划分分区后在分区内逐行保留
常用于	聚合统计（SUM、AVG、COUNT）	窗口函数（ROW_NUMBER、RANK）
结果行数	少于或等于原数据行数	等于原数据行数
示例函数	SUM(salary)	ROW_NUMBER() OVER(PARTITION BY dep ORDER BY salary)

44 Hive 内部表和外部表的区别？

区别项	内部表（Managed Table）	外部表（External Table）
数据管理	Hive 全权管理	数据由外部系统或用户管理
DROP TABLE	删除元数据 + 数据文件	只删除元数据，不删除数据文件

创建语法	CREATE TABLE	CREATE EXTERNAL TABLE
使用场景	临时数据、中间结果表	数据共享、数据导入外部管理系统
默认位置	/user/hive/warehouse/	用户指定的 HDFS 路径

45 Hive 怎么调优

优化方向	调优措施
分区裁剪	增加分区字段，WHERE 子句写在分区字段上
小表广播	使用 MAPJOIN 或开启 hive.auto.convert.join=true
文件合并	使用 hive.merge.smallfiles 合并小文件提高读写性能
压缩格式	存储使用 ORC / Parquet，配合向量化读取
并发调优	设置 mapreduce.reduce.tasks, tez.task.resource
数据倾斜	对热点 key 加前缀打散，或使用 hive.skewjoin
缓存中间结果	使用 INSERT OVERWRITE TABLE tmp_table 减少重复计算

46 hive 中的分区和分桶有什么区别

共同点：都是为了 加快查询效率、减少扫描量

项目	分区（Partition）	分桶（Bucket）
本质	数据目录划分	数据文件划分
物理结构	每个分区一个 HDFS 目录	每个桶一个文件（一个分区内多个桶）
使用字段	明确字段，如 dt, province	一般是 id, user_id 等 hash 分布字段
查询过滤	WHERE partition_col = ...	Hive 会根据 hash 函数定位桶（较少用）
优点	减少扫描数据量（分区裁剪）	优化 Join（如 Bucket Join）、均衡计算负载
建表语法	PARTITIONED BY(dt STRING)	CLUSTERED BY (user_id) INTO 4 BUCKETS
典型场景	按天/省份划分	高效 Join、倾斜优化、大表多表分桶 Join

四 数据采集与同步

1 Kafka 和 Flink 在实时计算中的作用？

Kafka 的作用：实时数据通道 / 缓冲区

作用	描述
🛠 数据解耦	上游写入、下游消费异步解耦
🚦 流控缓冲	抗压能力强，天然流控机制

作用	描述
🔁 多订阅	一个 Topic 可被多个消费者消费
📦 存储保证	持久化日志 + offset 提供消费保障

✅ *Flink 的作用：实时计算引擎 / 状态处理机*

作用	描述
🧠 实时处理	实时 ETL、聚合、窗口计算、JOIN 等
📍 状态管理	Checkpoint + StateBackend 实现 Exactly-once
🔍 事件驱动	支持事件时间、乱序处理、水位线机制
🎛️ 多源输出	可写入 Kafka、ES、ClickHouse、MySQL 等多种 Sink

2 DataX 执行原理

原理框架：

```
text
复制编辑
Reader（读取器） → Channel（缓冲） → Writer（写入器）
```

✅ *执行流程：*

1. **配置文件解析**：读取 JSON 配置（源、目标、字段映射）
2. **Reader 启动**：如 `mysqlreader` 拉取数据块
3. **Channel 缓冲传输**：默认内存通道，也支持 stream/file
4. **Writer 写入**：如 `hdfswriter`、`postgresqlwriter` 写入目标表
5. **插件机制**：Reader/Writer 为可插拔式组件

3 Kafka 消费能力不足如何处理？

问题表现：

- 消费延迟积压
- Lag 累积快速上升
- Flink/消费者处理速度慢于生产速度

✅ *解决手段：*

手段	说明
✅ 提高并发	增加 Partition 数量 + 消费组并发数
✅ 优化消费者逻辑	减少 IO 操作、批量处理、异步 Sink
✅ 启用异步消费	如 KafkaConsumer 设置 `enable.auto.commit=false`
✅ 调整批量策略	提高 `fetch.min.bytes` 和 `fetch.max.wait.ms`
✅ 增设 Kafka 集群资源	硬件资源不足时需扩容 Topic / Broker

4 用 kafka 做了什么？为什么要用 kafka

实战应用场景：

场景	使用 Kafka 的方式
📝 日志采集	Logstash/Filebeat 收集日志入 Kafka，供下游消费
💡 实时 ETL	Flink 从 Kafka 消费事件流，实时加工入仓
📊 实时指标	Kafka → Flink → ClickHouse 实时指标系统
🔄 多系统解耦	Kafka 作为中间件连接多个系统（APP → 分析平台）

✅ *为什么用 Kafka？*

优势	描述
✅ 高吞吐	支持百万级 TPS
✅ 解耦	多消费者模型
✅ 持久化	可设置数据保留时间
✅ 宕机容错	offset 控制确保容错恢复
✅ 多端一致	多语言/平台支持好（Flink、Spark、Java 等）

5 kafka 数据丢失如何解决

常见丢失场景：

场景	描述

场景	描述
❌ 没有开启 ACK	Producer 端未确认就结束发送
❌ offset 提交过早	消费逻辑未完成就提交 offset
❌ Broker 崩溃 / 物理宕机	无副本、磁盘未持久化导致数据丢失
❌ 分区 leader 未同步	ISR 不完整导致写入不一致

✅ ***应对策略：***

手段	描述
✅ Producer 设置 `acks=all`	保障所有副本写成功再返回
✅ Broker 设置 `min.insync.replicas > 1`	写入必须有多个副本才成功
✅ 启用幂等性	设置 `enable.idempotence=true` 防止重复写入
✅ 消费端手动提交 offset	处理完逻辑后再提交，配合 checkpoint
✅ 增加 ISR 同步机制	防止副本不一致带来的数据断链

6 你在使用 kafka 时会关注什么参数

类型	参数	说明
✅ Producer 参数	`acks`	推荐设置为 `all` 保证副本一致
	`retries`	失败重试次数（避免消息丢失）
	`batch.size` / `linger.ms`	控制发送批次大小与等待时间，提高吞吐
	`enable.idempotence`	开启幂等性，避免重复发送
✅ Broker 参数	`num.partitions`	分区数量，影响并发能力
	`min.insync.replicas`	最小同步副本数，保障数据可靠性
	`log.retention.hours`	数据保留时长，影响容量与恢复能力
✅ Consumer 参数	`auto.offset.reset`	`earliest` / `latest`，控制消费起点
	`enable.auto.commit`	是否自动提交 offset，推荐关闭手动控制
	`max.poll.records`	每次 poll 消费的最大消息条数

7 kafka 中削峰的使用

场景：上游数据突发写入、下游处理能力有限

✅ *Kafka 削峰作用：*

- 利用 **Topic + 分区** 缓冲机制
- 实现 **流量错峰写入，平稳消费处理**

✅ *削峰方式：*

方式	描述
✅ Kafka 本身作为削峰组件	上游快速写入 → Kafka 存储 → 下游慢速消费
✅ 控制 Flink Sink 速率	利用异步 Sink + BackPressure 控制
✅ 提高 Partition 数	增强并发消费能力，降低单位消费压力
✅ 设置消费延迟告警	利用 Lag 指标提前发现堆积风险

8 Kafka 怎么做增量同步

本质：Kafka 承接增量变更事件（如 Binlog）

实现方式：

步骤	描述
✅ 数据源开启 Binlog	如 MySQL 配置 row 格式 Binlog
✅ 使用 Maxwell / Canal / Debezium 等组件	实时读取 Binlog 并写入 Kafka Topic
✅ 下游 Flink/Spark 消费 Kafka Topic	提取 insert/update/delete 类型做增量处理
✅ Sink 入目标系统	如 HBase、ClickHouse、Doris、ES 等

9 Maxwell 和 kafka 对比

维度	Maxwell	Kafka

维度	Maxwell	Kafka
类型	Binlog 解析工具	消息中间件 / 数据通道
数据来源	MySQL Binlog	任意系统推送数据
数据格式	JSON / Canal 格式事件流	任意二进制或 JSON 消息
功能	增量采集 / CDC 工具	实时传输、削峰、解耦
是否替代	✕ 不能替代 Kafka	✅ 是 CDC 工具的下游核心组件之一

10 Maxwell 和 binlog 如何合作

流程：

```
text
复制编辑
MySQL 开启 Binlog（row 格式）
    ↓
Maxwell 连接 MySQL，解析 Binlog（增量变更）
    ↓
转换为 JSON 结构消息
    ↓
发送到 Kafka Topic
    ↓
Flink/Spark/ETL 系统消费 Kafka，做入仓或更新
```

📌 注意事项：

项	要点
Binlog 必须开启 row 格式	Maxwell 不支持 statement 格式
MySQL 用户权限需有 replication	Maxwell 读取 binlog 需该权限
Maxwell 会记录 offset	保证消费位点恢复
支持 insert/update/delete 三种变更	可用于构建 UPSERT 流程

11 数据采集需要注意什？

注意点	说明
✅ 数据源权限	是否有读权限（Binlog、表结构）
✅ 网络稳定性	远程连接数据库/接口是否有断连重试机制
✅ 数据格式一致性	JSON、CSV、NULL 值、空字符串处理规范化
✅ 编码处理	防止乱码（UTF-8/GBK 混用）
✅ 异常容错	是否支持跳过坏数据或记录失败日志

注意点	说明
✅ 采集完整性校验	源目标行数比对、校验 hash 码

12 kafka 核心组件，如何保证数据有序

核心组件：

组件	描述
Producer	发送数据至 Topic
Broker	Kafka 服务器，负责存储 Topic 数据
Topic / Partition	主题 + 分区，实现并发与负载均衡
Consumer / Group	消费数据，Group 实现负载均衡
Zookeeper / KRaft	管理元数据、选主（Kafka 2.8 起支持无 ZK 模式）

✅ *数据有序保证方法：*

方法	说明
✅ 单分区写入	Producer 将相同 Key 写入同一 Partition，分区内部有序
✅ 发送按顺序	单线程顺序发送、不开启批量合并或异步写
✅ Consumer 单线程消费	保证分区内有序读取处理
✅ 不跨 Partition 聚合	聚合任务尽量分区内完成，否则需自定义排序

13 Datax 怎么解决数据依赖问题

问题场景：

多个表或任务之间存在上下游依赖关系（如维表先更新，事实表才能入仓）。

✅ *解决方法：*

方法	描述
✅ 任务调度平台控制依赖	使用 Airflow / Azkaban / DolphinScheduler 等工具配置任务依赖图
✅ 手动拆分任务流程	将多个表拆为单任务，严格顺序执行

方法	描述
✅ 加标志位文件 / 检查数据是否就绪	判断上游数据是否存在（如某分区已落地）后再执行下游任务
✅ 检查上游导入成功行数 / 校验值	脚本中通过 SQL 进行依赖检测，确保数据完整性后继续执行

14 Datax 跨系统数据同步问题

问题挑战：

- 数据类型不兼容（如 varchar → string、timestamp → string）
- 编码格式不同（MySQL 为 UTF-8，Hive 为默认编码）
- 分区表字段处理复杂
- 目标系统写入机制不一致（Hive 无 insert overwrite）

✅ 解决方案：

问题	处理方式
类型兼容	ader 中统一为 string 类型，或 writer 中做类型映
/NULL 处理	DataX 参数 nullFormat 和 defaultValue
表写入	临时表/中间目录，调度后执行 HQL 插入分区
方式不支持 ap	多为覆盖模式，可用分区覆盖规避重复
络访问慢	用中间层缓存（如 HDFS 临时落地，再导入 Hi

15 跨系统如何判断数据已经同步完成

通用判断方式：

方法	描述
✅ 分区检查	Hive / HDFS 是否存在指定分区数据（如 dt=20240516）
✅ 行数对比	源表与目标表行数对比（特别是全量表）
✅ 校验字段总和/哈希值	对字段做汇总（如 sum）或 md5 哈希后比对
✅ 标志位文件	数据同步完成后写 _SUCCESS 文件作为标志
✅ 日志/回执检查	Kafka + CDC 系统中查看 offset 消费状态是否一致

16 Kafka 如何保证幂等性

幂等性本质：多次发送同样的消息，只被消费一次且内容一致

✅ *Kafka Producer 幂等机制：*

参数	作用
`enable.idempotence=true`	开启幂等性，默认 Kafka 保证 “exactly-once per partition”
`acks=all`	所有副本都写入成功才返回 ack
`retries > 0`	保证失败自动重试
自动使用事务 ID + 序列号	Kafka 在 broker 端记录 producer 的序列号，避免重复写入

17 Kafka 在日志故障的时候如何保证幂等性

问题场景：

- Broker 崩溃 / 重启
- Producer 重试机制触发
- 磁盘损坏导致消息状态不一致

✅ *应对机制：*

机制	描述
✅ `acks=all` + ISR 限制	仅当所有副本写成功才确认 ack，防止主副本失步
✅ `min.insync.replicas ≥ 2`	提高高可用保障
✅ 幂等序列号记录在 Broker 上	重启后仍能识别重复写入
✅ 配合事务机制使用	Producer 提交/回滚事务防止半成功状态
✅ 存储层使用 WAL 日志	Kafka 自身日志顺序写+持久化保证宕机恢复能力

18 数据采集、清洗、分析的方法和原理

通用三段流程：

---

① 数据采集

方法	工具/手段
接口采集	HTTP + Python/Shell 脚本
数据库采集	JDBC + DataX / Sqoop
日志采集	Flume / Filebeat / Logstash + Kafka
Binlog 采集	Maxwell / Canal → Kafka/Flink

---

② 数据清洗

操作	描述
空值处理	替换、删除、填充（均值/众数）
类型转换	统一数据格式（int → string）
去重、异常值检测	hash 去重、3σ/箱型图检测
标准化字段	字段命名规范、编码格式统一
数据补全	与维表 join 填充字段缺失内容

---

③ 数据分析

分析方法	场景
聚合分析	用户数、GMV、访问次数等汇总
分组统计	各省市销售对比、各渠道转化率
漏斗/留存分析	用户行为路径/生命周期分析
建模分析	回归、聚类、分类等高级预测模型
可视化	用 Tableau、Echarts、FineBI 展示结论

19 如何从源数据同步到数仓，同步策略，如何更新，为什么 DataX 不可以同步增量表，maxwell 可以同步全量表吗？为什么要用两种组件同步全量表和增量表？

同步策略：

类型	说明	工具建议
同步	全表覆盖	X

类型	说明	工具建议
同步（基于字段	时间戳/主键等字段拉取增量	X（仅支持静态字段
同步（基于 Binl	监听变更事件	well / Canal / Flink (

✅ DataX 不适合增量同步的原因：

- 不支持监听 binlog 变更，只能通过字段条件筛选（如 update_time > xxx）
- **更新/删除操作会遗漏**，仅适合追加型全量/静态表

✅ Maxwell 不能同步全量表的原因：

- 只能监听增删改（binlog 事件）
- 启动时不读取已有表数据，只抓后续变更

✅ 为什么要用两种工具混合同步？

各有优缺点，职责不同：

场景	工具	理由
全量	X	性拉取全表
增量	well	binlog 的高效变更
更新	well + Sink 处理 UPSERT 逻辑	
合替	X 不能做实时，Maxwell 不能	

五 数据质量与监控

1 如果某天发现报表数据异常，如何排查？

步骤	检查项	举例
1️⃣ 报表层	是否是图表渲染错误、字段名错用？	e.g. 数据展示错误但表中无误
2️⃣ ADS 层	查询是否返回异常值？是否有空？	汇总错误、字段为 null
3️⃣ DWS 层	宽表指标是否异常？口径是否改动？	如 GMV 汇总少了优惠金额

步骤	检查项	举例
4️⃣ DWD 层	原子事实表是否缺失？	比如当天订单没入仓或被过滤了
5️⃣ 源系统	源表是否有延迟或未生成？	MySQL 表没有当天数据或接口挂了
6️⃣ 调度链路	是否任务失败、依赖断了？	DolphinScheduler / Airflow 中断

2 清洗什么样的异常值，如何清洗？

常见异常值类型及清洗方式：

异常类型	示例	清洗方式
❌ 空值 / NULL	金额为空、缺省字段	填充默认值、删除、插入占位符（如 “未知”）
❌ 格式不合法	手机号、时间戳格式错误	正则校验后替换/过滤
❌ 异常数值	订单金额为负数、年龄 > 200	阈值判断、箱线图/3σ 检测后剔除或置为中位数
❌ 重复值	订单号重复、主键重复	hash 去重 / window 函数 row_number() 筛选
❌ 逻辑冲突	创建时间 > 支付时间	逻辑校验 + 回滚标识处理
❌ 脏数据	json 结构错误、乱码	封装 parse 校验函数后过滤掉失败行

3 你们是怎么保证数据质量的？

保障机制：

层级	措施
🧱 源头控制	明确字段规则、接口结构稳定、及时更新文档
🧹 清洗流程	增加异常校验、值范围限制、类型强校验
📦 入仓校验	空值/唯一性校验、分区对齐、主键冲突校验
🔍 指标监控	每日 GMV/活跃数同比、环比监控、自动报警
📊 DQC 规则	配置质量规则：非空、唯一、行数、值范围等
🔁 回流对账	与源系统对比核心指标，如订单笔数、金额总值等

4 数据质量监控的角度？

角度	说明
✅ 数据值角度	字段是否为空、是否异常跳变、是否重复
✅ 数据结构角度	表结构变动、字段缺失、类型变化
✅ 数据流转角度	分区是否齐全、任务是否准点、ETL 是否成功
✅ 指标角度	与历史数据同比、环比，是否跳动
✅ 数据对账角度	与源系统指标（如订单量、金额）是否一致

5 DQC 里面具体校验了什么东西

类型	校验项
✅ 空值校验	某字段是否为 null、空字符串、0 值
✅ 唯一性校验	主键是否唯一、有无重复行
✅ 行数校验	是否与源表行数一致，或与昨日持平
✅ 业务范围校验	数值是否在合理区间（如年龄 [0, 120]）
✅ 分区完整性	是否成功落地当天分区 dt = '2024-05-16'
✅ 关联性校验	维表字段是否匹配，如省份 ID 与名称是否能 join 上
✅ 异常波动检测	环比/同比跳变是否超出 3σ 范围

6 如何保证数据质量的规范

层面	内容
✅ 命名规范	表名、字段名统一命名规则（驼峰/下划线）
✅ 类型规范	金额类字段统一为 decimal，时间统一 timestamp
✅ 分层规范	ODS → DWD → DWS → ADS 严格层级定义
✅ 开发规范	SQL 模板化、调度任务命名统一
✅ 校验规则标准化	常用 DQC 模板、指标异常监控模块沉淀复用
✅ 字典/元数据平台	字段含义、维度指标说明明确、可搜索

7 零点漂移

*定义：**在定时执行的数据报表/指标中，**每日“0 点数据”与实际应有的数据偏移、缺失、延迟**的现象。*

📌 产生原因：

原因	描述
⏰ 任务调度延迟	0 点任务失败或延迟跑完
📦 上游数据未及时落地	ODS → DWD 延迟，DWS 无法汇总
🧯 DQC 执行滞后	报错未及时捕捉、分区为空未报警
🧩 数据源抖动	MySQL 写延迟、Kafka Lag 累积

8 为什么要解决零点漂移问题？

影响	说明
📉 报表数据不准	业务看到的数据不完整、决策错误
⚠️ 错误报警频发	异常监控大量误报，影响运维效率
😡 信任度下降	数据团队失去业务团队信任
📊 留存/趋势类指标偏离	用户新增、GMV 等分析失真
📅 跨天追溯困难	后续补数据后无更新机制，形成数据孤岛

六 数仓建模与分层

1 说一下数仓项目的架构

模块	说明
数据源	业务数据库（MySQL、Oracle）、日志、API 等
ODS（原始数据层）	保留原始数据结构、字段清洗、字段脱敏，支持回溯
DWD（明细数据层）	按业务主题建模后的明细表（宽表/拉平）
DWB（汇总数据层）	基于 DWD 聚合形成宽表（天、周、月级别）
ADS（应用数据层）	面向报表的主题数据，如用户画像、营收统计等
BI 展现层	ECharts、Tableau、FineBI 等展示工具

2 说一下数仓分层及其作用

分层	英文缩写	作用
ODS	Operational Data Store	保留原始数据，全量/增量同步，便于数据追溯
DWD	Data Warehouse Detail	业务过程明细层，主题化处理，字段标准化
DWB	Data Warehouse	汇总层，按照统计口径生成各类汇总宽表

分层	英文缩写	作用
	Business	
ADS	Application Data Store	应用层，直接服务报表系统、接口系统或运营平台

每层都解决什么问题？

- **ODS → DWD**：结构统一、标准字段命名，数据脱敏清洗
- **DWD → DWB**：减少查询压力，提前聚合处理
- **DWB → ADS**：面向具体业务场景优化展示结构

3 维度建模和范式建模的区别和联系

项目	维度建模（维度+事实表）	范式建模（第一/第二/第三范式）
目标	面向查询优化	面向数据一致性、冗余控制
模型	星型模型、雪花模型	一张张规范化表
使用场景	BI、OLAP 分析系统	OLTP 系统（如业务库）
冗余	有一定冗余（利于查询）	尽量无冗余
可读性	高，可视化友好	抽象、技术性强

  - 都是数据建模方法，目标不同

- 在实际项目中，**常将业务系统范式建模数据，通过 ETL 转换为维度建模模型（数仓）**

4 数据库三大范式

范式	条件	作用
1NF（第一范式）	字段不可再分（原子性）	保证数据基本结构合理
2NF（第二范式）	满足 1NF，且每个字段完全依赖主键	消除部分依赖，减少冗余
3NF（第三范式）	满足 2NF，且字段不依赖其他非主属性	消除传递依赖，提高数据一致性

5 DWD 层和 DWS 层的区别

对比项	DWD（明细数据层）	DWS（汇总/宽表层）
作用	业务过程的明细数据（细粒度）	维度聚合的宽表，适合建模分析
数据粒度	原始行为级，如一条订单、一笔支付	聚合层级，如每日用户维度汇总
表结构	主题拆分、字段规范化	按业务指标建宽表，字段打平
存储方式	规范化结构，适合重用	面向查询和分析优化，可能冗余
用途	提供通用计算中间层	直接服务于 ADS 或 BI 系统

6 事实表有哪些，为什么要设置周期快照事实表

类型	说明
事务型事实表	一条记录代表一次事件，如下单、支付
周期快照事实表	定期记录一次汇总快照，如每日库存、账户余额
累积型事实表	跟踪一个过程的整个生命周期（有开始、结束）
聚合事实表	从事务表按维度聚合而来，减少查询成本

为什么要设置周期快照事实表？

- 解决**随时间变化的状态型数据分析问题**
- 无法通过简单聚合还原的数据（如每天库存余量）
- 支持周期环比、同比分析

7 说一下 ADS 层完成的比较难的需求及其实现（离线数仓的）

举例：用户消费漏斗分析

需求描述：

- 统计每个渠道的用户，在 7 天内是否完成 注册 → 浏览 → 下单 → 支付
- 输出转化率路径、人数分布

实现方式：

1. 从 DWD 层提取相关行为数据（注册、浏览、下单、支付）
2. 使用 `user_id + event_time` 进行事件串联
3. 使用窗口函数判断转化顺序和时间范围
4. 聚合各步骤完成人数，计算转化率

5. 将最终结果写入 ADS 层表 ads_user_funnel_7d

8 对数据建模的理解

据建模的核心含义：

数据建模是将业务流程与指标维度进行抽象，用结构化表模型表示业务逻辑的过程。

类型	内容
概念建模	明确业务主题、实体、指标，如订单、用户、支付等
逻辑建模	维度与事实表的组织形式，建立星型/雪花模型
物理建模	表字段定义、分区字段设计、文件格式设置

9 数仓分层与指标分类之间有什么关系

分层	指标分类	描述
DWD	原子指标	最基本的行为数据，如“下单金额”、“浏览次数”
DWS	衍生指标	聚合而来的中间层指标，如“月均订单数”
ADS	展现指标	按报表口径组织的主题指标，如“当日 GMV”、“人均消费”

10 什么是总线矩阵

总线矩阵是维度建模中的一个核心概念，用于**规划维度与业务过程之间的关系**。

业务过程（事实表）	维度：时间	维度：用户	维度：商品	维度：渠道
下单	✅	✅	✅	✅
浏览	✅	✅	✅	✅
支付	✅	✅	✅	✅

用途：

- 规范所有业务主题下需要使用的维度
- 保证维度口径一致（统一的“用户维度”、“渠道维度”）
- 为后续建模提供参考，避免重复开发

11 构建业务总线矩阵的目的和注意事项？梳理总线矩阵最难的地方？

目的：

- 明确 **哪些维度作用于哪些业务过程**
- 规范维度粒度、命名规则，确保跨主题一致性
- 指导建模时的维度复用、模型复用

注意事项：

项目	说明
统一	要能适配多个主题，如“用户”维度不可混用会员 ID 与匿名
命名统	同义异名，如 area_code vs province_id
可复用	、商品、时间、渠道等核心维度应抽象为通用模型
可扩展	未来扩展，如新业务接入或新的维度口径

最难的地方：

梳理跨系统的业务过程，确认**每个事实表的粒度与核心维度绑定关系**。尤其在集团级系统中，不同 BU 对“用户”、“渠道”的理解可能不同，统一口径极具挑战。

12 主题域划分

主题域划分（维度建模中）：

将整个业务系统的指标 / 数据按功能、组织、业务过程划分成若干模块

比如：

- 用户行为域（用户浏览、点击、下单）
- 商品域（商品基本信息、类目）
- 订单域（订单主表、明细表）
- 营销域（优惠券、活动数据）

主题域划分是构建数仓时的第一步，有助于代码逻辑和数据责任归属清晰。

13 为什么划分成五个域？

五个主题域一般是：用户域、商品域、订单域、营销域、行为域（或支付域）

原因	说明
模块解耦	以业务边界划分，便于分工协作
数据分治	不同域内数据粒度、指标、维度不同，方便集中建模
复用共享	用户、商品、时间等维度可作为共享维度跨域使用
易于治理	每个域可独立开发、测试、上线，支持多团队并行
分析清晰	报表可围绕域分类展示，如用户画像、商品分布等

14 事实表怎么建模，数仓建模过程

建模流程：

1. **明确业务过程（事实）**：如“下单”、“支付”、“浏览”
2. **确认粒度**：是“每笔订单”还是“每个用户每天”
3. **识别维度**：用户、商品、渠道、时间、地区等
4. **定义指标字段**：金额、次数、数量等度量值
5. **确定表类型**：事务型 / 累积型 / 快照型
6. **关联维度表建星型模型**：提高查询效率

15 维度表增长缓慢怎么办

维度表增长慢的常见情形：

- 用户、商品类维度变化不频繁，但字段有更新
- 时间维度是静态的（如日期、节假日等）

应对策略：

方法	场景	说明
Slowly Changing Dimension（SCD）	有历史追踪需求	常用 SCD Type 1 / 2 管理变更
增量更新	更新字段但保留主键不变	适合无历史记录需求
数据缓存	常驻缓存中（如 Redis）	提高实时查询效率
Hash 校验字段	只更新发生变动的记录	比较新旧字段的 Hash 值

16 在入仓的时候，增量抽取的场景下，周期快照表最新分区的数据是如何产生的？

原则：

周期快照表（如每日库存快照），即使没有新增/变更，也需要**每天生成一次最新快照数据**，用于后续对比分析。

实现方式：

方法	描述
拉取快照源表（如库存	跑一次 Job 全表拉取当前状态
中增加分区字段	tl_date = current_date 写入分区表中
生成逻辑写死周期	赖数据变更，只依赖调度周期执行一次快照快照生成
：保留昨日快照字段	做库存变化比较（环比、留存等）

17 设计模型的时候，设计的原则？

原则	含义
稳定性优先	结构和字段尽量稳定，避免频繁改表
可读性优先	字段命名清晰，易于理解与复用
冗余适度	在 DWS/ADS 层允许适当冗余，换取查询性能
维度抽象清晰	用户、时间、地区等维度单独建模
数据一致性	指标口径标准统一，维度引用一致
适配分层	表结构与数仓分层解耦，粒度清晰（DWD 粒度不可变）

18 Dws 和 dwt 的宽表都有哪些？并且都是什么？DWS 的表是怎么放的？如何整合的？如何评判？主观的还是有什么标准？同一个主题的表放在一起吗？

DWS（明细汇总宽表）与 DWT（通用分析宽表）的差异：

	DWS	**DWT**
细		级汇总
层宽表，便于衍生		直接复用，支持多分析

DWS	DWT
用户行为宽表	画像宽表、订单分析宽

表放置与整合：

- 一般按**主题域划分文件夹/数据库**（如：dws_user/、dws_order/）
- 同一主题的宽表通常会放在同一个文件夹或 schema 下
- 可采用多粒度表组合设计：如 dws_user_day, dws_user_week

如何评判宽表设计质量？

评估维度	标准
清晰	维度组合（如：user_id +
复用度高	维度/指标尽量复用
性能好	过多嵌套/无必要关联
展性好	标追加容易，不影响旧结
层报表强	DS 表支撑报表，DWT 要

---

19 拉链表有什么缺点？拉链表有哪些字段必须要有的？

必备字段：

字段	说明
主键	ser_id，用于唯一标识实体
日期（start_date）	该版本的生效时间
日期（end_date）	该记录的失效时间，当前有效数据一般设为 9999-12
业务字段	机号、状态等会变化的字段
：是否当前有效标记	urrent=1，便于直接查当前版本

拉链表的缺点：

缺点	说明
复杂	需要判断字段是否变化并更新历史版本

缺点	说明
较慢	额外条件筛选“当前有效”记录
并发	持链条完整，容易出现错链/环链等数据
冗余	实体有多个版本记录

20 缓慢变化维，除了拉链表还有哪些方式

类型	SCD 类型	描述
覆盖更新	SCD Type 1	不保留历史，直接更新
拉链表	SCD Type 2	保留历史记录，记录变化生效/失效时间
添加新字段	SCD Type 3	增加字段如 old_value 保留部分历史值
历史快照表	自定义	每日全量拉快照，形成版本记录
外部缓存维度	外挂维表 join 最新值	如用 Redis / MySQL 保留最近一次更新

21 如何判断链表成环

检查标准：

- 成环即某条记录的 end_date 对应的 start_date 是当前记录自身或重复形成回路

✅ 检查方法（SQL）：

```sql
-- 检查某主键是否出现多个相交生效时间段
SELECT user_id
FROM dim_user_chain
GROUP BY user_id, start_date, end_date
HAVING COUNT(*) > 1
```

或递归/窗口函数方式检查 start_date 是否出现在另一个版本中 start_date < x < end_date

22 拉链表初始化的速度

初始化阶段：

- 会将全量数据转为历史版本
- 一般只保留当前版本 + 加入 start_date/end_date 字段

✔ ***加速方法：***

方法	说明
分区处理	键 hash 或时间切片加速
合并逻辑	化阶段所有记录都是有效版本，start_date=etl_date，end_date=9

23 拉链表初始化与查询速度影响因素

影响项	说明
是否加索引	user_id + end_date 上有索引可极大提高效率
表大小	大量历史版本会导致行数膨胀
是否走过滤条件	查询中是否指定 end_date='9999-12-31' 等条件
是否打宽	如果字段太多、存储冗余大，会影响扫描速度
分区字段设计	若按天/ID hash 分区合理可降低 I/O 压力

24 如何评价数仓建设的好坏？如何快速建成一个“好数仓”？

衡量数仓建设好坏的五个维度：

维度	评价标准
☑ 模型合理性	表结构清晰、命名规范、粒度合理、口径一致
☑ 数据质量	准确性、完整性、一致性（有监控/校验机制）
☑ 性能表现	查询速度、资源利用、宽表合理、预聚合有无
☑ 可维护性	脚本分层清晰、依赖透明、调度稳定
☑ 复用性	维表、宽表、指标支持多个业务复用

✔ ***快速建好数仓的方法论：***

方法	关键点
📐 先建“业务总线矩阵”	明确主题域、事实过程与维度关系
🎯 明确核心指标	由 BI 需求倒推数据源与模型粒度

方法	关键点
自下而上 + 自上而下结合	同时基于源数据和业务需求建模设计
建立规范	字段命名、任务命名、分层规范一致
引入模板化脚本	建表 + 调度 + 衍生指标标准化脚本组件化

25 如何衡量表好用不好用？有制定一些量化的指标吗？

维度	量化指标
查询性能	平均查询耗时 < 1s；扫描数据量 < X GB
字段清晰	字段含义完整、注释齐全、命名规范
可复用性	同一表服务多个报表/主题 > 3 次
设计合理性	粒度清晰、主键唯一、命名统一
数据质量	无空值、重复率低、与源表一致性 ≥ 99.9%
查询热度	被访问次数 / 被复用脚本数（可从元数据平台中取）

预测某个街区某个商品的销量，选什么模型？

典型建模思路：时间序列 + 回归 + 空间建模结合

模型	描述	适合场景
✅ ARIMA/Prophet	时间序列建模预测销量趋势	有明显季节性/周期性
✅ XGBoost/LightGBM	多特征回归预测（加入街区、天气、促销等）	多因子场景
✅ 时空图模型（ST-GCN）	考虑时间+空间位置联合建模	高级精度场景
✅ RNN/LSTM	深度学习序列预测	有大量历史数据可训练场景

26 数仓分层一定是要 5 层吗？什么场景下解决什么问题可以多一层或者少一层？

通用 5 层：ODS → DWD → DWS → DWT → ADS

场景	分层策略
简单数据报表/日志分析	可省略 DWT，直接 DWS → ADS
多租户、跨部门共享指标	增加 DIM 层，专门管理维度（可单独维护）

场景	分层策略
指标统一管理	增加 DMI 层（指标中间层）
实时 + 离线混合场景	分离 Realtime Layer（RT_ODS, RT_DWS）

27 现有的大环境下数仓的分层会原来越多还是越来越少？

趋势是逻辑越来越清晰，但物理层次：

趋势	原因
✅ 分层逻辑更细（如加 DMI 层）	增加指标治理、可维护性
✅ 物理层合并执行脚本（多表共存）	降低 ETL 成本、减少资源调度
✅ 模型层“逻辑多、物理少”	用 Lakehouse + Delta/Table Format 合并数据层

回答模板：

随着数仓治理需求提升，分层逻辑将更细化（如增加指标中间层、实时层），但实际执行层可能通过 Lakehouse、任务合并等手段进行优化简化，达到“逻辑分、物理合”的平衡目标。

28 现有大环境下 ER 建模更多还是维度建模更多？

场景	建模方式
业务系统（交易库）	ER 建模为主（强调规范）
分析型数仓 / OLAP	维度建模为主（强调性能和复用）
中台类大数据平台	混合建模，先 ER 后转换为维度建模结构
Realtime OLAP（Doris/ClickHouse）	直接以维度建模存储宽表

29 数仓和 AI 大模型结合

核心连接点：数据驱动 + 知识组织 + 特征供给

结合点	描述	价值

结合点	描述	价值
✅ 特征生成	数仓中宽表、DWS、标签数据可作为 LLM 模型的结构化特征输入	增强 LLM 推理上下文
✅ Prompt 数据仓	将数仓中的“报表语义”、“口径指标”组织成 Prompt / Embedding	用于语义检索和自然语言问答
✅ 向量化标签	将用户标签或商品标签向量化供大模型使用	强化推荐/搜索/多模态任务
✅ RAG 场景下数据源	数仓可作为 RAG（检索增强生成）系统的结构化知识底座	增强大模型 factual correctness（事实正确性）

30 AI 大模型对数据研发的影响

方面	影响
💡 工具层	SQL 自动生成、数据质量规则自动补全、元数据自动文档化
🔎 数据理解层	利用 Embedding + 向量搜索实现指标搜索、字段问答
🧠 业务抽象	从字段层开发向“语义层开发”转变，强调知识组织能力
🎯 标签体系智能生成	通过大模型辅助构建用户标签、商品标签系统

31 维度建模理论这里的冗余是指什么，如果维度变了怎么办（缓慢变化维）

冗余含义：

为了查询性能和可读性，在事实表或宽表中复制维度字段（如用户性别、地域）

不是范式中的“坏冗余”，而是**刻意冗余**

为什么冗余？

- 避免多表 Join
- 简化查询语句
- 提升性能

如果维度发生变化（缓慢变化维）：

处理方式	描述
表（SCI 变更历史，事实表中关联当前维度版	
更新	历史追踪需求，维表直接更新
维表	生成维表快照，事实表带上快照时间

32 说说做了哪些指标，做了哪些标签

常见指标举例（按主题域）：

主题域	示例指标
用户域 日活 DAU、周留存率、流失率	
订单域 GMV、订单数、客单价、支付转化率	
行为域 点击率 CTR、跳出率、访问深度	
渠道域 各渠道带来转化人数、新增用户占比	

常见标签举例（按用户行为）：

标签类型	示例标签
人口属性标签 性别、年龄段、省份、职业	
行为偏好标签 高频下单、夜间活跃、偏好手机数码	
价值标签	最近消费金额、生命周期阶段、RFM 模型
活跃标签	最近活跃天数、近 7 日登录次数、日均浏览数

33 你提到 DWS 层存储一些指标，这个层可以删掉吗

视项目复杂度而定：

情况	是否需要 DWS
✅ 多报表共用中间指标	必须保留，支持指标复用
✅ 跨主题分析宽表	必须保留，用于统一抽象
❌ 报表简单、数据少、粒度固定	可省略，DWD → ADS 直通
✅ 实时场景中	通常弱化 DWS，直接 DWD→ADS 计算指标

34 数据质量监控，主要监控哪些内容，监控哪些指标

类型	内容	示例
✅ **完整性**	数据是否缺失	是否有空字段、分区是否齐全
✅ **准确性**	数据是否正确	订单金额是否为负、订单数是否为整数
✅ **一致性**	多表字段值是否一致	user_id 在多个表中是否能对应
✅ **及时性**	数据是否按时到达	T+1 任务是否准点完成，是否延迟
✅ **异常检测**	值是否异常跳变	今日 GMV 与昨日对比波动过大、超出设定阈值

手段	工具
SQL 监控 + 比对脚本	Airflow, Azkaban 中脚本监控
数据质量平台	如滴滴 DQC、字节 QualityGuard、阿里 DataQuality
指标波动预警	与昨日/上周同比异常自动报警（可结合 LLM 自动解释）

35 数仓中有哪些主题，哪些维度

常见业务主题（按模块划分）：

模块	主题
用户域	注册、登录、活跃、留存、生命周期
商品域	上架、下架、浏览、收藏、销售
订单域	下单、支付、退款、取消
行为域	浏览、点击、曝光、埋点事件
渠道域	来源渠道、投放渠道、转化数据
营销域	优惠券、满减、活动参与率等
财务域	收入、成本、毛利、发票等

📌 通用维度：

维度	示例字段
用户维度	用户 ID、性别、年龄、地域、等级

维度	示例字段
商品维度	商品 ID、分类、品牌、价格
时间维度	日期、周、季度、是否节假日
地理维度	城市、省份、大区、渠道
活动维度	活动 ID、类型、时间段

36 拿到一个数据需求后，如何转化成一个数据模型？

标准流程：

1. **理解业务需求**

   明确“分析什么”、“指标口径”、“维度切片”

2. **拆解核心指标**

   明确“订单量”、“退款率”是否已有；计算口径是否一致

3. **确定粒度**

   比如按“用户-日”粒度汇总，还是“商品-城市-小时”

4. **寻找数据源 & 对应主题域**

   来源表在哪？在 DWD 还是 DWS 层？

5. **模型设计**

   需要新建事实表？宽表？中间聚合表？是否已有复用基础

6. **是否接入总线矩阵 / 指标平台**

   能否沉淀为公用模型或指标

37 共性指标如何下沉？

什么是“下沉”？

将常用、复用率高的指标从上层报表/ADS 下沉至 DWS/DWT 层。

操作流程：

步骤	描述
✅ 识别高频指标	通过报表、任务脚本分析出重复指标
✅ 统一口径、梳理逻辑	明确所有人使用 GMV 是“下单金额”还是“支付金额”
✅ 建统一 DWS/DWT 表	将指标封装入宽表中并提供字段注释
✅ 逐步替换上层重复计算逻辑	减少代码重复，降低维护成本
✅ 接入指标平台 / 字典系统	提供搜索、预警、说明书能力

38 围绕指标体系建设和治理说一说

建设维度：

方向	内容
📦 指标标准化	字段命名规范、计算公式统一
🧱 指标结构化	原子 → 衍生 → 复合指标的层级
📚 指标字典管理	建指标平台，支持检索、权限控制
🔍 指标溯源能力	支持从报表字段回溯至指标公式与源表
📈 指标监控体系	支持变更提醒、异常预警、血缘分析
🔁 指标复用治理	一处定义，多处引用（防“同名异义”）

39 数据分析的模型

模型类型	示例
📊 描述性分析	同比/环比、留存分析、RFM 模型、漏斗模型
📈 诊断性分析	异常波动原因分析、AB Test
🔮 预测性建模	回归模型、时间序列、ARIMA、Prophet、LSTM
🎯 分类模型	用户偏好预测、购买意愿识别
🎯 聚类模型	用户画像（K-means）、商品聚类
🎯 关联规则	推荐系统、购物篮分析（Apriori）

40 星型模型和雪花模型对比

项目	星型模型	雪花模型
构成	一张事实表 + 多个维度表	一张事实表 + 维度表再细分子维表
冗余度	有冗余（如重复的地区名）	冗余低
查询性能	高，Join 少	低，Join 多
可读性	好	差
建模场景	OLAP 报表系统	OLTP 规范系统或对冗余特别敏感场景

41 处理大规模维度表的拉链

难点：

- 数据量大，链表容易错链/断链/重链
- 更新开销高，需对比字段是否变动
- 历史版本多，查询慢

实践优化策略：

方法	描述
🚀 **字段级 Hash 比对**	每次抽取时对比字段 Hash，提升判断效率
分区 + 主键排序	加快判断/覆盖匹配效率
✅ **增量处理**	仅处理变动记录，减少全量扫描
🧠 **构建 CDC 中间表**	增量变更集中处理后写入拉链逻辑
📦 **外部缓存比对**	如 Redis 存储旧版本摘要值进行快速匹配

42 数据建模和数学建模的区别联系

对比项	数据建模	数学建模
目标	构建可复用、可查询的数据表结构	构建数学方程/函数以解释现象或做预测
对象	表、字段、维度、事实、指标	方程、分布、变量、损失函数
场景	数仓、数据平台、数据中台	预测、优化、机器学习
方法论	维度建模、ER 建模、范式建模	回归、分类、线性规划、概率模型

43 数据建模和日常数据开发的区别

项目	数据建模	数据开发
目标	设计结构与规范	实现数据流转与加工
成果	表模型、维度设计、指标体系	ETL 脚本、调度任务、数据表落地
职责	抽象业务过程为事实维度模型	按建模规范实现数据处理流程
工具	绘图工具（PowerDesigner）、指标平台	SQL、Shell、调度平台（Airflow、Dolphin）

44 拉链表和版本表

对比维度	拉链表	版本表
是否记录历史	✅ 有效期区间	✅ 每日快照
数据量	较少（仅变化）	较多（每天全量）
查询成本	复杂（需判断时间区间）	简单（过滤 dt）
用途	精准历史分析、变更追踪	快照统计、补数
更新方式	Upsert 合并（多用于 Hudi）	Insert overwrite

七 数据存储与分析平台

1 ES 的使用场景？

2 ES 搜索比较快，为什么？

3 数据报表存储这块用过哪些产品，用过哪些存储引擎？

- **产品：**
  - **MySQL / MariaDB**：关系型数据库，支持 InnoDB 存储引擎，适合结构化数据报表存储。
  - **PostgreSQL**：高级关系型数据库，丰富的扩展支持复杂查询和报表。
  - **Oracle**：企业级关系型数据库，强大的报表和分析能力。
  - **ClickHouse**：列式数据库，适合大规模在线分析处理（OLAP）场景，快速报表查询。
  - **Hive / Presto / Spark SQL**：基于 Hadoop 生态的大数据 SQL 查询，适合海量数据离线报表。
  - **Elasticsearch**：对半结构化日志类报表做实时分析。
  - **Redis**：缓存型存储，常用于实时报表的缓存加速。
- **存储引擎：**
  - **InnoDB**（MySQL 默认，支持事务、行级锁）
  - **MyISAM**（适合读多写少的场景，但不支持事务）

    - Columnar 引擎（ClickHouse 本身是列式存储）
- Parquet / ORC（数据仓库中的列式存储格式，用于大数据存储与分析）

4 OLAP 引擎有哪些？

引擎名称	特点
ClickHouse	列式存储，适合实时分析，支持 SQL，部署灵活。
Apache Kylin	构建 Cube 以加速查询，适合维度较多的复杂报表分析。
Doris / StarRocks	支持实时明细和聚合一体化查询，兼容 MySQL 协议，易于接入。
Presto / Trino	分布式查询引擎，可联邦查询多种数据源。
Druid	擅长流式数据分析，适合监控看板等需求。
Hive + Tez/Spark	离线 OLAP 场景，支持大数据批处理，性能相对较慢。

5 OLAP 和 OLTP 使用场景

- ***OLTP（联机事务处理）***

适合日常业务系统的在线事务处理，如银行系统、订单系统、库存管理。特点是大量短小、频繁的写读操作，数据一致性要求高。

- ***OLAP（联机分析处理）***

适合复杂查询和多维分析，比如报表统计、数据仓库、业务决策支持。特点是大批量数据读取，聚合和多维分析为主，写入频率低但查询复杂。

6 Hbase 介绍

HBase 是基于 Hadoop 生态的分布式、可扩展、面向列的 NoSQL 数据库，适合海量结构化和半结构化数据存储。它支持随机读写，强一致性，常用作大数据实时存储与查询，适合时序数据、日志数据存储。

7 Doris 和 clickhouse 的区别，使用场景

- ***Doris***

基于 MPP 架构，支持高性能的实时数据分析，适合复杂 SQL，提供较好的事务支持和数据更新能力，适合实时数据仓库和交互式分析。

- ***ClickHouse***

高性能的列式数据库，适合海量数据的快速查询和复杂聚合，擅长离线报表分析，写入相对批量化，实时性不如 Doris。

8 Hbase 和 clickhouse 在选型的时候各自适用于什么场景

HBase：

- 典型的分布式 NoSQL 列式数据库，基于 Hadoop 生态。
- 适用场景：
  - 海量结构化或半结构化数据存储（PB 级别）
  - 高频写入、实时随机读写场景（如用户画像、日志数据存储）
  - 对强一致性有要求
  - 实时计算场景（配合 Flink、Spark Streaming 使用）
  - 支持事务级别较弱，适合大数据存储和查询，不适合复杂多表关联

ClickHouse：

- 高性能分布式列式数据库，主要面向分析型查询（OLAP）
- 适用场景：
  - 大规模数据的实时分析和报表生成
  - 复杂聚合和多维分析，尤其是时间序列和日志分析
  - 高并发读，数据批量导入，读多写少
  - 不适合高频写入和实时更新，写操作一般是批量追加
  - 典型业务如用户行为分析、广告监控、大数据报表

总结：

HBase 更适合海量随机写和实时查询的场景，ClickHouse 更适合海量批量写入且快速分析查询的场景。

9 数据压缩

- ***目的：减少存储空间，降低 I/O，提升传输效率***
- ***常见压缩算法：***
  - 无损压缩：Snappy（速度快，压缩率适中），Zlib（压缩率高，速度慢），LZ4（压缩速度快，解压快），Zstandard（新型高效压缩算法）
  - 有损压缩：主要用于图像、音视频，不涉及数据库数据

- ***应用场景：***
  - 列式存储数据库（ClickHouse、Parquet）常用压缩，减少磁盘占用，提升查询速度
  - 流式系统中压缩消息（Kafka 中使用 Snappy、LZ4）
  - HBase 默认支持数据块压缩（Snappy、LZO、GZIP）
- ***选型原则：***
  - 读写性能 vs 压缩率权衡
  - 业务对实时性的要求
  - CPU 资源消耗限制

10 AB test

- ***定义：***

将用户随机分配到两个或多个版本组（A、B 组），比较各组的行为指标差异，评估产品改动效果。

- ***流程：***
  - 设计实验，确定指标（点击率、转化率等）
  - 随机分组，保证统计学上的独立性
  - 收集数据，统计分析
  - 判断是否有显著性差异（t 检验、卡方检验等）
- ***应用场景：***
  - 产品功能优化（按钮颜色、推荐算法）
  - 用户界面设计测试
  - 营销活动效果验证
- ***注意点：***
  - 样本量足够大，避免假阳性
  - 统计显著性与业务实际意义结合考虑
  - 实验期间避免干扰因素

11 Flink、doris 和 hbase 的关系

- **Apache Flink**：流式计算引擎，擅长实时数据处理和事件驱动计算。
- **Apache Doris**：分布式 MPP 分析型数据库，面向实时 OLAP 场景，支持低延迟大数据分析。
- **HBase**：分布式 NoSQL 数据库，海量存储，随机读写。

关系和协同：

- Flink 可作为实时计算平台，将实时数据计算结果写入 Doris 或 HBase。
- Doris 负责 OLAP 分析，支持大规模复杂查询，适合报表和 BI。
- HBase 负责存储大量实时数据，适合随机查询和数据持久化。
- 整个数据架构中，Flink 是计算层，Doris 和 HBase 是存储层，根据业务需求选择。

12 Hbase 可以做 dwd 层吗

- **DWD 层（明细宽表层）**是数据仓库设计中的一层，保存详细的、结构化的业务数据，要求实时或准实时更新。
- **HBase 特点：**

支持大规模实时写入，随机读写能力强

适合存储宽表格式，尤其是稀疏数据

- **因此**，HBase 完全可以作为 DWD 层的存储介质，尤其是需要低延迟访问和写入时。但需要结合数据建模和二级索引设计，配合计算引擎完成数据清洗和加工。
- **缺点**：不适合复杂 SQL 查询，通常配合 Presto/Impala 等 SQL 引擎使用。

13 Hbase 适用场景和不适用场景

- ***适用场景：***
  - 海量数据存储（PB 级别）
  - 实时随机读写
  - 大规模日志存储和分析
  - 用户画像、行为轨迹存储
  - IoT 数据存储
- ***不适用场景：***
  - 复杂 SQL 联表查询、事务处理
  - 低延迟强一致性事务场景
  - 小规模数据或者对查询延迟极高要求的场景
  - 传统关系型数据库适用的业务

14 Hbase 和 doris 使用场景的异同

方面	HBase	Doris
数据模型	NoSQL，列式存储	分布式 MPP 数据库，关系模型
主要用途	实时随机读写，海量数据存储	高并发分析查询，实时 OLAP
写入特性	高频写入，低延迟	批量写入为主，支持流式写入
查询类型	Key-value 检索，单行/范围扫描	复杂 SQL 查询，支持多表关联
聚合分析	支持但不擅长复杂聚合	强大，支持复杂多维聚合
典型应用	用户画像、日志存储	BI 报表、实时数据分析

总结:

HBase 更适合**海量实时写读存储**，Doris 更适合**复杂实时分析与报表**，两者在大数据生态中常常互补使用。

15 Doris 和 clickhouse 解决什么问题

Apache Doris 和 **ClickHouse** 都是面向大数据分析的分布式列式数据库，主要解决以下问题：

高性能实时分析

传统关系型数据库在面对海量数据时，查询效率低、响应慢，难以满足实时 BI 和大数据分析需求。Doris 和 ClickHouse 通过列式存储、MPP 架构优化了大规模数据的读写性能，支持秒级甚至亚秒级响应。

复杂多维分析和大规模聚合

面对海量数据，传统数据库难以高效处理复杂聚合、多维度切片分析。

两者都支持高效的向量化执行引擎，优化了聚合计算，满足实时报表、用户行为分析、广告监控等需求。

弹性扩展性

大数据场景下，数据规模快速增长。

Doris 和 ClickHouse 都支持分布式扩容，节点横向增加，保证性能和容量的线性提升。

成本与运维效率

提供更简化的架构，减少对复杂 Hadoop 生态的依赖。

更友好的 SQL 接口，方便业务人员和数据分析师直接使用。

区别侧重点：

Doris 偏重于传统数仓用户和企业级应用，提供兼容 MySQL 协议、支持复杂 SQL、事务等功能，易于迁移传统数仓。

ClickHouse 更偏向于极致的分析速度和高吞吐量，适合日志、事件流处理和互联网行业的实时分析。

16 Doris 为什么能解决高并发问题

Doris 解决高并发的关键技术点：

MPP 架构（Massively Parallel Processing）

查询任务分布到多个计算节点并行执行，大幅度提升查询吞吐量和响应速度。

每个节点负责数据局部计算，最后汇总结果，避免单点瓶颈。

列式存储

只读取查询涉及的列，减少磁盘 IO 和内存使用，提升查询效率。

列式压缩技术减少存储空间和数据传输量。

向量化执行引擎

支持批量数据处理，充分利用 CPU 缓存和 SIMD 指令，极大提升 CPU 效率。

减少函数调用开销，提升单节点的查询速度。

存储和计算分离设计（支持多种部署模式）

允许独立扩展计算资源和存储资源，灵活应对不同的负载需求。

提升整体资源利用率和并发处理能力。

数据预聚合和物化视图

支持预计算常用聚合，减少运行时计算负担。

物化视图加速复杂查询，避免重复计算。

高效的缓存机制和负载均衡

多级缓存减少磁盘访问，提升响应速度。

智能负载均衡防止热点数据过载。

支持多租户和资源隔离

通过资源池管理和调度，保证多个用户查询同时进行时的性能稳定。

八 场景题与项目经验

1 时间复杂度分析，如何优化？

- **分析步骤：**
  - 理解算法或 SQL 的核心操作，判断其复杂度（如线性 O(n)、平方 $O(n^2)$等）。
  - 关注嵌套循环、排序、连接操作等高耗时部分。
  - 观察数据规模和增长趋势，判断瓶颈环节。
- **优化方法：**

- 减少不必要的计算和重复访问。
- 使用高效数据结构，如哈希表、索引、B 树。
- 优化 SQL，避免笛卡尔积，合理使用 JOIN 和 WHERE 条件。
- 利用分区、分桶减少扫描范围。
- 并行计算和分布式处理，利用集群资源。
- 预计算和缓存热点数据，减少重复计算。

2 如果数据量增加 10 倍，如何优化你的方案？

优化思路：

- 水平扩展集群：增加节点，提升存储和计算能力。
- 数据分区细化：更精细的分区策略减少扫描量。
- 索引优化：增加合适的索引，减少查询扫描范围。
- 分布式计算：利用 Flink/Spark 等引擎进行分布式处理。
- 优化存储格式：采用列式存储、压缩格式（Parquet、ORC）减少 I/O。
- 异步和增量计算：避免全量重算，提高计算效率。
- 流批一体架构：实现实时和批量数据协同处理。
- 缓存策略：利用 Redis 等缓存热点数据，降低数据库压力。

3 如果某天发现报表数据异常，如何排查？

- **排查流程：**
  - **确认异常表现**：具体异常指标，波动幅度。
  - **回溯数据源**：检查原始数据是否异常（采集是否正常、数据质量）。
  - **检查 ETL 流程**：是否存在任务失败、重跑、参数变更。
  - **排查代码变更**：SQL、脚本更新是否引入逻辑错误。
  - **分析依赖系统状态**：依赖数据库、服务是否正常。
  - **版本和配置核对**：确认环境一致性。
  - **数据对比**：与历史数据做趋势和分布对比，定位异常范围。
  - **沟通业务**：确认业务逻辑或活动是否变化。
- **工具支持：**

日志监控（如 ELK）

数据质量监控（如 Great Expectations）

任务调度监控（DolphinScheduler）

4 数据报表存储这块用过哪些产品，用过哪些存储引擎？

- **产品举例：**
  - 传统关系型数据库：MySQL（InnoDB 存储引擎）、PostgreSQL。
  - 分布式列式数据库：ClickHouse、Apache Doris、Apache HBase。
  - 大数据仓库：Hive（基于 HDFS 存储，支持 ORC/Parquet）。
  - 云原生数据仓库：Amazon Redshift、Google BigQuery。
- **存储引擎：**
  - 行存储引擎：InnoDB（支持事务），MyISAM。
  - 列存储格式：Parquet、ORC，用于大数据分析优化 I/O。
  - LSM-Tree 引擎：HBase 使用的底层存储结构，适合写密集场景。
  - MPP 引擎：Doris、ClickHouse 采用的分布式并行处理。
- **根据业务需求选型：**实时性、数据规模、查询复杂度等。

5 场景题：商家同类产品竞争太多，怎样设计方案控制在合理区间？

问题理解：避免市场过度饱和，保护商家利益，保证用户体验。

设计方案：

- **规则限制**：控制同类产品的上架数量或频率，设置门槛。
- **智能推荐算法**：根据用户画像和购买历史，个性化展示，避免重复暴露同类产品。
- **产品分类细化**：细化类目，控制每个子类目内的竞争度。
- **价格和库存监控**：动态调整价格，避免恶性竞争。
- **流量分配策略**：合理分配流量资源，防止集中流量挤兑。
- **引入竞价机制**：商家通过竞价获得更多展示机会。
- **运营干预**：定期评估同类产品竞争情况，适时调整策略。

技术实现：

- 建立实时监控和预警系统。
- 利用大数据分析发现异常竞争行为。
- 开发后台管理系统支持规则配置。

6 在入仓的时候，增量抽取的场景下，周期快照表最新分区的数据是如何产生的？

- **周期快照表**一般用于记录某一时间点的全量状态数据。
- **最新分区数据产生流程：**
  - 从增量抽取数据源中，提取当前周期的变化数据（新增、更新、删除）。
  - 对增量数据进行**去重和合并**，通常利用主键或业务唯一键，合并历史快照数据。
  - 应用合并逻辑后，生成当前周期的最新状态数据，写入新的分区。
  - 保留历史分区，实现历史版本追溯。
- **技术实现：**
  - 利用 Spark、Flink 等流批一体计算框架完成合并计算。
  - 数据库或数据仓库支持分区管理和分区替换。
- **优势：**只处理增量数据，减少全表扫描，快速获得最新状态。

7 有没有一张表实现所有的分析？

- **理论上不推荐**用一张表做所有分析，原因：
  - 数据粒度和维度需求不同，分析维度多样化。
  - 大表读写性能瓶颈明显，且复杂查询难以优化。
  - 数据治理和权限管理复杂。
- **实际做法：**

设计多层数据模型，明细层（DWD）、汇总层（DWS）、应用层（ADS）分层管理。

各分析场景选用不同物化视图或专题表。

- **但在某些小规模或简单业务中，**可以设计宽表或事实表满足大部分分析需求。

8 数据和业务是怎么协作的？数据对业务做哪些反馈和支持？

- **协作方式：**
  - 数据团队与业务团队紧密沟通，明确业务需求和数据指标定义。
  - 通过数据产品（报表、仪表盘、API）提供业务洞察。
  - 及时反馈数据异常和趋势变化。
- **数据对业务支持：**
  - **决策支持**：通过指标分析指导运营策略。
  - **用户行为分析**：精准营销和个性化推荐。

- **风险控制**：欺诈检测和异常告警。
- **产品优化**：AB 测试结果分析，产品迭代依据。
- **效率提升**：自动化报表减少人工统计。

9 场景题：调研某城市各手机品牌市场占比

- **思路：**
  - 收集数据来源，如电商销售数据、线下调研数据、第三方市场报告。
  - 统一数据格式，清洗去重。
  - 设计数据模型，按品牌和城市维度聚合销售量或销售额。
  - 计算品牌市场占比 = 品牌销量/该城市总销量。
  - 利用 BI 工具或自定义报表展示市场占比趋势和细分。
- **注意点：**
  - 保证数据来源可靠性和代表性。
  - 处理时间跨度和统计口径一致。
  - 支持多维度切片分析（如按时间、渠道）。

10 如果一个 spark 任务跑得慢，怎么去排查分析？

排查步骤：

- 查看 Spark UI，分析 Stage 和 Task 的执行时间，定位慢点。
- 检查数据倾斜，部分 Task 处理数据过多。
- 资源分配是否合理，CPU 和内存是否充足。
- Shuffle 数据量是否过大，导致网络瓶颈。
- 代码是否存在不合理的操作（如重复计算、宽依赖过多）。
- 输入数据量是否异常增大。
- 查看 GC 日志，判断是否频繁 GC 影响性能。

优化建议：

- 使用广播 Join 减少 Shuffle。
- 调整并行度，合理分配资源。
- 使用缓存（persist）避免重复计算。
- 优化 Shuffle 操作，减少数据倾斜。

- 精简和优化代码逻辑。

11 数据结构的优缺点，使用场景

数组

优点：随机访问快，存储紧凑。

缺点：插入删除慢，大小固定。

适用：定长数据，频繁随机访问。

链表

优点：插入删除快，动态大小。

缺点：随机访问慢，内存开销大。

适用：频繁插入删除操作。

哈希表

优点：查找插入平均 O(1)，效率高。

缺点：空间浪费，哈希冲突处理复杂。

适用：快速查找和唯一键映射。

树结构（如 B 树、红黑树）

优点：有序数据，高效范围查询。

缺点：实现复杂，维护开销。

适用：数据库索引，多级分类。

图结构

优点：复杂关系表达，灵活。

缺点：存储和查询复杂。

适用：社交网络、推荐系统。

12 Row_number()里的 order by 怎么优化

- **减少排序的数据量**：先通过 WHERE 或分区过滤，缩小排序范围。
- **合理利用索引**：确保排序字段有索引支持，避免全表排序。
- **分区排序**：结合分区字段，实现局部排序减少数据量。
- **避免不必要的 ROW_NUMBER 计算**：如果只取 TopN，结合 LIMIT 和索引。
- **使用物化视图或预排序表**：提前计算排序结果，减少实时排序。

13 数据倾斜有哪些场景？如何优化？

- 数据倾斜场景：
  - Join 时部分 key 数据量远大于其他 key。
  - GroupBy 时某些 key 过于集中。
  - Repartition 或 Shuffle 时分区不均。
- 优化手段：
  - 广播 Join：对小表广播，避免 Shuffle。
  - 盐值法：给 key 加随机前缀，打散倾斜数据，后续再聚合。
  - 增加并行度：细分分区，提高并发。
  - 过滤和预聚合：减少倾斜数据量。
  - 合理设计数据模型：避免热点 key 产生。

14 如何从不同角度对任务进行优化

- **资源角度**：合理分配 CPU、内存、磁盘 IO，避免资源瓶颈。
- **算法角度**：优化代码逻辑，减少复杂度。
- **数据角度**：减少数据量，过滤无用数据，压缩数据格式。
- **架构角度**：分布式处理、缓存机制、异步计算。
- **调度角度**：错峰执行，避免资源争抢。
- **监控角度**：实时监控指标，快速定位问题。

15 场景题：直播成交波动分析

数据收集：采集直播间流量、成交订单、用户行为数据。

数据处理：计算成交量、成交率，关联流量和成交波动。

分析指标：

- 不同时段成交量和转化率。
- 观众峰值和成交波动关联。
- 商品热度和成交趋势。

洞察：找出成交下降或暴增的原因（主播表现、活动促销、技术问题）。

优化建议：

- 优化直播内容和互动。

- 设计合理的促销活动。
- 监控关键指标，快速响应异常。

16 场景题：拦截器反应时间和天的任务调度时间冲突怎么办？

- **问题**：系统拦截器响应时间过长，影响当天任务调度的准时执行。
- **解决方案**：
  - **优化拦截器性能**：排查慢响应原因，代码优化、缓存或异步处理。
  - **调整调度时间**：合理规划任务执行时间，避开高峰。
  - **任务优先级管理**：关键任务优先执行，减少冲突。
  - **分布式调度**：多集群或多节点调度，减轻单点压力。
  - **监控预警**：及时发现响应异常，自动告警处理。

17 如何界定业务流程？举例说明？

- **界定方法**：
  - 识别业务活动的关键步骤和顺序。
  - 明确输入、输出及参与角色。
  - 结合业务目标和规则划分流程边界。
- **举例**：

电商订单流程：

  - 用户下单
  - 支付确认
  - 订单处理与发货
  - 物流跟踪
  - 售后服务

  该流程涵盖订单生命周期的关键节点，便于数据采集和分析。

18 数据采集、清洗、分析的方法和原理

- **数据采集**：
  - 批量采集（ETL）、实时采集（Kafka、Flink）。

- 采集层监控保证数据完整性和准确性。

- **数据清洗：**
  - 缺失值处理、异常检测、格式标准化、重复数据剔除。
  - 自动化规则和人工校验相结合。
- **数据分析：**
  - 描述性分析、诊断性分析、预测性分析、规范性分析。
  - 使用 SQL、统计模型、机器学习算法等。
  - 结合业务场景，设计合理指标和 KPI。

九 其他通用类问题

1 你比较熟悉什么架构

- **数据采集层（Data Ingestion）**

负责从多源（业务系统、日志、第三方 API 等）采集原始数据，使用工具如 Flink、Kafka、Logstash 实现实时或批量采集。

- **数据存储层（Data Storage）**

包含分布式文件系统（如 HDFS）、数据库（如 HBase、ClickHouse、Doris）和数据湖，支持海量数据的高效存储和访问。

- **数据处理层（Data Processing）**

使用 Flink、Spark、Hive 等计算引擎完成数据清洗、转换（ETL/ELT），构建明细层（DWD）、汇总层（DWS）、主题层（ADS）等数据模型。

- **数据服务层（Data Serving）**

提供 SQL 查询、API 接口、报表工具支持，常用 ClickHouse、Doris、Presto 等作为高性能分析引擎。

- **数据治理与调度层**

使用 Apache Airflow、DolphinScheduler 管理作业调度、数据血缘和质量监控。

- **数据展示层**

通过 BI 工具（如 Tableau、Superset）实现数据可视化，支持业务决策。

2 集群配置，并行度

- **集群配置**

根据数据规模和业务需求，配置合理的计算节点和存储节点数量。常见包括 Master 节点负责调度和元数据管理，Worker 节点负责计算和存储。

- **资源分配**

CPU 核数、内存大小、网络带宽等直接影响作业性能。对计算资源进行隔离与弹性扩缩容，提升资源利用率。

- **并行度设置**
  - 在 Flink/Spark 中，并行度决定任务分割的数量，越高并行度通常带来更高吞吐，但也带来调度和网络开销。
  - 并行度设置要结合集群规模、任务复杂度和资源情况动态调整。
  - 在 Doris/ClickHouse 等 MPP 系统中，数据分片和节点数决定并行查询能力。

3 dophinshedule 的底层实现了解吗？

- **简介**

DolphinScheduler 是一款开源的分布式工作流调度平台，专注于大数据作业的调度和管理。

- **底层实现**

基于 Master-Worker 架构：Master 负责调度和协调，Worker 负责执行任务。

任务依赖通过有向无环图（DAG）表示，支持复杂任务依赖管理。

使用 ZooKeeper 进行分布式协调，保证高可用和负载均衡。

支持多种任务类型（Shell、Spark、Flink、MR 等）和灵活的重试策略。

UI 和 API 提供作业管理、监控、告警功能。

4 微服务了解吗？

- **微服务概念**

将应用拆分为多个独立部署、功能单一的服务，通过轻量级通信（通常是 HTTP REST 或消息队列）协作完成业务。

- **优点**
  - 独立开发与部署，提高开发效率和系统可维护性。
  - 容错性好，单个服务故障不影响整体。
  - 支持技术多样性，不同服务可用不同技术栈。
  - 容易水平扩展，适应不同负载需求。
- **在数据仓库中的应用**
  - 数据采集、数据处理、调度管理、报表服务各模块独立微服务化。
  - 方便持续集成、持续交付（CI/CD）。
  - 结合容器化（Docker、Kubernetes）实现弹性伸缩。

5 怎么提高 UV 点击率

UV（独立访客）点击率提升，实质是提升用户活跃度和转化率，常见策略包括：

- **优化用户体验**

  页面加载速度快，交互流程顺畅，移动端兼容性好。

- **精准推荐和个性化**

  利用用户画像和行为数据，推荐相关内容或产品，增加用户停留和点击。

- **内容质量**

  提供有价值、吸引人的内容，利用 A/B 测试持续优化标题和页面布局。

- **营销活动**

  利用限时优惠、弹窗提醒、推送通知等手段刺激点击。

- **数据驱动迭代**

  通过数据分析发现用户行为瓶颈，不断调整运营策略。

6 如何处理跨天数据？

- **定义：**跨天数据指发生在两个自然日时间点的业务数据（如交易在 23:59 开始，00:01 结束）。
- **处理策略：**
  - **基于时间范围切分**：按照业务需求，切分数据到对应日期分区（分区表设计）。
  - **使用时间窗口聚合**：例如基于事件时间（event-time）做滚动窗口或滑动窗口处理，避免因处理时间导致数据跨天错乱。
  - **标记业务时间字段**：业务流水中保留交易发生时间，作为分区键或过滤字段。
  - **数据补偿机制**：针对晚到数据（如延迟日志），设计补偿和重跑机制，保证跨天数据完整。
  - **ETL 调度合理安排**：确保当天数据处理在次日凌晨完成，避免遗漏。
- **实战中**，常结合分区表+流批一体架构（Flink）做精准跨天数据处理。

7 一个任务，平常 10-20 分钟就完成了，今天 1-2 个小时都没有完成，如何解决？

- **排查思路：**
  - **资源瓶颈**：检查集群 CPU、内存、磁盘、网络是否资源紧张。
  - **数据量异常**：是否今日数据量暴增，导致处理时间增长。
  - **任务调度冲突**：是否存在资源争用，任务被抢占或限流。

- **数据倾斜**：数据分布不均，部分节点处理压力过大。
- **代码或配置变更**：最近是否有代码更新或参数调整影响性能。
- **系统故障**：节点宕机、网络抖动、依赖服务异常。

- **解决方案：**
  - 通过监控（如 Grafana、Prometheus）分析指标，定位瓶颈。
  - 优化 SQL 或作业逻辑，排查数据倾斜问题。
  - 扩容集群资源或调整并行度。
  - 重启异常节点或服务。
  - 针对数据量大，考虑分批次增量处理。
  - 如果问题短时间难解，考虑回滚至稳定版本。

8 缓慢变化维/层级维表

- ***缓慢变化维：***

处理维度属性随时间缓慢变动的情况，常见类型：

- SCD Type 1：直接覆盖旧值，不保留历史。
- SCD Type 2：保留历史版本，通过生效时间或版本号区分。
- SCD Type 3：保留部分历史（如前一个值）。

- ***层级维表：***

表示层级关系，如组织架构、产品分类。

设计时考虑树形结构存储，常用方式：

- 父子关系表
- 路径枚举法
- 嵌套集模型

- ***应用：***

报表查询时准确反映历史维度信息。

分析时支持跨层级聚合。

9 架构选型

- **选型原则：**
  - **业务需求**：实时性、数据规模、查询复杂度。

- **技术生态**：团队技术栈、运维经验。
- **成本预算**：硬件资源、云服务费用。
- **扩展性和容错性**。

- **常见架构：**
  - 离线批处理架构（Hive、Spark on Hadoop）
  - 实时流批一体（Flink + Doris/ClickHouse）
  - Lambda 架构（批处理+流处理分开）
  - Kappa 架构（纯流处理）
- **结合场景：**
  - 实时要求高，用 Flink 流处理+实时数仓（Doris）；
  - 离线大数据，Hive+Spark 更合适。

10 说说 udf udaf udtf

- **UDF（User Defined Function）：**用户自定义函数，实现单行输入单行输出的标量函数，比如自定义字符串处理、数值计算。
- **UDAF（User Defined Aggregate Function）：**用户自定义聚合函数，对一组数据进行聚合计算，返回单个结果，比如自定义求和、平均值、复杂统计。
- **UDTF（User Defined Table Function）：**用户自定义表函数，输入一行数据，输出多行数据或多列结果，常用于数据拆分、爆炸等操作。

11 和 dolphinSchedular 类似的工具

- Apache Airflow

Python 编写，支持复杂 DAG 任务调度，生态丰富，插件多。

- AzKaban

LinkedIn 开源的批处理调度平台，适合 Hadoop 生态。

- Oozie

Hadoop 生态的作业协调器，支持 MapReduce、Hive、Pig 等。

- Luigi

Spotify 开源的任务调度工具，支持复杂依赖，使用 Python 编写。

- Kubernetes CronJob

用于云原生环境，轻量调度定时任务。

12 中间层怎么设计比较好？

- **中间层（Data Mart / DWS 层）**通常指介于明细层（DWD）和应用层（ADS）之间的数据层，主要作用是对数据进行清洗、整合和预聚合，供应用层快速查询使用。
- **设计原则：**
  - **数据模型清晰**：按业务主题划分，支持多维分析。
  - **数据质量保障**：清洗重复、缺失、脏数据。
  - **灵活的更新机制**：支持全量和增量更新。
  - **合理的分区与索引**：提升查询效率。
  - **预聚合设计**：减少应用层计算压力。
  - **接口统一**：方便上层访问和权限管理。

13 应用层怎么设计？应用层和汇总层的区别在哪里？

- 应用层（ADS）设计：
  - 面向具体业务场景，提供报表和分析指标。
  - 聚合和计算逻辑进一步简化，直接供业务使用。
  - 设计时关注查询性能和易用性，通常伴随多维度报表或仪表盘。
- 应用层与汇总层（DWS）的区别：
  - **汇总层（DWS）**侧重数据整合和预聚合，作为中间计算层，数据较为细致，支撑多种分析需求。
  - **应用层（ADS）**是最终展现层，数据更贴近业务，通常是汇总层结果的二次加工，直接服务于业务决策和用户。

14 利用数据库表介绍 exists 关键字

- **exists** 用于判断子查询是否返回结果，语法：

```
sql
SELECT * FROM table1 WHERE EXISTS (SELECT 1 FROM table2 WHERE condition);
```

- 如果子查询有结果，则 exists 返回 true，该条记录会被保留；否则被过滤。

15 Exists 返回 true 还是 false 的结果

- exists 返回布尔值：
  - 返回 true，当子查询结果至少有一条记录。
  - 返回 false，当子查询结果为空。
- 在 WHERE 子句中，exists 作为条件，决定外层查询是否包含该条记录。

16 DML 和 DDL 是什么

- **DML（Data Manipulation Language）数据操作语言**

用于数据的增删改查，如：SELECT, INSERT, UPDATE, DELETE。

- **DDL（Data Definition Language）数据定义语言**

用于数据库结构定义和管理，如：CREATE TABLE, ALTER TABLE, DROP TABLE。

17 非等值关联

非等值关联指连接条件中使用了非等号的比较符，如>, <, >=, <=，而非普通的等值匹配（=）。

```
sql
SELECT a.*, b.*
FROM tableA a
JOIN tableB b
ON a.value > b.value
```

应用场景：

- 时间范围匹配
- 阈值区间关联
- 层级关系匹配

注意：非等值关联通常比等值关联性能差，优化时需谨慎。

18 头部主播热点问题

- **问题描述：**头部主播通常流量大，数据访问和分析需求集中，可能出现热点数据访问瓶颈，导致系统压力大。
- **解决方案：**
  - **缓存机制**：使用 Redis 等缓存热点数据，减少数据库压力。

- **数据分片和负载均衡**：将热点数据分散存储，避免单节点过载。
- **异步处理**：将热点数据分析异步化，减少实时计算压力。
- **预计算和物化视图**：提前计算热点指标，快速响应查询。
- **限流和降级**：对热点请求做限流策略，保障整体服务稳定。

19 长周期退款和热点问题叠加解决方案

- **背景：**长周期退款数据涉及时间跨度长、数据量大，结合热点数据访问，系统负载和复杂度较高。
- **解决思路：**
  - **分层存储**：将长周期数据存入归档层或冷数据层，热点数据存热数据层。
  - **分区和分桶**：合理设计时间分区，优化查询范围。
  - **增量更新**：避免全量重算，使用增量数据处理。
  - **多级缓存策略**：热点数据和长周期常用数据分别缓存，减少重复计算。
  - **调度优化**：合理调度长周期数据的批处理时间，避开高峰期。

20 平时如何学习新技术

21 数据预处理时需要注意的点

- **数据质量检查：**缺失值、异常值、重复数据的识别与处理
- **数据格式规范：**统一时间格式、编码格式、字段类型
- **数据清洗：**剔除脏数据，填补缺失值或删除
- **数据转换：**字段拆分、合并、标准化、归一化
- **数据一致性：**保证主键唯一，数据之间关联完整
- **性能优化：**合理使用批处理和流处理，避免重复计算
- **记录数据变更：**日志记录数据清洗步骤，支持回溯和审计