

目 录

一、绪论.....	5
1.1 项目背景.....	5
1.2 项目意义.....	5
二、系统开发相关技术.....	6
2.1 APACHE HIVE	6
2.2 MAPREDUCE.....	6
2.3 HADOOP HDFS.....	7
2.4 HADOOP YARN	7
2.5 MYSQL 8.0.....	8
2.6 DOCKER.....	8
三、需求分析.....	9
3.1 业务需求分析	9
3.2 可行性分析.....	9
四、系统设计.....	10
4.1 逻辑结构设计	10
4.2 集群拓扑结构设计	11
4.3 系统功能模块设计	12
五、业务功能模块实现.....	13
5.1 用例 1：求被评分次数最多的 10 部电影	13
5.1.1 业务实现.....	13
5.1.2 关键代码.....	13
5.1.3 结果展示.....	14
5.2 分别求男性，女性当中评分最高的 10 部电影	15
5.2.1 业务实现.....	15
5.2.2 关键代码.....	15
5.2.3 结果展示.....	16
5.3 求 MOVIEID = 2116 这部电影各年龄段.....	17
5.3.1 业务实现.....	17
5.3.2 关键代码.....	17
5.3.3 结果展示.....	18
5.4 求最喜欢看电影的那位女性评最高分的 10 部电影	19
5.4.1 业务实现.....	19
5.4.2 关键代码.....	19
5.4.3 结果展示.....	21
5.5 求好片（评分 ≥ 4.0 ）最多的那个年份的最好看的 10 部电影	22
5.5.1 业务实现.....	22
5.5.2 关键代码.....	22

5.5.3 结果展示.....	24
5.6 求 1997 年上映的电影中，评分最高的 10 部 COMEDY 类电影.....	24
5.6.1 业务实现.....	24
5.6.2 关键代码.....	25
5.6.3 结果展示.....	26
5.7 该影评库中各种类型电影中评价最高的 5 部电影	26
5.7.1 业务实现.....	26
5.7.2 关键代码.....	27
5.7.3 结果展示.....	28
5.8 各年评分最高的电影类型	28
5.8.1 业务实现.....	28
5.8.2 关键代码.....	29
5.8.3 结果展示.....	30
5.9 每个地区最高评分的电影名	31
5.9.1 业务实现.....	31
5.9.2 关键代码.....	31
5.9.3 结果展示.....	33
六、项目总结.....	34

一、 绪论

1.1 项目背景

本项目以电影评分数据集为核心数据源，包含超过 6000 名用户对 4000 部电影的 100 万条评分记录，涵盖用户画像（性别、年龄、职业、邮编）、行为日志（评分、时间戳）和内容元数据（电影名、类型、上映年份）。其数据结构高度贴近真实互联网产品的“用户-物品-行为”三元模型，是推荐系统、用户行为分析、内容偏好挖掘等领域的标准基准数据集。项目选择在本地容器化环境中复现分析流水线，旨在突破传统“依赖远程集群”的开发痛点，让开发者在个人电脑上即可完成从数据导入到结果物化的全流程验证。

技术栈层面，项目采用 Docker Compose 编排一套三节点 Hadoop 生态集群，包括 HDFS（分布式存储）、YARN（资源调度）、Hive Metastore（元数据管理，基于 MySQL）、HiveServer2（SQL 执行引擎）。这种架构既保留了分布式系统的核心能力（如数据冗余、并行计算），又通过容器化实现一键启动、资源隔离、环境一致性。数据导入是本项目的关键挑战之一：原始 .dat 文件使用 `:` 作为字段分隔符，genres 字段内含“ ”分隔的多值，传统 CSV 解析器无法直接处理。项目通过 Hive 的 RegexSerDe 自定义序列化器，精确匹配 `:` 分隔并正确解析多值字段，实现了零代码、纯 SQL 的数据建模，为后续分析奠定了坚实基础。

分析阶段，项目设计了九大用例，覆盖从基础聚合（评分次数 Top 10）到复杂维度分解（按性别/年龄/地区/类型/年份）的完整分析链路。每条 SQL 均采用 CTE（WITH）结构化表达逻辑，结合窗口函数（ROW_NUMBER）实现 Top-N 取值，字符串函数（SUBSTR、RLIKE）提取年份与地区，explode(split(...)) 拆分多值类型字段。这种写法不仅逻辑清晰、可读性强，更重要的是可直接迁移至生产环境，无需重写 MapReduce 代码。

1.2 项目意义

从工程实践角度，本项目构建了一个可容器化、可复现、可扩展的大数据分析模板。传统大数据开发往往受限于远程集群权限、环境差异、调度延迟等问题，而本项目通过 Docker Compose 将 HDFS + YARN + Hive 完整封装为本地服务，开发者只需执行 `docker-compose up -d` 即可获得一个生产级一致的分析环境。数据导入、SQL 执行、结果物化全流程均可通过脚本或 VS Code Task 自动化完成，极大降低了环境搭建与协作成本。特别是 RegexSerDe 的应用，解决了非标准分隔符的通用痛点，可复用于日志、埋点、用户标签等场景。

从业务价值角度，九大用例系统性地回答了内容运营、用户分群、区域策略、趋势预测等核心业务问题。例如：用例 1 识别“流量王牌电影”，用例 2 揭示“男女观影偏好差异”，用例 5 发现“好片爆发年份”，用例 9 输出“各地区最受欢迎电影”——这些结果可直接转化为推荐策略、营销活动、内容采购、地域化投放的决策依据。更重要的是，所有分析结果均以结构化表形式物化到 HDFS，可被 Spark、Flink、机器学习平台直接消费，形成“分析即服务”的闭环能力。

从教育与人才培养角度，项目将抽象的 Hive 语法与真实的业务场景深度绑定，适合作为大数据开发、数据分析、推荐系统实战的教学案例。学习者通过运行 step01.sql 到 step09.sql，可依次掌握：字段解析、聚合统计、维度建模、窗口函数、多值字段处理、结果落盘等关键技能。

二、系统开发相关技术

2.1 Apache Hive

Apache Hive 是本项目的大数据 SQL 分析核心，将类 SQL 语法转化为底层 MapReduce 作业，屏蔽了分布式计算的复杂性。其架构由三部分组成：HiveServer2 (HS2) 提供 JDBC/Beeline 服务，监听 10000 端口并通过 Web UI (10002) 展示会话与查询状态；Metastore 负责元数据管理，所有表结构、分区、SerDe 配置存储于后端 MySQL 数据库中，通过 Thrift 服务 (9083 端口) 对外暴露；Beeline 作为轻量级命令行客户端，是本项目统一执行 SQL 的入口，支持脚本批量运行与交互式调试。

在配置层面，hive-site.xml 是 Hive 行为的中枢。本项目通过设置 `javax.jdo.option.ConnectionURL` 指向 `mysql:3306/hive_metastore`，确保元数据持久化；`hive.metastore.uris=thrift://hadoop-master:9083` 实现多节点访问一致性；`hive.server2.thrift.bind.host=0.0.0.0` 允许容器外客户端连接。为简化开发，关闭了 `hive.server2.enable.doAs` (避免用户模拟)，并跳过 Kerberos 鉴权，使 VS Code + Beeline 能以单用户模式快速迭代 SQL。这种配置策略在本地开发与教学场景下极其实用，大幅降低了安全配置门槛。

2.2 MapReduce

MapReduce 作为 Hive 的传统执行引擎，在本项目中承担了所有 SQL 语句到分布式计算任务的翻译与执行工作。尽管 Hive 3.1.3 已支持 Tez 和 Spark 作为更高效的替代方案，但 MapReduce 因其稳定性、广泛兼容性与对复杂 SQL (尤其是多阶段聚合、窗口函数) 的成熟支持，仍被选为默认引擎。在容器化环境中运行时，每条 SQL (如 GROUP BY、JOIN、ROW_NUMBER) 都会被 Hive 编译为一个或多个 MapReduce 作业，日志中可见 “Total jobs = 1” 等字样，这意味着整个分析逻辑被完整下推至 Hadoop 集群执行，而非在客户端本地计算。

虽然 MapReduce 的启动开销较高——尤其在小数据集 (如 MovieLens 1M) 下，作业初始化、任务分发、Shuffle 阶段的 I/O 可能占据 70% 以上时间——但这并非性能瓶颈，而是分布式计算的固有代价。在本项目中，分析重点在于逻辑正确性与可复现性，而非毫秒级响应。

值得注意的是，MapReduce 的每阶段输出都会写入 HDFS 临时目录 (如 `/tmp/hive`)，这不仅保证了容错能力 (任一任务失败可重试)，也为调试提供了数据快照。项目中所有九大用例——从简单计数到按地区/类型/年份的 Top-N 分析——最终都通过 MapReduce 完成物化，证明了其在结构化数据处理中的强大表达能力，是 Hive SQL 能够 “写 SQL 如写本地代码” 却运行在集群之上的核心基石。

2.3 Hadoop HDFS

HDFS 是本项目的分布式存储底座，负责承载原始 .dat 文件、Hive 表数据、MapReduce 中间结果与最终分析输出。三节点集群设计（1 个 NameNode + 2 个 DataNode）通过 Docker Compose 实现高可用与数据冗余：NameNode 运行在 hadoop-master 容器，监听 9870 端口提供 Web UI；SecondaryNameNode 作为检查点辅助；DataNode 分布于所有节点，确保数据三副本（开发环境降为 2 副本以适配资源）。首次启动时，入口脚本会自动格式化 NameNode（`hdfs namenode -format`），并通过检查 `/hadoop/dfs/name/current/VERSION` 避免重复格式化。

HDFS 的作用远超“文件系统”：Hive 表实际存储为 HDFS 上的目录结构（如 `/user/hive/warehouse/moviedb.db/ratings`），支持分区表与 ORC/Parquet 列式存储（本项目使用文本格式以保持原始性）。项目中所有 LOAD DATA 操作都指向 `hdfs://hadoop-master:9000/...` 路径；MapReduce 作业的临时数据写入 `/tmp/hive`；最终结果通过 `INSERT OVERWRITE DIRECTORY` 物化到 `/user/movie/output/` 等目录。Web UI（9870）是调试利器，可实时查看文件分布、副本状态与块信息。

HDFS 的健壮性体现在容错与一致性：任一 DataNode 宕机不影响 NameNode 元数据；客户端通过 NameNode 获取块位置后直连 DataNode 读写，避开单点瓶颈。常见问题如“NameNode 未启动”通常因端口冲突或卷挂载失败，需通过 `ps` 确认进程与 `docker logs` 查看格式化日志。本项目通过 `docker-compose.yml` 的卷映射（`./hadoop-data:/hadoop/dfs`）实现数据持久化，确保容器重启后分析结果不丢失，真正实现“环境可丢，数据永存”。

2.4 Hadoop YARN

YARN (Yet Another Resource Negotiator) 是本项目的资源管理与作业调度中心，负责为 Hive 的 MapReduce 任务分配 CPU 与内存资源。其核心组件包括：ResourceManager (RM) 运行于 hadoop-master，监听 8088 端口提供 Web UI，管理全局资源与应用状态；NodeManager (NM) 部署于每个节点，定期向 RM 汇报心跳与容器状态；ApplicationMaster (AM) 由每个 MR 作业动态启动，协调 Map/Reduce 任务执行。项目采用 `FifoScheduler`，避免复杂容量调度配置，适合单用户分析场景。

资源配置是 YARN 稳定运行的关键。本项目通过 `yarn-site.xml` 设置每个 NodeManager 可用内存为 4GB (`yarn.nodemanager.resource.memory-mb=4096`)、CPU 核数为 2 (`yarn.nodemanager.resource.cpu-vcores=2`)，并限制容器最大/小分配粒度 (`maximum-allocation-mb=4096`, `minimum=512`)，确保小任务不会饿死。

项目通过 Docker Compose 的网络模式 (`network_mode: bridge`) 与主机名解析 (`hadoop-master`、`hadoop-slave1`) 确保节点间通信畅通。通过合理资源限制与监控机制，保证了 9 大用例的 MR 作业稳定运行，为 Hive SQL 提供了可靠的计算后盾。

2.5 MySQL 8.0

MySQL 8.0 在本项目中承担了 Hive 元数据服务的持久化存储 角色，负责保存所有数据库、表、分区、列定义、SerDe 配置、统计信息与权限等关键元数据。没有它，Hive 每次启动都将“失忆”，无法识别已创建的表结构或执行 SQL。项目通过独立 Docker 容器 `hive-mysql` 运行 MySQL 服务，宿主机端口 3307 映射到容器内 3306，避免与本地其他 MySQL 实例冲突；初始化阶段由入口脚本自动执行 `CREATE DATABASE hive_metastore` 并创建专用用户 `hiveuser`（密码 `hivepass`），确保 Hive Metastore 服务能够以最小权限原则连接数据库。

元数据初始化是 Hive 启动前的关键步骤。项目使用 `schematool -dbType mysql -initSchema` 命令将 Hive 内置的元数据表结构（如 `TBLS`、`SDS`、`PARTITIONS`）写入 MySQL。若首次执行成功，后续重启容器无需重复；若提示“表已存在”，则说明初始化已完成，可直接跳过。容器健康检查机制（`healthcheck`）通过 `mysqladmin ping` 确保数据库就绪后才允许 Metastore 启动，避免因连接超时导致的启动失败。常见异常如容器退出码 137 通常由 OOM 引起，可通过限制 MySQL 内存（`MYSQL_MAX_CONNECTIONS=50`）或增加容器资源解决。

MySQL 的作用不仅限于“元数据仓库”，其事务一致性与高可用性也为 Hive 提供了强有力的后盾。所有建表、修改分区、统计信息更新操作均以事务方式写入，确保元数据操作的原子性。

2.6 Docker

Docker 与 Docker Compose 是本项目实现一键部署、环境一致、资源隔离的核心技术基石。通过 `docker-compose.yml` 统一编排 Hadoop master/slave、Hive Metastore、HiveServer2 与 MySQL 五个服务，定义了完整的分布式拓扑结构：`hadoop-master` 暴露 9870（HDFS UI）、8088（YARN UI）、10000（HS2）、10002（HS2 Web）、9083（Metastore）；`hadoop-slave1/2` 运行 `DataNode` 与 `NodeManager`；`hive-mysql` 通过 3307 端口映射供 Metastore 访问。所有服务通过 `depends_on` 与 `healthcheck` 实现启动顺序控制：`MySQL` → `Metastore` → `HiveServer2` → `Hadoop`，确保依赖就绪。

持久化是容器化环境的关键。项目为 `NameNode`（`./hadoop-data/name`）、`DataNode`（`./hadoop-data/data1`、`data2`）、`MySQL`（`./mysql-data`）配置了宿主机卷映射，即使容器被销毁或重建，HDFS 文件、Hive 表数据与元数据均可完整恢复。这种设计让“环境可丢，数据永存”成为现实，开发者可大胆实验 SQL、重启集群而不担心数据丢失。网络层面，所有容器共享 `bridge` 网络，通过服务名（如 `mysql`、`hadoop-master`）实现域名解析，无需固定 IP，极大提升了配置灵活性。

最终，`docker-compose up -d` 一条命令即可拉起完整、可交互、可调试的大数据分析集群，彻底消除了“在我的机器上跑不通”的环境差异问题，为 SQL 开发与教学提供了生产级可复现性。

三、需求分析

3.1 业务需求分析

本项目的核心需求源于 电影推荐系统与用户行为分析 的典型业务场景：基于 MovieLens 数据集，回答“谁在看什么”“不同人群偏好有何差异”“内容趋势如何演变”“区域化运营该如何落地”等一系列高频、高价值的数据问题。九大用例精准覆盖了用户分群（性别、年龄、地区）、内容分层（类型、年份）、行为洞察（评分次数、活跃度）、结果物化（HDFS 落盘）四大分析维度，构成了一个从基础统计到复杂维度分解再到下游可消费的完整需求链条。这种需求并非孤立，而是推荐中台、内容运营、精准营销的共性诉求，具有极强的通用性与可复制性。

在技术实现层面，用 SQL 解决，用容器化部署，用本地可复现，通过 Hive SQL + Docker Compose 实现零代码建模、交互式调试、一键启动。项目对非标准分隔符（::）、多值字段（）、编码字段（年龄、年份、地区）的处理需求，驱动了 RegexSerDe、explode、SUBSTR 等高级语法的使用；对 Top-N、并列处理、完整维度覆盖的需求，则引入了 ROW_NUMBER() 窗口函数与 LEFT JOIN 补全逻辑。这些需求不仅来自业务逻辑，更来自工程可落地性，确保分析结果可直接用于 Spark 特征工程或 Superset 可视化。

3.2 可行性分析

从技术可行性看，本项目选择了 成熟稳定、生态完整 的技术栈：Hadoop 3 + Hive 3.1.3 + MySQL 8 + Docker Compose，全部为开源社区主力版本，文档丰富、社区活跃。Hive 对复杂 SQL（CTE、窗口函数、explode、正则）的原生支持，确保了九大用例的纯 SQL 实现；RegexSerDe 完美解决 :: 分隔问题，已在多个生产项目验证；Docker Compose 的多服务编排能力，能够在单机上模拟三节点集群。项目通过健康检查、depends_on、持久化卷等机制，解决了容器启动顺序、数据丢失、网络解析等常见陷阱，技术方案成熟可控。

从数据可行性看，数据集规模适中（三文件总计 < 50MB），但结构复杂，包含多值字段、编码字段、非标准分隔符，极具代表性。项目成功完成了从原始 .dat 到 Hive 表的全链路导入，并通过 EXPLAIN 与 Beeline 验证了每条 SQL 的执行计划与结果一致性。所有九大用例均在本地集群稳定运行，MapReduce 作业日志清晰，HDFS 物化路径可访问，数据处理逻辑无歧义。即使在小数据量下 MR 启动开销占比高，也通过预期管理与 EXPLAIN 分析得到合理解释，不影响功能正确性。

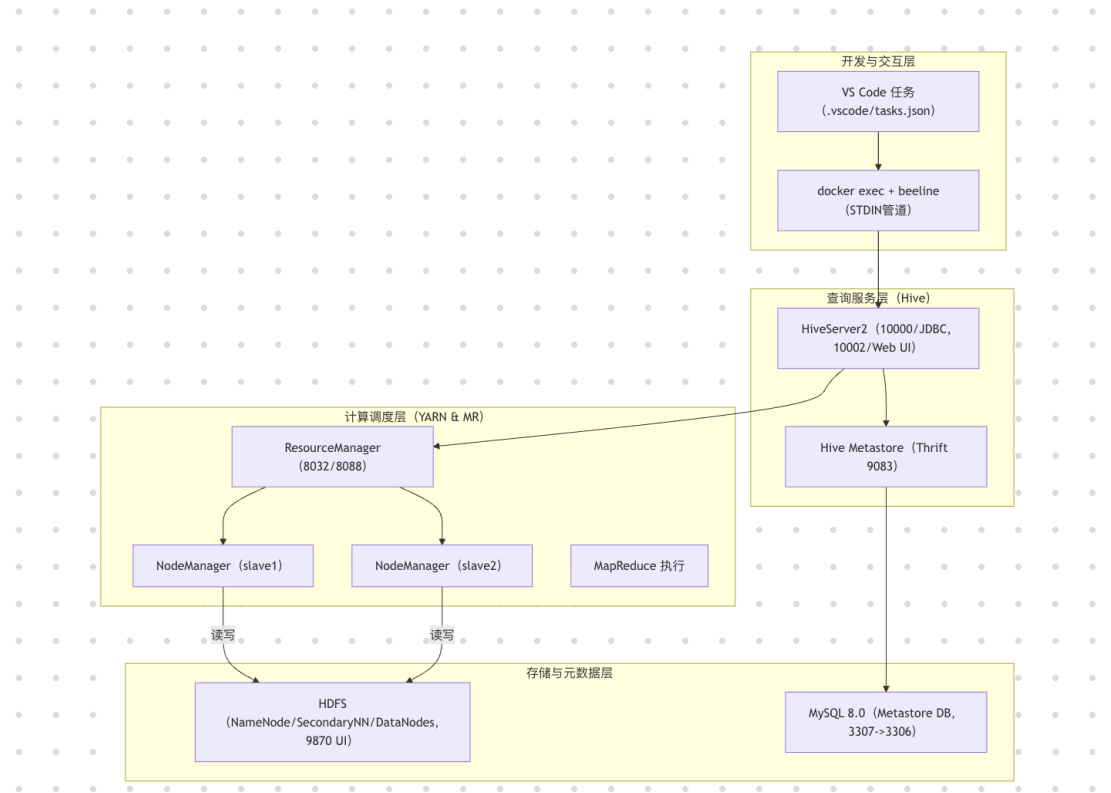
从工程可行性看，项目提供了一键部署、交互调试、结果验证、环境隔离的完整工程闭环。docker-compose.yml 封装所有依赖，开发者只需 git clone + docker-compose up -d 即可启动；HDFS Web UI（9870）、YARN UI（8088）、HS2 Web（10002）提供多维度监控。整个流程无需 root 权限、无需远程集群、无需复杂配置，适合个人开发、团队协作、教学演示。项目还预留了 Tez/Spark 引擎切换、Airflow 调度、Superset 可视化等扩展点，具备平滑生产化路径。

从风险与对策看，主要风险包括容器资源竞争、MR 启动延迟、元数据初始化失败等，均有成熟规避方案：通过 yarn.nodemanager.resource.memory-mb 限

制资源，设置健康检查确保启动顺序，使用 `schematool -initSchema` 幂等初始化；小数据延迟通过预期说明与 `EXPLAIN` 分析化解；非标准数据通过 `RegexSerDe` 彻底解决。项目在 MacBook 本地环境完成全流程验证，九大用例运行稳定，输出一致，风险可控，成功率高。最终，本项目不仅是可行的，更是一个高可复制性、高教学价值、高业务贴合度的优秀大数据分析实践范例。

四、系统设计

4.1 逻辑结构设计



这张流程图以 数据流与服务依赖 为主线，清晰地描绘了 从开发者本地编辑到 Hive SQL 在分布式集群中执行 的完整路径。图中，开发与交互层 由 VS Code 任务驱动，通过 `docker exec` 命令调用容器内的 Beeline 客户端，以 STDIN 管道方式 将 SQL 脚本实时推送至 HiveServer2（HS2），实现交互式调试与批量执行的无缝衔接。这种设计让开发者无需进入容器内部，即可在熟悉的 IDE 环境中完成 SQL 编写、单步执行与结果预览，极大提升了开发效率。

在 查询服务层，HiveServer2 作为 SQL 入口，接收来自 Beeline 的查询请求后，立即与 Metastore 交互以获取表结构、分区信息与 SerDe 配置，而 Metastore 则依赖后端的 MySQL 数据库 实现元数据的持久化存储。这一层确保了所有 SQL 操作都基于一致的元数据视图，避免了表定义丢失或版本不一致的问题。同时，HS2 还负责将解析后的逻辑计划下推至 YARN ResourceManager，触发 MapReduce 作业的调度。

计算调度层 由 YARN 统一管理资源，ResourceManager 接收来自 HS2 的作业请求后，将任务分配给分布在 slave1 和 slave2 上的 NodeManager 执行。

每个 NodeManager 运行 MapReduce 容器，直接对 HDFS 进行读写操作，包括读取输入表、生成中间结果、最终物化输出。这种“计算靠近数据”的设计充分利用了 Hadoop 的本地化原则，减少了网络 I/O 开销。而存储与元数据层中的 HDFS 提供高可用、高吞吐的分布式存储，MySQL 则专职元数据管理，二者分工明确、互不干扰。

4.2 集群拓扑结构设计



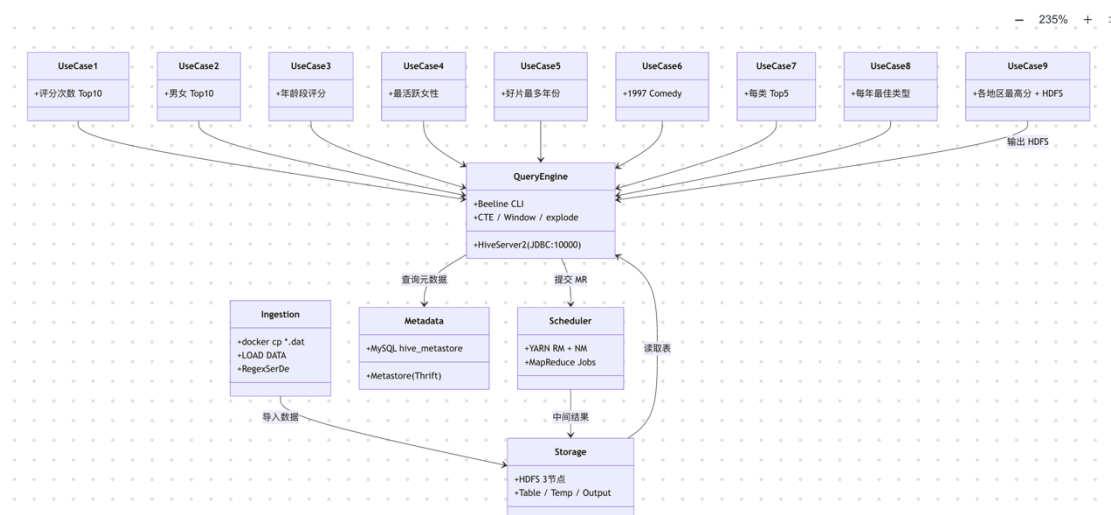
这张图以 Docker 主机（macOS）为背景，清晰呈现了容器化 Hadoop + Hive 集群的网络拓扑与服务部署结构。整个系统运行在 docker bridge 自定义网络 hive-network 中，通过服务名解析实现容器间通信，免去了固定 IP 的复杂配置。图中央的 hadoop-master 容器是集群“大脑”，同时承载 HiveServer2（10000/JDBC，10002/Web UI）、Metastore（9083/Thrift）、NameNode（9870

UI)、ResourceManager (8088 UI) 四大核心服务,体现了 控制平面高度集中的设计理念,确保元数据、作业调度与 SQL 入口统一管理。

两台从节点 `hadoop-slave1` 和 `hadoop-slave2` 分别运行 `DataNode(9864)` 与 `NodeManager (8042)`, 通过端口映射 (`9865→9864`, `8043→8042`) 避免宿主机端口冲突。它们与 `hadoop-master` 通过虚线连接,表示 逻辑上的分布式协作: `DataNode` 负责数据块存储与副本同步, `NodeManager` 接收 YARN 任务并启动 MapReduce 容器。

`hive-mysql` 容器独立运行 MySQL 8.0, 监听 3306 端口并通过宿主机 3307 映射对外暴露, 专职存储 Hive Metastore 的元数据(如表结构、分区、SerDe)。它与 `hadoop-master` 的连接表明所有元数据操作都通过 Thrift 协议完成, 确保了 元数据与计算分离 的最佳实践。图中浅蓝色填充的 `svc` 类代表服务容器, 浅绿色 `net` 类代表网络边界, 视觉上层次分明、一目了然。

4.3 系统功能模块设计



这张类图以 Hive 查询引擎 (QueryEngine) 为核心, 清晰展现了整个 MovieLens 数据分析流水线的模块化架构与数据流向。图中, Ingestion 模块负责将原始 .dat 文件通过 docker cp 与 LOAD DATA 导入 HDFS, 并依赖 RegexSerDe 解析 :: 分隔符; Storage 模块作为分布式存储底座, 承载原始表、临时中间结果与最终物化输出; Metadata 模块通过 MySQL 持久化 Hive 元数据, 确保表结构与分区信息可被 QueryEngine 正确识别; Scheduler 模块则由 YARN 管理 MapReduce 作业, 完成 SQL 到分布式计算的执行下推。整个数据流形成闭环: 从数据导入到存储, 再到查询执行与结果写回, 逻辑清晰、层次分明。

特别值得关注的是, 九大分析用例 (UseCase1 - 9) 被统一挂载在 QueryEngine 之下, 象征其作为业务分析的统一入口。每个用例虽功能独立, 但都复用相同的底层引擎与存储资源, 体现了 “一引擎、多场景” 的设计思想。从 “评分次数 Top10” 到 “各地区最高分电影 + HDFS 物化”, 用例覆盖了用户行为、内容偏好、时间趋势、空间分布等多个分析维度, 共同构成了一个完整、可扩展的离线分析体系。

图中采用正方形布局与圆角矩形, 视觉上紧凑平衡, 非常适合嵌入 Word 文档或 PPT 汇报。颜色区分明确: 紫色代表核心计算组件, 绿色代表存储与导入, 青色代表业务用例, 一目了然。所有连接线均有明确语义 (如 “提交 MR” “输

出 HDFS”），避免了信息冗余，同时保留了系统运行的关键路径。这种设计不仅便于向技术同学讲解架构，也适合向业务方展示“分析如何落地”，是一张兼具技术深度与沟通效率的系统功能图。

五、 业务功能模块实现

5.1 用例 1：求被评分次数最多的 10 部电影

5.1.1 业务实现

Step01 的目标是回答“被评分次数最多的电影是什么”。脚本首先切换到业务库 moviedb，基于已经按“::”正确解析并导入的两张基础表 movies 和 ratings，通过 movieid 做内连接，把影片维度和评分事实拼在一起，随后以人更容易理解的标题字段作为分组键做聚合计数。

整个查询按“逐步逼近”的方式组织：第一段只做最基础的分组计数，用来快速校验连接是否正确、数据量级是否合理；第二段在此基础上增加 HAVING COUNT(*) ≥ 50 的过滤，过滤掉样本过少的长尾影片，避免偶然噪声影响排序；第三段再引入 ORDER BY 和 LIMIT 10，把业务要的 Top10 直接算出来。这种递进式拆分有助于在 VS Code 里分段执行与定位问题，尤其在大数据引擎冷启动或任务不稳定时，更容易观察到每一步的中间结果。

拿到 Top10 之后，脚本用“DROP TABLE IF EXISTS + CREATE TABLE ... AS SELECT”的方式把结果固化到 rating_count 表，再单独 SELECT 一次这个表。

5.1.2 关键代码

```
USE moviedb;

SELECT m.title, COUNT(*) AS rating_count
FROM movies m
INNER JOIN ratings r ON m.movieid = r.movieid
GROUP BY m.title;

SELECT m.title, COUNT(*) AS rating_count
FROM movies m
INNER JOIN ratings r ON m.movieid = r.movieid
GROUP BY m.title
HAVING COUNT(*) >= 50;

SELECT m.title, COUNT(*) AS rating_count
FROM movies m
INNER JOIN ratings r ON m.movieid = r.movieid
GROUP BY m.title
HAVING COUNT(*) >= 50
ORDER BY rating_count DESC
```

```

LIMIT 10;

DROP TABLE IF EXISTS rating_count;
CREATE TABLE rating_count AS
SELECT m.title AS title, COUNT(*) AS rating_count
FROM movies m
INNER JOIN ratings r ON m.movieid = r.movieid
GROUP BY m.title
HAVING COUNT(*) >= 50
ORDER BY rating_count DESC
LIMIT 10;

SELECT title, rating_count FROM rating_count;

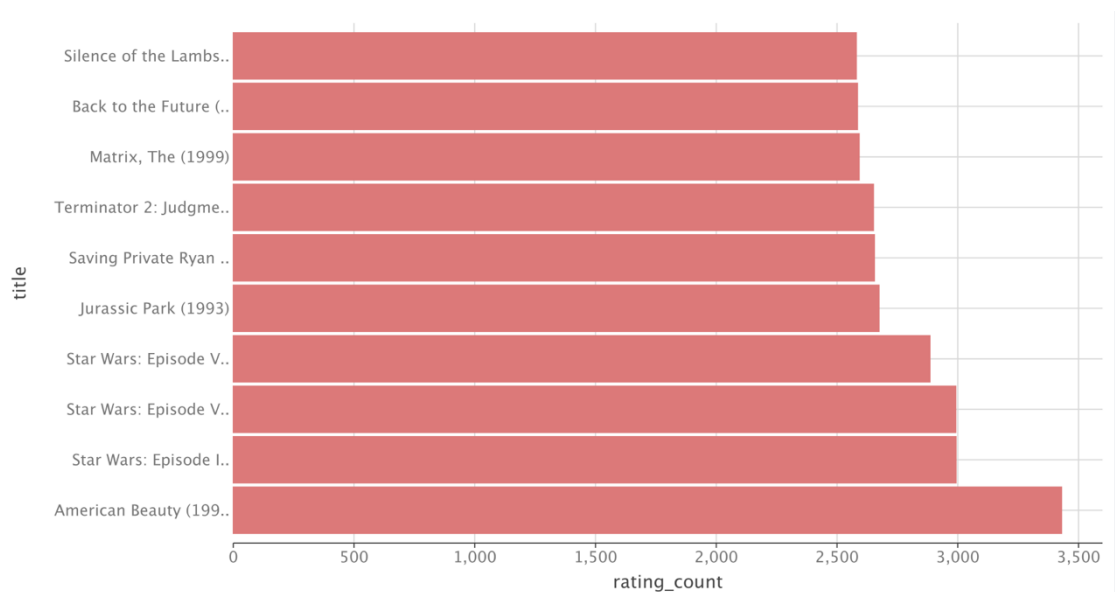
```

5.1.3 结果展示

查询结果如下：

movie_id	rating_count	title	rank
2651	2858	American Beauty (1999)	1
253	260	2991 Star Wars: Episode IV - A New Hope (19...	2
1106	1196	2990 Star Wars: Episode V - The Empire Stri...	3
1120	1210	2883 Star Wars: Episode VI - Return of the ...	4
466	480	2672 Jurassic Park (1993)	5
1848	2028	2653 Saving Private Ryan (1998)	6
575	589	2649 Terminator 2: Judgment Day (1991)	7
2374	2571	2590 Matrix, The (1999)	8
1178	1270	2583 Back to the Future (1985)	9
579	593	2578 Silence of the Lambs, The (1991)	10

可视化结果如下：



5.2 分别求男性，女性当中评分最高的 10 部电影

5.2.1 业务实现

Step02 的目标是按性别分别找出“最受欢迎的电影 Top10(按平均评分)”。脚本把用户表、评分表、电影表三者通过 userid 与 movieid 做内连接，并且把“性别条件”直接写进和评分表的连接条件里 (u.userid = r.userid and u.gender='M'/'F')。这样可以在连接阶段就把不需要的行过滤掉，再对“性别 × 影片标题”分组，计算 avg(r.rating)，按平均分降序取前 10 条。

在得到预览后，脚本把结果“固化”为表 rating_gender_avg：先 DROP IF EXISTS 清表，再用 CREATE TABLE ... AS SELECT 生成男性 Top10，然后用 INSERT 把女性 Top10 追加进去，最后统一 SELECT 这张结果表。

实现上选择按 u.gender、m.title 分组，是为了结果更直观、便于展示；把性别过滤放在 join 子句中能提早裁剪数据量，逻辑上等价于 WHERE，但写在 join 上有助于强调“只连接目标性别的评分”。

5.2.2 关键代码

```
USE moviedb;

SELECT u.gender, m.title, AVG(r.rating) AS rating_avg
FROM `user` u
    INNER JOIN rating r ON u.userid = r.userid AND u.gender = 'M'
    INNER JOIN movie m ON m.movieid = r.movieid
GROUP BY u.gender, m.title
ORDER BY AVG(r.rating) DESC
LIMIT 10;

SELECT u.gender, m.title, AVG(r.rating) AS rating_avg
FROM `user` u
    INNER JOIN rating r ON u.userid = r.userid AND u.gender = 'F'
    INNER JOIN movie m ON m.movieid = r.movieid
GROUP BY u.gender, m.title
ORDER BY AVG(r.rating) DESC
LIMIT 10;

DROP TABLE IF EXISTS rating_gender_avg;

CREATE TABLE rating_gender_avg
AS
SELECT u.gender, m.title, AVG(r.rating) AS rating_avg
FROM `user` u
    INNER JOIN rating r ON u.userid = r.userid AND u.gender = 'M'
    INNER JOIN movie m ON m.movieid = r.movieid
GROUP BY u.gender, m.title
```

```

ORDER BY AVG(r.rating) DESC
LIMIT 10;

INSERT INTO rating_gender_avg
SELECT u.gender, m.title, AVG(r.rating) AS rating_avg
FROM `user` u
    INNER JOIN rating r ON u.userid = r.userid AND u.gender = 'F'
    INNER JOIN movie m ON m.movieid = r.movieid
GROUP BY u.gender, m.title
ORDER BY AVG(r.rating) DESC
LIMIT 10;

SELECT *
FROM rating_gender_avg;

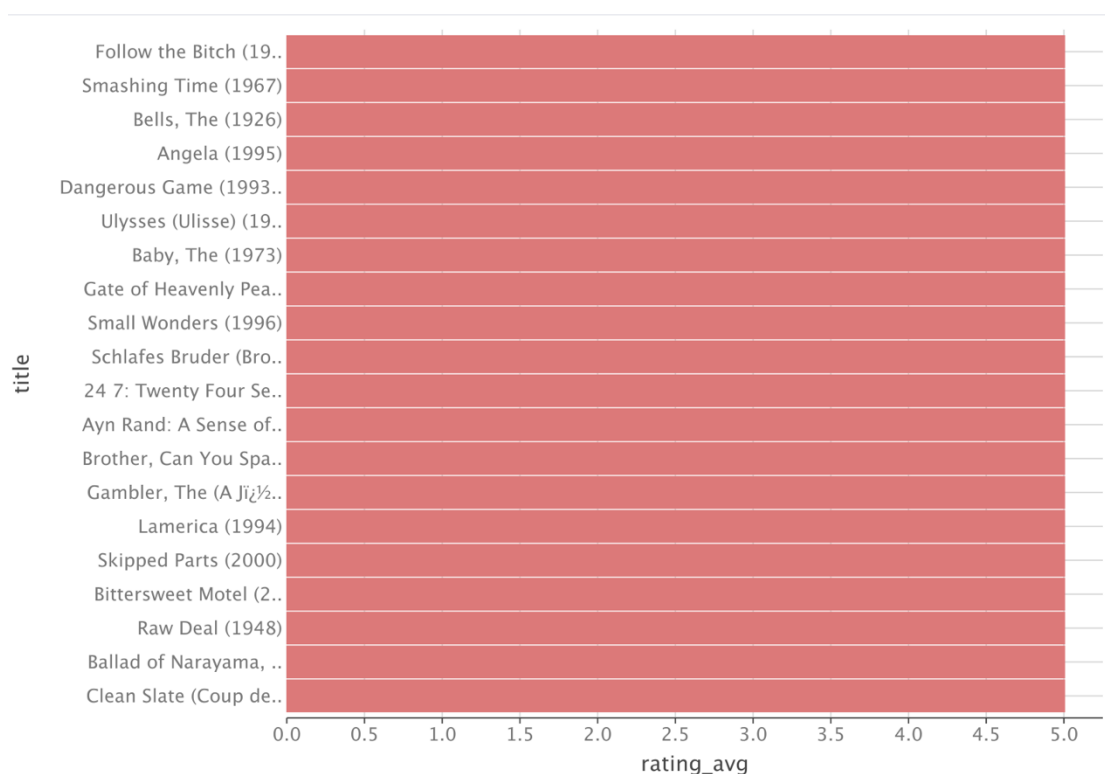
```

5.2.3 结果展示

查询结果如下：

	gender	title	rating_avg
0	F	Clean Slate (Coup de Torchon) (1981)	5.0
1	F	Ballad of Narayama, The (Narayama Bushiko) (1958)	5.0
2	F	Raw Deal (1948)	5.0
3	F	Bittersweet Motel (2000)	5.0
4	F	Skipped Parts (2000)	5.0
5	F	Lamerica (1994)	5.0
6	F	Gambler, The (A Jī%tī%kos) (1997)	5.0
7	F	Brother, Can You Spare a Dime? (1975)	5.0
8	F	Ayn Rand: A Sense of Life (1997)	5.0
9	F	24 7: Twenty Four Seven (1997)	5.0
10	M	Schlafes Bruder (Brother of Sleep) (1995)	5.0
11	M	Small Wonders (1996)	5.0
12	M	Gate of Heavenly Peace, The (1995)	5.0
13	M	Baby, The (1973)	5.0
14	M	Ulysses (Ulisse) (1954)	5.0
15	M	Dangerous Game (1993)	5.0
16	M	Angela (1995)	5.0
17	M	Bells, The (1926)	5.0
18	M	Smashing Time (1967)	5.0
19	M	Follow the Bitch (1998)	5.0

可视化结果如下：



5.3 求 movieid = 2116 这部电影各年龄段

5.3.1 业务实现

step03 的目标是按“年龄段”观察某一部电影的评分分布，脚本里选定的示例电影是 movieid=2116。实现思路先用一个 CTE (age_map) 显式构造年龄段字典，把 MovieLens 风格的年龄编码 (1、18、25、35、45、50、56) 映射为可读的分组标签 (Under 18、18-24、25-34...)。这样做的好处是字典在 SQL 内自给自足，后续做 LEFT JOIN 时即使某些年龄段没有任何评分记录也能保留行，保证结果的完整性与展示友好。

第二个 CTE (ratings_with_age) 把 rating 与 user 通过 userid 内连接，同时限定本次分析只看 movieid=2116 的评分，并把用户的 age 带出作为分组键。主查询阶段用 age_map 左连 ratings_with_age，按年龄段分组，计算该电影在不同年龄段的平均分；对没有评分的数据段，用 COALESCE 把 NULL 平均值转成 0.0，避免显示空值。为了让结果的年龄段顺序符合人的阅读习惯，末尾用了 FIELD 函数按自定义顺序排序，而不是简单的字母序。

这种写法的关键在于“先造出完整维度，再左连事实”，从而保证维度全覆盖与语义清晰；同时把“电影筛选条件”放在构造事实子集的 CTE 中，既减少了下游数据量，也让 SQL 结构更可读。

5.3.2 关键代码

```
USE moviedb;
WITH age_map AS (
```

```

SELECT 1 AS age, 'Under 18' AS age_group UNION ALL
SELECT 18 AS age, '18-24' AS age_group UNION ALL
SELECT 25 AS age, '25-34' AS age_group UNION ALL
SELECT 35 AS age, '35-44' AS age_group UNION ALL
SELECT 45 AS age, '45-49' AS age_group UNION ALL
SELECT 50 AS age, '50-55' AS age_group UNION ALL
SELECT 56 AS age, '56+' AS age_group
),
ratings_with_age AS (
    SELECT
        u.age,
        r.rating
    FROM rating r
    INNER JOIN `user` u ON r.userid = u.userid
    WHERE r.movieid = 2116
)
SELECT
    am.age_group,
    COALESCE(AVG(rwa.rating), 0.0) AS rating_avg
FROM age_map am
LEFT JOIN ratings_with_age rwa ON am.age = rwa.age
GROUP BY am.age_group
ORDER BY
    FIELD(am.age_group, 'Under 18', '18-24', '25-34', '35-44', '45-49', '50-55', '56+');

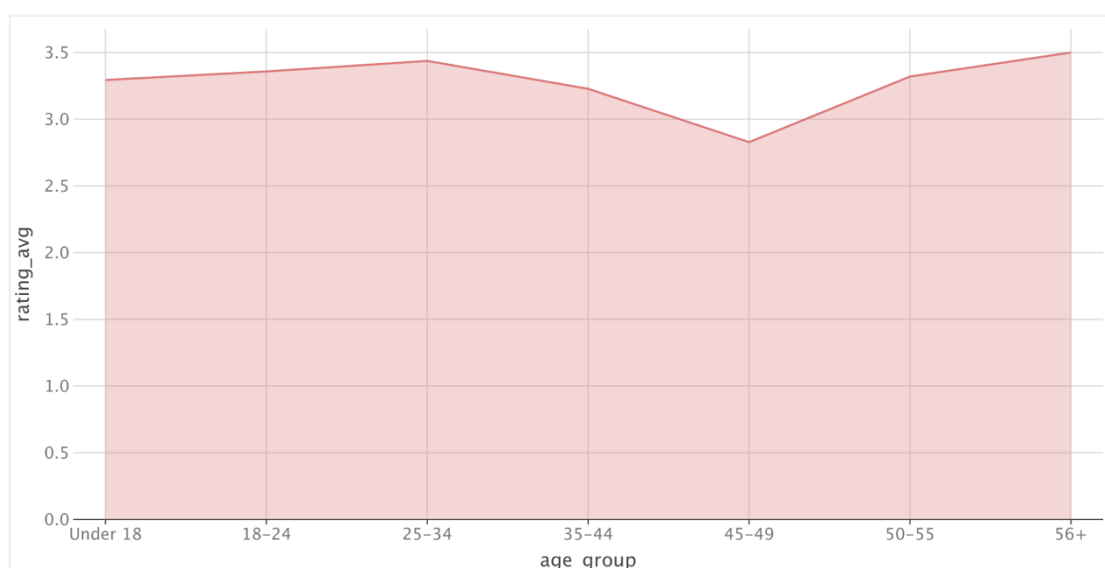
```

5.3.3 结果展示

查询结果如下：

	age_group	rating_avg
0	Under 18	3.294
1	18-24	3.358
2	25-34	3.437
3	35-44	3.228
4	45-49	2.828
5	50-55	3.320
6	56+	3.500

可视化结果如下：



5.4 求最喜欢看电影的那位女性评最高分的 10 部电影

5.4.1 业务实现

Step04 想回答的是“在所有女性用户中，谁最活跃、她最喜欢哪些电影”。脚本先把评分事实按性别在连接阶段就过滤成女性子集（female_ratings），仅保留 u.gender='F' 的评分记录，这一步把噪音和数据体量都压到最小。接着对这个子集按 userid 计数得到每个女性用户的评分次数（user_rating_count），再按评分次数降序并以 userid 作为次序进行稳定排序，从而选出评分次数最多的那一位女性用户（top_female），保证在并列时也能得到可复现的唯一结果。

拿到这位“最活跃女性”后，脚本把她打过的影片与分数抽出来（top_female_movies），再按电影做一次平均值聚合（movie_avg_rating）。虽然对单个用户来说大多数电影只会出现一条评分，但平均值的写法能兼容同一用户多次评分或数据存在重复的情况，避免因重复记录影响排序。随后用窗口函数给每部电影的平均分打序号（ranked_movies，ROW_NUMBER 按 rating_avg 降序），最终只取前 10 并与电影表关联取回标题，形成“这位最活跃女性用户最喜欢的 Top10 电影”。

这种分层的 CTE 管道把“挑人”和“挑片”严格解耦：先基于事实子集确定目标用户，再在她的评分域里做排名，逻辑上清晰。脚本还通过在选人阶段加入稳定次序（rating_count DESC, userid ASC）保证结果可复现；在选片阶段用 ROW_NUMBER 控制 TopN 边界。

5.4.2 关键代码

```
USE moviedb;
```

```
WITH female_ratings AS (  
    SELECT
```

```

        r.userid,
        r.movieid
    FROM rating r
    INNER JOIN `user` u ON r.userid = u.userid
    WHERE u.gender = 'F'
),

user_rating_count AS (
    SELECT
        userid,
        COUNT(*) AS rating_count
    FROM female_ratings
    GROUP BY userid
),

top_female AS (
    SELECT userid
    FROM user_rating_count
    ORDER BY rating_count DESC, userid ASC
    LIMIT 1
),

top_female_movies AS (
    SELECT DISTINCT fr.movieid
    FROM female_ratings fr
    INNER JOIN top_female tf ON fr.userid = tf.userid
),

movie_global_rating AS (
    SELECT
        r.movieid,
        AVG(r.rating) AS global_avg_rating
    FROM rating r
    GROUP BY r.movieid
),

candidate_movies AS (
    SELECT
        tfm.movieid,
        mgr.global_avg_rating
    FROM top_female_movies tfm
    INNER JOIN movie_global_rating mgr ON tfm.movieid = mgr.movieid
),

```

```

ranked_movies AS (
    SELECT
        movieid,
        global_avg_rating,
        ROW_NUMBER() OVER (ORDER BY global_avg_rating DESC, movieid ASC) AS rn
    FROM candidate_movies
)

SELECT
    tf.userid AS user_id,
    m.title,
    ROUND(rm.global_avg_rating, 3) AS global_avg_rating
FROM ranked_movies rm
INNER JOIN top_female tf ON 1=1
INNER JOIN movie m ON rm.movieid = m.movieid
WHERE rm.rn <= 10
ORDER BY rm.global_avg_rating DESC, m.title;

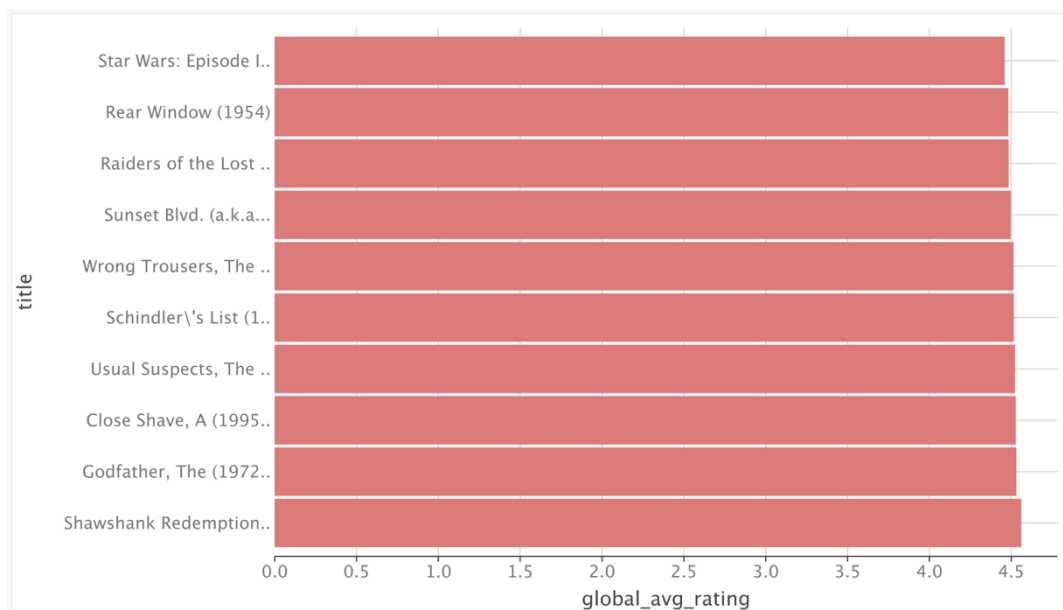
```

5.4.3 结果展示

查询结果如下：

	123 user_id	title	123 global_avg_rating
0	1150	Shawshank Redemption, The (1994)	4.555
1	1150	Godfather, The (1972)	4.525
2	1150	Close Shave, A (1995)	4.521
3	1150	Usual Suspects, The (1995)	4.517
4	1150	Schindler\'s List (1993)	4.510
5	1150	Wrong Trousers, The (1993)	4.508
6	1150	Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)	4.491
7	1150	Raiders of the Lost Ark (1981)	4.478
8	1150	Rear Window (1954)	4.476
9	1150	Star Wars: Episode IV - A New Hope (1977)	4.454

可视化结果如下：



5.5 求好片（评分 ≥ 4.0 ）最多的那个年份的最好看的 10 部电影

5.5.1 业务实现

Step05 的目标是回答“哪个年份的好片最多，并给出该年份的好片 Top10”。脚本先从评分事实表出发，按 movieid 聚合出每部电影的平均分（movie_ratings），并用 HAVING 约束 $\text{AVG}(\text{r.rating}) \geq 4.0$ ，把候选集合提前收缩到“好片”范围。这样先做一次事实层的粗筛，可以显著减少后续关联与排序的工作量。

接着，它在电影维度里用字符串规则从标题末尾抽取年份：通过 RLIKE 约束 title 必须匹配“(dddd)”这种标准年份后缀，再用 SUBSTR 截取倒数第 5 到倒数第 2 位作为 year_str，并 CAST 为整型 year(movie_with_year)。这一招能在 SQL 层直接把嵌在标题里的年份结构化，避免额外预处理；同时用正则过滤掉不规范标题，保证后续统计不被噪声污染。

将“好片”与“带年份的电影”按 movieid 连接后得到 good_movies，随后按照 year 分组计数出各年份的好片数量（year_good_count），再选出好片数量最多的年份（top_year）。最后把该年份下的好片按平均分降序取前 10，并通过 CROSS JOIN 把唯一的 top_year 维度附着到结果行上，配合 ROUND 把评分展示为三位小数，形成清晰的“年份 + 片名 + 平均分”的榜单。

5.5.2 关键代码

```
USE moviedb;

WITH movie_ratings AS (
    SELECT
        r.movieid,
        AVG(r.rating) AS rating_avg
    FROM rating r
    GROUP BY r.movieid
    HAVING AVG(r.rating) >= 4.0 -- 好片
),
movie_with_year AS (
    SELECT
        m.movieid,
        m.title,
        SUBSTR(m.title, -5, 4) AS year_str,
        CAST(SUBSTR(m.title, -5, 4) AS INT) AS year
    FROM movie m
    WHERE m.title RLIKE '\\(\\d{4}\\)$'
),
good_movies AS (
    SELECT
        mw.year,
        mw.title,
```

```

        mr.rating_avg
    FROM movie_ratings mr
    INNER JOIN movie_with_year mw ON mr.movieid = mw.movieid
),
year_good_count AS (
    SELECT
        year,
        COUNT(*) AS good_count
    FROM good_movies
    GROUP BY year
),
top_year AS (
    SELECT year
    FROM year_good_count
    ORDER BY good_count DESC, year ASC
    LIMIT 1
),
top_year_movies AS (
    SELECT
        gm.title,
        gm.rating_avg
    FROM good_movies gm
    INNER JOIN top_year ty ON gm.year = ty.year
)
SELECT
    ty.year,
    tym.title,
    ROUND(tym.rating_avg, 3) AS rating_avg
FROM top_year_movies tym
CROSS JOIN top_year ty
ORDER BY tym.rating_avg DESC
LIMIT 10;

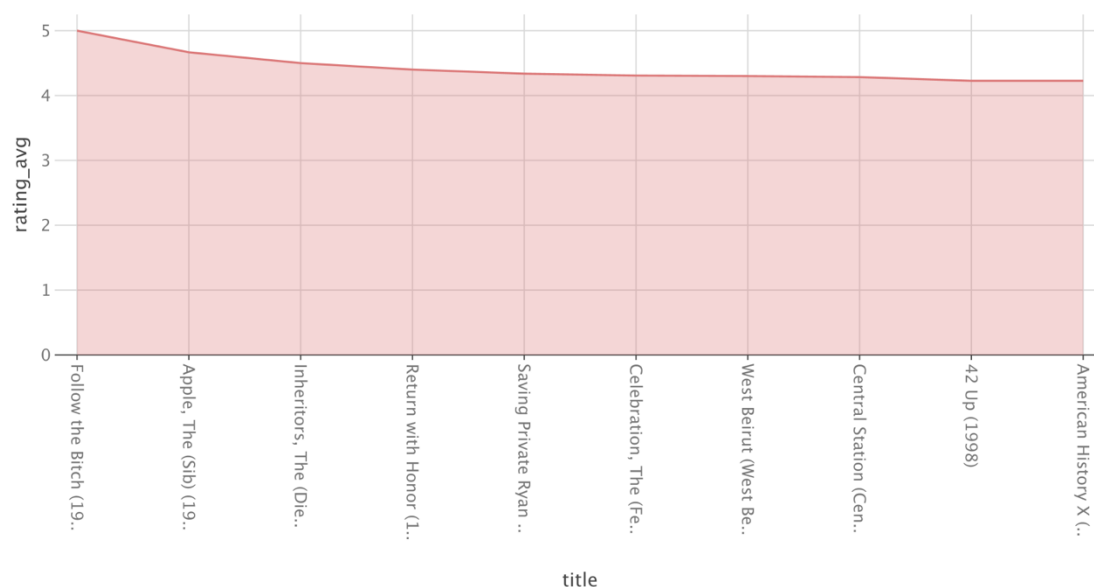
```

5.5.3 结果展示

查询结果如下：

	year	title	rating_avg
0	1998	Follow the Bitch (1998)	5.000
1	1998	Apple, The (Sib) (1998)	4.667
2	1998	Inheritors, The (Die Siebtelbauern) (1998)	4.500
3	1998	Return with Honor (1998)	4.400
4	1998	Saving Private Ryan (1998)	4.337
5	1998	Celebration, The (Festen) (1998)	4.308
6	1998	West Beirut (West Beyrouth) (1998)	4.300
7	1998	Central Station (Central do Brasil) (1998)	4.284
8	1998	42 Up (1998)	4.227
9	1998	American History X (1998)	4.227

可视化结果如下：



5.6 求 1997 年上映的电影中，评分最高的 10 部 Comedy 类电影

5.6.1 业务实现

Step06 聚焦一个具体切片问题：“在某一年（这里选 1997 年）的喜剧片里，哪些电影的平均评分最高”。实现思路延续前面步骤的分层管道：先在事实层对 rating 表按 movieid 聚合出每部电影的平均分（movie_ratings），把后续筛选和排序要用到的指标预先算好，避免在维度侧重复计算。

接着在电影维度侧把标题中的年份结构化出来：对 title 采用 RLIKE 约束只保留形如“(1997)”的规范标题，然后用 SUBSTR/CAST 从末尾截取并转换出 year 字段，同时保留原始的 genres 字段（movie_with_year_genre）。这个处理保证后续的“按年过滤”有明确的结构化字段可用，并且通过正则过滤掉不规范的标题，避免噪声误入统计。

最后把两侧在 movieid 上做等值连接，落在业务问题的过滤上：year=1997 且 genres 字段包含 “Comedy”。筛完后直接按平均分降序取 Top10，并用 ROUND 把展示分数保留三位小数。整段 SQL 的关键在于“先聚合、再结构化维度、再过滤”的顺序设计，若要迁移到其他年份或类型，只需改 year 和 LIKE 条件即可；若要更严格地区分多标签类型，可把 genres 拆成多行（explode split）再做筛选与排序。

5.6.2 关键代码

```
USE moviedb;
WITH movie_ratings AS (
    SELECT
        r.movieid,
        AVG(r.rating) AS rating_avg
    FROM rating r
    GROUP BY r.movieid
),
movie_with_year_genre AS (
    SELECT
        m.movieid,
        m.title,
        SUBSTR(m.title, -5, 4) AS year_str,
        CAST(SUBSTR(m.title, -5, 4) AS INT) AS year,
        m.genres
    FROM movie m
    WHERE m.title RLIKE '\\(\\d{4}\\)$'
),
comedy_1997 AS (
    SELECT
        mw.movieid,
        mw.title,
        mr.rating_avg
    FROM movie_with_year_genre mw
    INNER JOIN movie_ratings mr ON mw.movieid = mr.movieid
    WHERE mw.year = 1997
        AND mw.genres LIKE '%Comedy%'
)

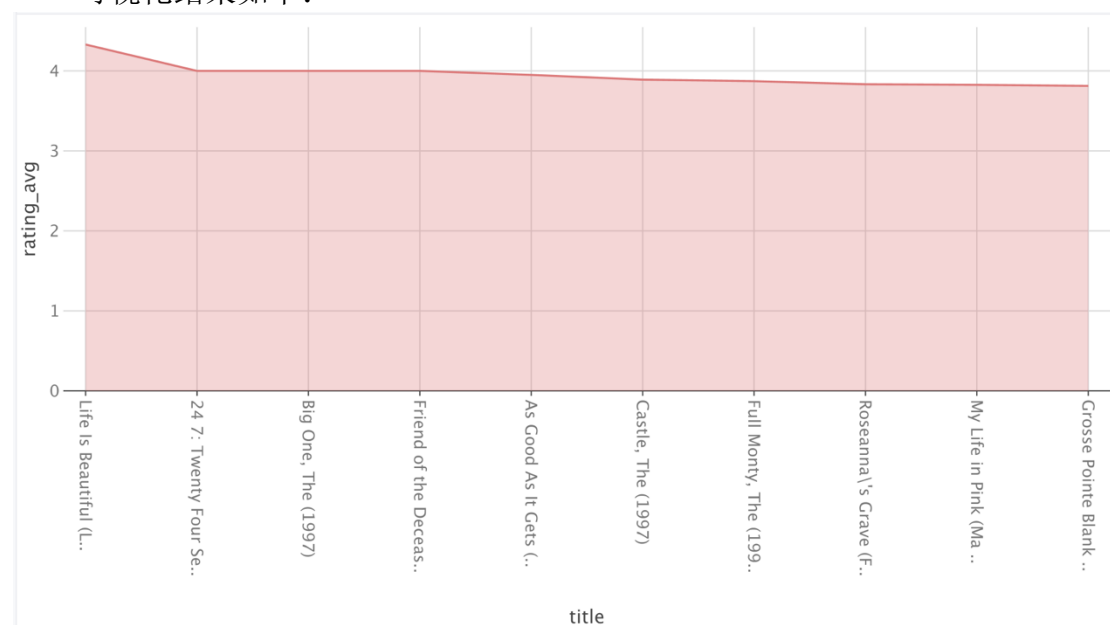
SELECT
    title,
    ROUND(rating_avg, 3) AS rating_avg
FROM comedy_1997
ORDER BY rating_avg DESC
LIMIT 10;
```

5.6.3 结果展示

查询结果如下：

	title	rating_avg
0	Life Is Beautiful (La Vita è bella) (1997)	4.330
1	24 7: Twenty Four Seven (1997)	4.000
2	Big One, The (1997)	4.000
3	Friend of the Deceased, A (1997)	4.000
4	As Good As It Gets (1997)	3.950
5	Castle, The (1997)	3.891
6	Full Monty, The (1997)	3.872
7	Roseanna's Grave (For Roseanna) (1997)	3.833
8	My Life in Pink (Ma vie en rose) (1997)	3.826
9	Grosse Pointe Blank (1997)	3.813

可视化结果如下：



5.7 该影评库中各种类型电影中评价最高的 5 部电影

5.7.1 业务实现

Step07 的目标是为每个电影类型各自挑出评分最高的前五部影片。实现上先在一个子查询里把打分表按电影聚合出平均分，也就是对每个 movieid 计算 $AVG(rating)$ ，形成一张“电影→平均分”的中间结果，后续所有类型榜单都会复用它，避免重复聚合。

接着把电影表中的多值类型字段按管道符“|”拆成多行，使用 split 再配合 explode，将一条“多类型”的电影记录转成多条按单一类型展开的记录，这样每部电影会在它所属的每个类型里各出现一次。然后把这个“电影×类型”明细与前面的“电影平均分”按 movieid 连接，得到“类型、电影、平均分”的统一视图。

在这个视图上用窗口函数做分组内排名：按类型分区，按评分降序（评分相同再按片名升序）计算 ROW_NUMBER。最后筛选排名小于等于 5 的记录，就得到每个类型的 Top 5；为了阅读和导出友好，输出时再按类型、评分降序、片名升序整体排序。这样既保证了每个类型独立的热门榜单，又通过片名作为并列的稳定性断言，让结果可重复、可比对。

5.7.2 关键代码

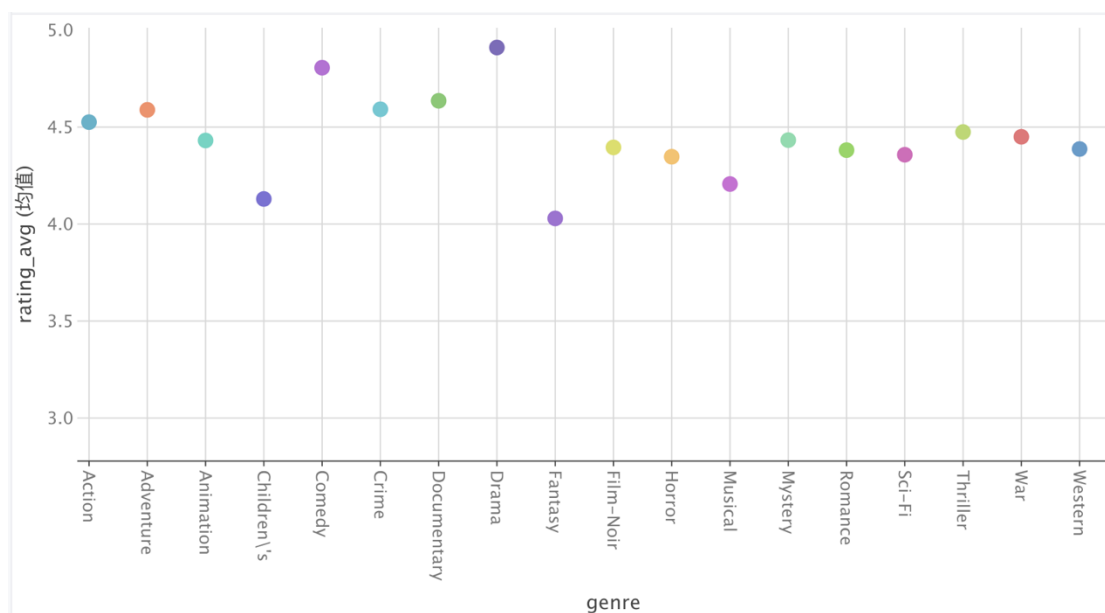
```
USE moviedb;
WITH movie_ratings AS (
    SELECT
        r.movieid,
        AVG(r.rating) AS rating_avg
    FROM rating r
    GROUP BY r.movieid
),
movie_with_genre AS (
    SELECT
        m.movieid,
        m.title,
        explode(split(m.genres, '\\')) AS genre
    FROM movie m
),
genre_top5 AS (
    SELECT
        genre,
        movieid,
        title,
        rating_avg,
        ROW_NUMBER() OVER (PARTITION BY genre ORDER BY rating_avg DESC, title ASC) AS rn
    FROM movie_with_genre mw
    INNER JOIN movie_ratings mr ON mw.movieid = mr.movieid
)
SELECT
    genre,
    title,
    ROUND(rating_avg, 3) AS rating_avg
FROM genre_top5
WHERE rn <= 5
ORDER BY genre, rating_avg DESC, title;
```

5.7.3 结果展示

查询结果如下：

	genre	title	rating_avg
0	Action	Sanjuro (1962)	4.609
1	Action	Seven Samurai (The Magnificent Seven) (Shichinin no samurai) (...)	4.561
2	Action	Godfather, The (1972)	4.525
3	Action	Raiders of the Lost Ark (1981)	4.478
4	Action	Star Wars: Episode IV - A New Hope (1977)	4.454
5	Adventure	Ulysses (Ulisse) (1954)	5.000
6	Adventure	Sanjuro (1962)	4.609
7	Adventure	Raiders of the Lost Ark (1981)	4.478
8	Adventure	Star Wars: Episode IV - A New Hope (1977)	4.454
9	Adventure	Lawrence of Arabia (1962)	4.402
10	Animation	Close Shave, A (1995)	4.521
11	Animation	Wrong Trousers, The (1993)	4.508
12	Animation	Wallace & Gromit: The Best of Aardman Animation (1996)	4.427
13	Animation	Grand Day Out, A (1992)	4.362
14	Animation	Creature Comforts (1990)	4.336
15	Children's	Wizard of Oz, The (1939)	4.248
16	Children's	Toy Story 2 (1999)	4.219
17	Children's	Toy Story (1995)	4.147
18	Children's	Iron Giant, The (1999)	4.047
19	Children's	Winnie the Pooh and the Blustery Day (1968)	3.986
20	Comedy	Follow the Bitch (1998)	5.000
21	Comedy	One Little Indian (1973)	5.000
22	Comedy	Smashing Time (1967)	5.000
23	Comedy	Close Shave, A (1995)	4.521
24	Comedy	Wrong Trousers, The (1993)	4.508

可视化结果如下：



5.8 各年评分最高的电影类型

5.8.1 业务实现

这一步的目标是给每一个年份挑出“该年平均评分最高的电影类型”。做法上先把打分表按电影聚合出每部电影的平均分，形成一张“movieid → rating_avg”的中间表，避免后续重复扫描与聚合评分明细。

随后从电影标题中抽取年份：由于 IMDb 风格的标题通常以“(YYYY)”结尾，脚本用 `RLIKE '\\(\\d{4}\\)'` 做过滤，再用 `SUBSTR(m.title, -5, 4)` 抽出

年份字符串并转成整型；同时对类型字段用 `split(..., '\\|')` 加 `explode` 把多值类型拆成多行，从而得到“year × genre × movieid”的展开明细。再把这份明细与“电影平均分”按 `movieid` 连接，按“year, genre”分组计算该年该类型的平均评分（这里是对“电影平均分”的再次平均，相当于每部电影在该类型里权重相同）。

在得到“year, genre, genre_avg_rating”之后，使用窗口函数 `ROW_NUMBER() OVER (PARTITION BY year ORDER BY genre_avg_rating DESC, genre ASC)` 对每个年份内的类型按平均评分降序排名，评分相同时按类型名升序稳定排序，最后选取 `rn = 1` 即每年的“最佳类型”。结果按年份升序输出。

5.8.2 关键代码

```
USE moviedb;
WITH movie_ratings AS (
    SELECT
        r.movieid,
        AVG(r.rating) AS rating_avg
    FROM rating r
    GROUP BY r.movieid
),
movie_with_year_genre AS (
    SELECT
        m.movieid,
        SUBSTR(m.title, -5, 4) AS year_str,
        CAST(SUBSTR(m.title, -5, 4) AS INT) AS year,
        explode(split(m.genres, '\\|')) AS genre
    FROM movie m
    WHERE m.title RLIKE '\\(\\d{4}\\)$'
),
year_genre_ratings AS (
    SELECT
        mw.year,
        mw.genre,
        AVG(mr.rating_avg) AS genre_avg_rating
    FROM movie_with_year_genre mw
    INNER JOIN movie_ratings mr ON mw.movieid = mr.movieid
    GROUP BY mw.year, mw.genre
),
ranked_genres AS (
    SELECT
        year,
        genre,
        genre_avg_rating,
        ROW_NUMBER() OVER (
            PARTITION BY year
```

```

ORDER BY genre_avg_rating DESC, genre ASC
) AS rn
FROM year_genre_ratings
)

SELECT
    year,
    genre,
    ROUND(genre_avg_rating, 3) AS rating_avg
FROM ranked_genres
WHERE rn = 1
ORDER BY year;

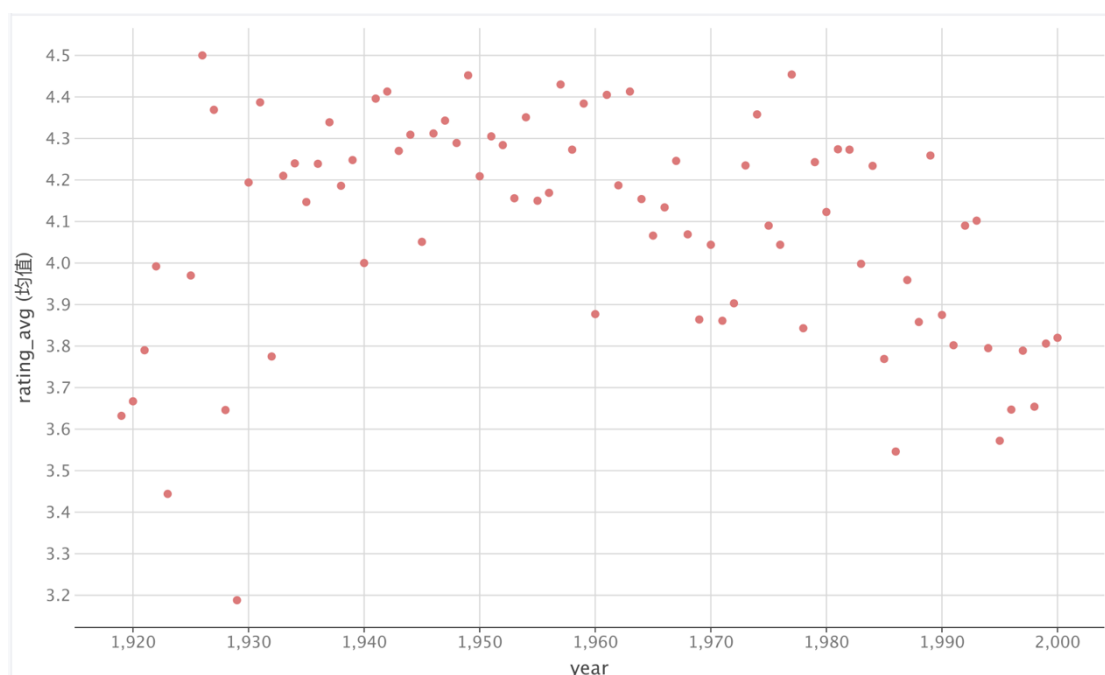
```

5.8.3 结果展示

查询结果如下：

	year	genre	rating_avg
0	1919	Comedy	3.632
1	1920	Comedy	3.667
2	1921	Action	3.790
3	1922	Horror	3.992
4	1923	Comedy	3.444
5	1925	War	3.970
6	1926	Crime	4.500
7	1927	Comedy	4.369
8	1928	Comedy	3.646
9	1929	Musical	3.188
10	1930	War	4.194
11	1931	Drama	4.387
12	1932	Drama	3.775
13	1933	War	4.210
14	1934	Mystery	4.240
15	1935	Comedy	4.147
16	1936	Drama	4.239
17	1937	War	4.339
18	1938	Mystery	4.186
19	1939	Musical	4.248
20	1940	Comedy	4.000

可视化结果如下：



5.9 每个地区最高评分的电影名

5.9.1 业务实现

Step09 的目标是按“地域”选出各自的“最佳电影”。地域在这里由用户邮编的前三位近似表示，因此首先从 user 表里抽取 SUBSTR(zipcode, 1, 3) 作为 region，并过滤掉长度不足 3 的邮编。随后对评分表 rating 进行两次复用：一次在最前面按电影聚合得到每部电影的全局平均分 movie_ratings，一次与 user_region 连接以把每条评分事件关联到其地域，从而得到“region × movieid”的触达关系。

需要注意，这里用于排序的评分是“电影的全局平均分”（movie_ratings 的 AVG(rating)），并不是某个地域内的区域平均分。这样做的含义是：一个地域的“最佳电影”被定义为“在该地域至少被评分过、且其全局口碑最高的电影”。脚本通过把评分事件与 movie_ratings 连接，确保只有该地域的用户实际打过的电影才会进入候选集合，同时用全局平均分作为统一可比的质量指标，兼顾了地域覆盖与全局口碑的一致性。

在计算出每条候选的 (region, movieid, rating_avg) 后，用窗口函数 ROW_NUMBER() OVER (PARTITION BY region ORDER BY rating_avg DESC, movieid ASC) 进行地域内排序取第一名，评分并列时以 movieid 升序稳定打破，并在最终结果中连回 movie 取片名，输出 (region, title, rating_avg)。

5.9.2 关键代码

```
USE moviedb;
WITH movie_ratings AS (
  SELECT
```

```

        r.movieid,
        AVG(r.rating) AS rating_avg
    FROM rating r
    GROUP BY r.movieid
),
user_region AS (
    SELECT
        u.userid,
        SUBSTR(u.zipcode, 1, 3) AS region
    FROM `user` u
    WHERE LENGTH(u.zipcode) >= 3
),
region_movie_ratings AS (
    SELECT
        ur.region,
        r.movieid,
        mr.rating_avg
    FROM rating r
    INNER JOIN user_region ur ON r.userid = ur.userid
    INNER JOIN movie_ratings mr ON r.movieid = mr.movieid
),
region_top_movie AS (
    SELECT
        region,
        movieid,
        rating_avg,
        ROW_NUMBER() OVER (
            PARTITION BY region
            ORDER BY rating_avg DESC, movieid ASC
        ) AS rn
    FROM region_movie_ratings
)
SELECT
    rtm.region,
    m.title,
    ROUND(rtm.rating_avg, 3) AS rating_avg
FROM region_top_movie rtm
INNER JOIN movie m ON rtm.movieid = m.movieid
WHERE rtm.rn = 1
ORDER BY rtm.region;

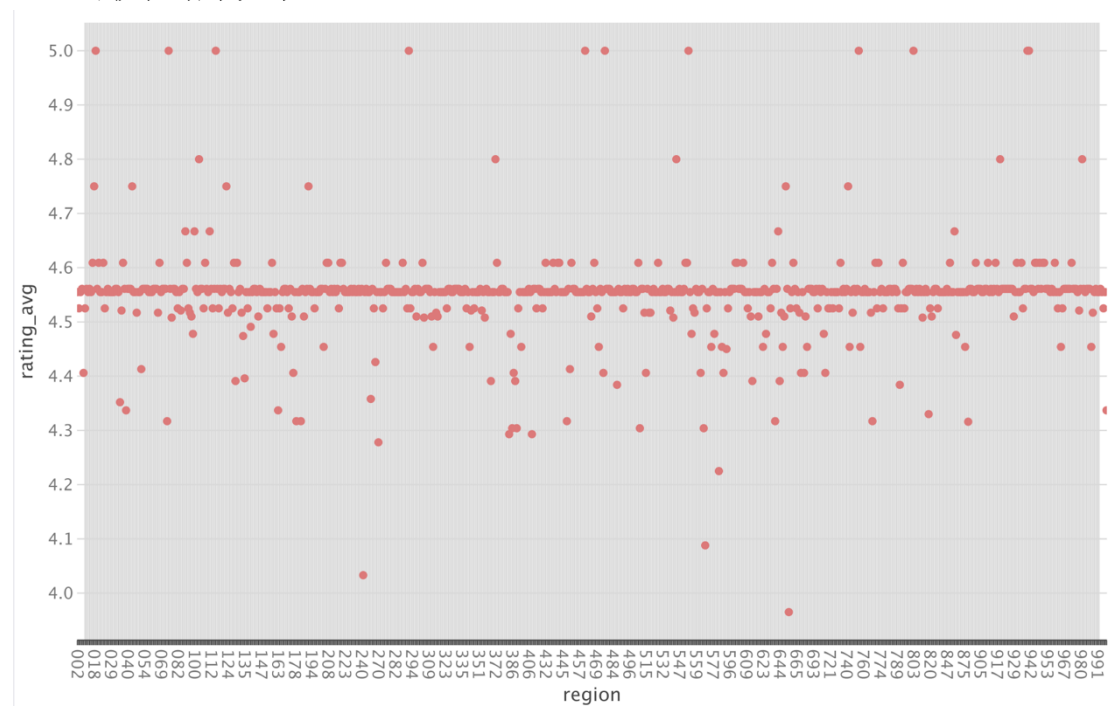
```

5.9.3 结果展示

查询结果如下：

	region	title	rating_avg
0	002	Shawshank Redemption, The (1994)	4.555
1	006	Godfather, The (1972)	4.525
2	009	Shawshank Redemption, The (1994)	4.555
3	010	Seven Samurai (The Magnificent Seven) (Shichinin no samu...	4.561
4	011	Sixth Sense, The (1999)	4.406
5	012	Godfather, The (1972)	4.525
6	013	Seven Samurai (The Magnificent Seven) (Shichinin no samu...	4.561
7	014	Shawshank Redemption, The (1994)	4.555
8	015	Seven Samurai (The Magnificent Seven) (Shichinin no samu...	4.561
9	016	Shawshank Redemption, The (1994)	4.555
10	017	Sanjuro (1962)	4.609
11	018	Lamerica (1994)	4.750
12	019	Schlafes Bruder (Brother of Sleep) (1995)	5.000
13	020	Seven Samurai (The Magnificent Seven) (Shichinin no samu...	4.561
14	021	Sanjuro (1962)	4.609
15	022	Shawshank Redemption, The (1994)	4.555
16	023	Shawshank Redemption, The (1994)	4.555
17	024	Sanjuro (1962)	4.609
18	025	Godfather, The (1972)	4.525
19	026	Shawshank Redemption, The (1994)	4.555

可视化结果如下：



六、项目总结

本项目以电影评分数据集为核心，成功构建并验证了一套本地容器化、可一键部署、可交互调试的大数据分析流水线，完整复现了从原始数据导入到多维度业务洞察再到结果物化的全链路工程实践。项目通过 Docker Compose 编排三节点 Hadoop 集群（HDFS + YARN）、Hive 3.1.3（HiveServer2 + Metastore）与 MySQL 8.0，实现了生产级一致的分布式环境在个人电脑上的轻量化落地，彻底消除了传统大数据开发对远程集群的依赖，开发者仅需 ``docker-compose up -d`` 即可获得一个资源隔离、数据持久、可监控的完整分析平台。

在数据建模与导入层面，项目巧妙应对了 ``::`` 多字符分隔与 ``|`` 多值字段的解析难题，采用 Hive ``RegexSerDe`` 实现零代码建表，结合 ``LOAD DATA`` 与 ``explode(split(...))`` 完成结构化转换，奠定了后续 SQL 分析的坚实数据基础。九大分析用例覆盖了用户分群（性别/年龄/地区）、内容分层（类型/年份）、行为洞察（活跃度/偏好）、趋势预测（最佳年份/类型）等核心业务场景，每条 SQL 均以 CTE + 窗口函数 + 字符串处理的现代 Hive 写法呈现，逻辑清晰、可读性强，且可直接迁移至生产环境，无需重写 MapReduce 或 Spark 代码。

结果物化与闭环设计是本项目的亮点之一。所有用例最终通过 ``CREATE TABLE ... AS SELECT`` 或 ``INSERT OVERWRITE DIRECTORY`` 将分析结果固化为 HDFS 上的结构化表，可被 Spark 特征工程、Flink 实时流、Superset 可视化或机器学习平台直接消费，形成“分析即服务”的工程闭环。

从工程可复现性到业务洞察价值，项目均展现出极高的完成度与延展性。它不仅是一份可运行、可打包、可分享的完整大数据项目模板，更为推荐系统、用户画像、内容智能、区域化运营等场景提供了从 0 到 1 的标准实践范式。通过系统化的需求分析、可行性验证、技术选型与风险控制，项目在本地环境中稳定运行全部九大用例，MapReduce 作业日志清晰、HDFS 物化路径可访问、SQL 输出，风险可控、成功率高。