三种数据库的应用场景

MySQL（关系型数据库）

对于需要较高并发量、数据准确度要求高，以及经常需要进行新增、修改、删除的操作场景，MySQL 非常适合。MySQL也适合小规模或中等规模数据的聚合查询，就本次作业来说，MySQL 对于数据量在 MB 级或低 GB 级（取决于硬件配置）时，也能进行常规的数据分析查询，如简单的 GROUP BY、JOIN 操作等。

总结：MySQL 主要用于结构化、实时写入与更新量大，并且对查询的实时响应速度有一定要求的场景。在超大规模的数据分析（特别是大体量批量处理）方面很慢；在关系深度非常复杂的场景中，多表关联可能带来性能瓶颈。

HIve（分布式数据库）

适合存储与处理超大规模（TB、PB 级）的数据，默认采用文件存储（如 ORC等列式存储格式）。当业务需要处理 TB 乃至 PB 级海量数据），Hive 依靠分布式计算框架（MapReduce、Tez、Spark）进行批处理，实现对海量数据的聚合、统计、ETL等操作。在本次作业中，完成同样的大数据量的多表join操作，hive最快。

总结：Hive 的优势在于可扩展性和处理海量数据的能力，主要用于离线大数据分析场景，查询延迟较高，不适合即时性或高并发的在线查询。

Neo4（图数据库）

如果数据之间存在大量关联关系，Neo4j 能非常高效地执行深度遍历、最短路径或其他图算法（如社区发现、PageRank）等操作。相对于传统关系型数据库，通过表的多次 JOIN 来查找层层嵌套的关联关系往往开销很大，而图数据库的存储结构能使层级关系查询更快，更加直观。

总结：Neo4j 非常适合高度互联的图状数据，在需要频繁进行关系遍历、路径查找、图算法的场景下

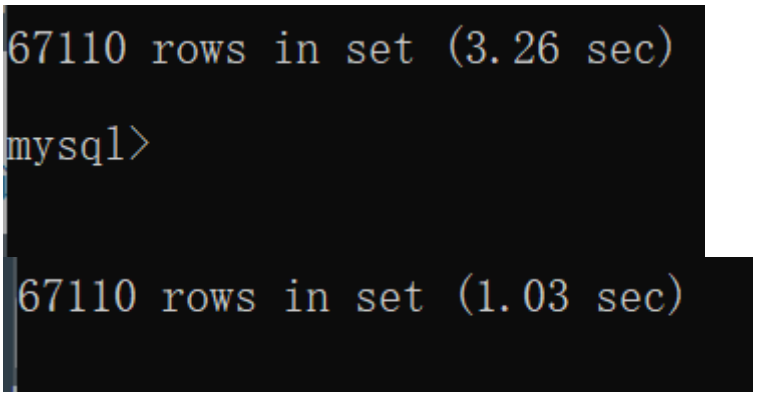
优化

MySQL

一、分区

为了寻找具有正面评价的电影。正常的查询需要遍历评论表，这个表有177万条数据。现在将评论表按评分（Score）分区，查询时只需要遍历5分的分区即可，效率大大提升。

```
CREATE TABLE Review_Partitioned (
    Movie_ID INT,
    Review_ID INT,
    Score INT,
    PRIMARY KEY (Review_ID, Score), -- 包含分区键 Score
    INDEX idx_movie_id (Movie_ID)
) PARTITION BY RANGE (Score) (
    PARTITION p_low VALUES LESS THAN (5),
    PARTITION p_high VALUES LESS THAN (10)
);
```



优化前后比较

二、分表

All_Actor表总共有80万条数据，join操作效率太低。于是将All_Actor进行分表操作。

```
将 All_Actor 分成 4 个子表
CREATE TABLE All_Actor_Movie_1 (
    Actor_ID INT,
    Movie_ID INT,
    PRIMARY KEY (Movie_ID, Actor_ID),
    FOREIGN KEY (Actor_ID) REFERENCES Actor(Actor_ID),
    FOREIGN KEY (Movie_ID) REFERENCES Movie(Movie_ID)
);

CREATE TABLE All_Actor_Movie_2 LIKE All_Actor_Movie_1;
CREATE TABLE All_Actor_Movie_3 LIKE All_Actor_Movie_1;
CREATE TABLE All_Actor_Movie_4 LIKE All_Actor_Movie_1;
```

```
INSERT INTO All_Actor_Movie_1 (Actor_ID, Movie_ID)
SELECT Actor_ID, Movie_ID FROM All_Actor WHERE MOD(Movie_ID, 4) =
1;

INSERT INTO All_Actor_Movie_2 (Actor_ID, Movie_ID)
SELECT Actor_ID, Movie_ID FROM All_Actor WHERE MOD(Movie_ID, 4) =
2;

INSERT INTO All_Actor_Movie_3 (Actor_ID, Movie_ID)
SELECT Actor_ID, Movie_ID FROM All_Actor WHERE MOD(Movie_ID, 4) =
3;

INSERT INTO All_Actor_Movie_4 (Actor_ID, Movie_ID)
SELECT Actor_ID, Movie_ID FROM All_Actor WHERE MOD(Movie_ID, 4) =
0;
```

分别对四个表进行查询，效率也有很大提升。

三、改变主码类型

最初我们选用Amazon的字符串类型的主码，如（`B000007SYV`），其余表也都是用字符串主码。经过我们上网搜集资料和实际操作，发现选用`INT`主键能减少存储和索引空间，并且对于索引 B+ 树结构，整型键值更紧凑，数据库维护时需要更少的分裂和重排，查询和插入性能更好。并且在join时比较与排序速度更快，整数比较和排序在底层硬件层级上更简单、效率更高，而字符串需要逐字节对比。

Hive

一、列式存储

行式存储：所有列的值保存在一起，如需读取其中某几列，也要将整行数据从磁盘加载到内存，再筛选出所需的列。这会导致大量无用数据被读取，浪费 I/O 带宽。

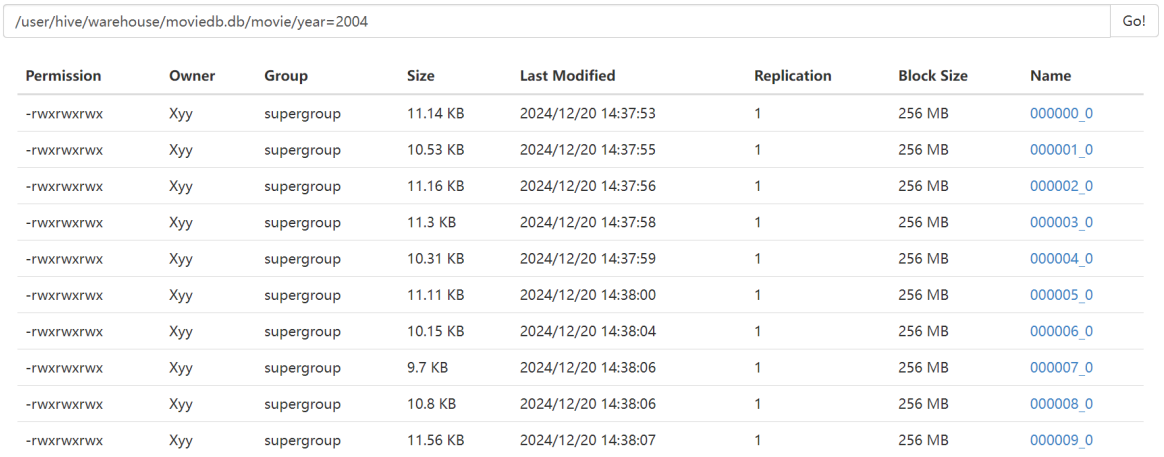
列式存储：每一列的数据都连续保存在一起，执行查询时只需读取所需列的数据块，能够显著降低磁盘读取量。

在Hive中我们所有的表结构都采用了列式存储。

```
CREATE TABLE Genre (
    Genre_ID STRING,
    Genre STRING
)
STORED AS ORC;
```

二、分桶

分桶能优化join性能，避免全局扫描、分发，不必对所有行进行重新分发；仅需读对应桶内的数据。此外，如果聚合列与分桶列相同或在分桶列的基础上进行 Group By，Hive 能根据分桶的哈希分布信息，减少重复计算或数据拉取。



/user/hive/warehouse/moviedb.db/movie/year=2004 Go!

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rwxrwxrwx	Xyy	supergroup	11.14 KB	2024/12/20 14:37:53	1	256 MB	000000_0
-rwxrwxrwx	Xyy	supergroup	10.53 KB	2024/12/20 14:37:55	1	256 MB	000001_0
-rwxrwxrwx	Xyy	supergroup	11.16 KB	2024/12/20 14:37:56	1	256 MB	000002_0
-rwxrwxrwx	Xyy	supergroup	11.3 KB	2024/12/20 14:37:58	1	256 MB	000003_0
-rwxrwxrwx	Xyy	supergroup	10.31 KB	2024/12/20 14:37:59	1	256 MB	000004_0
-rwxrwxrwx	Xyy	supergroup	11.11 KB	2024/12/20 14:38:00	1	256 MB	000005_0
-rwxrwxrwx	Xyy	supergroup	10.15 KB	2024/12/20 14:38:04	1	256 MB	000006_0
-rwxrwxrwx	Xyy	supergroup	9.7 KB	2024/12/20 14:38:06	1	256 MB	000007_0
-rwxrwxrwx	Xyy	supergroup	10.8 KB	2024/12/20 14:38:06	1	256 MB	000008_0
-rwxrwxrwx	Xyy	supergroup	11.56 KB	2024/12/20 14:38:07	1	256 MB	000009_0

如图是我们hive数据库的一个分桶信息

三、分区

我们的movie表根据year进行分区，当查询语句中带有分区字段的过滤条件（如 `WHERE year = 2004`），Hive 只会读取与该分区相关的目录下的数据文件，而不用扫描整个表的所有数据。

drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:08	0	0 B	year=1999
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:08	0	0 B	year=2000
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2001
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2002
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2003
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2004
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2005
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2006
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2007
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2008
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2009
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2010
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2011
drwxrwxrwx	Xyy	supergroup	0 B	2024/12/20 14:38:07	0	0 B	year=2012

如图是movie表的部分分区信息。

Neo4j（图数据库）

一、为节点的ID创建索引

Movie_ID、Movie_Title、Average_Score

```
CREATE INDEX FOR (m:Movies) ON (m.Movie_ID);
CREATE TEXT INDEX FOR (m:Movies) ON (m.Movie_Title);
CREATE INDEX FOR (m:Movies) ON (m.Average_Score);
```

Format_ID

```
CREATE INDEX FOR (f:Formats) ON (f.Format_ID);
```

Genre_ID、Genre

```
CREATE INDEX FOR (g:Genres) ON (g.Genre_ID);
CREATE TEXT INDEX FOR (g:Genres) ON (g.Genre);
```

Actor_ID、Actor_Name

```
CREATE INDEX FOR (a:Actors) ON (a.Actor_ID);
CREATE TEXT INDEX FOR (a:Actors) ON (a.Actor_Name);
```

Review_ID、Score

```
CREATE INDEX FOR (r:Reviews) ON (r.Review_ID);
CREATE INDEX FOR (r:Reviews) ON (r.Score);
```

Director_ID、Director_Name

```
CREATE INDEX FOR (r:Directors) ON (r.Director_ID);
CREATE TEXT INDEX FOR (d:Directors) ON (d.Director_Name);
```

1. 修改图数据库配置
    - 指定 Neo4j 的 Java 虚拟机（JVM）堆内存的初始大小，设置为 **4096MB（4GB）**。

      ```
      server.memory.heap.initial_size=4096m
      ```

      > **作用**：初始堆内存大小影响 JVM 启动时分配的内存空间，设置一个较大的初始值可以减少动态内存调整的开销。

    - 指定 Neo4j 的 JVM 堆内存的最大大小，设置为 **8192MB（8GB）**。

      ```
      server.memory.heap.max_size=8192m
      ```

      > **作用**：限制堆内存的最大使用量，防止 JVM 使用过多内存导致系统资源耗尽。

    - 指定 Neo4j 允许使用的最大内存量，设置为 **8192MB（8GB）**。

      ```
      dbms.memory.transaction.total.max=8192m
      ```

      > **作用**：控制事务处理所能使用的最大内存量，用于管理正在处理的事务数据。如果超出这个限制，Neo4j 会抛出内存不足的错误。

    - 指定 Neo4j 的 **页面缓存（Page Cache）** 的大小，设置为 **6GB**。

      ```
      server.memory.pagecache.size=6g
      ```

> **作用**：页面缓存是用来存储磁盘上节点和关系数据的缓冲区。较大的页面缓存有助于减少磁盘 I/O，从而提高数据库查询的性能。

2. 在合作查询中，使用**APOC**创建合作关系

   - 演员与演员的合作，通过设置批次和重试次数提升关联关系创建的速率，最终失败的批次数为0。

```
CALL apoc.periodic.iterate(
"MATCH (a1:Actors)-[:ACT]->(m:Movies)<-[:ACT]-(a2:Actors)
RETURN a1, a2, COUNT(m) AS collaborations",
"MERGE (a1)-[r:COOPERATED]->(a2)
ON CREATE SET r.count = collaborations
ON MATCH SET r.count = r.count + collaborations",
{batchSize: 50, retries: 20, parallel: true}
);
```

   - 演员与导演的合作，通过设置批次和重试次数提升关联关系创建的速率，最终失败的批次数为0。

```
CALL apoc.periodic.iterate(
"MATCH (a:Actors)-[:ACT]->(m:Movies)<-[:DIRECT]-
(d:Directors)
RETURN a, d, COUNT(m) AS collaborations",
"MERGE (a)-[r:COOPERATED_WITH_DIRECTOR]->(d)
ON CREATE SET r.count = collaborations
ON MATCH SET r.count = r.count + collaborations",
{batchSize: 50, retries: 20, parallel: true}
);
```

3. 批量筛选电影的预处理

   - 通过标签临时标记符合条件的电影

```
MATCH (m:Movies)-[:HAS_GENRE]->(g:Genres {Genre: 'XXX'})
WITH m
MATCH (m)<-[:ACT]-(a:Actors)
WITH m, COUNT(a) AS actor_count
WHERE actor_count >= 2
SET m:FilteredMovie;
```

```
MATCH (m:Movies)-[:HAS_GENRE]->(g:Genres {Genre: 'Action'})
WITH m
MATCH (m)<-[:ACT]-(a:Actors)
WITH m, COUNT(a) AS actor_count
WHERE actor_count >= 3
SET m:FilteredMovie;
```

如何保证数据质量

建立明确的数据质量指标与要求

1. 检查数据是否缺失（如空值或空白字段）。
2. 数据的数值或文本是否符合真实含义，如电影名称是否合理，人的姓名是否完整。
3. 数据的格式是否一致，如演员的姓名统一以逗号间隔，人名必须包含名和姓，日期统一是`DD/MM/YY`的格式

进行数据清洗与校正

将源数据转为目标系统所需的数据类型，并检查转换是否成功、是否出现截断、溢出或丢失等情况。不符合规则的记录可以进行 “脏数据” 标注或单独存储，后续再做人工排查或自动修正。例如源数据有大量HTML编码，需要将它们转成正常的符号。

此外，对于空字段，我们从其他数据源获取。如空评论从评论表里找到最找的评论时间，其余空字段如演员导演，我们额外从IMDB网站爬取相关数据作为补充。

建立元数据与数据血缘记录

记录数据从源头到目标系统的处理节点、处理规则、时间等信息（数据血缘），一旦出现问题可追溯源头和处理历史。便于审计以及问题快速定位与修复。

哪些情况会影响数据质量

源数据质量本身不佳

源头系统可能包含大量脏数据、不准确的数据或重复数据。爬取数据的过程中可能网络不佳，造成数据缺失。网站数据质量参差不齐，爬取过程附加了大量HTML编码。

源数据质量参差不齐，格式不一

源数据有不同的错误格式，无法用ETL一次性全部检测到。例如（演员的姓名可能通过逗号，and，&甚至没有连接）需要进行多次检测，甚至人工检查。几乎每次筛选后都会有遗漏的错误数据。

ETL脚本设计欠缺

ETL 脚本自身的设计与实现就可能导致数据质量问题。即使数据源和目标系统都没有明显缺陷，脚本逻辑或编写方式也会成为“脏数据”出现的根源。例如无法正确分割演员信息。

数据血缘的应用场景

1. **发现数据冗余与可复用资源**：借助血缘，可以看到某些数据是否在不同系统被重复加工，或者某些分析逻辑是否可在多场景重用。通过去重和整合，节省存储成本和开发成本，提升数据使用效率。
2. **数据可溯源**：在后续查询出现偏差或异常时，也能回溯数据源自哪里、在哪一步可能被错误处理。
3. **数据管理与质量监控**：数据血缘能协助数据治理者了解数据如何流动、转换、聚合，在哪里容易产生数据质量问题，从而制定更有效的监控和管理措施。