The MySQL Workbook

Michael Alexander Kaufmann

Manuscript DRAFT, 23.8.2023

6 Chapter 6: Optimizing queries

6.1 Learning outcomes

In this chapter, you'll continue to develop your own movie recommendations database application.

In Chapter 1, we outlined a plan for our database system, and in Chapter 2 you installed the necessary software components. In Chapter 3, you downloaded the source files, analyzed their structure, derived corresponding table structures, and created those tables in the database server. Then, in Chapter 4, you loaded the source files into the tables and transformed them into the desired format. In Chapter 5, you analyzed the data to derive first movie recommendations. However, some of these queries took quite some time to complete!

In this chapter, we will speed up the queries with a few tricks. We will also try to improve the quality of the results so that they're more attuned to the movies for which the users are requesting recommendations. As usual, please complete all exercises on your own to achieve the optimal learning effect.

In this chapter you will learn to speed up database queries by:

- understanding the query optimizer,
- analyzing execution plans with `EXPLAIN`,

- saving intermediate results as materialized views with `CREATE TABLE AS SELECT …`,
- and by creating indexes on search columns with `CREATE INDEX`.

Before we manually optimize queries, let's look at how a database server automatically optimizes query execution. We'll use that information to implement our own optimizations.

6.2 The Query Optimizer

Within our MySQL database server, the *query optimizer* [19], [20] is a piece of software that plans the steps of the execution of a database query for optimal performance. In fact, most SQL-based database servers do automatic query optimization. When scheduling queries, the optimizer tries to plan the most efficient execution possible. As shown in Figure 6.1it draws on three pieces of information to do this:

- Statistics about the number of rows of tables and index entries involved in the query.
- Keys or indexes that can be used as efficient access paths during data filtering.
- A cost estimate for various execution variants based on the above two points.

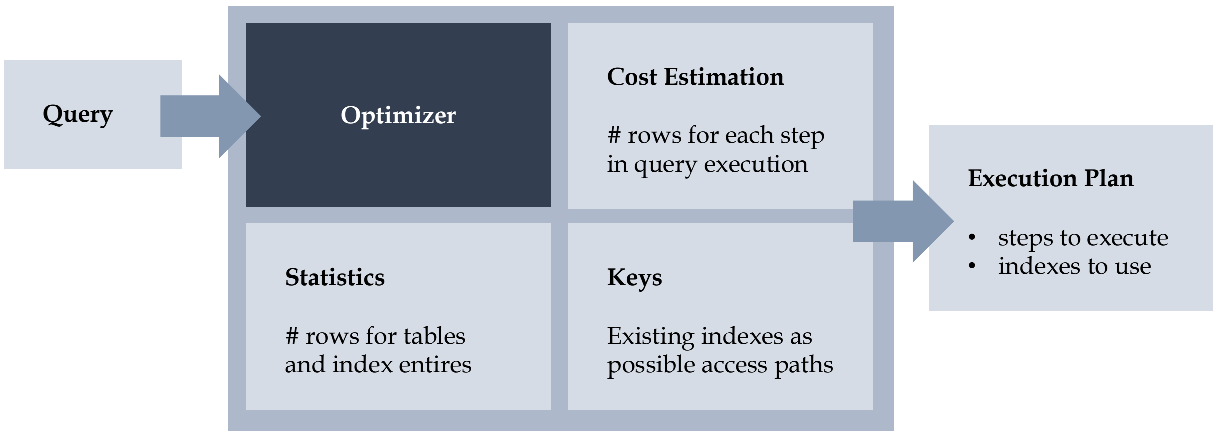

Query
Optimizer
Cost Estimation
# rows for each step in query execution
Statistics
# rows for tables and index entires
Keys
Existing indexes as possible access paths
Execution Plan
• steps to execute
• indexes to use

Figure 6.1: The query optimizer of the database server

Based on the tables used and the existing index structures, the optimizer can use the information from the table statistics to create a cost estimate. This calculates approximately how many rows of data need to be searched for the different variants of query execution. Based on cost estimates, the optimizer then selects the optimal variant, which it represents in the execution plan. Wow, so much more complicated than we expected!

The *execution plan* decided upon by the server *specifies* which calculation steps will be performed in which order, and which keys or indexes will be used for efficient access. By *access*, I mean the way the database server can find and gather all relevant data from the database to answer the SQL-query. For each `SELECT` statement, we can take a look at the execution plan by placing the `EXPLAIN` keyword in front of the statement and executing the entire statement.

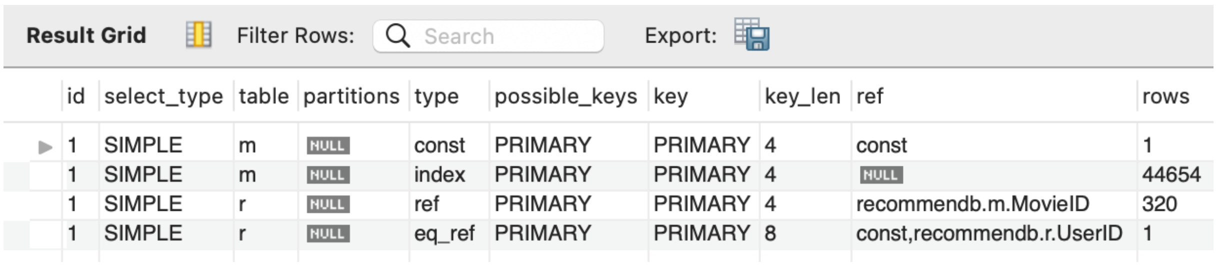
Show the execution plan for movie recommendations

Code6.1 demonstrates how to query the execution plan for the calculation of the *Star Trek* movie recommendations from Chapter 5.

- Execute the following SQL statement in MySQL Workbench. As a result, you should see the table in Figure 6.2.

```
EXPLAIN
SELECT MovieId, COUNT(*) AS N5R
FROM Ratings5
WHERE UserID IN (
  SELECT UserId AS TargetGroup
  FROM Ratings5
  WHERE MovieID = 1371 )
GROUP BY MovieID
ORDER BY COUNT(*) DESC;
```

Code6.1: Calling the execution plan for a query with `EXPLAIN`


Result Grid   Filter Rows: Search   Export:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
1	SIMPLE	m	NULL	const	PRIMARY	PRIMARY	4	const	1
1	SIMPLE	m	NULL	index	PRIMARY	PRIMARY	4	NULL	44654
1	SIMPLE	r	NULL	ref	PRIMARY	PRIMARY	4	recommendb.m.MovieID	320
1	SIMPLE	r	NULL	eq_ref	PRIMARY	PRIMARY	8	const,recommendb.r.UserID	1

Figure 6.2: Execution plan for move recommendations for Star Trek

The `EXPLAIN` keyword requests the execution plan for whatever query we provide subsequently. Here, we will get the final execution plan decided upon by the server for the `SELECT` query that requests movie recommendations for *Star Trek*.

What does the execution plan look like?

As we can see in Figure 6.2, the execution plan is presented in a table. Here's how to read this: it's the final execution plan for each part of the full SELECT query, listing the steps the query will run in.

Each row is one step that the server will take in executing the full query. The definition of what each column shows and all the possible values it can hold can be found in the documentation[10] of MySQL. Here I detail the most important columns:

- `Table` says which table (or table alias) the step processes. Here, `m` stands for the alias we gave the table Movies in Code 5.5 back in Chapter 5, and `r` stands for the alias for the table Ratings in Code 5.5. This means that the optimizer takes the SQL in the definition of the view Ratings5 and estimates the number of rows to process it. That's why the aliases from the view definition appear here in the execution plan.
- `Type` shows how tables are joined. For the details of the join types, refer to the MySQL documentation. For example, here we see
    - `const` – There is only one single matching row, and the values are considered constants.

[10] https://dev.mysql.com/doc/refman/8.0/en/explain-output.html

  - `Index` – each combination of rows from the previous tables is compared using index lookups.
  - `ref` – for each combination of rows from the previous tables, all rows with matching index values are read from this table.
  - `eq_ref` – One row is read from this table for each combination of rows from the previous tables.
- `Possible_keys` lists which indexes or keys from the table can be used as access paths to speed up the queries. So this will sometimes have multiple values.
- `Key` determines which index the server has chosen to use as access path.
- `Ref` shows which column will be accessed to get the relevant values.
- `Rows` shows an estimate of the number of rows expected to be searched, which is our cost estimate for the whole step. The fewer rows, the more efficient the query.

How can we speed up queries?

The optimizer does its own work of choosing the quickest or most cost-efficient way to execute a query, but there are things we as users can do to help it.

First, for efficient querying, all data accesses should be based on an index or a key if possible. A database is like a book: if I must search through all the pages to find the right place, it will take much longer than if I can jump directly to the right page via an index. So, we can try the query out with different indexes to find the execution plan variant that is fastest.

Second, as few rows as possible should be searched. We want to keep the cost estimate as low as possible. We can compare different variants of the query here to find the most efficient one.

In the execution plan for our *Star Trek* recommendation query in Figure 6.2, we see in the column `keys` that for all substeps the `PRIMARY` key of the table can be used. Since `PRIMARY` is the index for the table, this is already using the optimum key, and so this cannot be further optimized from an indexing point of view. We move on to the next significant column, `type`.

The third line in `type` shows the access type `ref`, for reference. The fact that the ref type has to read all matching rows in one table for each combination of rows results in the previous table, this results a multiplication of the effort required to execute that substep. For each of the 44,654 movies, the optimizer expects that 320 associated ratings must all be searched to find the five-star ratings for the movie Star Trek. This number, 320, is the cost estimation of the optimizer based on the statistics it gathered. We can improve the join type by rewriting the query. For example, we can store intermediate results of a subquery in a new table, which is then called a *materialized view*, which we will look at in the next subsection.

Here we only focus on these four items, the key, the type, the table, and the number of rows, because in my experience, indexing and query rewriting are the best ways to change the content of the key and type to reduce the number of rows processed for a table to increase speed.

6.3 Create materialized views

What is a materialized view?

A *materialized view* is a database object that contains the results of a query as a stored (i.e. “materialized”) set of records. It looks like this is what a regular view did. But in fact, for a regular view the content is calculated using the query that defines it each time it is used, which makes it slow. In contrast, in a materialized view the results are not recalculated each time. The results of a `SELECT` statement are precalculated once and then stored in a table for later reuse. This is therefore much quicker to re-run and query.

Whenever the calculation of a subquery takes a long time, we may be able to speed up the query if we store intermediate results for parts of the query in a materialized view. We’ll try this out and measure the run times.

Optimize performance with a materialized view

In MySQL, we create materialized views using the following instruction:

```
CREATE TABLE <name> AS SELECT <query>
```

Code 6.2: Schema to create a materialized view in MySQL

Let's try this out now.

- Execute the statement in Code 6.3 in the MySQL Workbench. This creates the materialized view `Ratings5M` (`5` for five stars and `M` for materialized).

This saves all five-star ratings as results in a separate table that approximately is five times smaller than the original ratings table, which increases speed by reducing the number of rows to process. When we call `Ratings5M`, we therefore eliminate the need to search through all reviews to find those with five stars. Also, Code 6.3 stores the movie titles with the corresponding reviews, so the lookup for titles can an also be eliminated from the final query.

```
CREATE TABLE Ratings5M AS
SELECT UserID, m.MovieID, Title
FROM Ratings r, Movies m
WHERE rating = 5
AND r.MovieID = m.MovieID;
```

Code 6.3: Materialized view with `CREATE TABLE ... AS SELECT`

Now we can use this newly created materialized view `Ratings5M` instead of the original view `Ratings5` in our movie recommendation query to see if this speeds things up.

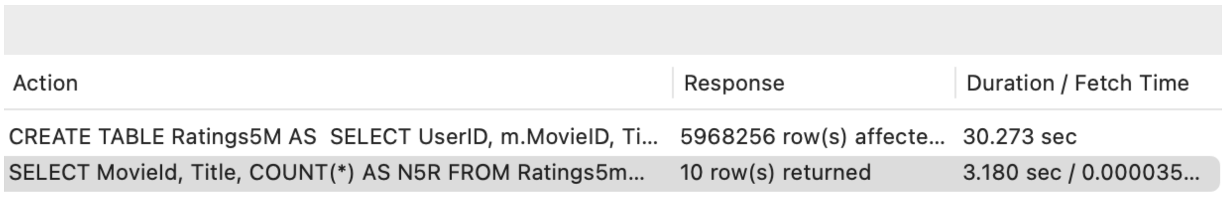
- Execute the `SELECT` statement from Code 6.4 in MySQL Workbench to calculate movie recommendations using the new materialized view.

```
SELECT MovieId, Title, COUNT(*) AS N5R
FROM Ratings5M
WHERE UserID IN (
  SELECT UserId AS TargetGroup
  FROM Ratings5m
  WHERE MovieID = 1371 )
GROUP BY MovieID, Title
ORDER BY COUNT(*) DESC
LIMIT 10;
```

Code 6.4: Optimizing the runtime with a materialized view

After executing these two queries, we can see feedback from the database server in the `Action Output` window, as shown in Figure 6.3. According to the `Duration / Fetch Time` column, the creation of the materialized view takes about 30 seconds in my system. But then, the movie recommendation only takes 3 seconds, which means it now runs twice as fast as the original query that we executed in Chapter 5, which took 6 seconds, as you can see in Figure

5.10. Your own times will vary a little depending on your system setup.



Action	Response	Duration / Fetch Time
CREATE TABLE Ratings5M AS SELECT UserID, m.MovieID, Ti...	5968256 row(s) affecte...	30.273 sec
SELECT MovieId, Title, COUNT(*) AS N5R FROM Ratings5m...	10 row(s) returned	3.180 sec / 0.000035...

Figure 6.3: Improved query runtime with materialized view.

Now that the materialized view is made, we can run the recommendation query using this view for faster results. This shows that this kind of optimization is suitable for apps that will use the same query or similar queries many times. This is certainly the case for a movie recommendation system.

Analyze the execution plan with materialized view

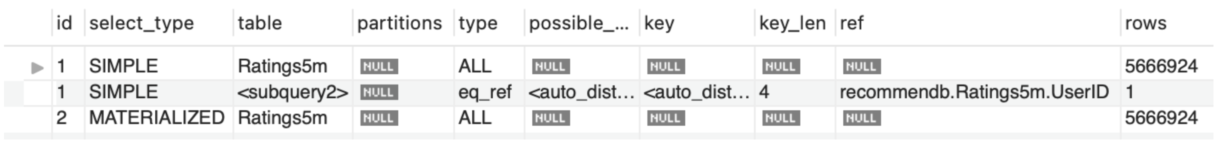
Let's look at the execution plan of the optimized query for comparison. Note that this does not include the creation of the view, but is just the recommendation query that uses the view.

- Execute the following query (Code 6.5). You should results as in Figure 6.4.

```
EXPLAIN
SELECT MovieId, Title, COUNT(*) AS N5R
FROM Ratings5m
WHERE UserID IN (
  SELECT UserId AS TargetGroup
  FROM Ratings5M
  WHERE MovieID = 1371 )
```

```
GROUP BY MovieID, Title
ORDER BY COUNT(*) DESC
LIMIT 10;
```

Code 6.5: Execution plan with materialized view



id	select_type	table	partitions	type	possible_...	key	key_len	ref	rows
1	SIMPLE	Ratings5m	NULL	ALL	NULL	NULL	NULL	NULL	5666924
1	SIMPLE	<subquery2>	NULL	eq_ref	<auto_dist...	<auto_dist...	4	recommendb.Ratings5m.UserID	1
2	MATERIALIZED	Ratings5m	NULL	ALL	NULL	NULL	NULL	NULL	5666924

Figure 6.4: Execution plan with materialized view

We have created a materialized view, we have not yet created indexes to make lookups faster. It is interesting to see: although the `NULL` value in the key column in the first row tells us that this plan is not using an index to access the `Ratings5m` table, and although the cost estimate for this row is much higher than in any of the rows in the original execution plan, our query is still twice as fast. That's because, as we see in the execution plan, this requires three steps and not four: the use of the materialized view eliminates the need to use the reference from the `Movies` table to the (very large) rating table `Ratings` for all films, and so the multiplier effect we saw above disappears. The query linearly searches the materialized view in two *full table scans*. This is shown with the value `ALL` in the `type` column. That is, every row of the `Ratings5M` table is searched, but because `Ratings5M` uses a much smaller amount of pre-filtered data, the query is now noticeably faster.

But we can see in Figure 6.4 from the `NULL` values in the `keys` column that no indexes are yet used for searching our materialized view (substeps 1 and 3). That means we can optimize this further by using indexes.

6.4 Create indexes on search columns

What is an index good for? Once again, we fall back on an analogy: Books tend to have indexes that list keywords and the book pages on which those keywords occur. Using the index, we can jump directly to the right page for a given keyword, because we only need to look at the page numbers and not at the content of the pages.

An *index* in a database system works in the same way. The server manages data on so-called *data pages*. If no index or key exists, all data pages must be searched to find the correct data records. This is a full table scan, or an execution step of type `ALL` as we saw in Figure 4. If an index is used as access path, then instead the server has a list of which data values of a column or column combination occur on which data pages. Accordingly, the database server can access the relevant data pages directly.

The values might also be structured in a tree-like manner, which additionally accelerates the search process, especially with a large amount of different data values. Instead of searching a very large list of index terms, it follows branches of the tree on which the value can possible exist, for example using alphabetic or numeric categorization, discarding any branches that cannot hold the value and reducing the amount of searching needed.

Create a materialized view with indexes

We can create an index in SQL with the `CREATE INDEX` instruction. We will now create an index for the materialized view:

- Execute the following SQL statements (Code 6.6). If you want to execute several statements at once in MySQL Workbench, you use the other lightning button to execute a selected portion or all of the content in a code window.

```
CREATE TABLE Ratings5MI AS
SELECT UserID, m.MovieID, Title
FROM Ratings r, Movies m
WHERE rating = 5
AND r.MovieID = m.MovieID;

CREATE INDEX IX_MovieID ON Ratings5MI(MovieID);
CREATE INDEX IX_UserID ON Ratings5MI(UserID);
```

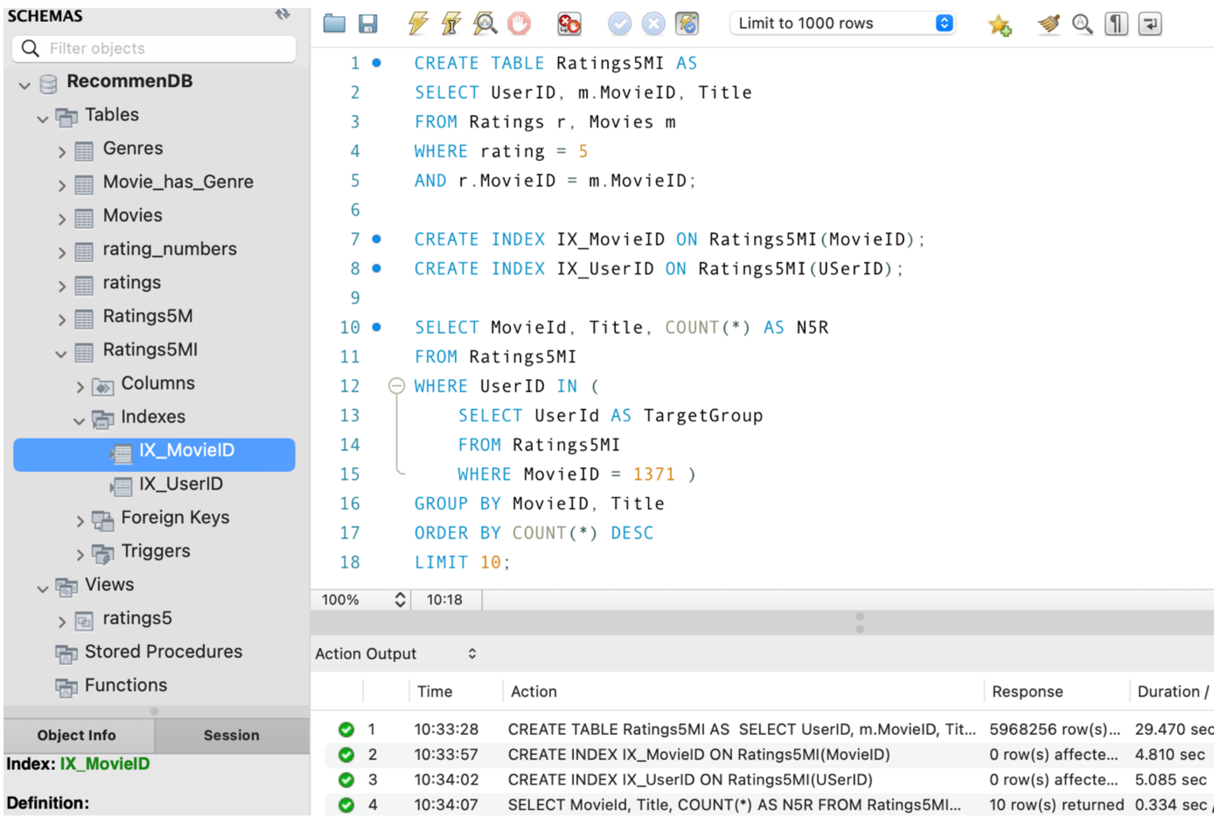
Code 6.6: Create indexes with `CREATE INDEX`

This creates the materialized view (`M`) `Ratings5MI` of five-star ratings (`5`) with two indexes (`I`), one for the `MovieID` and one for each `UserId` columns. This means that the movieIDs and the UserIDs become like keywords in a book index, and the rows in the Ratings5MI table are like the page numbers in a book. Now we can adjust the query for movie recommendations accordingly.

- Execute the SQL statement in the Code 6.7. We can see the result in Figure 6.5.

```
SELECT MovieId, Title, COUNT(*) AS N5R
FROM Ratings5MI
WHERE UserID IN (
  SELECT UserId AS TargetGroup
  FROM Ratings5MI
  WHERE MovieID = 1371 )
GROUP BY MovieID, Title
ORDER BY COUNT(*) DESC
LIMIT 10;
```

Code 6.7: Further optimization with indexed materialized views



```
CREATE TABLE Ratings5MI AS
SELECT UserID, m.MovieID, Title
FROM Ratings r, Movies m
WHERE rating = 5
AND r.MovieID = m.MovieID;

CREATE INDEX IX_MovieID ON Ratings5MI(MovieID);
CREATE INDEX IX_UserID ON Ratings5MI(USerID);

SELECT MovieId, Title, COUNT(*) AS N5R
FROM Ratings5MI
WHERE UserID IN (
    SELECT UserId AS TargetGroup
    FROM Ratings5MI
    WHERE MovieID = 1371 )
GROUP BY MovieID, Title
ORDER BY COUNT(*) DESC
LIMIT 10;
```

```
     Time      Action                                                 Response             Duration /
✓ 1  10:33:28  CREATE TABLE Ratings5MI AS  SELECT UserID, m.MovieID, Tit...  5968256 row(s)...  29.470 sec
✓ 2  10:33:57  CREATE INDEX IX_MovieID ON Ratings5MI(MovieID)        0 row(s) affecte...  4.810 sec
✓ 3  10:34:02  CREATE INDEX IX_UserID ON Ratings5MI(USerID)          0 row(s) affecte...  5.085 sec
✓ 4  10:34:07  SELECT MovieId, Title, COUNT(*) AS N5R FROM Ratings5MI...  10 row(s) returned  0.334 sec
```

Figure 6.5 Accelerated query with materialized view with index

We see in the navigation panel in Figure 6.5, under `SCHEMAS -> RecommenDB -> Tables`, that the table `Ratings5MI` has been created. This contains two indexes: `IX_MovieID` and `IX_UserID`. In the `Action Output` we see on line 4 that the 10 movie recommendations have been returned in 0.334 seconds (your time result may differ slightly). That means that indexing made the query ten times faster than the original!

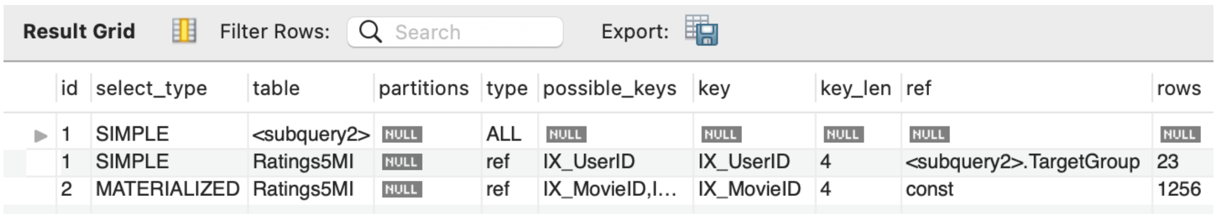
Analyze the execution plan with the new indexes

Let's take a look at the new execution plan:

- Execute the following SQL statements on the server (Code 6.8). We can see the result in Figure 6.6.

```
EXPLAIN
SELECT MovieId, Title, COUNT(*) AS N5R
FROM Ratings5MI
WHERE UserID IN (
  SELECT UserId AS TargetGroup
  FROM Ratings5MI
  WHERE MovieID = 1371 )
GROUP BY MovieID, Title
ORDER BY COUNT(*) DESC
LIMIT 10;
```

Code 6.8: Execution plan for materialized view with indexes


Result Grid | Filter Rows: Search | Export:

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
▶	1	SIMPLE	<subquery2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL
	1	SIMPLE	Ratings5MI	NULL	ref	IX_UserID	IX_UserID	4	<subquery2>.TargetGroup	23
	2	MATERIALIZED	Ratings5MI	NULL	ref	IX_MovieID,I...	IX_MovieID	4	const	1256

Figure 6.6: Execution plan with materialized view and indexes

As we see in the execution plan in Figure 6.6, now the server does not have to search the whole table of five-star ratings twice anymore. The new execution plan is to simply search all results of the subquery (five-star ratings for the movie with ID 1371) row by row, then use references to `UserID` and `MovieID` to search the rating tables for matching ratings of other movies. Thanks to the indexes, significantly fewer rows now need to be searched. This explains the speedup.

Thus, we have already accelerated the query from 6 seconds to 0.3 seconds, which corresponds to a factor of about 20. For further acceleration, we can reduce the amount of data with so-called *subsampling*.

6.5 Optimize the quality of recommendations

Now that we have very fast queries, we can incorporate additional information to increase query quality. We'll use more indexed materialized views to add additional data to our analysis to improve the quality of our recommendations without reducing speed.

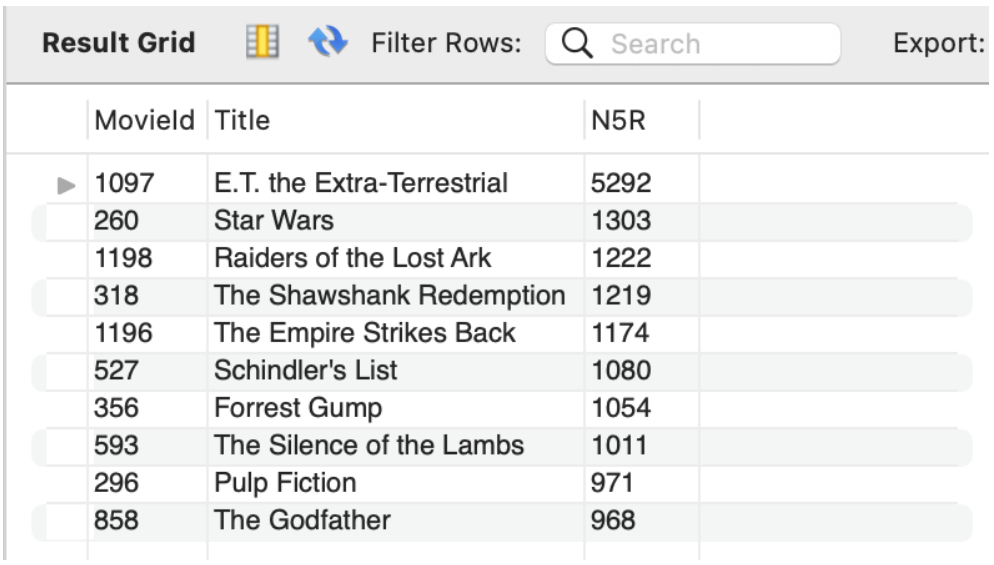
Generate recommendations relevant to a particular film

- Execute the following query (Code 6.9). This generates recommendations for people who liked the movie *E.T.*. You can see the result in Figure 6.7.

```
SELECT MovieID,Title FROM Movies
WHERE Title LIKE '%E.T.%'; -- 1097

SELECT MovieID, Title, COUNT(*) AS N5R
FROM Ratings5MIS
WHERE UserID IN (
  SELECT UserId AS TargetGroup
  FROM Ratings5MI
  WHERE MovieID = 1097 )
GROUP BY MovieID, Title
ORDER BY COUNT(*) DESC
LIMIT 10;
```

Code 6.9: Movie Recommendations with alternative `MovieID`



MovieId	Title	N5R
1097	E.T. the Extra-Terrestrial	5292
260	Star Wars	1303
1198	Raiders of the Lost Ark	1222
318	The Shawshank Redemption	1219
1196	The Empire Strikes Back	1174
527	Schindler's List	1080
356	Forrest Gump	1054
593	The Silence of the Lambs	1011
296	Pulp Fiction	971
858	The Godfather	968

Figure 6.7: Recommendations for the film E.T.

If we compare the list of recommendations of the movie E.T. in Figure 6.7 with the list of the most popular movies in Figure 5.8in Chapter 5, we can see that seven of the top ten recommendations for the movie E.T. are also among the highest rated movies in general. This isn't all that helpful to the user, and supports the assumption that popular films are weighted too much.

To adjust the weighting, we can measure the overall number of five-star ratings for the entire dataset. Then, we can compare the attractiveness to the target audience to the overall attractiveness to all audiences, and give more weight to movies that are less popular to create more specific and individualized recommendations.

Generate alternative ratings

For this purpose, we first need to determine the general popularity of a movie. To do this, we count the number of five-star ratings it receives, regardless of any user group, and store the result in another indexed materialized view for speedy usage.

- Execute Code 6.10 to create the new materialized view of the number of five-star ratings per movie id so that we can look this number up later for particular `MovieID`s.

```
CREATE TABLE Rating_Numbers AS
SELECT MovieID,
COUNT(*) AS N_Ratings5_All
FROM Ratings
WHERE Rating = 5
```

```
GROUP BY MovieID;

CREATE INDEX IX_MovieID
ON RatingNumbers(MovieID);
```

Code 6.10: Indexed materialized view for overall rating numbers

Then we have to modify the query for movie recommendations from Code 6.9 accordingly. Now, we'll sort by a new formula that compares the popularity of a movie for the target group with the overall popularity. I'll explain this later.

- Now, execute Code 6.11 that modifies the query. The adjusted rows are highlighted in bold.

```
SELECT MovieId, r.Title,
N_Ratings5_all, COUNT(*) AS N_Ratings5_target
FROM Ratings5MI r
JOIN rating_numbers USING (MovieID)
WHERE UserID IN (
  SELECT UserId AS TargetGroup
  FROM Ratings5MIS
  WHERE MovieID = 1097 )
GROUP BY MovieID, Title, N_Ratings5_all
ORDER BY COUNT(*)*COUNT(*)/N_Ratings5_all DESC
LIMIT 10;
```

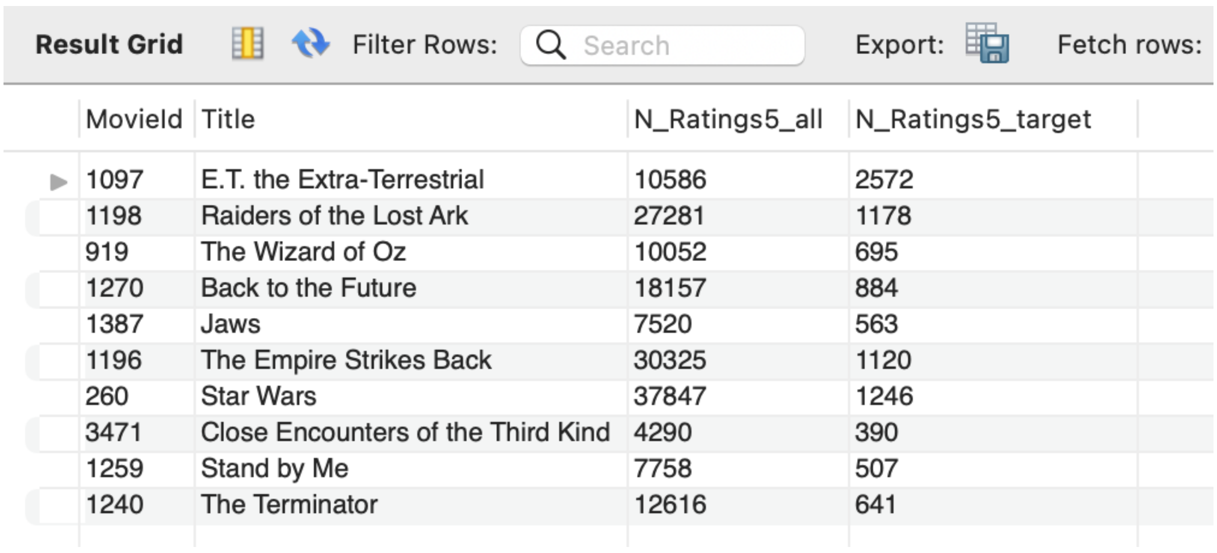
Code 6.11: Alternative sorting with a different relevance measure

On the fourth line of Code 6.11, we have added the new table `RatingNumbers` to include the additional information how popular a movie is in general. Then, on line 10 of Code 6.11, we can adjust the sorting to discount the general popularity of the recommended movies . Here's how this works: We weight the number of five-star ratings of the target group quadratically with the function `COUNT(*)*COUNT(*)`. Practically speaking, we multiply the number of five-star ratings from our target group by itself, to give this measure more weight. Then we set this number in relation to the total number of five-star ratings using the column `N_Ratings5_all` that we get from the newly joined materialized view `Ratings5MI`. You can see the result in Figure 6.8.

The recommendations now seem to contain significantly fewer of the generally popular movies, and so contain more movies that are particularly relevant to E.T. fans. In fact, only one of the top ten recommendations for the target group are among the ten most popular movies: *Star Wars*.

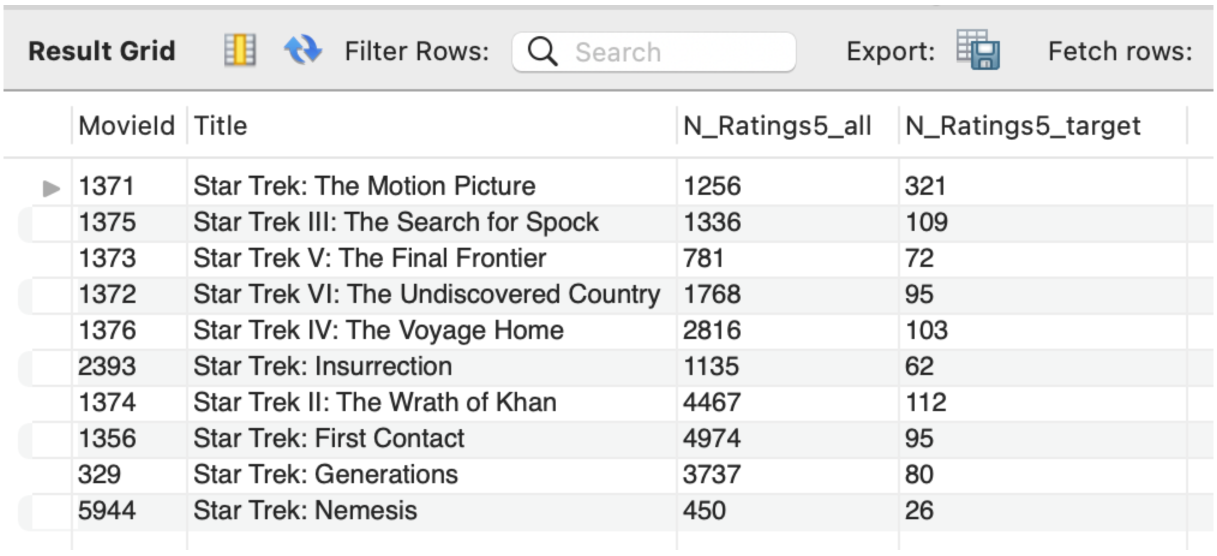
The inclusion of the entry *Close Encounters of the Third Kind* suggests that the adaptation worked, since this is also a Spielberg film about extra-terrestrials and is therefore clearly related.

We can rerun this query for our earlier target movie, Star Trek, by exchanging the `MovieID` on line 8 of Code 6.11. Then we can compare the new results in Figure 6.9 with the old results in Figure 5.10. In my opinion, the results are now better because we now only get Star Trek movies as recommendations. That's Interesting!



MovieId	Title	N_Ratings5_all	N_Ratings5_target
1097	E.T. the Extra-Terrestrial	10586	2572
1198	Raiders of the Lost Ark	27281	1178
919	The Wizard of Oz	10052	695
1270	Back to the Future	18157	884
1387	Jaws	7520	563
1196	The Empire Strikes Back	30325	1120
260	Star Wars	37847	1246
3471	Close Encounters of the Third Kind	4290	390
1259	Stand by Me	7758	507
1240	The Terminator	12616	641

Figure 6.8: Alternative recommendations for the movie E.T.



MovieId	Title	N_Ratings5_all	N_Ratings5_target
1371	Star Trek: The Motion Picture	1256	321
1375	Star Trek III: The Search for Spock	1336	109
1373	Star Trek V: The Final Frontier	781	72
1372	Star Trek VI: The Undiscovered Country	1768	95
1376	Star Trek IV: The Voyage Home	2816	103
2393	Star Trek: Insurrection	1135	62
1374	Star Trek II: The Wrath of Khan	4467	112
1356	Star Trek: First Contact	4974	95
329	Star Trek: Generations	3737	80
5944	Star Trek: Nemesis	450	26

Figure 6.9: Improved Recommendations for Star Trek

As we see, we have been able to recommend more specialized movies that seem to be thematically more related to the target movie by including a discount on the movie popularity. So, we have now created an SQL query that provides us with movie recommendations.

For a complete database application, however, we do not want to write code. We want a visual, interactive interface. Therefore, in the next two chapters, we'll look at how to make the data in the database accessible to other applications safely, and how to implement a graphical user interface for movie recommendations that accesses our database system.

6.6 Questions

Performance bottlenecks

What do you think are the most principally influential or fundamental sources of database query performance bottlenecks?

Joining tables

How would you go about looking up reference values from one table in another table? To what extent is this a performance bottleneck? How can it be circumvented?

Phone books

Who still remembers the old telephones and telephone books that were once the quickest way to find a phone number? How does the index and sorting in a phone book increase the reading speed? What do a phone book and a database system have in common?

Hardware optimization

We have looked at optimizing queries using the database software. How can hardware help optimize the performance of an SQL query?

6.7 Challenges

Another alternative relevance measure

- Implement another alternative measure of recommendation relevance:
    - Instead of the number of five-star ratings, take into account the squared average ratings, divided by the overall number of ratings, regardless of the number of stars.
    - How does this method compare in terms of speed and quality of recommendations?

Steam games analysis

- In Chapter 3 you analyzed the steam games dataset and created a data model and a database schema. In chapter 4 you loaded the data. Now, create a query that computes an item-to-item collaborative filter for Steam games. Due to the large amount of data, apply the techniques of materialized views, indexes, and subsampling to get the runtime under control.

- Tip: For such large amounts of data, the database server needs long runtimes, lots of memory, and a large disk swap file. You can achieve these by adjusting the following parameters in the MySQL configuration file[11].
    - `innodb_buffer_pool_size=15G`
        - This sets the size of the cache; it should be about 50-60% of your computer's RAM. If this is too high, MySQL will crash.
    - `tmp_table_size=12G`
        - This sets the size of the temporary table in RAM, which should be about 40-50% of RAM.
    - `innodb_buffer_pool_instances=35`
        - This sets the number of parallel threads; more is faster, but needs more CPU.
    - `connect_timeout=500000`
        - This sets the wait time until timeout for long queries in sec; the default is only 10 min.

[11] For instruction, see: https://dev.mysql.com/doc/refman/8.0/en/option-files.html

Your own project

- If you have started your personal project in Chapter 1 and have continued working on it throughout the book, ou can now apply the techniques shown in this chapter to optimize the runtime of the queries.

References

[1] M. Kaufmann, “Big Data Management Canvas: A Reference Model for Value Creation from Data,” *Big Data and Cognitive Computing*, vol. 3, no. 1, p. 19, Mar. 2019, doi: 10.3390/bdcc3010019.

[2] A. Wegrzynek, “InfluxDB at CERN and Its Experiments,” Influxdata, Case Study, 2018. [Online]. Available: https://www.influxdata.com/customer/cern/

[3] D. Meiss, “Image recognition and search at Adobe with Elasticsearch and Sensei,” *Elastic Blog*, Dec. 11, 2019. https://www.elastic.co/blog/image-recognition-and-search-at-adobe-with-elasticsearch-and-sensei (accessed Jan. 09, 2022).

[4] MySQL, “University of Toronto Empowers Astronomers to Research Dark Matter with Massive Space Image Database.” https://www.mysql.com/why-mysql/case-studies/university-toronto-astronomers-massive-mysql-database.html (accessed Feb. 10, 2023).

[5] OECD, *Data-Driven Innovation: Big Data for Growth and Well-Being*. Paris: OECD Publishing, 2015. Accessed: Sep. 29, 2016. [Online]. Available: http://www.oecd.org/sti/data-driven-innovation-9789264229358-en.htm

[6] M. Hilbert and P. López, “The World’s Technological Capacity to Store, Communicate, and Compute Information,” *Science*, vol. 332, no. 6025, pp. 60–65, Apr. 2011, doi: 10.1126/science.1200970.

[7] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*, 1st edition. Upper Saddle River, N.J: Prentice Hall, 2002.

[8] P. P.-S. Chen, “The Entity-relationship Model - Toward a Unified View of Data,” *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, Mar. 1976, doi: 10.1145/320434.320440.

[9] D. D. Chamberlin and R. F. Boyce, “SEQUEL: A Structured English Query Language,” in *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access*

and Control, in SIGFIDET '74. New York, NY, USA: ACM, 1974, pp. 249–264. doi: 10.1145/800296.811515.

[10] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," Internet Engineering Task Force, Request for Comments RFC 8259, Dec. 2017. doi: 10.17487/RFC8259.

[11] M. Kaufmann and A. Meier, *SQL and NoSQL Databases*. Wiesbaden: Springer International Publishing, 2023.

[12] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970, doi: 10.1145/362384.362685.

[13] W. L. Chang, "NIST Big Data Interoperability Framework: Volume 1, Definitions," NIST Big Data Public Working Group, NIST Special Publication 1500–1, 16 2015. Accessed: Sep. 29, 2016. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.1500-1.pdf

[14] S. K. Bansal and S. Kagemann, "Integrating Big Data: A Semantic Extract-Transform-Load Framework," *Computer*, vol. 48, no. 3, pp. 42–50, Mar. 2015, doi: 10.1109/MC.2015.76.

[15] S. Henry, S. Hoon, M. Hwang, D. Lee, and M. D. DeVore, "Engineering trade study: extract, transform, load tools for data migration," in *2005 IEEE Design Symposium, Systems and Information Engineering*, Apr. 2005, pp. 1–8. doi: 10.1109/SIEDS.2005.193231.

[16] A. Gorelik, *The Enterprise Big Data Lake: Delivering on the Promise of Hadoop and Data Science in the Enterprise*, Illustrated Edition. Sebastopol, California: O'Reilly UK Ltd., 2019.

[17] A. Simitsis and P. Vassiliadis, "Extraction, Transformation, and Loading," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds., New York, NY: Springer, 2018, pp. 1432–1440. doi: 10.1007/978-1-4614-8265-9_158.

[18] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, Jan. 2003, doi: 10.1109/MIC.2003.1167344.

[19] Y. E. Ioannidis, “Query optimization,” *ACM Comput. Surv.*, vol. 28, no. 1, pp. 121–123, Mar. 1996, doi: 10.1145/234313.234367.

[20] S. Chaudhuri, “An overview of query optimization in relational systems,” in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, in PODS ’98. New York, NY, USA: Association for Computing Machinery, May 1998, pp. 34–43. doi: 10.1145/275487.275492.