

Stage 3 Report:

Here's a screenshot of the GCP cloud shell, as well as the SHOW TABLES query.

```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to release-watch-355801.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
ntsiones@cloudshell:~ (release-watch-355801)$ gcloud sql connect releasewatch-sql --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 736
Server version: 8.0.26-google (Google)

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> USE releasewatch;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_relasewatch |
+-----+
| Companies               |
| Games                  |
| Media                  |
| News                   |
| Submission              |
| Users                  |
| WorkedOn               |
+-----+
7 rows in set (0.00 sec)

mysql> █
```

Here's 1000+ entries in each table with data:

```
mysql> SELECT COUNT(*) FROM Companies;
+-----+
| COUNT(*) |
+-----+
|      1000 |
+-----+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) FROM Games;
+-----+
| COUNT(*) |
+-----+
|      1256 |
+-----+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) FROM Submission;
+-----+
| COUNT(*) |
+-----+
|      2238 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM Users;
+-----+
| COUNT(*) |
+-----+
|      1101 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM WorkedOn;
+-----+
| COUNT(*) |
+-----+
|      1625 |
+-----+
1 row in set (0.00 sec)

mysql> █
```

Advanced Query 1 (Games submitted by users with a below-average activity rate):

```
mysql> SELECT DISTINCT name, submittedBy, points FROM Games JOIN Users ON submittedBy = username WHERE points < (SELECT AVG(points) FROM Users) ORDER BY points ASC limit 15;
```

name	submittedBy	points
Fairy Fencer F: Refrain Chord	SplendidGrandpa808	1
Serious Sam HD: Gold Edition	SplendidGrandpa808	1
Thymesia	SplendidGrandpa808	1
10 Ways From Sunday	GrimyCat92	6
Blossom Tales 2: The Minotaur Prince	GrimyCat92	6
Streets of Rage 2	NecessaryGamer47	7
Hyphen	InnocentGrandpa04	10
Swords & Soldiers	BrightGuy676	14
Check vs. Mate	LustrousWoman259	26
Trip to Vinelands	LustrousWoman259	26
No Return	BigMan276	50
Tyrron Cuthbert: Attorney of the Arcane	PracticalDog474	55
100 Doors Games: Escape from School	PracticalDog474	55
The Ascent: Cyber Edition	PoliticalGrandma24	58
007 Legends: Eve	PoliticalGrandma24	58

15 rows in set (0.01 sec)

Advanced Query 2 (Companies active before 2000 and companies active after 2010):

```
mysql> (select DISTINCT c.companyName,g.name, g.releaseDate FROM Games g JOIN WorkedOn w ON g.gameID = w.game JOIN Companies c ON w.company = c.companyID where g.releaseDate < 946684801 limit 8 ) UNION (select DISTINCT c.companyName,g.name,g.releaseDate FROM Games g JOIN WorkedOn w ON g.gameID=w.game JOIN Companies c ON w.company = c.companyID where g.releaseDate > 1262304001 limit 8) ;
```

companyName	name	releaseDate
Crytek Frankfurt	Quake II	881625600
(Archive) EA Games - duplicate	Quake II	881625600
Slant Six Games	Patrician III: Rise of the Hanse	946684800
Gray Matter Interactive	Quake	835401600
505 Games	Quake	835401600
Morning Star Multimedia	101 Airborne: The Airborne Invasion of Normandy	907113600
EA Tiburon	101 Airborne: The Airborne Invasion of Normandy	907113600
Compu-Games A/S	Age of Wonders	942278400
VU Games	Star Wars: The Force Unleashed II	1288051200
Nintendo EAD Group No. 4	Rift	1298937600
BMV! Entertainment	Command & Conquer 4: Tiberian Twilight	1268687600
(Archive) Konami Eighting	Command & Conquer 4: Tiberian Twilight	1268687600
Firefly Studios	Red Orchestra 2: Heroes of Stalingrad	1315872000
Redlynx	Red Orchestra 2: Heroes of Stalingrad	1315872000
FormGen	Men of War: Assault Squad	1298592000
Pocketeers	Men of War: Assault Squad	1298592000

16 rows in set (0.00 sec)

DDL Commands:

```
CREATE DATABASE /*!32312 IF NOT EXISTS*/ `releasewatch` /*!40100 DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci */ /*!80016 DEFAULT ENCRYPTION='N' */;
```

```
CREATE TABLE `Companies` (  
  `companyID` int NOT NULL,  
  `companyName` varchar(100) DEFAULT NULL,  
  `companyType` varchar(30) DEFAULT NULL,  
  `companySize` varchar(30) DEFAULT NULL,  
  `submittedBy` varchar(30) DEFAULT NULL,  
  PRIMARY KEY (`companyID`),  
  KEY `submittedBy` (`submittedBy`),  
  CONSTRAINT `Companies_ibfk_1` FOREIGN KEY (`submittedBy`) REFERENCES `Users`  
  (`username`)  
)
```

```
CREATE TABLE `Games` (  
  `gameID` int NOT NULL,
```

```

`name` varchar(255) DEFAULT NULL,
`description` varchar(2000) DEFAULT NULL,
`releaseDate` int DEFAULT NULL,
`submittedBy` varchar(30) DEFAULT NULL,
PRIMARY KEY (`gameID`),
KEY `fk_submits` (`submittedBy`),
CONSTRAINT `fk_submits` FOREIGN KEY (`submittedBy`) REFERENCES `Users`
(`username`),
CONSTRAINT `fk_submittedBy` FOREIGN KEY (`submittedBy`) REFERENCES `Users`
(`username`)
)

```

```

CREATE TABLE `Media` (
  `mediaID` int NOT NULL,
  `altText` varchar(1000) DEFAULT NULL,
  `link` varchar(1000) DEFAULT NULL,
  `isVideo` tinyint(1) DEFAULT NULL,
  `linkedTo` int DEFAULT NULL,
  PRIMARY KEY (`mediaID`),
  KEY `linkedTo` (`linkedTo`),
  CONSTRAINT `Media_ibfk_1` FOREIGN KEY (`linkedTo`) REFERENCES `News` (`newsID`)
)

```

```

CREATE TABLE `News` (
  `newsID` int NOT NULL,
  `text` varchar(2000) DEFAULT NULL,
  `submittedBy` varchar(30) DEFAULT NULL,
  PRIMARY KEY (`newsID`),
  KEY `submittedBy` (`submittedBy`),
  CONSTRAINT `News_ibfk_1` FOREIGN KEY (`submittedBy`) REFERENCES `Users`
(`username`)
)

```

```

CREATE TABLE `Submission` (
  `submissionID` int NOT NULL,
  `submissionType` varchar(10) DEFAULT NULL,
  `rawData` varchar(4000) DEFAULT NULL,
  `approved` tinyint(1) DEFAULT NULL,
  `submitDate` int DEFAULT NULL,
  `createdBy` varchar(30) DEFAULT NULL,
  PRIMARY KEY (`submissionID`),
  KEY `createdBy` (`createdBy`),
  CONSTRAINT `Submission_ibfk_1` FOREIGN KEY (`createdBy`) REFERENCES `Users`
(`username`)
)

```

)

```
CREATE TABLE `WorkedOn` (  
  `game` int NOT NULL,  
  `company` int NOT NULL,  
  PRIMARY KEY (`game`,`company`),  
  KEY `company` (`company`),  
  CONSTRAINT `WorkedOn_ibfk_1` FOREIGN KEY (`game`) REFERENCES `Games`  
  (`gameID`),  
  CONSTRAINT `WorkedOn_ibfk_2` FOREIGN KEY (`company`) REFERENCES `Companies`  
  (`companyID`)  
)
```

Original query: SELECT DISTINCT name, submittedBy, points FROM Games JOIN Users ON submittedBy = username WHERE points < (SELECT AVG(points) FROM Users) ORDER BY points ASC limit 15;

Unoptimized:

[illegible]

Add index on username:

```

-> Limit: 15 row(s) (actual time=5.447..5.450 rows=15 loops=1)
-> Sort: Users.points, limit input to 15 row(s) per chunk (actual time=5.446..5.448 rows=15 loops=1)
-> Table scan on <temp> (cost=0.02..8.94 rows=515) (actual time=0.002..0.101 rows=818 loops=1)
-> Temporary table with deduplication (cost=269.83..278.76 rows=515) (actual time=5.139..5.305 rows=818 loops=1)
-> Nested loop inner join (cost=218.28 rows=515) (actual time=0.456..4.518 rows=818 loops=1)
-> Filter: (Users.points < (select #2)) (cost=37.91 rows=367) (actual time=0.432..0.986 rows=716 loops=1)
-> Table scan on Users (cost=37.91 rows=1100) (actual time=0.030..0.397 rows=1101 loops=1)
-> Select #2 (subquery in condition; run only once)
-> Aggregate: avg(Users.points) (cost=221.25 rows=1100) (actual time=0.392..0.392 rows=1 loops=1)
-> Table scan on Users (cost=111.25 rows=1100) (actual time=0.018..0.281 rows=1101 loops=1)
-> Index lookup on Games using fk_submits (submittedBy=Users.username) (cost=0.35 rows=1) (actual time=0.004..0.005 rows=1 loops=716)

```

Adding an index on Users(username) did not change the efficiency of the query, which can be seen by comparing the costs of the two queries via explain analyze. For instance, both queries cost 111.25 to do a table scan on Users, meaning they have the same efficiency despite what the actual times are, since actual times vary with each run. This holds true for all of the costs in the two queries. The actual times for each action changes little for both, overall.

Add index on points:

```

-> Limit table scan on <temporary> (cost=0.01..15.07 rows=1006) (actual time=0.001..0.003 rows=15 loops=1)
-> Temporary table with deduplication (cost=603.62..618.68 rows=1006) (actual time=0.188..0.191 rows=15 loops=1)
-> Limit table size: 15 unique row(s)
-> Nested loop inner join (cost=502.96 rows=1006) (actual time=0.071..0.151 rows=15 loops=1)
-> Filter: (Users.points < (select #2)) (cost=150.71 rows=716) (actual time=0.033..0.038 rows=11 loops=1)
-> Index scan on Users using myIndex (cost=150.71 rows=716) (actual time=0.031..0.034 rows=11 loops=1)
-> Select #2 (subquery in condition; run only once)
-> Aggregate: avg(Users.points) (cost=221.25 rows=1100) (actual time=0.381..0.381 rows=1 loops=1)
-> Index scan on Users using myIndex (cost=111.25 rows=1100) (actual time=0.035..0.246 rows=1101 loops=1)
-> Index lookup on Games using fk_submits (submittedBy=Users.username) (cost=0.35 rows=1) (actual time=0.009..0.010 rows=1 loops=1)

```

Adding an index on `ONLY Users(points)`, on the other hand, changed the efficiency drastically. Although the costs for each action say they are higher, the actual time each action takes is much shorter, as can be seen by the actual times. While there is some variance in the runtime, the difference between 0.101 (indexed on points) and 5.305 (indexed on usernames/ no index) runtimes (from Temporary table with duplication) is significant. Using this index also removes an entire action, `Sort: Users.points`, which undoubtedly sped things up. I think that this is because indexing on points removes the need to sort through each point, removing that sorting action and making other actions involving points much faster. The username index might not have been as significant because there seems to automatically be indexes for all primary and foreign keys, making our index redundant.

Add index on username, points, submitted by:

This point can be seen with the final EXPLAIN ANALYZE, where we indexed Users.points, Users.username, and Games.submittedBy. It seems to have a similar efficiency to/ runtime with the indexing with only points, which makes sense if keys are automatically indexed for us. Since submittedBy is a foreign key, this would have already been indexed and its addition, along with username, does not change the efficiency of just indexing points. By comparing the costs and runtimes of the query with just the points Indexed and the query with everything indexed, we can see that the costs are the same, their structure is the same, and the runtimes are very similar. Indexing by just points, in this case, is just as good as indexing them all.

=====

Unoptimized:

Add index on releaseDate:

```

1 -> Table scan on <union temporary> (cost=0.177..2.70 rows=16) (actual time=0.0011..0.002 rows=16 loops=1)
   -> Union materialize with deduplication (cost=1249.44..1251.97 rows=16) (actual time=0.2331..0.236 rows=16 loops=1)
       -> Limit: 8 row(s) (cost=107.63..107.90 rows=8) (actual time=0.1471..0.150 rows=8 loops=1)
           -> Table scan on <temporary> (cost=0.041..3.73 rows=98) (actual time=0.0011..0.002 rows=8 loops=1)
               -> Temporary table with deduplication (cost=107.631..111.32 rows=98) (actual time=0.1471..0.149 rows=8 loops=1)
                   -> Limit table size: 8 unique row(s)
                       -> Nested loop inner join (cost=97.75 rows=98) (actual time=0.0891..0.128 rows=8 loops=1)
                           -> Nested loop inner join (cost=63.31 rows=98) (actual time=0.0811..0.104 rows=8 loops=1)
                               -> Index range scan on g using myindex1, with index condition: (g.releaseDate < 946684801) (cost=34.46 rows=76) (actual time=0.0691..0.072 rows=7 loops=1)
                                   -> Index lookup on w using PRIMARY (game=g.gameID) (cost=0.25 rows=1) (actual time=0.0041..0.004 rows=1 loops=7)
                                       -> Single-row index lookup on c using PRIMARY (companyId=w.company) (cost=0.25 rows=1) (actual time=0.0031..0.003 rows=1 loops=8)
                                           -> Limit: 8 row(s) (cost=1139.671..1139.77 rows=8) (actual time=0.0691..0.071 rows=8 loops=1)
                                               -> Table scan on <temporary> (cost=0.011..19.49 rows=1360) (actual time=0.0001..0.001 rows=8 loops=1)
                                                   -> Temporary table with deduplication (cost=1139.671..1159.15 rows=1360) (actual time=0.0691..0.070 rows=8 loops=1)
                                                       -> Limit table size: 8 unique row(s)
                                                           -> Nested loop inner join (cost=1003.70 rows=1360) (actual time=0.0221..0.057 rows=8 loops=1)
                                                               -> Nested loop inner join (cost=527.86 rows=1360) (actual time=0.0191..0.041 rows=8 loops=1)
                                                                   -> Fill(0.0001) (g.releaseDate < 126230401) (cost=123.25 rows=1050) (actual time=0.0111..0.012 rows=5 loops=1)
                                                                       -> Table scan on g (cost=123.25 rows=1050) (actual time=0.0101..0.019 rows=21 loops=1)
                                                                           -> Index lookup on w using PRIMARY (game=g.gameID) (cost=0.25 rows=1) (actual time=0.0031..0.003 rows=2 loops=5)
                                                                               -> Single-row index lookup on c using PRIMARY (companyId=w.company) (cost=0.25 rows=1) (actual time=0.0021..0.002 rows=1 loops=8)

```


Add index on game name and companyName:

```
-----+-----
| -> Table scan on <union temporary> (cost=0.17..2.70 rows=16) (actual time=0.000..0.002 rows=16 loops=1)
|   -> Union materialize with deduplication (cost=934.10..936.63 rows=16) (actual time=0.204..0.208 rows=16 loops=1)
|     -> Limit: 8 row(s) (cost=466.04..466.16 rows=8) (actual time=0.119..0.121 rows=8 loops=1)
|       -> Table scan on <temporary> (cost=0.02..8.16 rows=453) (actual time=0.001..0.002 rows=8 loops=1)
|         -> Temporary table with deduplication (cost=466.04..474.18 rows=453) (actual time=0.118..0.120 rows=8 loops=1)
|           -> Limit table size: 8 unique row(s)
|             -> Nested loop inner join (cost=420.71 rows=453) (actual time=0.064..0.097 rows=8 loops=1)
|               -> Nested loop inner join (cost=262.11 rows=453) (actual time=0.053..0.070 rows=8 loops=1)
|                 -> Filter: (g.releaseDate < 946684801) (cost=129.25 rows=350) (actual time=0.038..0.041 rows=5 loops=1)
|                   -> Table scan on g (cost=129.25 rows=1050) (actual time=0.035..0.037 rows=8 loops=1)
|                     -> Index lookup on w using PRIMARY (game=g.gameID) (cost=0.25 rows=1) (actual time=0.004..0.005 rows=2 loops=5)
|                       -> Single-row index lookup on c using PRIMARY (companyID=w.company) (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=8)
| -> Limit: 8 row(s) (cost=466.04..466.16 rows=8) (actual time=0.065..0.067 rows=8 loops=1)
|   -> Table scan on <temporary> (cost=0.02..8.16 rows=453) (actual time=0.000..0.001 rows=8 loops=1)
|     -> Temporary table with deduplication (cost=466.04..474.18 rows=453) (actual time=0.065..0.066 rows=8 loops=1)
|       -> Limit table size: 8 unique row(s)
|         -> Nested loop inner join (cost=420.71 rows=453) (actual time=0.017..0.054 rows=8 loops=1)
|           -> Nested loop inner join (cost=262.11 rows=453) (actual time=0.013..0.036 rows=8 loops=1)
|             -> Filter: (g.releaseDate > 1262304001) (cost=129.25 rows=350) (actual time=0.009..0.021 rows=5 loops=1)
|               -> Table scan on g (cost=129.25 rows=1050) (actual time=0.009..0.019 rows=21 loops=1)
|                 -> Index lookup on w using PRIMARY (game=g.gameID) (cost=0.25 rows=1) (actual time=0.002..0.003 rows=2 loops=5)
|                   -> Single-row index lookup on c using PRIMARY (companyID=w.company) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=8)
|
+-----+-----
```

Indexing the company name and games name did not change the query structure compared with no index at all, as seen by comparing the cost of the actions. This is likely because the query never needs to search for the names of the games or company, it already has access to the tuple and is simply grabbing the data, not changing anything. However, I suspect that this indexing might be faster if we decided to order the data by one of the attributes.

Add index on release date, game name, and company name:

```
-----+-----
| -> Table scan on <union temporary> (cost=0.17..2.70 rows=16) (actual time=0.000..0.002 rows=16 loops=1)
|   -> Union materialize with deduplication (cost=1249.44..1251.97 rows=16) (actual time=0.236..0.239 rows=16 loops=1)
|     -> Limit: 8 row(s) (cost=107.63..107.90 rows=8) (actual time=0.147..0.150 rows=8 loops=1)
|       -> Table scan on <temporary> (cost=0.04..3.73 rows=98) (actual time=0.001..0.002 rows=8 loops=1)
|         -> Temporary table with deduplication (cost=107.63..111.32 rows=98) (actual time=0.147..0.149 rows=8 loops=1)
|           -> Limit table size: 8 unique row(s)
|             -> Nested loop inner join (cost=97.75 rows=98) (actual time=0.089..0.128 rows=8 loops=1)
|               -> Nested loop inner join (cost=63.31 rows=98) (actual time=0.081..0.105 rows=8 loops=1)
|                 -> Index range scan on g using myIndex2, with index condition: (g.releaseDate < 946684801) (cost=34.46 rows=76) (actual time=0.070..0.072 rows=7 loops=1)
|                   -> Index lookup on w using PRIMARY (game=g.gameID) (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=7)
|                     -> Single-row index lookup on c using PRIMARY (companyID=w.company) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=8)
| -> Limit: 8 row(s) (cost=1139.67..1139.77 rows=8) (actual time=0.069..0.071 rows=8 loops=1)
|   -> Table scan on <temporary> (cost=0.01..19.49 rows=1360) (actual time=0.000..0.001 rows=8 loops=1)
|     -> Temporary table with deduplication (cost=1139.67..1159.15 rows=1360) (actual time=0.069..0.071 rows=8 loops=1)
|       -> Limit table size: 8 unique row(s)
|         -> Nested loop inner join (cost=1003.70 rows=1360) (actual time=0.022..0.057 rows=8 loops=1)
|           -> Nested loop inner join (cost=527.86 rows=1360) (actual time=0.019..0.040 rows=8 loops=1)
|             -> Filter: (g.releaseDate > 1262304001) (cost=129.25 rows=1050) (actual time=0.012..0.023 rows=5 loops=1)
|               -> Table scan on g (cost=129.25 rows=1050) (actual time=0.011..0.020 rows=21 loops=1)
|                 -> Index lookup on w using PRIMARY (game=g.gameID) (cost=0.25 rows=1) (actual time=0.002..0.003 rows=2 loops=5)
|                   -> Single-row index lookup on c using PRIMARY (companyID=w.company) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=8)
|
+-----+-----
```

And so, if we combine all three indexes (releaseDate, Games.name, Companies.companyName), the two '%name%' indexes will not change the efficiency, but the release date indexing will. This results in an efficiency that is the same as if we only indexed with releaseDate, as seen if you compare the two queries.