Table of Contents:

Implementation

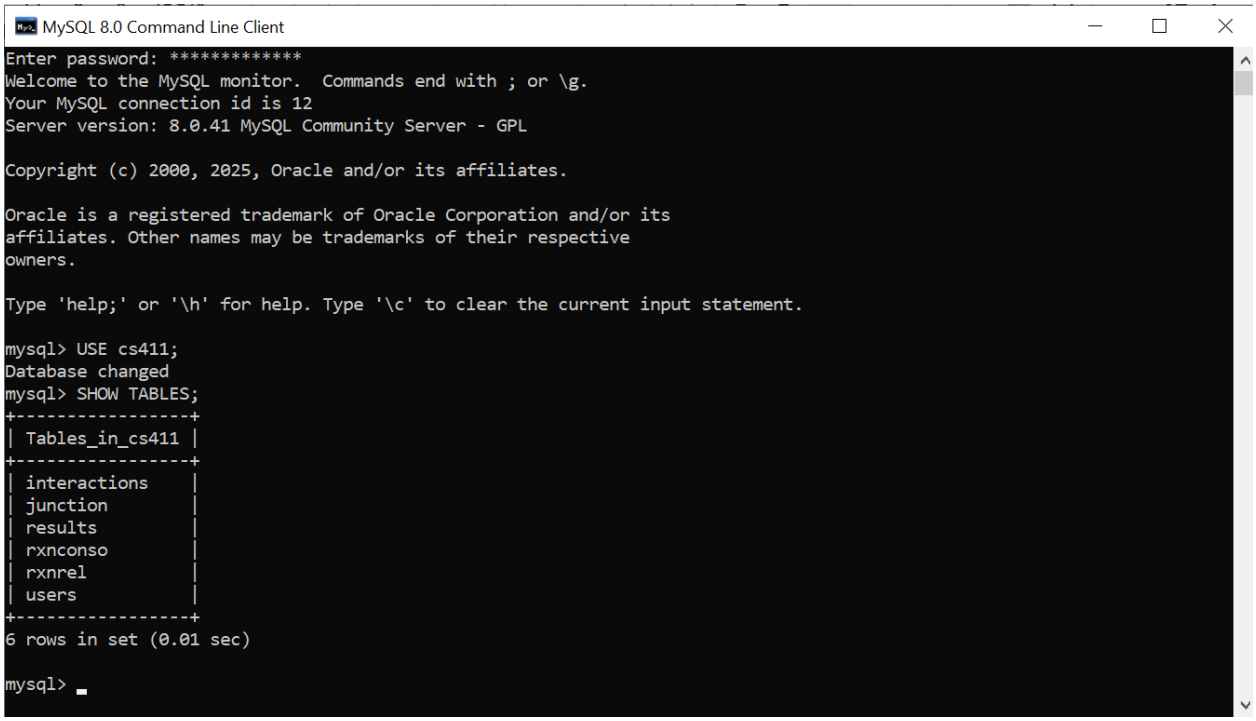
DDL commands used to create the tables:

```
CREATE TABLE rxnconso (
   RXCUI             VARCHAR (8) NOT NULL,
   LAT               VARCHAR (3) DEFAULT 'ENG' NOT NULL,
   TS                VARCHAR (1),
   LUI               VARCHAR (8),
   STT               VARCHAR (3),
   SUI               VARCHAR (8),
   ISPREF            VARCHAR (1),
   RXAUI             VARCHAR (20) NOT NULL,
   SAUI              VARCHAR (50),
   SCUI              VARCHAR (50),
   SDUI              VARCHAR (50),
   SAB               VARCHAR (20) NOT NULL,
   TTY               VARCHAR (20) NOT NULL,
   CODE              VARCHAR (50) NOT NULL,
   STR               VARCHAR (3000) NOT NULL,
   SRL               VARCHAR (10),
   SUPPRESS          VARCHAR (1),
   CVF               VARCHAR(50),
   PRIMARY KEY(RXAUI, RXCUI)
);
```

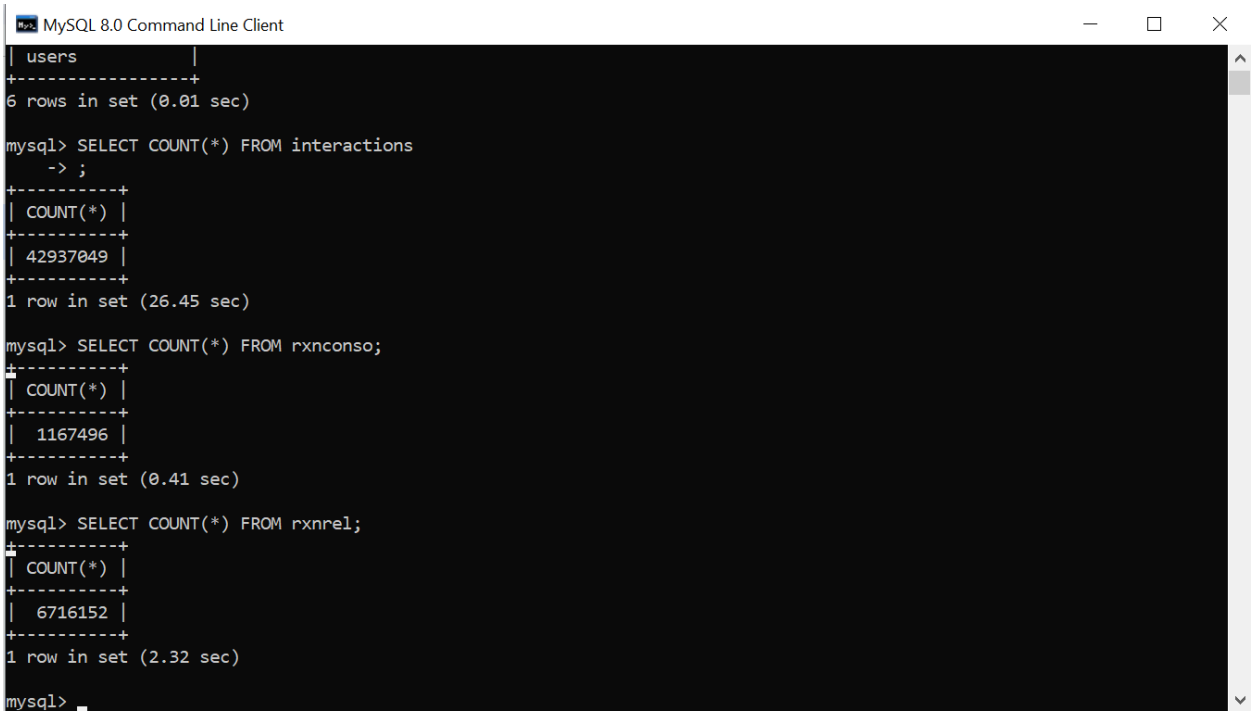
```sql
CREATE TABLE Interactions (
    RXCUI1 VARCHAR(8),
    drug_1_concept_name VARCHAR(150),
    RXCUI2 VARCHAR(8),
    drug_2_concept_name VARCHAR(150),
    condition_meddra_id VARCHAR(8),
    condition_concept_name VARCHAR(100),
    A VARCHAR(15),
    B VARCHAR(15),
    C VARCHAR(15),
    D VARCHAR(15),
    PRR VARCHAR(15),
    PRR_error VARCHAR(15),
    mean_reporting_frequency FLOAT,
    PRIMARY KEY(RXCUI1, RXCUI2, condition_meddra_id)
);

CREATE TABLE rxnrel (
    RXCUI1 VARCHAR(10) NOT NULL,
    RXAUI1 VARCHAR(10) NOT NULL,
    STYPE1 VARCHAR(50),
    REL VARCHAR(4),
    RXCUI2 VARCHAR(10) NOT NULL,
    RXAUI2 VARCHAR(10) NOT NULL,
    STYPE2 VARCHAR(50),
    RELA VARCHAR(100),
    RUI VARCHAR(10),
    SRUI VARCHAR(50),
    SAB VARCHAR(20) NOT NULL,
    SL VARCHAR(1000),
    DIR VARCHAR(3),
    RG VARCHAR(10),
    SUPPRESS VARCHAR(1),
    CVF VARCHAR(50),
    PRIMARY KEY(RUI)
);

CREATE TABLE Users (
    user_id INT,
    email VARCHAR(100),
    password_hash VARCHAR(100),
    PRIMARY KEY (user_id)
);
```

```
CREATE TABLE Results (
    dt_generated DATETIME,
    result_name VARCHAR(50),
    result_id INT,
    user_id INT,
    PRIMARY KEY (result_id),
    FOREIGN KEY (user_id) REFERNCES Users(user_id) ON DELETE CASCADE
);

CREATE TABLE Junction (
    RXCUI1 VARCHAR(10) NOT NULL,
    RXCUI2 VARCHAR(10) NOT NULL,
    result_id INT NOT NULL,
    condition_meddra_id VARCHAR(8) NOT NULL,
    PRIMARY KEY (RXCUI1, RXCUI2, result_id),
    FOREIGN KEY (RXCUI1, RXCUI2, condition_meddra_id) REFERENCES
Interactions(RXCUI1, RXCUI2, condition_meddra_id),
    FOREIGN KEY (result_id) REFERENCES Results(result_id) ON DELETE CASCADE
);
```

Screenshot of local MySQL server



Screenshot of table row counts



Advanced Queries

Query 1:

This query is used to select top 15 interactions which have the highest PRR with their name, occurrences and interaction count so that it can serve as a recommendation/fast add feature the users could quickly check if what they are taking could be dangerous.

```
WITH eng_names AS (
    SELECT RXCUI, MIN(STR) AS STR
    FROM rxnconso
    WHERE LAT = 'ENG' AND STR IS NOT NULL AND STR != ''
    GROUP BY RXCUI
)
SELECT
    top.avg_prr,
    top.interaction_count,
    rc1.STR AS drug_1_name,
    rc2.STR AS drug_2_name
FROM (
    SELECT
        RXCUI1, RXCUI2,
        AVG(CAST(PRR AS DECIMAL(10,4))) AS avg_prr,
        COUNT(*) AS interaction_count
    FROM interactions
    WHERE PRR IS NOT NULL AND PRR != ''
    GROUP BY RXCUI1, RXCUI2
    HAVING interaction_count > 5
    ORDER BY avg_prr DESC
    LIMIT 15
) AS top
LEFT JOIN eng_names rc1 ON top.RXCUI1 = rc1.RXCUI
LEFT JOIN eng_names rc2 ON top.RXCUI2 = rc2.RXCUI;
```

Query Output:

```
mysql>
mysql> WITH eng_names AS (
    ->     SELECT RXCUI, MIN(STR) AS STR
    ->     FROM rxnconso
    ->     WHERE LAT = 'ENG' AND STR IS NOT NULL AND STR != ''
    ->     GROUP BY RXCUI
    -> )
    -> SELECT
    ->     top.avg_prr,
    ->     top.interaction_count,
    ->     rc1.STR AS drug_1_name,
    ->     rc2.STR AS drug_2_name
    -> FROM (
    ->     SELECT
    ->         RXCUI1, RXCUI2,
    ->         AVG(CAST(PRR AS DECIMAL(10,4))) AS avg_prr,
    ->         COUNT(*) AS interaction_count
    ->     FROM interactions
    ->     WHERE PRR IS NOT NULL AND PRR != ''
    ->     GROUP BY RXCUI1, RXCUI2
    ->     HAVING interaction_count > 5
    ->     ORDER BY avg_prr DESC
    ->     LIMIT 15
    -> ) AS top
    -> LEFT JOIN eng_names rc1 ON top.RXCUI1 = rc1.RXCUI
    -> LEFT JOIN eng_names rc2 ON top.RXCUI2 = rc2.RXCUI;
```

avg_prr	interaction_count	drug_1_name	drug_2_name
276.36740303	33	AHF	Antihaemophilic factor B
104.56830000	10	Clavulanate	Hexetidina
89.08163333	21	Aprotinin	Insulin Lispro
88.56912917	24	(2-butyl-4-chloro-1-{[2'-(1H-tetrazol-5-yl)biphenyl-4-yl]methyl}-1H-imidazol-5-yl)methanol	Protamine Sulfate
86.23003750	8	(-)-quinine	(-)-Cocaine
83.39526000	30	Aprotinin	Celecoxib
82.33999130	23	Aprotinin	Ioversol
81.12988065	31	Aprotinin	1-165-Erythropoietin (human clone λHEPOFL13 protein moiety), glycoform α
80.08973846	26	1-(4-Amino-6,7-dimethoxy-2-chinazolinyl)-4-(2,3-dihydro-1,4-benzodioxixin-2-ylcarbonyl)piperazin	Aprotinin
78.75000000	8	Ferrous Sulfate	4-Sulfanilamido-5,6-dimethoxypyrimidine
78.62067586	29	(±)-5-((4-(2-(5-ethyl-2-pyridinyl)ethoxy)phenyl)methyl)-2,4-thiazolidinedione	Aprotinin
77.22337647	17	acipimox	Frusemide
77.22221667	6	D-glucose	Aminoethylsulfonic acid
75.05457619	21	Ioversol	Protamine Sulfate
74.91500000	25	N,N,6-Trimethyl-2-(4-methylphenyl)imidazo(1,2-a)pyridine-3-acetamide	Aprotinin

```
15 rows in set (18.29 sec)
```

Query 1 Indexing:

By running the explain analyze function, here's the baseline performance:

```
| EXPLAIN |

| -> Nested loop left join  (cost=4.36e+6 rows=0) (actual time=19513..19513 rows=15 loops=1)
    -> Nested loop left join  (cost=4042 rows=0) (actual time=19513..19513 rows=15 loops=1)
        -> Table scan on top  (cost=2.5..2.5 rows=0) (actual time=15278..15278 rows=15 loops=1)
            -> Materialize  (cost=0..0 rows=0) (actual time=15278..15278 rows=15 loops=1)
                -> Limit: 15 row(s)  (actual time=15278..15278 rows=15 loops=1)
                    -> Sort: avg_prr DESC  (actual time=15278..15278 rows=15 loops=1)
                        -> Filter: (interaction_count > 5)  (actual time=0.576..15203 rows=210133 loops=1)
                            -> Stream results  (cost=11.5e+6 rows=431341) (actual time=0.573..15194 rows=211989 loops=1)
                                -> Group aggregate: avg(cast(interactions.PRR as decimal(10,4))), count(0)  (cost=11.5e+6 rows=431341) (actual time=0.557..15154 rows=211989 loops=1)
                                    -> Filter: ((interactions.PRR is not null) and (interactions.PRR <> ''))  (cost=4.2e+6 rows=31.6e+6) (actual time=0.504..9768 rows=42.9e+6 loops=1)
                                        -> Index scan on interactions using PRIMARY  (cost=4.2e+6 rows=39e+6) (actual time=0.486..7830 rows=42.9e+6 loops=1)
        -> Index lookup on rc1 using <auto_key0> (RXCUI = top.RXCUI1)  (cost=0.257..276 rows=1077) (actual time=282..282 rows=1 loops=15)
            -> Materialize CTE eng_names if needed  (cost=0..0 rows=0) (actual time=4235..4235 rows=401464 loops=1)
                -> Table scan on <temporary>  (actual time=3717..3762 rows=401464 loops=1)
                    -> Aggregate using temporary table  (actual time=3717..3717 rows=401463 loops=1)
                        -> Filter: ((rxnconso.LAT = 'ENG') and (rxnconso.STR <> ''))  (cost=129672 rows=107727) (actual time=0.405..358 rows=1.17e+6 loops=1)
                            -> Table scan on rxnconso  (cost=129672 rows=1.2e+6) (actual time=0.043..272 rows=1.17e+6 loops=1)
    -> Index lookup on rc2 using <auto_key0> (RXCUI = top.RXCUI2)  (cost=0.25..269 rows=1077) (actual time=0.00303..0.00316 rows=1 loops=15)
        -> Materialize CTE eng_names if needed (query plan printed elsewhere)  (cost=0..0 rows=0) (never executed)
 |

1 row in set (19.51 sec)
```

Try to improve performance indexing on interactions using:

```sql
CREATE INDEX idx_prr_rxcui ON interactions (PRR, RXCUI1,
RXCUI2);
```

In the actual result there's no significant impact on costs, but the actual running time increased significantly. Runtime for the query degraded from 19s to more than 2 minutes:

```
| EXPLAIN
| -> Nested loop left join  (cost=4.36e+6 rows=0) (actual time=19513..19513 rows=15 loops=1)
    -> Nested loop left join  (cost=4042 rows=0) (actual time=19513..19513 rows=15 loops=1)
        -> Table scan on top  (cost=2.5..2.5 rows=0) (actual time=15278..15278 rows=15 loops=1)
            -> Materialize  (cost=0..0 rows=0) (actual time=15278..15278 rows=15 loops=1)
                -> Limit: 15 row(s)  (actual time=15278..15278 rows=15 loops=1)
                    -> Sort: avg_prr DESC  (actual time=15278..15278 rows=15 loops=1)
                        -> Filter: (interaction_count > 5)  (actual time=0.576..15203 rows=210133 loops=1)
                            -> Stream results  (cost=11.5e+6 rows=431341) (actual time=0.573..15194 rows=211989 loops=1)
                                -> Group aggregate: avg(cast(interactions.PRR as decimal(10,4))), count(0)  (cost=11.5e+6 rows=431341) (actual time=0.557..15154 rows=211989 loops=1)
                                    -> Filter: ((interactions.PRR is not null) and (interactions.PRR <> ''))  (cost=4.2e+6 rows=31.6e+6) (actual time=0.504..9768 rows=42.9e+6 loops=1)
                                        -> Index scan on interactions using PRIMARY  (cost=4.2e+6 rows=39e+6) (actual time=0.486..7830 rows=42.9e+6 loops=1)
        -> Index lookup on rc1 using <auto_key0> (RXCUI = top.RXCUI1)  (cost=0.257..276 rows=1077) (actual time=282..282 rows=1 loops=15)
            -> Materialize CTE eng_names if needed  (cost=0..0 rows=0) (actual time=4235..4235 rows=401464 loops=1)
                -> Table scan on <temporary>  (actual time=3717..3762 rows=401464 loops=1)
                    -> Aggregate using temporary table  (actual time=3717..3717 rows=401463 loops=1)
                        -> Filter: ((rxnconso.LAT = 'ENG') and (rxnconso.STR <> ''))  (cost=129672 rows=107727) (actual time=0.405..358 rows=1.17e+6 loops=1)
                            -> Table scan on rxnconso  (cost=129672 rows=1.2e+6) (actual time=0.043..272 rows=1.17e+6 loops=1)
    -> Index lookup on rc2 using <auto_key0> (RXCUI = top.RXCUI2)  (cost=0.25..269 rows=1077) (actual time=0.00303..0.00316 rows=1 loops=15)
        -> Materialize CTE eng_names if needed (query plan printed elsewhere)  (cost=0..0 rows=0) (never executed)
 |
1 row in set (19.51 sec)
```

Try to improve performance indexing on rxnconso:

```sql
CREATE INDEX idx_rxnconso_lat_rxcui_str255 ON rxnconso (LAT,
RXCUI, STR(255));
```

The filter cost decreased from 128,701 to 82,546, but the cost of Aggregate (MIN STR) increased significantly(From 129,854 to 206,655). The time remains comparable:

```
| -> Nested loop left join  (cost=109e+6 rows=0) (actual time=19211..19211 rows=15 loops=1)
    -> Nested loop left join  (cost=20201 rows=0) (actual time=19211..19211 rows=15 loops=1)
        -> Table scan on top  (cost=2.5..2.5 rows=0) (actual time=15627..15627 rows=15 loops=1)
            -> Materialize  (cost=0..0 rows=0) (actual time=15627..15627 rows=15 loops=1)
                -> Limit: 15 row(s)  (actual time=15627..15627 rows=15 loops=1)
                    -> Sort: avg_prr DESC  (actual time=15627..15627 rows=15 loops=1)
                        -> Filter: (interaction_count > 5)  (actual time=0.223..15561 rows=210133 loops=1)
                            -> Stream results  (cost=11.5e+6 rows=431341) (actual time=0.221..15553 rows=211989 loops=1)
                                -> Group aggregate: avg(cast(interactions.PRR as decimal(10,4))), count(0)  (cost=11.5e+6 rows=431341) (actual time=0.21..15512 rows=211989 loops=1)
                                    -> Filter: ((interactions.PRR is not null) and (interactions.PRR <> ''))  (cost=4.2e+6 rows=31.6e+6) (actual time=0.172..10014 rows=42.9e+6 loops=1)
                                        -> Index scan on interactions using PRIMARY  (cost=4.2e+6 rows=39e+6) (actual time=0.162..8035 rows=42.9e+6 loops=1)
        -> Index lookup on rc1 using <auto_key0> (RXCUI = top.RXCUI1)  (cost=208697..210079 rows=5386) (actual time=239..239 rows=1 loops=15)
            -> Materialize CTE eng_names if needed  (cost=208697..208697 rows=1094) (actual time=3584..3584 rows=401464 loops=1)
                -> Group aggregate: min(rxnconso.STR)  (cost=208445 rows=1094) (actual time=0.138..3084 rows=401464 loops=1)
                    -> Filter: (rxnconso.STR <> '')  (cost=84335 rows=538636) (actual time=0.135..2891 rows=1.17e+6 loops=1)
                        -> Index lookup on rxnconso using idx_rxnconso_lat_rxcui_str255 (LAT = 'ENG')  (cost=84335 rows=598484) (actual time=0.134..2843 rows=1.17e+6 loops=1)
    -> Index lookup on rc2 using <auto_key0> (RXCUI = top.RXCUI2)  (cost=208697..210044 rows=5386) (actual time=0.00344..0.00358 rows=1 loops=15)
        -> Materialize CTE eng_names if needed (query plan printed elsewhere)  (cost=208697..208697 rows=1094) (never executed)
 |
1 row in set (19.22 sec)
```

Try to improve performance indexing on both:

```
mysql> explain analyze WITH eng_names AS (     SELECT RXCUI, MIN(STR) AS STR     FROM rxnconso     WHERE LAT = 'ENG' AND STR IS NOT NULL AND STR != ''     GROUP BY RXCUI ) SELECT     top.avg_prr,     top.interac
tion_count,     rc1.STR AS drug_1_name,     rc2.STR AS drug_2_name FROM (     SELECT         RXCUI1, RXCUI2,         AVG(CAST(PRR AS DECIMAL(10,4))) AS avg_prr,         COUNT(*) AS interaction_count     FROM int
eractions     WHERE PRR IS NOT NULL AND PRR != ''     GROUP BY RXCUI1, RXCUI2     HAVING interaction_count > 5     ORDER BY avg_prr DESC     LIMIT 15 ) AS top LEFT JOIN eng_names rc1 ON top.RXCUI1 = rc1.RXCUI LEF
T JOIN eng_names rc2 ON top.RXCUI2 = rc2.RXCUI;
+------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                           |
+------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------+
| -> Nested loop left join  (cost=109e+6 rows=0) (actual time=130431..130431 rows=15 loops=1)
    -> Nested loop left join  (cost=20201 rows=0) (actual time=130431..130431 rows=15 loops=1)
        -> Table scan on top  (cost=2.5..2.5 rows=0) (actual time=126676..126676 rows=15 loops=1)
            -> Materialize  (cost=0..0 rows=0) (actual time=126676..126676 rows=15 loops=1)
                -> Limit: 15 row(s)  (actual time=126676..126676 rows=15 loops=1)
                    -> Sort: avg_prr DESC  (actual time=126676..126676 rows=15 loops=1)
                        -> Filter: (interaction_count > 5)  (actual time=126569..126602 rows=210133 loops=1)
                            -> Table scan on <temporary>  (actual time=126569..126595 rows=211989 loops=1)
                                -> Aggregate using temporary table  (actual time=126569..126569 rows=211988 loops=1)
                                    -> Filter: ((interactions.PRR is not null) and (interactions.PRR <> ''))  (cost=4.44e+6 rows=19.5e+6) (actual time=0.112..8924 rows=42.9e+6 loops=1)
                                        -> Covering index range scan on interactions using idx_prr_rxcui over (NULL < PRR < '') OR ('' < PRR)  (cost=4.44e+6 rows=19.5e+6) (actual time=0.109..6694 rows=42.9e+6 loo
ps=1)
        -> Index lookup on rc1 using <auto_key0> (RXCUI = top.RXCUI1)  (cost=208697..210079 rows=5386) (actual time=250..250 rows=1 loops=15)
            -> Materialize CTE eng_names if needed  (cost=208697..208697 rows=1094) (actual time=3755..3755 rows=401464 loops=1)
                -> Group aggregate: min(rxnconso.STR)  (cost=208445 rows=1094) (actual time=0.118..3235 rows=401464 loops=1)
                    -> Filter: (rxnconso.STR <> '')  (cost=84335 rows=538636) (actual time=0.113..3034 rows=1.17e+6 loops=1)
                        -> Index lookup on rxnconso using idx_rxnconso_lat_rxcui_str255 (LAT = 'ENG')  (cost=84335 rows=598484) (actual time=0.112..2984 rows=1.17e+6 loops=1)
    -> Index lookup on rc2 using <auto_key0> (RXCUI = top.RXCUI2)  (cost=208697..210044 rows=5386) (actual time=0.00401..0.00416 rows=1 loops=15)
        -> Materialize CTE eng_names if needed (query plan printed elsewhere)  (cost=208697..208697 rows=1094) (never executed)
 |
+------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------+
1 row in set (2 min 10.44 sec)
```

I feel like in this case the biggest bottleneck is just having to iterate through the database, adding indexes in this case will give us more overhead and slow down performance overall.

Query 2:

Highlight substances that invoke the most number of conditions, serve as advice to avoid certain drugs.

```
SELECT
    rc.STR AS drug_name,
    c.RXCUI,
    c.condition_count
FROM (
    SELECT RXCUI1 AS RXCUI, COUNT(DISTINCT condition_concept_name) AS
condition_count
    FROM interactions
    WHERE condition_concept_name IS NOT NULL AND
condition_concept_name != ''
      AND RXCUI1 IS NOT NULL AND RXCUI1 != ''
    GROUP BY RXCUI1
    ORDER BY condition_count DESC
    LIMIT 15
```

```sql
) AS c
LEFT JOIN (
    SELECT RXCUI, MIN(STR) AS STR
    FROM rxnconso
    WHERE STR IS NOT NULL AND STR != ''
    GROUP BY RXCUI
) rc ON c.RXCUI = rc.RXCUI;
```

Query Output:

```
mysql> SELECT
    ->     rc.STR AS drug_name,
    ->     c.RXCUI,
    ->     c.condition_count
    -> FROM (
    ->     SELECT RXCUI1 AS RXCUI, COUNT(DISTINCT condition_concept_name) AS condition_count
    ->     FROM interactions
    ->     WHERE condition_concept_name IS NOT NULL AND condition_concept_name != ''
    ->       AND RXCUI1 IS NOT NULL AND RXCUI1 != ''
    ->     GROUP BY RXCUI1
    ->     ORDER BY condition_count DESC
    ->     LIMIT 15
    -> ) AS c
    -> LEFT JOIN (
    ->     SELECT RXCUI, MIN(STR) AS STR
    ->     FROM rxnconso
    ->     WHERE STR IS NOT NULL AND STR != ''
    ->     GROUP BY RXCUI
    -> ) rc ON c.RXCUI = rc.RXCUI;
```

drug_name	RXCUI	condition_count
2-Acetoxybenzenecarboxylic acid	1191	9198
4-(Acetylamino)phenol	161	8733
Frusemide	4603	8089
(RS)-3-ethyl 5-methyl 2-[(2-aminoethoxy)methyl]-4-(2-chlorophenyl)-6-methyl-1,4-dihydropyridine-3,5-dicarboxylate	17767	7933
1,2-Dehydrocortisone	8640	7585
OMEP	7646	7452
2,2-dimethylbutyric acid, 8-ester with (4R,6R)-6-(2-((1S,2S,6R,8S,8aR)-1,2,6,7,8,8a-hexahydro-8-hydroxy-2,6-dimethyl-1-naphthyl)ethyl)tetrahydro-4-hydroxy-2H-pyran-2-one	36567	7180
1,1-Dimethylbiguanide	6809	7156
Acide folique	4511	7097
HCTZ	5487	6991
Atorvastatin	83367	6890
(S)-1-(N(2)-(1-Carboxy-3-phenylpropyl)-L-lysyl)-L-proline	29046	6805
Pantoprazol	40790	6794
Albuterol	435	6773
4-Hydroxy-3-(3-oxo-1-phenylbutyl)coumarin	11289	6730

15 rows in set (28.82 sec)

Query 2 Indexing:

Baseline performance:

```
    ->     WHERE condition_concept_name IS NOT NULL AND condition_concept_name != ''
    ->       AND RXCUI1 IS NOT NULL AND RXCUI1 != ''
    ->     GROUP BY RXCUI1
    ->     ORDER BY condition_count DESC
    ->     LIMIT 15
    -> ) AS c
    -> LEFT JOIN (
    ->     SELECT RXCUI, MIN(STR) AS STR
    ->     FROM rxnconso
    ->     WHERE STR IS NOT NULL AND STR != ''
    ->     GROUP BY RXCUI
    -> ) rc ON c.RXCUI = rc.RXCUI;

+--------------------------------------------------------------------------------+
| EXPLAIN                                                                        |
+--------------------------------------------------------------------------------+
| -> Nested loop left join  (cost=40400 rows=0) (actual time=31730..31730 rows=15 loops=1)
    -> Table scan on c  (cost=2.5..2.5 rows=0) (actual time=27384..27384 rows=15 loops=1)
        -> Materialize  (cost=0..0 rows=0) (actual time=27384..27384 rows=15 loops=1)
            -> Limit: 15 row(s)  (actual time=27384..27384 rows=15 loops=1)
                -> Sort: condition_count DESC, limit input to 15 row(s) per chunk  (actual time=27384..27384 rows=15 loops=1)
                    -> Stream results  (cost=7.56e+6 rows=166342) (actual time=0.433..27384 rows=1715 loops=1)
                        -> Group aggregate: count(distinct interactions.condition_concept_name)  (cost=7.56e+6 rows=166342) (actual time=0.429..27383 rows=1715 loops=1)
                            -> Filter: ((interactions.condition_concept_name is not null) and (interactions.condition_concept_name <> '') and (interactions.RXCUI1 <> ''))  (cost=3.92e+6 rows=15.8e+6) (actual time=0.155..11637 rows=42.9e+6 loops=1)
                                -> Index range scan on interactions using PRIMARY over (RXCUI1 < '') OR ('' < RXCUI1)  (cost=3.92e+6 rows=19.5e+6) (actual time=0.145..8847 rows=42.9e+6 loops=1)
    -> Index lookup on rc using <auto_key0> (RXCUI = c.RXCUI)  (cost=0.257..2765 rows=10773) (actual time=290..290 rows=1 loops=15)
        -> Materialize  (cost=0..0 rows=0) (actual time=4346..4346 rows=402800 loops=1)
            -> Table scan on <temporary>  (actual time=3815..3863 rows=402800 loops=1)
                -> Aggregate using temporary table  (actual time=3815..3815 rows=402799 loops=1)
                    -> Filter: (rxnconso.STR <> '')  (cost=128919 rows=1.08e+6) (actual time=0.0316..317 rows=1.17e+6 loops=1)
                        -> Table scan on rxnconso  (cost=128919 rows=1.2e+6) (actual time=0.0312..264 rows=1.17e+6 loops=1)
 |
+--------------------------------------------------------------------------------+
1 row in set (31.73 sec)

mysql>
mysql>
```

Indexing for this one: I used a composite index on interactions, but the result does not change by much. In the explain analysis, we can see identical performance with the baseline, the filtered rows are comparable and the total cost is also comparable.

```sql
 CREATE INDEX idx_interactions_rxcui1_condition ON
interactions(RXCUI1, condition_concept_name);
```

```
mysql> explain analyze SELECT     rc.STR AS drug_name,     c.RXCUI,     c.condition_count FROM (     SELECT RXCUI1 AS RXCUI, COUNT(DISTINCT condition_concept_name) AS condition_count     FROM interactions     WHERE condition_concept_name IS NOT NULL AND condition_concept_name != ''       AND RXCUI1 IS NOT NULL AND RXCUI1 != ''     GROUP BY RXCUI1     ORDER BY condition_count DESC     LIMIT 15 ) AS c LEFT JOIN (     SELECT RXCUI, MIN(STR) AS STR     FROM rxnconso     WHERE STR IS NOT NULL AND STR != ''     GROUP BY RXCUI ) rc ON c.RXCUI = rc.RXCUI;

+--------------------------------------------------------------------------------+
| EXPLAIN                                                                        |
+--------------------------------------------------------------------------------+
| -> Nested loop left join  (cost=40400 rows=0) (actual time=31907..31907 rows=15 loops=1)
    -> Table scan on c  (cost=2.5..2.5 rows=0) (actual time=27539..27539 rows=15 loops=1)
        -> Materialize  (cost=0..0 rows=0) (actual time=27539..27539 rows=15 loops=1)
            -> Limit: 15 row(s)  (actual time=27539..27539 rows=15 loops=1)
                -> Sort: condition_count DESC, limit input to 15 row(s) per chunk  (actual time=27539..27539 rows=15 loops=1)
                    -> Stream results  (cost=7.56e+6 rows=166342) (actual time=0.57..27539 rows=1715 loops=1)
                        -> Group aggregate: count(distinct interactions.condition_concept_name)  (cost=7.56e+6 rows=166342) (actual time=0.566..27538 rows=1715 loops=1)
                            -> Filter: ((interactions.condition_concept_name is not null) and (interactions.condition_concept_name <> '') and (interactions.RXCUI1 <> ''))  (cost=3.92e+6 rows=15.8e+6) (actual time=0.193..11738 rows=42.9e+6 loops=1)
                                -> Index range scan on interactions using PRIMARY over (RXCUI1 < '') OR ('' < RXCUI1)  (cost=3.92e+6 rows=19.5e+6) (actual time=0.179..8933 rows=42.9e+6 loops=1)
    -> Index lookup on rc using <auto_key0> (RXCUI = c.RXCUI)  (cost=0.257..2765 rows=10773) (actual time=291..291 rows=1 loops=15)
        -> Materialize  (cost=0..0 rows=0) (actual time=4368..4368 rows=402800 loops=1)
            -> Table scan on <temporary>  (actual time=3837..3884 rows=402800 loops=1)
                -> Aggregate using temporary table  (actual time=3837..3837 rows=402799 loops=1)
                    -> Filter: (rxnconso.STR <> '')  (cost=129822 rows=1.08e+6) (actual time=0.125..316 rows=1.17e+6 loops=1)
                        -> Table scan on rxnconso  (cost=129822 rows=1.2e+6) (actual time=0.124..264 rows=1.17e+6 loops=1)
 |
+--------------------------------------------------------------------------------+
1 row in set (31.92 sec)
```

Covering index worked great in this case:

```sql
CREATE INDEX idx_covering ON interactions(RXCUI1,
condition_concept_name, PRR);
```

In the analysis, the total cost went from 7.56e+6 to 2.02e+6, rows filtered reduced from 15.8 million to 1.14 million. In terms of time, we got about a 3x performance boost.

```
mysql> explain analyze SELECT      rc.STR AS drug_name,      c.RXCUI,      c.condition_count FROM (      SELECT RXCUI1 AS RXCUI, COUNT(DISTINCT condition_concept_name) AS condition_count      FROM interactions      WHERE condition_concept_name IS NOT NULL AND condition_concept_name != ''        AND RXCUI1 IS NOT NULL AND RXCUI1 != ''      GROUP BY RXCUI1      ORDER BY condition_count DESC      LIMIT 15 ) AS c LEFT JOIN (      SELECT RXCUI, MIN(STR) AS STR      FROM rxnconso      WHERE STR IS NOT NULL AND STR != ''      GROUP BY RXCUI ) rc ON c.RXCUI = rc.RXCUI;
+------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                    |
+------------------------------------------------------------------------------------------------------------+
| -> Nested loop left join  (cost=40400 rows=0) (actual time=9390..9390 rows=15 loops=1)
    -> Table scan on c  (cost=2.5..2.5 rows=0) (actual time=5149..5149 rows=15 loops=1)
        -> Materialize  (cost=0..0 rows=0) (actual time=5149..5149 rows=15 loops=1)
            -> Limit: 15 row(s)  (actual time=5149..5149 rows=15 loops=1)
                -> Sort: condition_count DESC, limit input to 15 row(s) per chunk  (actual time=5149..5149 rows=15 loops=1)
                    -> Stream results  (cost=2.02e+6 rows=166342) (actual time=0.743..5149 rows=1715 loops=1)
                        -> Group aggregate: count(distinct interactions.condition_concept_name)  (cost=2.02e+6 rows=166342) (actual time=0.74..5149 rows=1715 loops=1)
                            -> Filter: ((interactions.condition_concept_name is not null) and (interactions.condition_concept_name <> '') and (interactions.RXCUI1 <> ''))  (cost=1.76e+6 rows=1.14e+6) (actual time=0.0727..4363 rows=1.88e+6 loops=1)
                                -> Covering index skip scan for deduplication on interactions using idx_covering over (RXCUI1 < '') OR ('' < RXCUI1)  (cost=1.76e+6 rows=1.14e+6) (actual time=0.0535..4228 rows=1.88e+6 loops=1)
    -> Index lookup on rc using <auto_key0> (RXCUI = c.RXCUI)  (cost=0.257..2765 rows=10773) (actual time=283..283 rows=1 loops=15)
        -> Materialize  (cost=0..0 rows=0) (actual time=4241..4241 rows=402800 loops=1)
            -> Table scan on <temporary>  (actual time=3723..3769 rows=402800 loops=1)
                -> Aggregate using temporary table  (actual time=3723..3723 rows=402799 loops=1)
                    -> Filter: (rxnconso.STR <> '')  (cost=129854 rows=1.08e+6) (actual time=0.0467..308 rows=1.17e+6 loops=1)
                        -> Table scan on rxnconso  (cost=129854 rows=1.2e+6) (actual time=0.046..257 rows=1.17e+6 loops=1)
 |
+------------------------------------------------------------------------------------------------------------+
1 row in set (9.39 sec)
```

From comparing the 2, apparently the covering index has a lower cost, so I'm gonna go with that.

I've also tried using both the first and second indexing method, but the result turns out to be similar to only applying indexing method 2.

```
mysql> explain analyze SELECT      rc.STR AS drug_name,      c.RXCUI,      c.condition_count FROM (      SELECT RXCUI1 AS RXCUI, COUNT(DISTINCT condition_concept_name) AS condition_count      FROM interactions      WHERE condition_concept_name IS NOT NULL AND condition_concept_name != ''        AND RXCUI1 IS NOT NULL AND RXCUI1 != ''      GROUP BY RXCUI1      ORDER BY condition_count DESC      LIMIT 15 ) AS c LEFT JOIN (      SELECT RXCUI, MIN(STR) AS STR      FROM rxnconso      WHERE STR IS NOT NULL AND STR != ''      GROUP BY RXCUI ) rc ON c.RXCUI = rc.RXCUI;
+------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                    |
+------------------------------------------------------------------------------------------------------------+
| -> Nested loop left join  (cost=40400 rows=0) (actual time=9450..9450 rows=15 loops=1)
    -> Table scan on c  (cost=2.5..2.5 rows=0) (actual time=5048..5048 rows=15 loops=1)
        -> Materialize  (cost=0..0 rows=0) (actual time=5048..5048 rows=15 loops=1)
            -> Limit: 15 row(s)  (actual time=5048..5048 rows=15 loops=1)
                -> Sort: condition_count DESC, limit input to 15 row(s) per chunk  (actual time=5048..5048 rows=15 loops=1)
                    -> Stream results  (cost=1.6e+6 rows=166342) (actual time=1.37..5047 rows=1715 loops=1)
                        -> Group aggregate: count(distinct interactions.condition_concept_name)  (cost=1.6e+6 rows=166342) (actual time=1.36..5047 rows=1715 loops=1)
                            -> Filter: ((interactions.condition_concept_name is not null) and (interactions.condition_concept_name <> '') and (interactions.RXCUI1 <> ''))  (cost=1.4e+6 rows=900455) (actual time=0.115..4260 rows=1.88e+6 loops=1)
                                -> Covering index skip scan for deduplication on interactions using idx_interactions_rxcui1_condition over (RXCUI1 < '') OR ('' < RXCUI1)  (cost=1.4e+6 rows=900455) (actual time=0.0789..4127 rows=1.88e+6 loops=1)
    -> Index lookup on rc using <auto_key0> (RXCUI = c.RXCUI)  (cost=0.257..2765 rows=10773) (actual time=293..293 rows=1 loops=15)
        -> Materialize  (cost=0..0 rows=0) (actual time=4402..4402 rows=402800 loops=1)
            -> Table scan on <temporary>  (actual time=3866..3914 rows=402800 loops=1)
                -> Aggregate using temporary table  (actual time=3866..3866 rows=402799 loops=1)
                    -> Filter: (rxnconso.STR <> '')  (cost=129854 rows=1.08e+6) (actual time=0.0548..322 rows=1.17e+6 loops=1)
                        -> Table scan on rxnconso  (cost=129854 rows=1.2e+6) (actual time=0.0541..268 rows=1.17e+6 loops=1)
 |
+------------------------------------------------------------------------------------------------------------+
1 row in set (9.45 sec)
```

Query 3:

List every single drug a user is taking given a userId. Traditionally this would be simple, but drugs are stored in both the RXCUI1 and RXCUI2 columns, so in order to get the drugs from both of them you must perform a set union operation. This query is necessary to find out all of the drugs that a user is taking. This query performs multiple joins to get the user_id's from the junction table and set unions RXCUI1 and RXCUI2 to get all of the drugs. It also performs a subquery to union the columns and then perform a WHERE to extract the user_id.

```
SELECT DISTINCT RXCUI FROM
(SELECT RXCUI1 AS RXCUI, result_id, condition_meddra_id
FROM Junction
UNION
SELECT RXCUI2 AS RXCUI, result_id, condition_meddra_id
FROM Junction) AS union_table
JOIN Results ON union_table.result_id = Results.result_id
WHERE user_id = 1;
```

Query output:

```
mysql> SELECT DISTINCT RXCUI FROM
    -> (SELECT RXCUI1 AS RXCUI, result_id, condition_meddra_id
    -> FROM Junction
    -> UNION
    -> SELECT RXCUI2 AS RXCUI, result_id, condition_meddra_id
    -> FROM Junction) AS union_table
    -> JOIN Results ON union_table.result_id = Results.result_id
    -> WHERE user_id = 1
    -> LIMIT 15;
+---------+
| RXCUI   |
+---------+
| 1000492 |
| 1827    |
| 1895    |
| 2626    |
| 3109    |
| 342369  |
| 41144   |
| 5489    |
| 854974  |
| 8745    |
| 8785    |
| 10582   |
| 448     |
| 27169   |
| 2002    |
+---------+
15 rows in set (0.00 sec)
```

Query 3 Indexing:

This is the benchmark without any indexes:

```
-> Table scan on <temporary>  (cost=7.42..10.1 rows=24) (actual time=0.209..0.214 rows=16 loops=1)
    -> Temporary table with deduplication  (cost=7.3..7.3 rows=24) (actual time=0.208..0.208 rows=16 loops=1)
        -> Nested loop inner join  (cost=4.9 rows=24) (actual time=0.138..0.184 rows=16 loops=1)
            -> Filter: (results.user_id = 1)  (cost=1.75 rows=1.5) (actual time=0.0441..0.0553 rows=15 loops=1)
                -> Table scan on Results  (cost=1.75 rows=15) (actual time=0.0417..0.0493 rows=15 loops=1)
            -> Index lookup on union_table using <auto_key1> (result_id=results.result_id)  (cost=4.02..4.33 rows=2) (actual time=0.00726..0.00788 rows=1.07 loops=15)
                -> Union materialize with deduplication  (cost=3.7..3.7 rows=16) (actual time=0.0857..0.0857 rows=16 loops=1)
                    -> Covering index scan on Junction using RXCUI1  (cost=1.05 rows=8) (actual time=0.016..0.0241 rows=8 loops=1)
                    -> Covering index scan on Junction using RXCUI1  (cost=1.05 rows=8) (actual time=0.0104..0.0147 rows=8 loops=1)
```

Optimization idea 1: Indexing based on user_id. This should speed things up as it will make it easier for the database to extract all of the different drug relations given a specific user index.

```
CREATE INDEX user_id ON Results(user_id);
```

```
-> Table scan on <temporary>  (cost=13.8..16.3 rows=16) (actual time=0.115..0.117 rows=16 loops=1)
    -> Temporary table with deduplication  (cost=13.6..13.6 rows=16) (actual time=0.115..0.115 rows=16 loops=1)
        -> Nested loop inner join  (cost=12 rows=16) (actual time=0.0662..0.0872 rows=16 loops=1)
            -> Table scan on union_table  (cost=3.87..6.4 rows=16) (actual time=0.0558..0.0576 rows=16 loops=1)
                -> Union materialize with deduplication  (cost=3.7..3.7 rows=16) (actual time=0.0545..0.0545 rows=16 loops=1)
                    -> Covering index scan on Junction using RXCUI1  (cost=1.05 rows=8) (actual time=0.0205..0.0236 rows=8 loops=1)
                    -> Covering index scan on Junction using RXCUI1  (cost=1.05 rows=8) (actual time=0.00433..0.00534 rows=8 loops=1)
            -> Limit: 1 row(s)  (cost=0.256 rows=1) (actual time=0.00162..0.00164 rows=1 loops=16)
                -> Filter: (results.user_id = 1)  (cost=0.256 rows=1) (actual time=0.00152..0.00152 rows=1 loops=16)
                    -> Single-row index lookup on Results using PRIMARY (result_id=union_table.result_id)  (cost=0.256 rows=1) (actual time=0.00134..0.00134 rows=1 loops=16)
```

This one index alone had a drastic impact on the performance of the query. It ran in about 55% of the time taken to run it without an index, implying a 45% reduction. This makes sense as the query searches are made easier given a specific user_id because of the index.

Optimization idea 2: Indexing based on a composite index of (user_id, result_id). This might speed things up because I'm also joining based on result_id and having a composite index of both might make it easier for the database to join tables and identify users.

```
CREATE INDEX composite_index ON Results(user_id, result_id);
```

```
-> Table scan on <temporary>  (cost=13.8..16.3 rows=16) (actual time=0.0871..0.0885 rows=16 loops=1)
    -> Temporary table with deduplication  (cost=13.6..13.6 rows=16) (actual time=0.0867..0.0867 rows=16 loops=1)
        -> Nested loop inner join  (cost=12 rows=16) (actual time=0.0567..0.0775 rows=16 loops=1)
            -> Table scan on union_table  (cost=3.87..6.4 rows=16) (actual time=0.047..0.0489 rows=16 loops=1)
                -> Union materialize with deduplication  (cost=3.7..3.7 rows=16) (actual time=0.046..0.046 rows=16 loops=1)
                    -> Covering index scan on Junction using RXCUI1  (cost=1.05 rows=8) (actual time=0.0125..0.0149 rows=8 loops=1)
                    -> Covering index scan on Junction using RXCUI1  (cost=1.05 rows=8) (actual time=0.0136..0.0148 rows=8 loops=1)
            -> Limit: 1 row(s)  (cost=0.256 rows=1) (actual time=0.00156..0.00159 rows=1 loops=16)
                -> Filter: (results.user_id = 1)  (cost=0.256 rows=1) (actual time=0.00144..0.00144 rows=1 loops=16)
                    -> Single-row index lookup on Results using PRIMARY (result_id=union_table.result_id)  (cost=0.256 rows=1) (actual time=0.00128..0.00128 rows=1 loops=16)
```

This query was drastically faster than even the first index I came up with. The queries given this index run in 38% of the time, implying a 62% reduction in time. This is likely due to the fact that the composite index makes searching during the join stage faster as well.

Optimization idea 3: Indexing based on only result_id. This might cause a reduction as it indexes based on the results and this should make the result_id join even faster.

```
CREATE INDEX result_index ON Results(result_id);
```

```
-> Table scan on <temporary>  (cost=7.42..10.1 rows=24) (actual time=0.101..0.102 rows=16 loops=1)
    -> Temporary table with deduplication  (cost=7.3..7.3 rows=24) (actual time=0.1..0.1 rows=16 loops=1)
        -> Nested loop inner join  (cost=4.9 rows=24) (actual time=0.0667..0.0805 rows=16 loops=1)
            -> Filter: (results.user_id = 1)  (cost=1.75 rows=1.5) (actual time=0.0176..0.0212 rows=15 loops=1)
                -> Table scan on Results  (cost=1.75 rows=15) (actual time=0.0164..0.0188 rows=15 loops=1)
            -> Index lookup on union_table using <auto_key1> (result_id=results.result_id)  (cost=4.02..4.33 rows=2) (actual time=0.00349..0.00368 rows=1.07 loops=15)
                -> Union materialize with deduplication  (cost=3.7..3.7 rows=16) (actual time=0.0448..0.0448 rows=16 loops=1)
                    -> Covering index scan on Junction using RXCUI1  (cost=1.05 rows=8) (actual time=0.0103..0.0129 rows=8 loops=1)
                    -> Covering index scan on Junction using RXCUI1  (cost=1.05 rows=8) (actual time=0.00513..0.00629 rows=8 loops=1)
```

This query was faster than the user_id index but it was slower than the composite index. It ran in 49% of the time resulting in a 51% improvement which is similar to the index based on just user_id but slightly faster. This can likely be attributed to the faster joins.

Query 4:

Given a certain drug, we want to list what drugs interact the most with it. Each user is taking multiple drugs, so we want to show which other drugs have the most interactions for each one. As an example I am using the drug with RXCUI 1001. Similar to the previous query, the RXCUI can be in either column RXCUI1 or RXCUI2 so it requires a set operation to check both columns. Along with that we are using COUNT(*) aggregation to count the number of interactions.

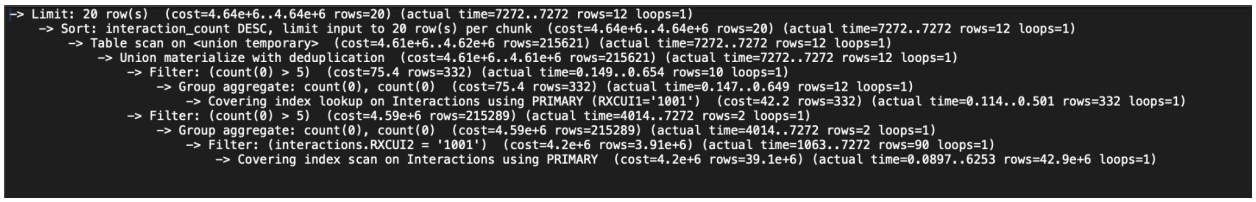
```
(SELECT RXCUI2, COUNT(*) AS interaction_count
FROM Interactions
WHERE RXCUI1 = '1001'
GROUP BY RXCUI2)
UNION
SELECT RXCUI1, COUNT(*) AS interaction_count
FROM Interactions
WHERE RXCUI2 = '1001'
GROUP BY RXCUI1
ORDER BY interaction_count DESC
LIMIT 15;
```

Query output:

```
+--------+-------------------+
| RXCUI2 | interaction_count |
+--------+-------------------+
| 161    |                73 |
| 1399   |                56 |
| 435    |                41 |
| 5640   |                40 |
| 41126  |                34 |
| 1191   |                32 |
| 10582  |                29 |
| 4124   |                24 |
| 723    |                24 |
| 88249  |                24 |
| 5489   |                22 |
| 39993  |                21 |
| F00104 |                 1 |
| H01102 |                 1 |
+--------+-------------------+
14 rows in set (1 min 53.78 sec)
```

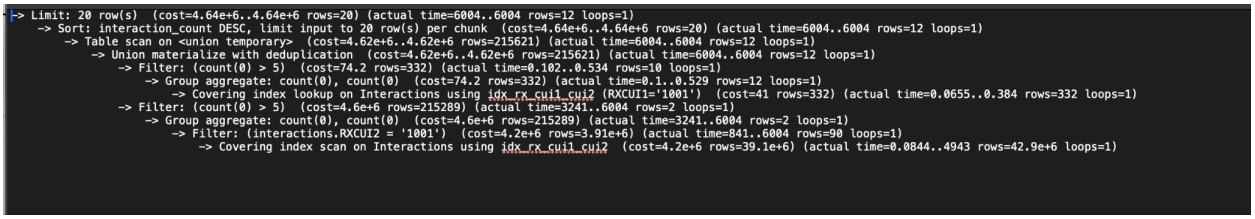
Query 4 Indexing

Here is a benchmark without any indexes:



Optimization idea 1: Indexing based on a composite key based off of a composite index on both rxcui1 and rxcui2. This might make it easier to find individual rxcui and their interactions. There are a lot of pairs where RXCUI1 and RXCUI2 are the same so this might help.

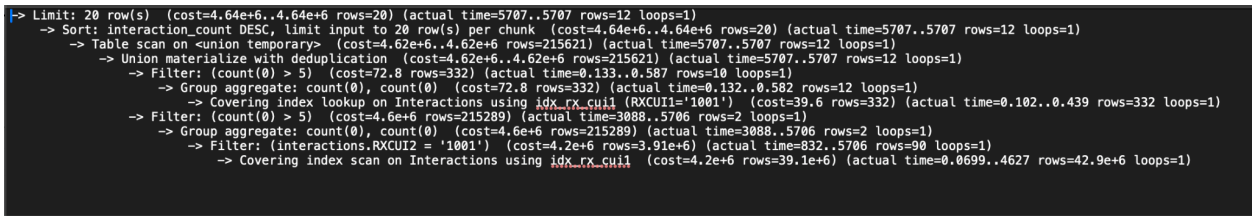
```
CREATE INDEX composite_index ON Interactions(RXCUI1, RXCUI2);
```



This does help a little bit. It runs 82.5% of the time resulting in a 17.5% reduction. The reason that the reduction is small is because there are simply far too many (RXCUI1, RXCUI2) pairs for this to be useful.

Optimization Idea 2: Indexing based on indexing just the RXCUI1 row. Indexing just one row should make things faster as it will make it easier to search for given just one RXCUI value. This issue is that the drug can show up on both RXCUI1 and RXCUI2 so it's not a perfect optimization but the index is cheaper.

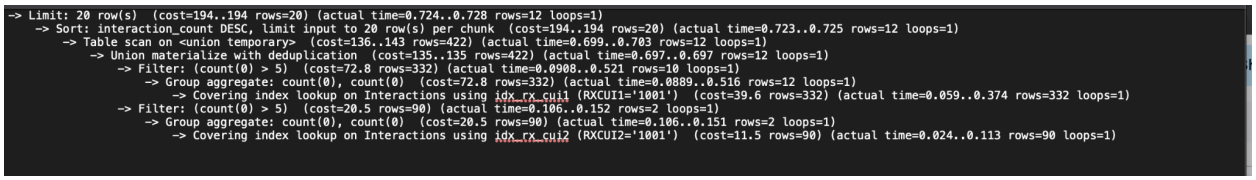
```
CREATE INDEX single_index ON Interactions(RXCUI1);
```



As we can see this did have a slightly better time reduction. This ran in 78.4% of the time meaning that it had a 21.5% reduction.

Optimization Idea 3: This RXCUI value can show up in both RXCUI1 and RXCUI2 so it makes sense to index both columns. Though generating this index will be more expensive, it will make each subsequent search much faster.

```
CREATE INDEX index_1 ON Interactions(RXCUI1);
CREATE INDEX index_2 ON Interactions(RXCUI2);
```



99990044% reduction. This makes sense as the RXCUI can appear in both indexes and two indexes should make it much faster.