SQL Query Optimizer

Pooja Bhandary and Jennifer Ma

Problem Statement

- Suggest relatively optimal alternatives for queries that are detected to have antipatterns.

Addressing Common Antipatterns

- Use published antipatterns.
- Write a rule-based identifier for the most common antipatterns.
- Map patterns to a list of possible solutions.
    - Solutions: efficient queries, guidelines on how the query should be

Progress

- Created solutions for 12 antipatterns:
    1) Jaywalker
    2) Keyless Entry
    3) ID Required
    4) Multivalued Attributes
    5) Fear of the Unknown
    6) Phantom Files
    7) Metadata Tribbles
    8) Random Selection
    9) Pattern Matching
    10) Rounding Errors
    11) Select Star
    12) Value in Definition

Progress

- Tested the scalability of the tool with real world queries
    - Manually checked most of the real world queries (~100+ per antipattern)
    - Grouped similar queries together
    - Coded solutions for each group

Revised Goals

75% goal/B grade:

- [Achieved] The program addresses 5 antipatterns.
- [Achieved] Some of the queries suggested may not work.
- [Achieved] We plan to test out the suggestions with at least 3 different real world queries for each antipattern.

100%/A grade:

- [Achieved] The program addresses at least 10 antipatterns and improves query processing speed.
- [Achieved] Most of the queries suggested are able to be processed.
- [Achieved] We plan to test out the suggestions with at least 5 different real world queries for each antipattern.

Revised Goals

- 125%/Wow:
    - The program addresses at least 15 antipatterns and improves query processing speed.
    - [Achieved] Most of the queries suggested are able to be processed.
    - [Achieved] Test each antipattern with 5 real world queries.
    - Incorporate a feedback mechanism from the user to determine if the suggestion actually helped and applied to them.

Testing Correctness

- Tested with ranking team's queries and solutions

Example: Metadata Tribbles

The ranking team's incorrect query for Metadata Tribbles is:

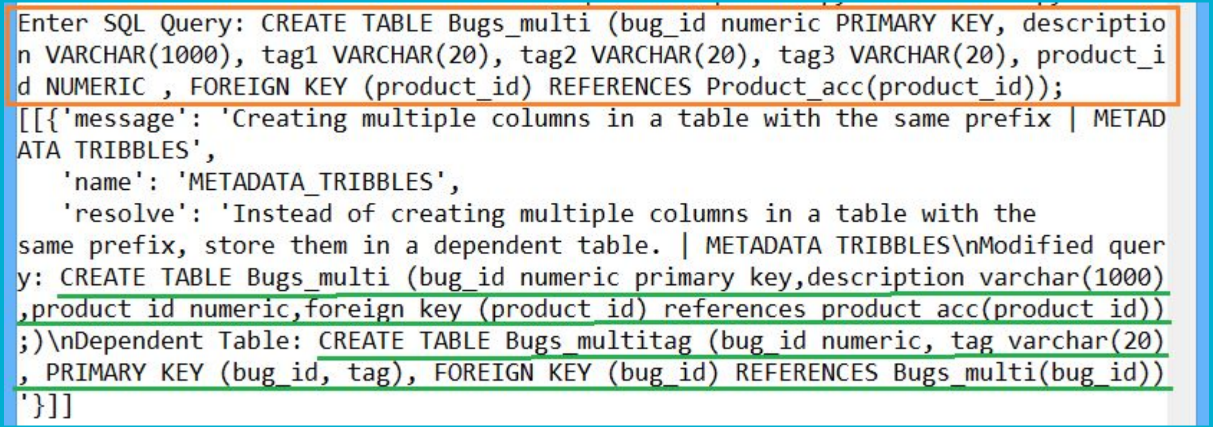
```
CREATE TABLE Bugs_multi (
    bug_id numeric PRIMARY KEY,
    description VARCHAR(1000),
    tag1 VARCHAR(20),
    tag2 VARCHAR(20),
    tag3 VARCHAR(20),
    product_id NUMERIC,
    FOREIGN KEY (product_id) REFERENCES Product_acc(product_id) );
```

Example: Metadata Tribbles

And their query for creating the dependent table is:

```
CREATE TABLE Tags (
    bug_id BIGINT NOT NULL,
    tag VARCHAR(20),
    PRIMARY KEY (bug_id, tag),
    FOREIGN KEY (bug_id) REFERENCES Bugs_multi(bug_id));
```

Rewriter's output:

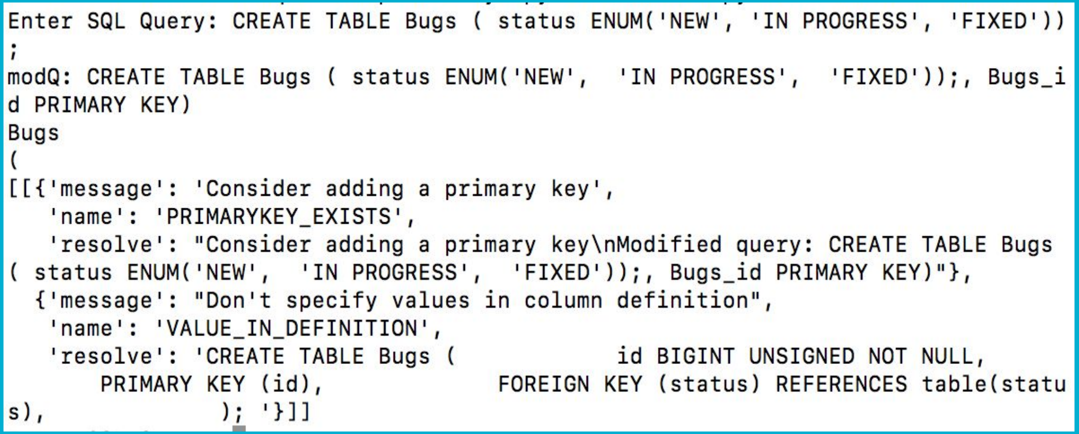


```
Enter SQL Query: CREATE TABLE Bugs_multi (bug_id numeric PRIMARY KEY, descriptio
n VARCHAR(1000), tag1 VARCHAR(20), tag2 VARCHAR(20), tag3 VARCHAR(20), product_i
d NUMERIC , FOREIGN KEY (product_id) REFERENCES Product_acc(product_id));
[[{'message': 'Creating multiple columns in a table with the same prefix | METAD
ATA TRIBBLES',
   'name': 'METADATA_TRIBBLES',
   'resolve': 'Instead of creating multiple columns in a table with the
same prefix, store them in a dependent table. | METADATA TRIBBLES\nModified quer
y: CREATE TABLE Bugs_multi (bug_id numeric primary key,description varchar(1000)
,product_id numeric,foreign key (product_id) references product_acc(product_id))
;)\nDependent Table: CREATE TABLE Bugs_multitag (bug_id numeric, tag varchar(20)
, PRIMARY KEY (bug_id, tag), FOREIGN KEY (bug_id) REFERENCES Bugs_multi(bug_id))
'}]]
```

Example: Value in Definition

CREATE TABLE Bugs (-- other columns, status ENUM('NEW', 'IN PROGRESS', 'FIXED');

Rewriters output:



```
Enter SQL Query: CREATE TABLE Bugs ( status ENUM('NEW', 'IN PROGRESS', 'FIXED'))
;
modQ: CREATE TABLE Bugs ( status ENUM('NEW',  'IN PROGRESS',  'FIXED'));, Bugs_i
d PRIMARY KEY)
Bugs
(
[[{'message': 'Consider adding a primary key',
   'name': 'PRIMARYKEY_EXISTS',
   'resolve': "Consider adding a primary key\nModified query: CREATE TABLE Bugs
( status ENUM('NEW',  'IN PROGRESS',  'FIXED'));, Bugs_id PRIMARY KEY)"},
  {'message': "Don't specify values in column definition",
   'name': 'VALUE_IN_DEFINITION',
   'resolve': 'CREATE TABLE Bugs (            id BIGINT UNSIGNED NOT NULL,
       PRIMARY KEY (id),             FOREIGN KEY (status) REFERENCES table(statu
s),             ); '}]]
```

Experimental Results

- Manually looked at most queries
- Grouped similar ones
- Coded solution for each group

Example: Fear of the Unknown

Real query:

- SELECT stat FROM sqlite_stat1 WHERE tbl= ? || '_rowid'

Modified version:

- SELECT stat FROM sqlite_stat1 WHERE tbl= COALESCE(?, ' ') || '_rowid'

Real query:

- SELECT x FROM t1 WHERE x LIKE ('ab' || 'c%') ORDER BY 1;

Modified version:

Here, the strings on either side of the '||' are string literals, and not columns, so we should not surround them with COALESCE.

- SELECT x FROM t1 WHERE x LIKE ('ab' || 'c%') ORDER BY 1;

Example: JayWalker

Real Query:

```
select alert_id, criteria from alerts where criteria not like "%speaker:%" and criteria like "%,%" and confirmed and not deleted');
```

Modified Query:

CREATE a intersection table with an id field and field criteria and set a foreign key with the table alerts.

Issues

1) Generalizing the template for each antipattern.

2) SQLParse Limitations

3) The suggestion tightly coupled with test queries.

4) The required context(eg: Schema) might not always be available.

Issues

1) Jaywalking:

```
SELECT x FROM t1 WHERE y LIKE ';%'
```

While the solution to this query would still be a new table with a referential integrity constraint, the pattern detection and template string generation would be different

Issues

2) Limitations of SQLParse - in the interest of flexibility, it would be worthwhile to develop a custom parser.

3) Enforcing rules might not scale well for variations in syntax and styles of writing queries.

Future Work

- Chain solutions for queries violating multiple antipatterns
- Cover last few antipatterns
- Cover edge case queries