DATA ANALYTICS USING DEEP LEARNING

GT 8803 // FALL 2018 // VENKATA KISHORE PATCHA & VARSHA ACHAR

RANKING DATABASE SCHEMA SMELLS

CREATING THE NEXT®

TODAY'S AGENDA

- Background
- Existing work
- Objectives
- Approach
- Experiment
- Resources

BACKGROUND

- Developers are not experts in database design and database administrators know little about application functionality.
- If recommended practices are not followed, database antipatterns are introduced.
- "Smelly Relations: Measuring and Understanding Database Schema Quality"
- How do we identify schema antipattern?
- What difference in performance can an antipattern cause, compared to 'good' alternate schemas?
- How do we rank these antipatterns?

EXISTING WORK

- Open source tool: DbDeo
- Identifies 9 types of schema smells:
  Compound Attribute, Adjacency list, Metadata as data, Multicolumn attribute, Clone tables, Values in attribute definition, Index abuse, God table, and Overloaded attribute names.
- DbDeo has a meta-model generator component that uses SQLParse to parse SQL statements and make a meta-model.

EXISTING WORK

- Details about antipatterns (including how to identify an antipattern and a respective solution) is found in the book by Bill Karwin, "*SQL Antipatterns: Avoiding the Pitfalls of Database Programming*".

http://www.r-5.org/files/books/computers/languages/sql/style/Bill_Karwin-SQL_Antipatterns-EN.pdf

OBJECTIVES

✓ Performance tables for 10 Antipatterns (75% goals)

✓ Rank antipatterns based on the DML statements in a project (based on existence of antipattern and sql usage from downloaded GitHub projects) (100% goals)

APPROACH

- Synthetic data load for each antipattern and the solution.
- Write queries that perform specific tasks for the good and bad schema.
- Execute queries and record the time taken for each query to execute (milliseconds).
- Rank

APPROACH

Classification of queries to experiment with

- **SELECT** a meaningful set of records. Example: get 5 customer details.
- Aggregation operation such as **COUNT** that visits every row in the table, making the complexity O(number of rows). Example: How many cars are there with each color in the cars table?
- **UPDATE** operation: Update some values in the table in a meaningful way. Example: update first name.
- **JOIN**: (if applicable to the particular antipattern) Perform a join operation between at least 2 tables where at least 1 table's schema has the antipattern. Example: How many customers have black car?

APPROACH

Developed a framework for easy execution and addition of a new antipattern using factory design pattern.

usage: measureMain.py [-h] [-d] [-l] [-e] Antipattern_name

optional arguments:

-h, --help show this help message and exit

-d Data will be generated for Given Antipattern.

-l Load Generated Data.

-e Run experiments.

APPROACH

Command line argument Antipattern_name is mapped with dedicated antipattern implementation through a mapping object (python dictionary).

We get the performance tables based on Antipattern_name argument.

APPROACH

```
class AntiPattern(object):

    def __init__(self):
        self.rows = 100000

    def data_gen(self):
        raise NotImplementedError()

    def load_data(self, antipattern_name):
    #Single implementation for all antipatterns

    def run_queries(self, antipattern_name):
    #Single implementation for all antipatterns
```

EXPERIMENT

- For each antipattern
  - For each Solution
    - try each relevant DML query (SELECT, JOIN, AGGREGATION, UPDATE).
      - Note the execution time of the antipattern and solution

LIST OF ANTIPATTERNS

- Jaywalking
- 31 Flavors
- ID Required
- Naive Trees
- Entity-Attribute Value
- Multicolumn
- Index Overuse
- Index Underuse
- Metadata Tribbles
- Keyless Entry

PERFORMANCE TABLE

Antipattern 1: Jaywalking

Number of records: 100,000

	SELECT	AGGREGATION	UPDATE	JOIN
Bad design	98 ms	115 ms	0 ms	440652 ms
Good design	7 ms	30 ms	0 ms	1982 ms

PERFORMANCE TABLE

Antipattern 2: 31 Flavors
Number of records: 1 million.
*Join operation is not applicable for this antipattern (it does not impact performance). This antipattern only demonstrates the ramifications of ENUM.

	SELECT	AGGREGATION	UPDATE	JOIN
Bad design	332 ms	672 ms	31678 ms	0 ms*
Good design	0 ms	178 ms	15891 ms	0 ms*

PERFORMANCE TABLE

Antipattern 3: Naive Trees

Number of records: 100,000

Baseline is better than expected since Postgres has inbuilt recursion support. (when the text book was written, only a few DBMS's had recursion support)

*Join operation does not impact performance.

	SELECT	AGGREGATION	UPDATE	JOIN
Bad design	4 ms	23 ms	2 ms	0 ms*
Good design 1 (closure table)	3 ms	6 ms	1 ms	0 ms*
Good design 2 (nested set)	2 ms	212 ms	1 ms	0 ms*
Good design 3 (path enumeration)	1 ms	672 ms	1 ms	0 ms*

PERFORMANCE TABLES

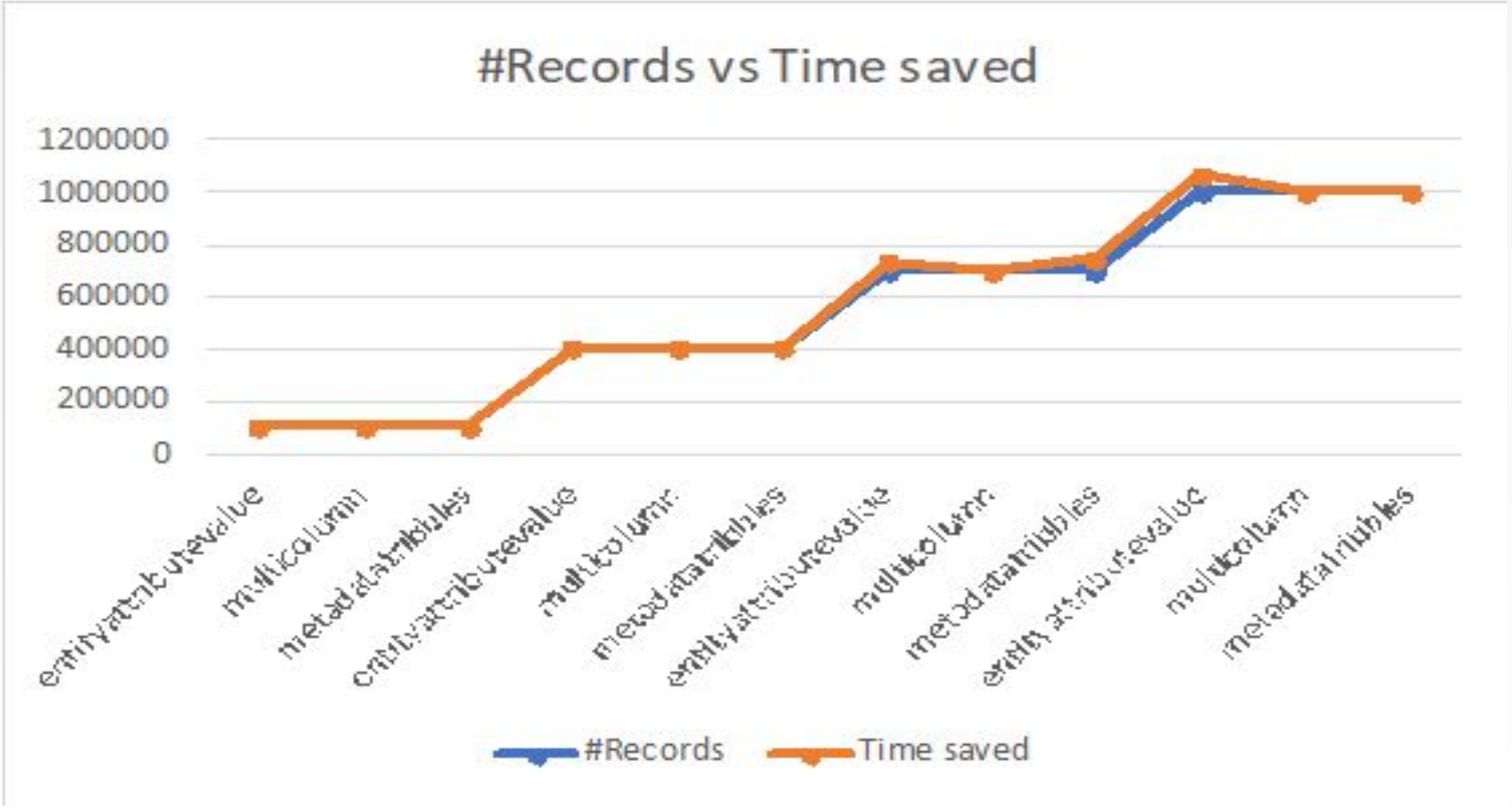
Rest of the numbers can be found here

PERFORMANCE TABLES

Time saved (best solution - baseline)

ANTIPATTERN	SELECT	AGGREGATE	UPDATE	JOIN	TOTAL
Jaywalking	91 ms	85 ms	0 ms	438670 ms	438846 ms
31 Flavors	332 ms	494 ms	15787 ms	-	11613 ms
Multicolumn	1 ms	4738 ms	1 ms	- 4 ms	4736 ms
Index Underuse	304 ms	59 ms	1 ms	92 ms	556 ms
Metadata Tribbles	302 ms	305 ms	-70 ms	- 128 ms	454 ms
ID Required	1 ms	61 ms	280 ms	31 ms	373 ms
Keyless Entry	-1 ms	110 ms	15 ms	-8 ms	116 ms
Index Overuse	3 ms	9 ms	2 ms	40 ms	54 ms
Naive Trees	1 ms	17 ms	1 ms	-	19 ms
Entity Attribute Value	-1 ms	11 ms	3 ms	5 ms	18 ms

Time saved Vs # records



WHAT'S DONE

Dynamic ranking based on existence of an antipattern and queries/usage from GitHub repositories.

Dynamic ranking steps

1. From our measurements, we made a measurement table which tells the user how much time they could save with best good design. The measurement table has five columns Antipattern, Select, Aggregation, update and join. Each cell is a time in milliseconds.
2. Create a count table with the same dimensions as measurement table. We need to update the count table for every occurrence of antipattern and type of usage (select, aggregation, update and join). We have a two-pass approach. Both passes will go through each of sql statement in a project.
   a. The first pass writes the required table and column details to a file (metadata details).
   b. The second pass will check the usage of tables and columns that have anti-pattern. Second pass updates counts table accordingly.
3. Dot multiply measurement table with count table. Sum each row. That will give us two column table (antipattern and summed time). Sort result table in descending order of time. Display order of antipatterns and respective time, that user can save, to the user.

why two passes?

select x, y from table T1 where y like '%2,%' group by y;

Above query can be classified in single pass. It's a **Jaywalking Aggregation.**

why two passes?

How about 31 flavours antipattern?

create type s as ENUM('NEW', 'IN PROGRESS', 'FIXED');

CREATE TABLE baseline_bugs (

id SERIAL PRIMARY KEY,

status s

);

When we parse create statement, we know antipattern exists but we don't know where to count in our count table? (which classification?)

Pass one detects and gets required metadata information.

Pass two uses metadata and counts.

Pass one metadata

Antipattern name	Table name with AP	Column name with AP
Jaywalking	Needed	Needed
Naive Tree	Needed	Two column names that form the tree. Needed
Id required	Needed	Needed
Entity-Attribute-Value	Needed	Needed (two column names)
Multicolumn Attributes	Needed	List of all tag columns
Metadata Tribbles	List of table names with the same name (postfixed with some number)	No need
Rounding Errors	Needed	Needed
31 Flavors	Needed	Needed

SQL parse

```
parsed = sqlparse.parse('select *
from foo')[0]
>>> parsed.tokens
[<DML 'select' at 0x7f22c5e15368>,
<Whitespace ' ' at 0x7f22c5e153b0>,
<Wildcard '*' … ]
>>>
```

SQL parse

Returned tokens can be analyzed with the help of:

`get_alias()`

`is_child_of(`*other*`)`

`get_parent_name()`

`get_real_name()`

What more can be done

1. Select statements with Subqueries are not tested. Most Likely needs more coding.
2. Counts can be tested on large set of projects and publish observations.
3. We don’t know user data loads. If data loads are available, that can be used as a factor for ranking.

RESOURCES

- Randomly populated database
- Postgres server
- DbDeo
- Linux server with 500GB disk
- Python

QUESTIONS
or
COMMENTS?


Happy Holidays