**Implementing an auto-indexing algorithm within an extension on PostgreSQL’s platform.**

**CS 5513 - Advanced Database Management - Spring 2018**

**YouTube Presentation: https://youtu.be/h5YKgJgQ9mw**

**Authors:**

Charlie Liu - charlie.liu-1@ou.edu

Dragon Tran - dragon.tran@ou.edu

Emily Black - emily@ou.edu

Charles Johnson - charles.t.johnson@ou.edu

**Abstract**

This report summarizes our work for the course ‘CS 5513 - Advanced Database Management’. The topic we chose to pursue is implementing an auto-indexing algorithm on top of an already existing open source platform, like PostgreSQL. This required us to use multiple open source libraries (HypoPG, pg\_query) and an open source extension called Dexter, written by Andrew Kane, which utilized these open source libraries. Our algorithm followed Mujiba Zaman’s algorithm very closely, with a few changes, including the removal of one of two thresholds. Zaman’s algorithm was originally written back in 2004 on Microsoft SQL’s platform, but still utilizes relevant technology to date - that technology being Artificial Intelligence, specifically clustering. Utilizing clustering lets us intelligently change our indices given a continuous workload. Our main goal is to let Dexter do the heavy lifting, implement our algorithm on top of Dexter, compare the results of what Dexter returns and of what Zaman’s algorithm returns, and send those new indices off to PostgreSQL. We mainly decided to go this way, not only because it would be less debugging with Dexter, but that we thought it would be more beneficial to have multiple algorithms running and comparing their results, which would result in greater accuracy of the new indices chosen. To further summarize our work and findings, this paper is broken up into five sections: **Introduction**, **Related Work**, **Proposed Work &** **Results** and **Conclusions & Future Work**.

**Section 1: Introduction**

Our project, our objectives and its significance will be discussed in this section of the paper. The idea behind our project is to implement an auto-indexing algorithm on PostgreSQL’s platform. Auto-indexing alone implies some sort intelligence behind the actual algorithm itself. It implies that an algorithm, given a continuous flow of work (queries), will intelligently re-calculate based on the current workload, and previous workloads before that. This is an intelligent way to go about the Index Selection Problem (ISP). The ISP is essentially the problem of finding the best set of indices given a database. The goal of the ISP “*is to choose a subset of given indexes to be created in a database, so that the response time for a given database workload is minimized.*” [1] This problem sounds incredibly easy to solve on the surface, but it actually turns out this is one of the biggest and hardest NP problems that modern day database administrators (DBA) still face. The algorithm we will be using will be a branch off of Mujiba Zaman’s algorithm, defined in her thesis work, and cited below.

Our primary objectives for this project include: implement and adapt Zaman’s algorithm as to not copy her work directly; modifying Dexter, a pre-existing auto-indexing tool, to load our python implementation; comparing results from Dexter and our algorithm and finding the most optimal set of new indices; increasing overall performance of the database given our new indices.

Before getting into the specifics of our algorithm, it should be pointed out as to why our algorithm and research topic is deemed unbelievably helpful. Our algorithm not only lessens the daily workload of an Database Administrator, but helps improve the overall performance of the database itself. “*As the pace of generating data gets faster and faster and the volume of data processed becomes bigger and bigger, the index technique has been playing a more critical role than ever before in order to support query processing and optimization.*” [4] Having an auto-indexing algorithm that utilizes modern day artificial intelligence techniques to continuously re-generate a set of new indices given workload changes makes a huge impact on a DBA’s job, and lets the DBA focus on other tasks at hand.

This is just a general high level summary of what this project is about, and why it’s important. For the rest of the report, we will discuss topics including: Related Work (Section 2), Proposed Work & Results (Section 3) and Conclusions & Future Work (Section 4).

**Section 2: Related Work**

**HypoPlans[6]**

HypoPlans is a non-intrusive and completely autonomous approach to make relational database systems able to execute self-tuning actions. It is based on the notion of hypothetical query execution plans. Almeida, Brayner, Monteiro, Lifschitz and Oliveira[7] classify databases as (*i*) continuous or non-continuous; (*ii*) autonomous or non-autonomous; and (*iii*) intrusive or non-intrusive. Using these classifications HypoPlans is presented as a continuous, autonomous, non-intrusive approach.

HypoPlans operates in three phases, observation, prediction, reaction. In the observation phase, HypoPlans monitors the workload of the database to identify candidate structures for physical design tuning. In the prediction phase, HypoPlans attempts to predict the outcome of physical design execution plan in which real and hypothetical structures are used. The reaction phase is where the autonomous feature plays its part. In this phase HypoPlans has the ability to implement the new hypothetical execution plan.

This autonomous behavior of HypoPlans has two major benefits. 1) It allows a DBMS to run more efficiently by adjusting to workflow as it arises, instead of waiting on a DBA to tune the DBMS. 2) in some instances, it is not beneficial to a company or an end user to have a DBA to maintain, or optimize a database. Given an autonomous solution eliminates this issue.

The difference in our work and HypoPlans is the cost model used to determine beneficial indexing. HypoPlans uses a canonical cost model, which is not as precise as using the DBMS internal cost model. Since our process takes an intersection of real indexes, using an imprecise cost model is unrealistic

**COLT[7]**

COLT, short for Continuous On-line Tuning, is a framework that supports the on-line materialization of index structures[8]. This framework is unique in the sense that not only does it continually analyze and index the database to handle differing workloads, it also maintains it overhead cost by increasing or decreasing the amount of work it does according to the DBMS workflow.

Colt can be broken down into three main components, the EQO (extended Query optimizer), the profiler, and the SO (self organizer). The EQO is an extension of the query optimizer that’s allows for what-if calculations. the what-if calculations are the reduction in query execution time given a certain index if it were materialized. It is interesting to note that most commercial DBMS have this functionality built in, making it easily accessible. The profiler gathers statistics about hypothetical indices. These statistics are then sent to the self organizer to determine if a hypothetical index is beneficial or “hot”. The self organizer is the reorganization component. It analyzes the current hypothetical indices to determine if they will be more or less beneficial. It forms a group of these beneficial indices to be analyzed by the EQO in the next phase.

While COLT is a novel idea, it is geared more towards a shifting workload environment. Our project is considered an off-line tuning paradigm, where our database is analyzed before it is live. COLT does have additional associated overhead costs due to monitoring the query distribution, and selecting indices. This additional cost does not outweigh the benefit for an off-line tuning paradigm.

**AISIO[8]**

In Pedrozo & Gomes Vaz’s paper, they tested the different performances achieved using an external tool versus an database-integrated tool. Pedrozo & Gomes Vaz hypothesised that “better results can be obtained by letting the database management system optimizer (DBMS) Optimizer itself internally manage the index selection”, and they implemented the Automatic Index Selection Integrated into Optimizer (AISIO) tool to test this. The key difference between external and integrated tool lies in how indexes are selected: external tools use its own metrics when selecting the indexes while integrated tools allow the DBMS to select the index.

The AISIO tool operates in 4 steps after detecting a change in workload: statistic collection, candidate indexes selection, generating hypothetical index configurations, and exhaustive enumeration of all the candidate indexes. During the last step, AISIO communicate with the DBMS optimizer continuously to passing statistics and hypothetical indexes for each query to the DBMS optimizer, and allowed the DBMS to select the best configuration.

Pedrozo & Gomes Vaz compared the performance between no-indexes, indexes created with an external toolkit “DBT-2” and indexes created by AISIO. For no-indexes, the transaction per minute noticeably decreased as the increase in database warehouses made the database larger and more complex, which is expected since each retrieval must travel the database sequentially. For both DBT-2 and AISIO, the transactions per minute are both much higher than no-indexes thanks to indexes enabling non-sequential access, with AISIO achieving slightly better results than DBT-2. This proves their hypothesis because a comparable performance was observed using a internal tool without the extra cost often associated with external tools.

**Semi-Automatic Index Tuning[9]**

Schnaitter & Polyzotis proposed a semi-automatic index selection system using an extended work function algorithm (WFA) which they called WFIT. WFA is an existing online algorithm, able to achieve high level of performance on its recommendations continuously and automatically by recording edge to edge cost as queries traversed the indexes and recommending the lowest cost path. WFIT extends on WFA by allowing database administrator(DBA) to provide feedback, positive or negative, to the algorithm, which is treated as an additional metric alongside the cost when selecting the candidate keys.

In their comparison study, the WFIT algorithm is compared with another online tuning algorithm referred to as “BC” (based on the works of Bruno and Chaudhuri) and “OPT” which has full knowledge of the workload and provide the baseline for best case scenarios. They first tested the WFA algorithm by providing no feedbacks, and found it performing better than the BC algorithm, and within 10% of OPT’s results. They also observed a larger overhead as it takes 300ms on average to analyze each query and create recommendation, but deemed this acceptable considering the higher savings in query execution cost with the right indexes. The WFIT algorithm is then tested again, but with DBA feedbacks on the recommendations, and the achieved results even closer to OPT, showing that DBA feedbacks can still improve the quality of recommendations significantly. However, Schnaitter & Polyzotis did remark on the effect of delayed feedback: if a DBA feedback is not immediately provided after each query, it will lead to an overall degradation in performance.

**Section 3: Proposed Work & Results**

**3.1: Past Attempts & Obstacles**

Before we jump into the specifics of our proposed work, we would like to point out our past attempts at finding a solution for this project and the obstacles we faced along the way. Originally, in our Progress Report 1, we detailed how we were going to implement two auto-indexing algorithms within PostgreSQL’s source files. The algorithms would consist of one clustering algorithm and one non-clustering algorithm. About half a month to a month into tediously looking through thousands upon thousands of PostgreSQL source files, we realized PostgreSQL actually didn’t automatically index, and that in fact we would need to create or use an extension to connect to PostgreSQL to automatically index. Out of all the roadblocks we hit along the way why developing a solution for our project, this was hands down the hardest one.

In Progress Report 2, we aimed at creating our own extension on PostgreSQL in order to implement an auto-indexing method on top of PostgreSQL. We had originally planned to structure it similarly to Dexter, but later found that this was a task far greater than we were expecting, mainly because Dexter uses multiple external libraries we’ve never heard about, and also because Dexter was written completely in another language none of us have ever coded in before - Ruby.

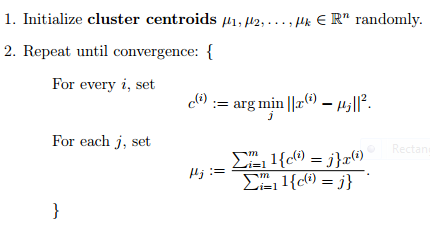
With these two roadblocks on either side of our group, we finally decided we would implement our algorithm on top of Dexter, which would then run our algorithm on top of PostgreSQL. While writing our algorithm, we found it incredibly hard to understand some of Dexter’s methods and variables, given that there was a lack of documentation. We focused more on inserting our algorithm within Dexter, than actually changing code within the Dexter framework, as to make things more difficult later in the development process.

The main obstacle we currently still face is the learning curve of Ruby. While it is a scripting language like Python, the syntax between Python and Ruby are a world apart. It’s also worth pointing out that Dexter alone was extremely difficult to set up, as it required various Ruby libraries installed to even get the Gem file to build.

**3.2: Proposed Work**

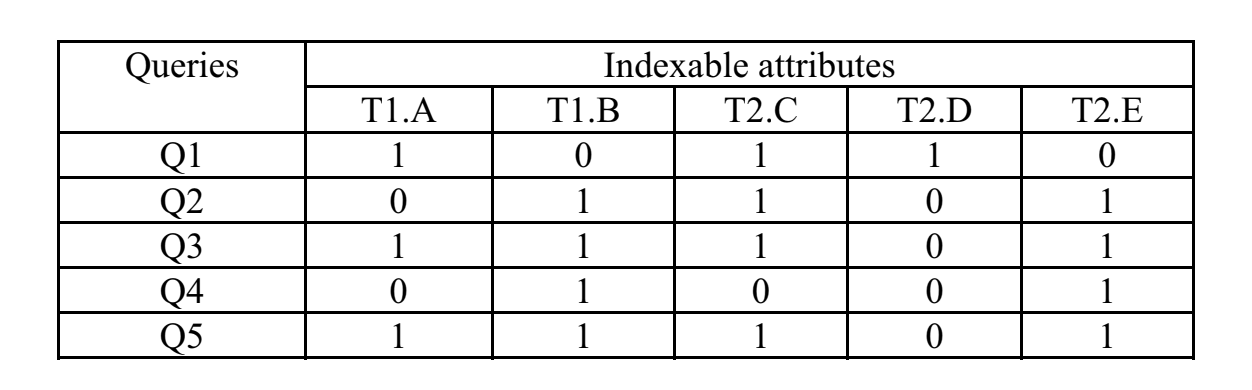
**3.2.1: Our Algorithm**

The algorithm we specifically used, which is derived from Mujiba Zaman’s algorithm, utilizes a common algorithm in found in data mining - K-means. K-means is classified as an unsupervised machine learning algorithm, but is also labeled as a clustering algorithm. “*K-means stores K centroids that it uses to define clusters. A point is considered to be in a particular cluster if it is closer to that cluster centroid than any other centroid.*” [2] Calculations for K-mean distances are done in an Euclidean space. Rough pseudo code for the algorithm is shown in **Figure 1**:

 [2]

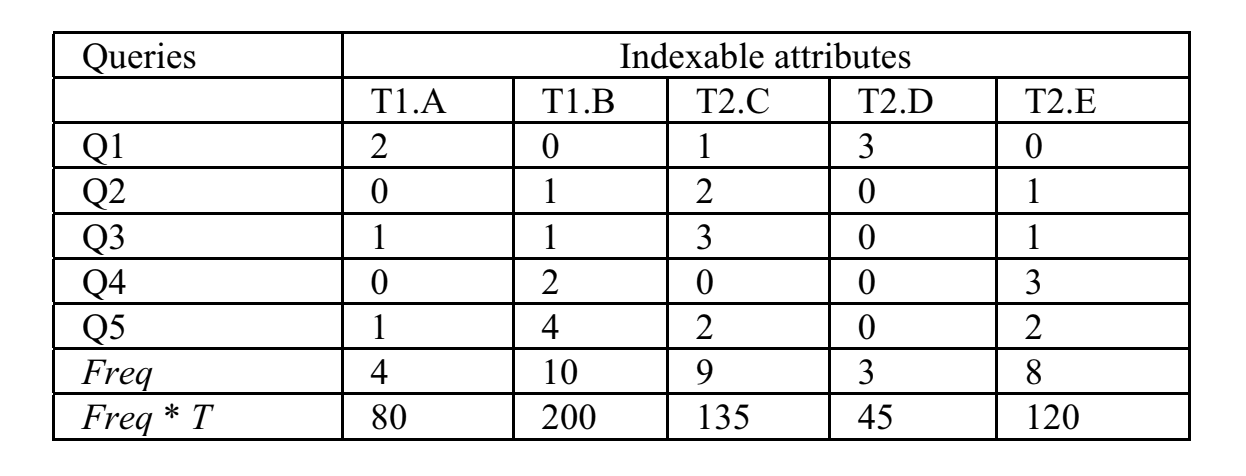
**Figure 1. K-means**

To get to the point of clustering data though, we need to pass K-means a dataset. This is done in our algorithm by implementation of both a Query-Attribute Matrix and a Query-Frequency Matrix. For each query, and for each attribute in a given database, a Query-Attribute Matrix displays either a 1 or 0, representing if an attribute is present in a query or not. An example of this is shown below in **Figure 2**. “*Let columns A and B belong to a table named T1 having 20 rows and C, D and E belong to a table named T2 having 15 rows*“ [3]:

[3]

**Figure 2. Query-Attribute Matrix**

With a given Query-Attribute Matrix, we are then able to traverse through the queries again based on attributes present and obtain exact numbers for their frequencies in the queries. This is called a Query-Frequency Matrix, and is used primarily for computing frequencies of all attributes in all queries. Given the example above, we continue with the data in the Query-Attribute Matrix by computing the data for the Query-Frequency Matrix, given in **Figure 3**:

[3]

**Figure 3. Query-Frequency Matrix**

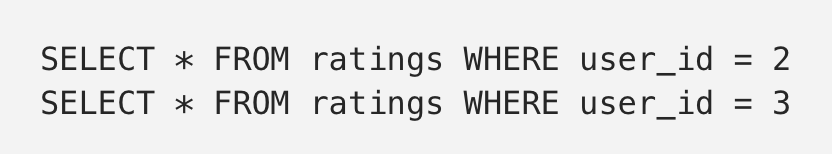
The Query-Frequency Matrix has two extra rows: Freq and Freq \* T. Freq is equal to the sum of that attribute across all queries. Freq \* T is equal to the sum of that attribute across all queries multiplied by the number of rows in the table where the attribute comes from. These two values let us filter out candidate indexable attributes, or attributes that are worthy to move onto the clustering phase. Before we can filter them out though, we must define two thresholds to compare them to. This is where our algorithm and Zaman’s algorithm differs. Zaman let threshold one be equal to the workload size / 2, and let threshold two be equal to the workload size / 4 (four was derived from multiple tests better described in Zaman’s thesis). For our algorithm, because we are using both Dexter’s results and our results to filter out the new indices, we decided that only having one threshold (workload size divided by two) would suffice for our needs, thus leaving the Freq \* T row unneeded.

Once we’ve populated both the Query-Attribute Matrix and the Query-Frequency Matrix, computed our thresholds and filtered out certain rows within our Query-Frequency Matrix, we then get to the final step of Zaman’s algorithm, which involves clustering the Query-Attribute Matrix, finding the most frequent attributes and queries across all clusters, then passing those new indices to the optimizer, and picking out which ones are most cost efficient. For our algorithm, we had Dexter, specifically indexer.rb call our algorithm file, zaman.py, which would return a new set of indices to be implemented. While Dexter utilizes both PostgreSQL’s optimizer and does cost calculation, the way we went about implementing Zaman’s algorithm does neither, though can be done through future work. The main feature that we wanted to emphasize on with our algorithm was clustering.

**3.2.2: Dexter’s Algorithm**

To give a general idea of the order of operations, Dexter runs its own algorithm, then calls our algorithm (written in Python because of incredible AI libraries), and lastly computes the intersect of both results, and gives priority to indices that are found in our algorithm’s results, and not Dexter’s results. It then passes that to another algorithm in Dexter that logs that there’s an index found in table X that should be indexed and displays a message to the user that that index was found.

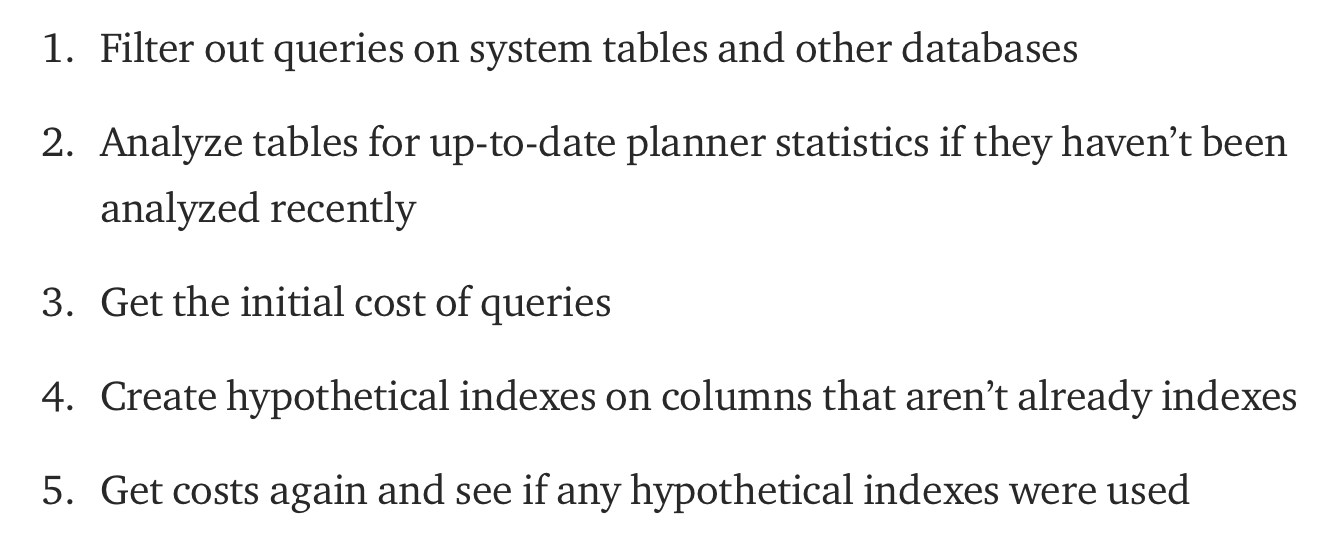
To further explain how Dexter works, it can be broken down into two phases: collection and generation. Collection starts off by utilizing an open source library, pg\_query, which filters through logs files, and parsing out the query itself and it’s duration to run. It then “*uses fingerprinting to group queries. Queries with the same parse tree but different values are grouped together.*” [5] Below in **Figure 4** is an example of two clearly different queries that have the same fingerprint:

 [5]

**Figure 4: Different Queries with Same Fingerprint**

Each fingerprint is then summed together by the total execution time. Dexter then sets a threshold for total execution time, so as to not over-index. The current threshold we used, and the threshold that Dexter defaults comes with is: new\_cost < query.initial\_cost \* savings\_ratio where new\_cost is the cost of the current query against the hypothetical index created, the initial\_cost is the initial cost of the query alone before the hypothetical index, and the savings\_ratio is 1 - @min\_cost\_savings\_pct / 100.0, where the minimum cost savings percent is determined by what the user sets it as.

From this point, we then move on to phase 2 - generation. After collecting the queries to speed up, Dexter then creates hypothetical indices via the open source library, HypoPG. “*Hypothetical indexes show how a query’s execution plan would change if an actual index existed. They take virtually no time to create, don’t require any disk space, and are only visible to the current session.*” [5] The main steps Dexter’s algorithm follows are listed below in **Figure 5**:

 [5]

**Figure 5: Steps of Dexter’s Algorithm**

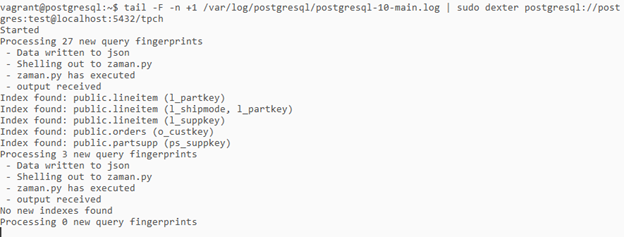
What makes Dexter so powerful though is that it utilizes PostgreSQL’s query planner to figure out only the BEST indices for a given query. In the end, “*Hypothetical indexes that were used AND significantly reduced cost are selected to be indexes.*” [5]

**3.2.3: Measuring performance**

When creating new indexes, we would like to evaluate its performance and see if the created indexes improve read / select query performance. Self-designing a sample database with arbitrary data and sample queries that is comprehensively tests the database while simulating transactions with varying workloads is challenge and will not result in reliable or valid metrics to use for measuring performance. The TPC, also known as the Transaction Processing Performance Council is an organization that defines transaction processing and database benchmarking standards and offers a range of different benchmarks for different database applications. We considered the TPC-C benchmark that is designed to simulate the workloads typically processed by a wholesale supplier application. [10] The metric that TPC-C uses is a measure of transaction per minute or business throughput called the tmp-C. The benchmark consists of 9 tables along with sample data and several queries that simulates a wide range of complex types of business activities performed on the database. [10] Although known as an accepted benchmarking standard we did not choose this benchmark. TPC-H is a decision support benchmark that consists of business-oriented ad-hoc queries and concurrent data modifications.[11] It models the typical transactions of a product supply business that consists of 8 tables. The benchmark includes a data generator that supplies the data to populate the 8 table. The generator uses a constant called the scale factor which is used to determine the size of the data generated for the database. The benchmark consists of 22 queries that are complex and simulates business decision-making questions along with 2 update procedures/refresh functions.[11] There are 2 test that are performed: the load test which measures time to insert the data into the database and the performance test which consists of 2 subtests that measure raw query execution of the system and the ability to process the most queries in the least amount of time. [11] The TPC-H benchmark uses only one metric called the composite query-per-hour performance metric which evenly distributes the weights of the metrics collected from the tests. Ultimately, we decided on TPC-H since it is known as the industry standard and is used by many database vendors and researchers. Also, since we are interested in measuring the read performance gains from our indexes, we believe that performing the power test (to measure query execution) will be sufficient in measuring the performance of this project.

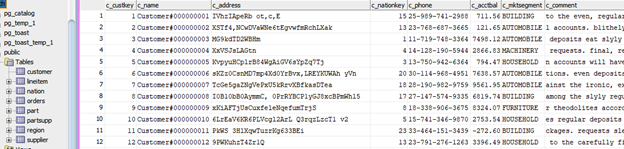
**3.2.4 Results**

We modified Dexter’s source code in Ruby to execute our Python script when the select\_index method is called. We have tested our implementation on PostgreSQL 10 using the TPC-H benchmark that uses 1 GB of generated data along with 22 benchmark queries. The Postgres instance was hosted on a Vagrant Virtualbox running Ubuntu 17.10, with its specifications as 1 GB RAM and 1 Virtual Core (Unable to retrieve clock speed from Virtualbox).



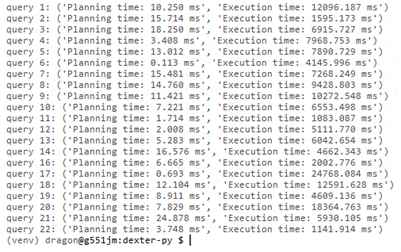
**Figure 6: Our Modified Dexter Implementation Execution Output**

In **Figure 7**, we show a screenshot of indices being suggested by the modified Dexter running our algorithm implementation. Dexter provides real time index analysis by streaming the Postgres log file to it. We used tail to retrieve the tail end of the log file and used | to pipe/direct the output it into Dexter. By default, it analyzes queries parsed from the Postgres log file and suggests indices but Dexter can also be specified to create the indices.

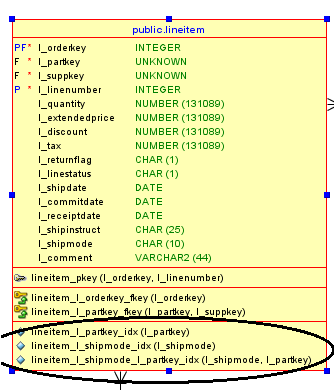


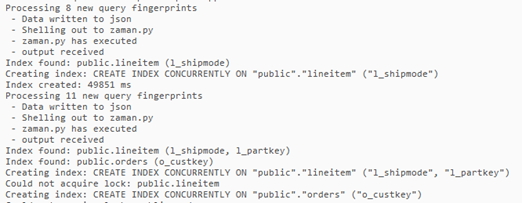
**Figure 7: Screenshot of Tables Used for Benchmark**

In **Figure 8**, we show a screenshot of data from one of the tables (list of tables on the left). As mentioned before, 1 GB of data was generated using the DBGEN tool that included in TPC-H. The data populated was randomly generated.

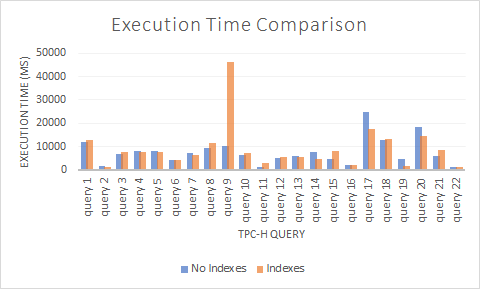


**Figure 8: Benchmark Driver Output**

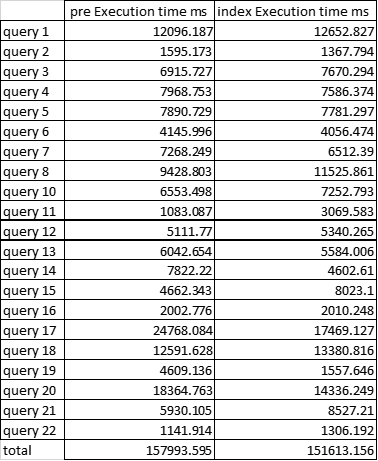
In **Figure 9/10**, we show a screenshot of our Python benchmark driver script performing the power test with our 22 benchmark queries. Outputs are planning time to execute the query and the total execution time to retrieve the data for the query. These metrics are obtained directly from Postgres by using EXPLAIN ANALYZE which runs an explain on the query and actually executes the query as well. 

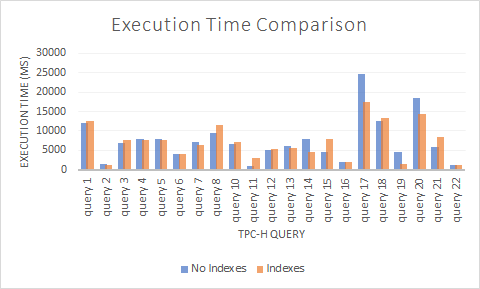
**Figure 9, 10: Indexes Creation Output and Verification in SQL Developer**

The benchmarking experiments results were nothing short of unexpected. We were expecting to receive overall better execution / retrieval times with the newly created indexes. The total execution time for running the 22 benchmark queries without any indexes is 158733.143  
 ms = 158 seconds. The execution time with the created indexes was unexpectedly at 197747.914 ms = 197 seconds. Depicted below is a comparison of the execution times of each benchmark query for query executions with no indexes and query executions with the indexes created by our implementation.



**Figure 11: Execution Time Before and After Indexing**

From our experiments, it seems that the implemented indexes did improve the execution times for certain queries (2,4,5,6,etc) but other for queries (notably query 9) the performance decreased and execution time took much longer. We are not certain on what whether is was an intended outcome by our implementation or a bottleneck in our VM. If we considered query 9 as an outlier and removed it, the total execution time for the benchmark queries with the created indexes is at 151.6 seconds compared to the total execution time for the non indexed database at 157.9 seconds, which is approximately a 4% decrease in total query execution / data retrieval times. We also observe that some of the queries with the highest execution times (17 & 20) actually improved noticeably, showing a 32.5% decrease for query 17 and a 28.3% decrease for query 20.



**Figure 12, 13: Execution Time Before and After Indexing, Without Query 9**

With this, we actually see a slight improvement in execution times with indexes created by our implementation. Other than query 9, the execution times can be further improved if optimal indexes for for queries 3,11, and 21 can be suggested through a database administrator or an update to our current implementation.

From these results, we believe that our implementation can be beneficial and effective at automatically selecting indexes, which was tested in Postgres given the TPC-H data set. Despite the indexed benchmark resulting in a longer execution time, we see that a query’s bad performance is enough to offset the improvements and benefits from indexes on the other tables/attributes. We also observed that several benchmark queries’ execution time improved and even decreased the overall total execution time the benchmark. Additionally, another observation was that our implementation created indexes that improved the longest executed benchmarks.

We believe we have addressed (some not fully accomplished) the objectives we listed. We implemented and adapted Zaman’s algorithm in python. We accomplished locating specific Dexter file(s) related to auto-indexing and hooking up our algorithm in the appropriate place. We accomplished to write the code to compute the intersect the between both Dexter’s results and our results, and return the suggested new indices, with priority to our algorithm. In our experiments, we didn’t outright get an improved total execution time with our created indexes. However we believe that, since that some of the longest executing queries were improved with the created indexes, we addressed this objective to some degree, although not completely.

**Section 4: Conclusions & Future Work**

In conclusion, this project has been an excellent learning experience. Throughout the course of the semester, we gained understanding of auto indexing and data mining through clustering by actively working with postgreSQL, modify and using its many open source extensions. The final result, and especially with query 9, was unexpected, but through this we did learn another aspect of auto indexing: for a given workload, not every individual query will receive benefit equally from the creation of a set of indexes. Overall, I believe our implementation can succeed in bringing a benefit to database performance by automatically creating indexes based on the workload. Additionally, our implementation also shows how an existing postgreSQL extension written in RUBY can be extended and modified with different algorithms implemented in python to change what indexes are created in a postgreSQL database.

As far as future work goes, there are tons of ways to expand past this project, but I’ll state a few examples of how to expand on just our algorithm. The first way that you could expand on our algorithm is either changing or bringing in more thresholds, and comparing them to more frequency calculations. We only used one threshold in our algorithm, which was workload size / 2, but you could either change the two to another metric, or just expand outward and create more thresholds that candidate indexable attributes must meet. Another way that you could expand on our algorithm is by using alternatives to K-means, like DBSCAN or HDBSCAN. Both algorithms are included in the python sklearn clustering suite, so it would be very simple to implement in our algorithm. Both DBSCAN and HDBSCAN offer benefits like non-globular clustering for more correct clustering, and are both designed to tackle large datasets. Lastly, our algorithm can be improved by continuing work on top of what we’ve accomplished here. This includes things like investigating and fixing any implementation issues that may have caused the unexpected result with query 9, adding overhead for creating indices to cost estimation, changing how you compare Dexter’s index suggestions and our algorithm’s index suggestion to provide a better set of indices to be indexed, or even rewriting the framework in a different language such as python to better accommodate the algorithm. We tried to add as much descriptive comments as possible to our algorithm, and parts of indexer.rb, so it will be easier for future work to be built upon this project.

**References:**

[1] Kratica, J., Ljubić, I., & Tošić, D. (2003, April). A genetic algorithm for the index selection problem. In *Workshops on Applications of Evolutionary Computation* (pp. 280-290). Springer, Berlin, Heidelberg.

[2] Piech, C. (2013). K Means. Retrieved April 28, 2018, from http://stanford.edu/~cpiech/cs221/handouts/kmeans.html

[3] Zaman, M., Surabattula, J., & Gruenwald, L. (2004, August). An auto-indexing technique for databases based on clustering. In *Database and Expert Systems Applications, 2004. Proceedings. 15th International Workshop on* (pp. 776-780). IEEE.

[4] Sun, J., & Hong, J. (2013, October). Index selection: a query pattern mining based approach. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems* (pp. 100-106). ACM.

[5] Kane, A. (2017, June 27). Introducing Dexter, the Automatic Indexer for Postgres. Retrieved April 29, 2018, from https://medium.com/@ankane/introducing-dexter-the-automatic-indexer-for-postgres-5f8fa8b28f27

[6] Ana Carolina Almeida, Angelo Brayner, José Maria Monteiro, Sérgio Lifschitz, Rafael Pereira de Oliveira. (2016). Automatic Physical Design Tuning based on Hypothetical Plans. Retrieved April 29, 2018, from https://pdfs.semanticscholar.org/8bb4/0a34465d46bcea5eac058b7bc8eb10b8a987.pdf

[7]Schnaitter, Karl, & Abiteoul, Serge & Milo, Tova & Polyzotis, Neoklis. (2007) “On-Line Index Selection for Shifting Workloads.” *ACM Digital Library*, IEEE Computer Society, dl.acm.org/citation.cfm?id=1547623.

[8] Pedrozo, Wendel & Gomes Vaz, Maria Salete. (2014). A Tool for Automatic Index Selection in Database Management Systems. Proceedings - 2014 International Symposium on Computer, Consumer and Control, IS3C 2014. 1061-1064. 10.1109/IS3C.2014.277.

[9] Schnaitter, Karl & Polyzotis, Neoklis. (2010). Semi-Automatic Index Tuning: Keeping DBAs in the Loop. Computing Research Repository - CORR. 5.

[10] Transaction Processing Performance Council. (1992, August 13). TPC Benchmark C. Retrieved from http://www.tpc.org/tpc\_documents\_current\_versions/pdf/tpc-c\_v5.11.0.pdf

[11] Transaction Processing Performance Council. (1999, June). TPC Benchmark H. Retrieved from http://www.tpc.org/tpc\_documents\_current\_versions/pdf/tpc-h\_v2.17.3.pdf