**PROJECT PROGRESS REPORT 1**

1. **Title**: Comparison of two different auto-indexing algorithms: clustering and non-clustering for efficient indexing on PostgreSQL’s platform.
2. **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)
3. **Project Objectives and Motivations**:

The objective for our project is to implement two different auto-indexing algorithms: clustering and non-clustering, within the source files of PostgreSQL and compare how efficiently they index tables. Clustering, by definition, is logically grouping a set of objects based on their characteristics. We decided to compare this up against a non-clustering algorithm because we wanted to see how efficient clustering is when it comes to auto-indexing. We chose PostgreSQL because unlike Oracle, PostgreSQL is open source and modifiable. We will be benchmarking our results with two different benchmark programs, one including a PostgreSQL built-in program called pgbench. It lets us enter custom parameters such as number of clients, number of threads and so on and let's PostgreSQL simulate workload and population on the database.   
 The overall objective of our project is incredibly important when it comes to lessening the daily workload of a Database Administrator, while improving the overall performance of the database itself. There’s a lot of work in this area of research, but a good chunk of them only compare one auto-indexing algorithm on a database, or improving existing auto-indexing algorithms, etc. Our paper differs because we’re not only comparing two auto-indexing algorithms, but at the same time branching out a little further and comparing what clustering/non-clustering auto-indexing algorithms can bring to the table.

1. **Literature Review**:

To improve our understanding on database auto-indexing and different implementation

approaches, we found six additional references on Continuous On-Line Tuning (COLT)[1], AISIO[3], several data mining implementations[4,5,6] and a semi-automatic implementation[2] where a Database Administrator’s (DBA) feedback can be used to refine the tuning algorithm.

In order to deal with the ever-increasing size and complexity[1, 2, 3, 4, 5, 6] thanks to the internet making data sets available to a large audience[1], most recent references involve the use of online tuning. The Schnaitter & co.[1] paper introduced a COLT implementation and outlined the COLT architecture, and it broke down the auto-index selection process into two major components: the extended query optimizer (EQO) and the self-tuning module (STM). The EQO has the additional function of creating hypothetical indexes during the query optimization process, on top of selecting the optimal physical execution plan of a query. Each time a query is ran, EQO communicates the hypothetical indexes with STM, which analyzes the cost of hypothetical queries and splits the indexes into three group: materialized, hot and cold. These groups represent a hierarchy on the potential performance benefits the indexes can bring; materialized indexes are already created and implemented in physical storage, hot indexes show high potential gain, and cold indexes show low potential gain. Based on how frequent an index is referenced in queries, it can be promoted or demoted amongst the three ranks, keeping the list of materialized indexes optimized over time.

The Pedrozo & Gomes Vaz[3] paper documented a comparison study between a database-integrated optimization tool versus an external tool. The key difference between integrated and external tools for automatic index selection is the use of Database Management System (DBMS) optimizer; whereas integrated tools allow DBMS to select the indexes, “external tools take from the DBMS the responsibility for selecting index, letting this assignment to be done by DBMS-external tools”[3]. The Automatic Index Selection Integrated into Optimizer (AISIO) tool proposed by Pedrozo & Gomes Vaz is similar to the COLT implementation described in the Schnaitter & co.[1] paper, with statistics and hypothetical index configurations from each query being passed to the DBMS optimizer, which then selects the best indexes to be implemented. Pedrozo & Gomes Vaz did state under “B. Heuristic for selection candidate indexes” that their choice of exhaustive enumeration of candidate indexes can be computationally expensive, but for systems where it is reasonable, it can produce the best choice for index selection. Pedrozo & Gomes Vaz compared the performance between no indexing, indexes created with DBT-2 external toolkit and AISIO. For no indexing, the transaction per minute (TPM) showed a noticeable decrease as the number of database warehouses increased, which is expected as increasing complexity and volume making sequential access slower. For both DBT-2 and AISIO, the increase in number of warehouses actually increased the performance with thanks to the indexing allowing for quicker, non-sequential access, and the increased number of access terminals which resulted from increase in the number of database records[3]. Between DBT-2 and AISIO, AISIO performed slightly better. While the improvement in TPM is not too significant, it did prove their hypothesis of “better results can be obtained by letting the DBMS Optimizer itself internally manage the index selection”[3] since external tools often associate with a higher cost investment.

In the Nimkanjana & co[4] paper, they focused on frequent itemset mining with the FP-Growth algorithm to improve automatic index selection given a limited space usage. Their implementation created and populated the frequent pattern tree (FP-tree) structure with attributes that surpassed a cardinality threshold they setup using the FP-Growth algorithm. The content of the FP-tree is passed through a r-filter so only items with a FP-tree value of ‘r’ or above are considered as the candidate indices. Index selection is done through an algorithm instead of the DBMS optimizer because Nimkanjana & co only indexes so that they can be created under the space usage constraint that will be considered for selection. This implementation had many variables such as the cardinality threshold, r-filter threshold and space usage constraint. While the paper stated these variables can be set automatically, it did not state how. Given this information, their implementation could be considered an offline automatic index selection, unlike COLT[1] and AISIO[3] mentioned above.

In the Ziani & Ouinten[5] paper, they proposed to further cut down on the number of candidate keys generated by traditional frequency-based data mining algorithms by only discovering the maximal frequent itemsets; a frequent itemset is called maximal if it has no superset that is frequent[5]. Their implementation of this utilized the FPMAX algorithm, which, based on a study they referenced, had the best performance amongst all maximal frequency data mining algorithms. The index selection method is similar to those described before, using FPMAX and DBMS optimizer. Their result section in this paper only compared no index versus index with FPMAX, thus a significant (36.18%) increase was expected. However, the proposal of using only the maximal frequency itemset seems sound, but consider the fact frequent itemsets are upward closed, meaning all supersets will contain the net content of all the connected lower sets. Their experiment was conducted on a fixed size database warehouse with a pre-defined workload, so it was another example of offline tuning.

The Junping & Jiman[6] paper offered a lot of insight to the Apriori algorithm for frequent itemset mining and utilization in automatic indexing. Apriori differs from both FP-Growth and FPMAX in that it use a hash tree structure to count the candidate itemsets instead of a FP-Tree. Because it uses a hash tree structure and a breadth-first traversal of subsets, redundancy is inevitable. Thus for auto indexing utilizing Apriori, it is imperative to reduce the hash-tree to avoid sending redundant candidate keys the index selection mechanism[6]. After removing redundant candidate indexes, their prior priority is calculated using histogram techniques that estimated the number of tuples for a single predict on the index. This priority value is used in the index selection mechanism where it iterated through the hash-tree and picked the candidate with the highest value. Compared to both FP-Growth and FPMAX, the amount of redundancy produced by the Apriori algorithm will definitely be considered as we pick our data mining algorithm to implement for our comparison studies.

The Schnaitter & Polyzotis[2] paper proposed a semi-automatic index selection system using WFIT, an extension of the WFA algorithm. Their definition of a semi-automatic index selection system is one where the system perform created candidate indexes based on statistics gathered from queries, but can also take positive or negative feedbacks from DBAs into consideration when performing the index selection. The way the WFA algorithm worked was to record information about the edge to edge costs as it traversed the indexes, and recommends the lowest costing path as the candidate indexes. WFIT extends on WFA by allowing an additional input from the DBA, positive or negative, alongside the recommendation WFA makes, which is used during the chooseCands method to determine the candidate keys instead of just passing the lowest cost. In the comparison study, they compared WFIT with a method called BC (after the authors Bruno and Chaudhuri) and OPT which has full knowledge of the workload and generally provided the baseline for best case scenarios. With no DBA feedbacks (basically running WFA) the performance of the resulting index was actually within 10% of OPT, at a cost of higher overhead as the system required 300ms on average to analyze each query and create recommendations[2]. When a DBA’s feedback was considered, the results became even closer to OPT’s best case scenario. However, if a DBA’s feedback is delayed, the overall responsiveness is greatly decreased. For the scope of our studies, we will likely perform only the WFA algorithm, as the WFIT algorithm’s performance weighed heavily on the DBA’s understanding of the workload and providing accurate feedbacks.

1. **Work completed**:

Up to this point in our project most of our time has been spent researching the topic of auto-indexing, how auto-indexing plays a critical role in database management systems, and making decisions on which algorithms would work best for our project. While we haven’t finalized the decision on algorithms to use, we have come to the conclusion of comparing a clustering algorithm and a non-clustering algorithm. This gives us an opportunity to compare an interesting database methodology alongside the auto-indexing algorithms.

We’ve also established some resources to work on our project with. A google drive cloud storage is being used to hold any information about our project i.e. progress reports, references, guides or general information we should find useful. A github repository was set up to maintain any source code and to enforce version control. A Microsoft Teams account was set up for quick collaboration if we weren’t available to meet in person as well.

Furthermore, we set up a central location for all of us to work from by creating virtual environment to work in. this is done by using a combination of Vagrant, a tool for building the virtual environment, and Oracle VirtualBox to host the virtual machine. This also helps eliminate issues we might run into because of conflicting operating systems or issues with inconsistent benchmarking results due to hardware specifications.

Finally, in addition to the initial research and references we cited, several more were included to support our work. Our initial research was somewhat dated. While relevant for the broad idea of auto-indexing, we attempted to find references more amicable to current auto-indexing methodologies.

1. **Work to be done**:

With our current progress, we are finalizing our decision on the 2 auto indexing algorithms. Our selection criteria for the algorithms will be based off their relevancy, resource availability, compatibility with Postgres, recency of publication, advantages, disadvantages, and etc. Currently, Charlie is finalizing preparations on the Ubuntu virtual environment using vagrant, where we will host an instance of Postgres running locally. The usage of virtual machines will ensure that the development environment is consistent and can be replicated. Dragon is researching suitable benchmarking databases to use for populating the database and benchmarking/testing/evaluating the performance of the database with the auto-generated indexes. Some that are being explored are Transaction Processing Council benchmarks (TPC); ANSI SQL Standard Scalable and Portable Benchmark (AS3AP); and Open Source Database Benchmark (OSDB). Some of these are known to be industry standards or widely used with constant support. Other considerations include compatibility, metrics that will be benchmarked, etc. The final decision will be discussed and made as a team. By using a benchmark database, this will ensure that the queries, transactions, workload, and other factors will remain controlled and consistent when running the Postgres instances using the different auto-indexing algorithms. Also, this will ensure that our benchmark results and comparisons are credible and realistic since these benchmark databases are designed to assess the performance of databases. After selecting the algorithms, Emily and TJ will begin the initial research on implementing the clustering and non-clustering auto-indexing algorithms respectively.

Development will begin when the development environment, Postgres instance, and benchmark database are prepared. The team will be split into pairs and work on implementing the clustering and non-clustering auto-indexing algorithms in Postgres. Dragon and Emily will be working on implementing the former algorithm while Charlie and TJ will be working on implementing the latter. It is expected for everyone to have a hands-on contribution to the implementation process but there will be a QA/Test role and Research/Dev role for each pair (which is to be determined). Once both algorithms have been implemented, we will start using the benchmark database to begin the testing and index creation. Performance metrics will be gathered through pgbench and another benchmarking application that has not been decided. After the database performance testing and monitoring, we will then compare and analyze the performance metrics gathered from the benchmark database and pgbench. Once that is complete, we will summarize and report our findings and conclusions on which auto-indexing algorithms is more optimal.

1. **References**:
2. Schnaitter, Karl & Abiteboul, Serge & Milo, Tova & Polyzotis, Neoklis. (2006). COLT: Continuous on-line tuning. Proceedings of the ACM SIGMOD International Conference on Management of Data. 793-795. 10.1145/1142473.1142592.
3. Schnaitter, Karl & Polyzotis, Neoklis. (2010). Semi-Automatic Index Tuning: Keeping DBAs in the Loop. Computing Research Repository - CORR. 5.
4. 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.
5. Nimkanjana, Klinsukon & Vanichayobon, Sirirut & Wettayaprasit, Wiphada. (2008). Auto-Indexing Selection Technique in Databases under Space Usage Constraint Using FP-Growth and Dynamic Programming. Computer and Electrical Engineering, International Conference on. 932-935. 10.1109/ICCEE.2008.132.
6. Ziani, B & Ouinten, Youcef. (2011). Improving Star Join Queries Performance: A Maximal Frequent Pattern Based Approach for Automatic Selection of Indexes in Relational Data Warehouses. Proceedings - 2011 International Conference on Internet Computing and Information Services, ICICIS 2011. 119 - 122. 10.1109/ICICIS.2011.36.
7. Junping Sun and Jiman Hong. 2013. Index selection: a query pattern mining based approach. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems* (RACS '13). ACM, New York, NY, USA, 100-106. DOI: https://doi.org/10.1145/2513228.2513253