Column-based Database Semantic Compression and Prediction-based Query Optimization

Zhang, Shu
szhang@cs.brown.edu

May 13, 2014

1 Introduction of the Project

Data compression is pretty important nowadays. Because the amount of data is just too huge and redundant, compression is a necessary for data-centric systems.

This project aims to build a system for (lossy) data compression and prediction basing on compressed data on column-based databases. The reason why it mainly aims on column-based databases is that our compression strategy makes more sense on column databases. Also, this project aims be compatible to regular SQL queries and aims to make predictions as precise as possible so the effect of lossy compression would be minimized.

We use machine learning based ideas to compress and decompress data, we also built a prototype columnar database to test our ideas. In the meantime, we used MySQL to store all relevant metadata information and developed an SQL optimizer to accept and parse SQL queries, make modification to it so it combines with our compressed data and metadata and return the answers. Finally, inspired by a paper called BlinkDB[1], we run experiments to compare the performance of our system and regular SQL queries to see the performance boost using our system while reducing the amount to data needed to read into memory.

This report is organized in the following manner. In the second part I introduce the data model of the database system and the principle of compression. In the third part, I introduce the SQL parsing technique and one innovation we made to make WHERE query a lot faster. The fourth part is more focused on system architecture and implementation. This part also introduces some small tools we used to make the development easier. In the fifth part, it exhibits the experimental results to prove that the system can make storage and time cost less than normal SQL queries, and the time cost is decreasing if the error bound is less tight. In the last, we conclude the work and propose some future possible improvements to make the system better.

2 Data Model and Learning-based Compression

The foundation of this project is to identify what kind of database we want to compress. Since we have some previous work on WEKA compression, so we are really inspired by WEKA and decided to build something on top of it. We choose to use columnar database because the compression is basing on columns and prediction will be easier in the column manner. For the data compression and decompression, the system replies on WEKA to train models for each column so that prediction is possible given its model and dependent columns and their values.

2.1 Column Database

We implemented a columnar database on top of Unix file system. It is kind of simple but still is the basis of all the following work. The prototype database is organized in folders' manner. Each folder in the assigned path is a table. Inside the folder, each file is a column.

The name of a folder is the name of a table. For each file inside the folder, it is actually an ARFF file. ARFF file is a file format used by WEKA. And the name of the file is name of the column. For each file, it should be self-descriptive so it has three fields inside:

- Relation: Defining the name of the column.
- Attribute: Defining the data type of this column.
- Data: The actual data of the column, one line per row.

Concretely, ARFF could represent multiple columns (a table), but in our system, it is only used to represent one column.

2.2 Data compression and decompression

In detail, the general idea of data compression/decompression is to reduce the amount of data read into memory. Instead of using time to read data from hard disk, we can use data already inside the memory and do computation to get the data which we formerly want to read from disk.

The data compression and decompression is basing on WEKA and we use two model representation format. The basic idea is to iteratively train for each column to get the model (the "prediction function" in WEKA) to deduce the possible value of it. Since it is a typical supervised learning problem, there is no much trick on it, and WEKA does a very good job in such kind of table-like supervised learning.

WEKA supports learning for two types of data : numeric and non numeric (like String). And whether the model supports either type of data is decided by the representation of the model. We adopted REPTree for both types of data and M5P rep for only numeric data. Concretely, REPTree is a decision tree which narrows the option from broad to narrow and finally gives the prediction answer. By contrast, M5P model is based on pure numeric calculation to get the prediction (mainly linear calculation) so it is pretty good for numeric predictions.

3 SQL Optimizer

One important part of the project is our SQL optimizer. The SQL optimizer accepts upstream SQL queries and translates them to commands the system understands and interact with other parts.

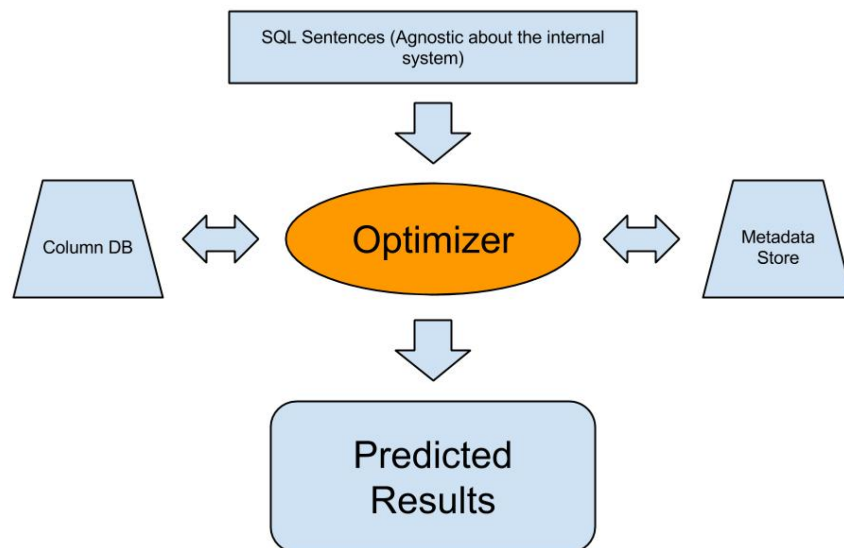


Figure 1: Optimizer and other parts

Inside the optimizer, it has several components. First is the SQL parser, it utilizes a package called ZQL to get elements out of the SQL query sentence. Then depending on the type of SQL queries (with or without WHERE), it decides whether to use a permutation calculator to get possible combinations sequences of columns. Then in order to produce the predicted results, the final part is the predictor

which talks with Metadata storage to get essential information in order to generate the results. The detailed functionalities of each module are described below.

We introduce the error bound concept in the SQL queries. it is not included in standard SQL syntax. But we decided to use it because sometimes users might want the result to be very accurate and don't need the system to predict for them. So they can set an error bound which indicates how much tolerant they are to the predict result. If the result is just not accurate and above the error bound, the optimizer will return the result directly. Importantly, we don't read out the original data out and do the comparison on the fly, we store these error bounds inside the metadata store so when the optimizer reads out the training model, it can also read out the error of a row stored inside the database. These work are done in the offline phase.

3.1 SQL Parsing

Now the system support SQL queries with two types like:

- SELECT {COLUMNS} FROM {TABLES};
- SELECT {COLUMNS} FROM {TABLES} WHERE {CONDITIONS};
- SELECT {COLUMNS} FROM {TABLES} WHERE {CONDITIONS} WITH {ERROR BOUNDS};

So the SQL parser should return elements including columns, tables and WHERE conditions. Concretely, ZQL parses out the elements basing on the SQL keywords like SELECT and FROM. Using these APIs, the SQL parser returns elements in a K-V manner, keys are the SQL keywords and values are lists of element names. For WHERE, the values are K-V pairs whose keys are column names and values are values in the WHERE. Now one constraint is that WHERE only supports equal sign, not including other signs of inequity. Future work might include those signs. The parser is robust to incorrect format of sentences - if returns an empty hash table if the sentence format is wrong.

3.2 Normal SELECT Parsing with Permutation

The normal SELECT sentence refers to sentences like "SELECT A,B,C FROM TABLE". Regularly, such queries will return the content for these three columns. But with predictions, the situation becomes more complex.

It is possible that for columns in the query, some columns could deduce the value of other columns, so the optimizer only need to read some columns instead of all, and use these columns to predict the value of the rest columns. In this way, the amount of data read into memory is reduced to only part of all.

There are something more to think about. For example, column A could be deduced by column B and the SQL query wants to get values from column A, B, C. In normal SQL queries, the amount of data read into memory would be the data of column A, B and C. In our optimizer case, first the optimizer will read B into memory, so the data would be less. Then the optimizer will calculate and estimate the value of column A. Then the optimizer knows that C is isolated so it can return results of A and B back to the user first, and after that, the optimizer cleans the memory and remove the data of A and B, and finally it gets C into memory and return it to the user. So in this example, the optimizer reduces the amount of data residing inside memory by (1) Reading partial data (2) Returning answers on the fly.

In practical implementation, we return the data to the SQL caller by writing the answers back into disk. So for each SQL query, its answer would be organized as a folder in hard disk. So in the above example, in fact, the behavior of the optimizer would be: Read B into Memory - Predict A using B - Write A and B back to hard disk as returning the answers - Read C into Memory - Write C back to hard disk.

So the normal SELECT is optimized in the above manner. But one question remains. The SQL parser should find a way to know which columns to read first so basing on the dependency, other columns could be predicted using data already inside the memory. The dependencies are stored in metadata database, but the Optimizer should investigate all possible combinations in order to get the optimized one. So this process is handled by the permutation calculator inside the optimizer.

3.2.1 Permutation And Query Plans

The idea of permutation calculator is pretty simple - it generates all possible sequence of columns which represent different query plans. Then the optimizer will scan the columns - from beginning to the end

of the sequence and see if the column being investigated could be deduced by columns also in the set. If hit, then this column could be predicted instead of being loaded from hard disk.

Concretely, different such kinds of plans will generate different answers. Using the above example again (A could be predicted by B), the query plan [A,B,C] and [B,A,C] outputs different results. For the first plan, the optimizer will behave exactly the way we described above. For the second plan, the optimizer will read B and output it first. Then when it sees A, it look the rest columns finding nothing to deduce it, so the optimizer also reads A from the hard disk instead of predicting the value of it. So these two plans has different space and time cost.

One limitation of the project is that the optimizer cannot find the best plan on the fly. Since we pay more on experiment, the optimizer will execute all query plans and output all these results. In the future, the optimizer could be improved to get the best query plan and only execute it.

3.3 SELECT parsing with WHERE clause

We also support SQL queries with WHERE. Firstly, we implemented the behavior of normal SQL queries with WHERE in our prototype columnar database. The naive WHERE queries are executed by scan all columns in WHERE and records the indices of rows which meet the conditions. Finally by intersecting these indices, we can get the result indices and thus return the answers.

Inspired by the BlinkDB paper, we got to know that human factor is also important in query patterns. Generally, people tend to query for certain attributes which are closely related to each other and have semantic meanings. For example, people tend to query for the prices of houses giving conditions like the size and location of the houses. Intuitively, the prices of the house could be learnt by size and location features and thus it makes precise prediction possible.

So in the project we pose an optimization of WHERE queries. Instead of scan for each attribute in WHERE to get the filtered results, we use the values in WHERE to predict the answer directly. In order to get the precise answer, the optimizer needs to look at the queried attribute could be has the dependency of the columns in the WHERE part. We don't require the columns in the WHERE are exactly the same as the the dependency set, but the more close they are , the more precise the prediction will be. In the example of housing price prediction, it is very possible that the model of housing prices has the dependency of attributes including space and location. SO the optimizer will run WEKA API to get the predicted answer directly.

There are two major limitations in this feature. First is that only equity sign is permit in WHERE so the prediction is based on actual and precise values. Secondly, the attributes in WHERE should be included in the dependency of the target attribute so WEKA can predict the value. Also, we expect the attributes in WHERE combines to form the primary key of the table so the answer will be unique instead of multiple rows.

3.4 Error Bound Processing Mixed in

We took a step further and give user the flexibility in selecting whether to use WEKA to predict the answers or not. We introduce the the functionality of letting users set their error tolerance which represents their expectation of precision of the answer. The user can add the error bound to the end of SQL queries. The error stands for the difference percentage of the predicted answers to the original data. The larger the error bound the user sets, the more tolerant the user is to the prediction, the more possible for the optimizer to return the expected answer instead of read the actual value. The error bound property is only valid for WHERE queries because error bound only makes sense for a particular row, not for all rows in one column.

The optimizer processes queries with error bound are reduced in two possible manners. Basically, if the prediction is beyond the error bound, the optimizer will run regular SQL query to scan the database and fetch the original data. If the prediction is below the error bound, the optimizer runs the prediction procedure.

It is not practical to run prediction and calculate the error on the fly. Actually we stored these error information inside the metadata store. The detail will be discussed in next chapter. But the general idea is to get the errors in offline phase.

4 System Architecture and Design Principles

4.1 Online and Offline Separation

We treat the online and offline phases differently and seriously in order to make the system have good performance. In the offline phase, we train the models for each columns and store these models in MySQL database. Also we run error calculation for target columns and store errors so optimizer can get the errors out directly. In the online phase, we have optimizer as the major component which talks to process users' queries and talk to different parts of the system. We use some mature infrastructure like MySQL to manage the offline phase and their results. So in this chapter we discuss the design of table layouts in the metadata store and also mention some of other small tools we developed to make the work easier.

4.2 Metadata Management System

One thing to clarify is that the project is dealing with data stored in columnar database so we have made a simple one to test. But for metadata like models' raw data and dependencies, we use MySQL to store it (which is not a column database).

4.2.1 Model Store

The models of each column is stored in metastore. For each table, there are two tables in MySQL representing its models for two types of model types (REPTree and M5P). For example, if the table name is "house", then the two metadata tables is named as "house_REPTree" and "house_M5P".

For the metadata table, it has three attributes. The first is the column name, which is equal to the column name of the table in the column DB. The second is the dependency, which are columns required to predict the target column. The third attribute is the model. The model is understandable for WEKA, but in MySQL, we just store them in the binary format (long blob type). So everytime JDBC stores the model, it needs to (de)serialize it to translate it between WEKA java type and binary blob type.

4.2.2 Error Store

In order to support the error bound feature in SQL queries, we also need to store errors in MySQL. For each table in column DB, it has a corresponding error table for each model type. Each error store table has five attributes. First is the column name. For the second to fifth attribute, they are sets of K-V pairs. The second attribute stands for rows with 1% or less error percent. The third is for error percent from 1% to 5%. The fourth is for error percent from 5% to 10%. The last one is for error percent from 10% to 25%. For rows included in above, they have error percent higher than 25%.

The K-V pairs are the values for columns of the dependency. For example, we have three columns A,B and C. A could be predicted using B and C (B and C are dependencies of A). The error predictions are shown in the following:

B Value	C Value	Prediction Error Percent
1	2	0.75%
3	4	5.6%
10	20	26%

So for the first row, {B=1, C=2} will be put in the first bucket which is the second column of the error table because it has error bound lower than 1%. The second will be put in the fourth column and the last error will not be put in the table. So when user set an error bound like 50%, and inside WHERE clause B=10 and C=20, the optimizer will read the original data out because it cannot find such K-V pair in the error store (although 26% is less than 50% which is set by the user).

Concretely, these K-V are combined as an element in a hash set. So the optimizer can load these four hash sets into memory and do the "hit detection" in constant time. The hash sets are also encoded as binary blob and will be translated to Java HashSet in program.

We generate these error bounds by reusing the procedure of optimizer querying SELECT without WHERE. That is to say, we run these queries in offline phase to get the predicted values for the target column and calculate the errors.

4.3 Other Tools

We also write a bunch of small tools.

- A program splitting an ARFF file to a folder containing all columns split to several columns files.
- A metadata storing program putting classifiers and related metadata into MySQL.
- A simple SQL parser to extract different components of a SQL sentence.
- An offline program to get the error bounds by comparing predictions to original values.

5 Experimental Results

We focused our experiments on the optimizer executing WHERE queries with error bounds. The reason is that it is the major innovation of our project and it does provide speed boost without sacrificing precision a lot.

Our methodology is to randomly generate WHERE sentences with controlled portions of accuracies. In our cases, 10% of the SQL queries will hit the rows with errors with 1% or less, 20% have 5% error or less, 40% have 5% - 10% error and the rest have 10% to 25% error. We generate different sizes of such sets but keep the portion the same. For experiment we have sets with size 10, 100 and 1000.

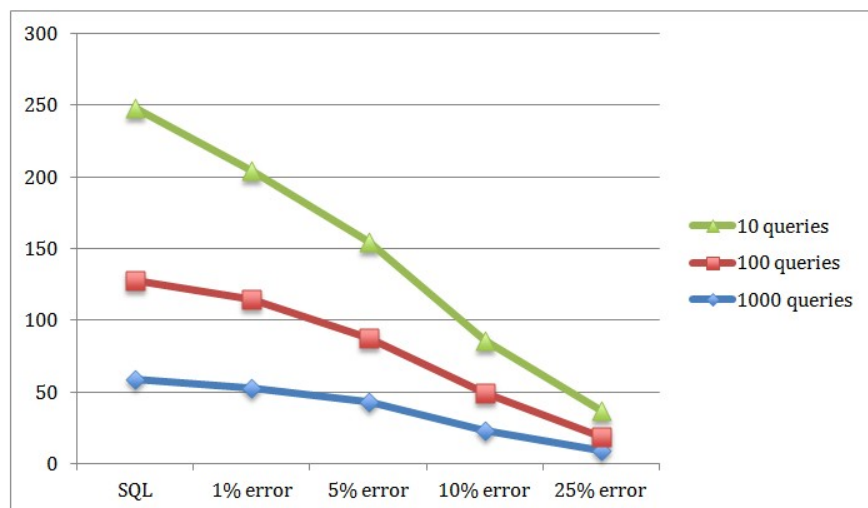
So we try to run these sets of queries with different error bound requirements. For example, we set 1% error bound to these queries, so in this case, 10% of the queries will be using prediction to get the answers, but the rest of them will all be reading directly from database. We set error bounds from 1% to 25% (which all queries will be using prediction) and compare the running time. Intuitively, the larger error bound we set, the faster the queries will be executed.

We also compare the overall speed improvement by running these queries using regular SQL engines which does not support error bound and will be reading values from databases. So we record the running time and compare it with optimizer's.

The following is the experiment results. For now we focused on numeric values. We generate two types of graphs, first is the execution time of different sets of queries with different error bounds. The second is the percentage of speedup compared to normal WHERE queries. We investigated three databases and ran evaluation on them to ensure the results are convincing enough.

Table Name: House16H

Column Name: P1



Improvement percentage over the SQL case:

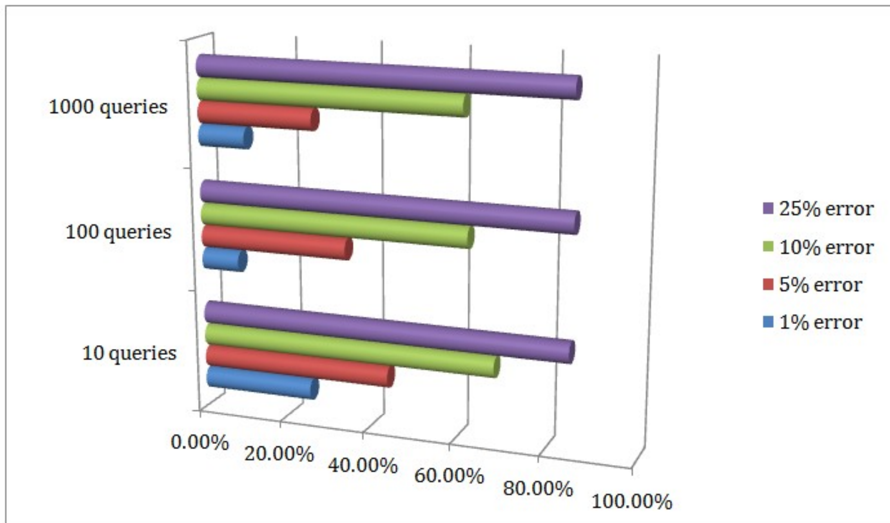
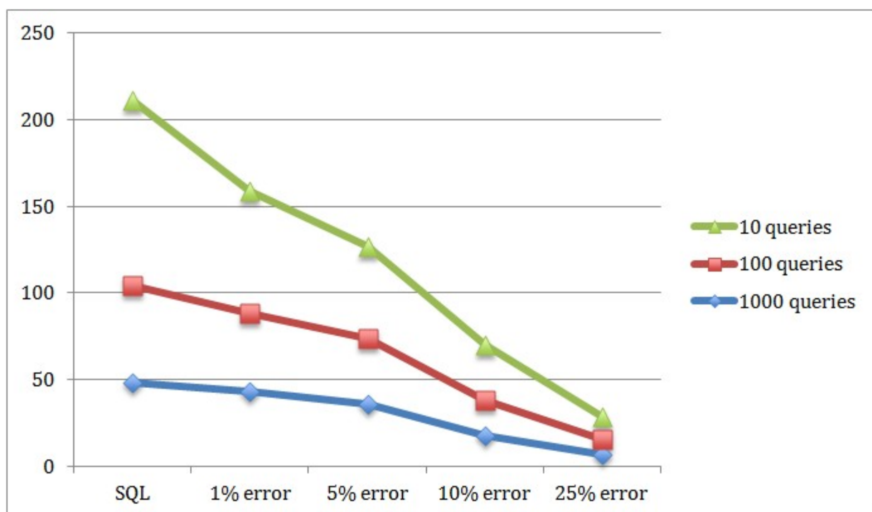
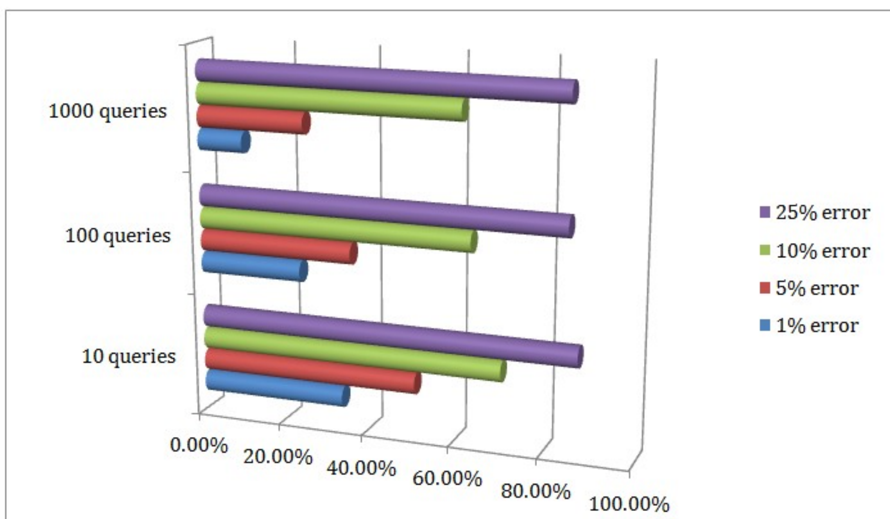


Table Name: House8L

Column Name: H18pA



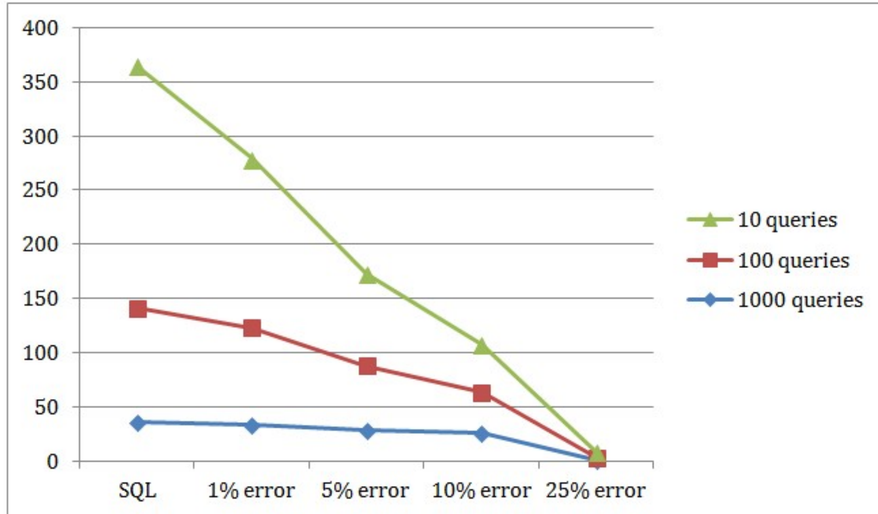
Improvement percentage over the SQL case:



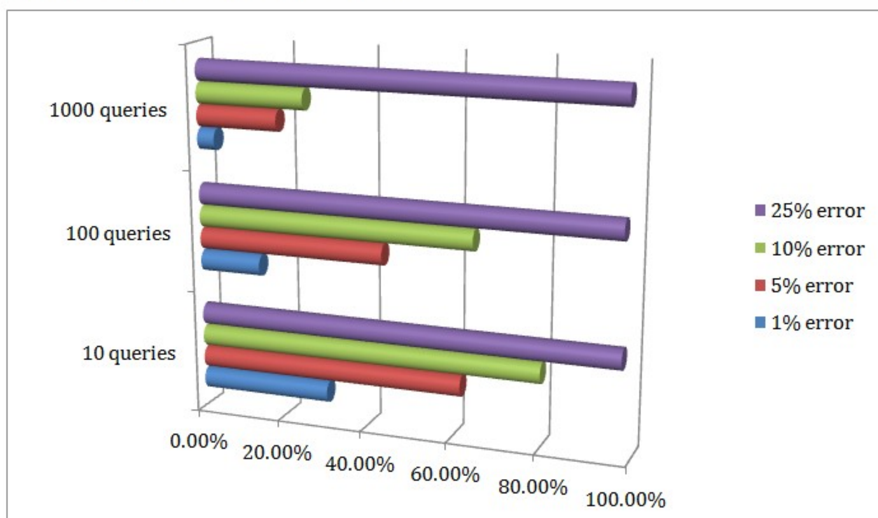
Data set: Current Population Survey (CPS)

Column Name: PEERNHRO

Very Strong Correlateted Column 0.8960424104064841 [0% - 1%]



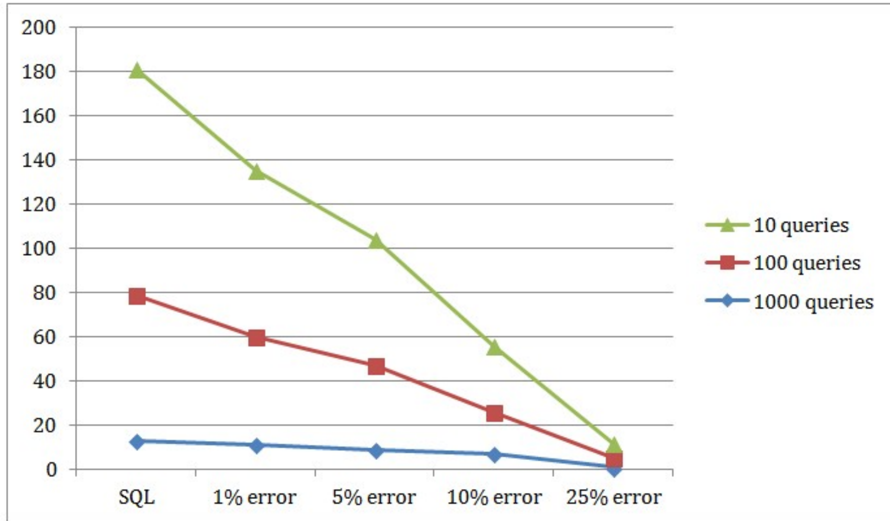
Improvement percentage over the SQL case:



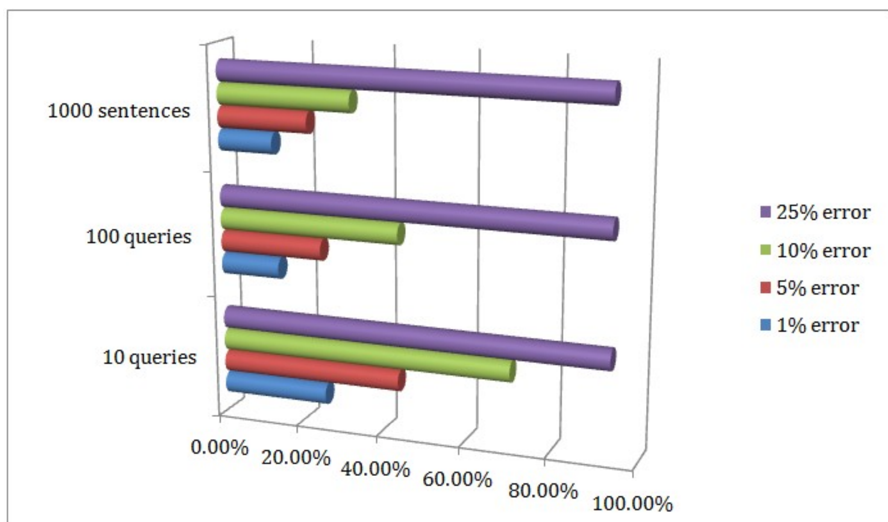
Data set: bank32nh (A family of datasets synthetically generated from a simulation of how bank-customers choose their banks.)

Column Name: b1x

Weak Correlateted Column 0.0015869140625 [0% - 1%]



Improvement percentage over the SQL case:



6 Conclusion and Future work

In this project I introduce a compression and query optimization method basing on columnar databases. I have built the system from scratch, and evaluated the performance of the major functionality - WHERE prediction with error bound settings. The results prove that using prediction would save I/O time and make the queries faster without sacrificing precision.

However, there remain a lot of interesting features to improve and explore in the future. First is that the optimizer cannot pick the best query plan from all possible plans generated by the permutation, so making it smarter to pick the best one (online) will be an important functionality to finish. Secondly, now we only support queries on one table which means the optimizer can't handle queries with JOIN keywords. Also, the optimizer doesn't support aggregation calculators like SUM and AVG. But one of my previous project (SqueezeDB) dealt with such queries so it might be possible to merge the idea into here. The last one is that the system can be extended to real world column databases.

References

[1] Agarwal, Sameer, et al. "BlinkDB: queries with bounded errors and bounded response times on very large data." Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013.

[2] http://www.cs.waikato.ac.nz/ml/weka

[3] http://zql.sourceforge.net