UNIVERSITY OF TWENTE.

Academic Research Skills
Conference Proceedings

internal
edition

Benchmarking MayBMS based on hardware specifications and query complexity

Marnick Q.T.P. van der Arend, Jelle F. Beerten, M. van Keulen.

Pre-master track: CS

Submission date: January 24, 2020

This research proposes a new kind of database that can store uncertain information: a probabilistic database. To this day, no standardized benchmark is available to assess the performance of a probabilistic database. This paper examines a benchmark for the probabilistic database 'MayBMS'. The benchmark assesses the execution time of probabilistic queries based on the database size. An experiment is run on two hardware platforms to assess the validity of the benchmark. The benchmark creates probabilistic data and runs the benchmark queries. From the results is derived that the number of matches for a query and the type of hardware are of equal importance. Even though the instance with the worse hardware took considerably longer to execute, the ratio between the execution time of the queries stayed the same as on the better hardware. Thus, proving the validity of the benchmark.

Keywords: Probabilistic data, MayBMS, data generator, benchmark, database hardware

1. Introduction

Consider a hospital, patients are diagnosed with different kinds of diseases based on a set of key symptoms. In some cases, not all key symptoms are visible or can otherwise not be absolutely determined. Predicting if a patient has a certain disease could be of great value in this case. This prediction would be based on historical data of patients with likewise symptoms for a disease.

If this data is entered into a database, many rows and columns will be produced. Rows can either be completely filled, partially filled or empty. These partially filled rows contain so-called missing values.

Through comparing the symptoms of different patients, it is possible to calculate the probability of our patient having disease A or disease B. This creates a so-called probabilistic database. These probabilistic databases could be of great value to minimize the amount of missing values.

Advances in probabilistic databases have the potential of providing for more automatic and robust data integration. Currently, there is no standard for comparing the scalability and efficiency of these probabilistic databases. The performance of the probabilistic databases cannot be measured correctly. This research will fill this gap by delivering a benchmark for measuring the performance of the probabilistic database 'MayBMS'.

2. Related work

2.1 Probabilistic Data

Probabilistic or uncertain data is data that has multiple possible values, with each value having a certain probability of being the correct value. Data integration is the process of combining data from multiple sources into a single, unified view. As the amount of data generated by both humans and machines is ever increasing, the field of Computer Science has shifted its focus to data-intensive applications rather than computation-intensive ones. Most of this data, however, is uncertain, and therefore has become necessary for many modern applications. [1].

As stated by Halevy, Rajaraman, & Ordille 'While in traditional database management managing uncertainty and lineage seems like a nice feature, in data integration it becomes a necessity' [2]. Researches have come up with multiple frameworks and applications to aid the process of data integration with uncertain data and to improve the performance of said process [1] [3] [4]. However, it is not possible to directly compare these techniques, as each of them uses a different methodology and starting point.

2.2 Probabilistic Databases

There are many database management systems (DBMS) freely available on the internet such as MySQL, PostgreSQL and BaseX. These databases do not take into account the occurrence of probabilistic data. Probabilistic data therefore needs to be stored in a probabilistic database management system. Van Keulen describes a probabilistic database as 'a specific kind of DBMS that allows storage, querying and manipulation of uncertain data. It keeps track of alternatives and the dependencies among them.' [5] There are multiple research groups that have developed such systems, for example Trio [6] and MayBMS [7].

Trio is 'a robust prototype that supports uncertain data and data lineage' [6]. Trio uses a new database model called the Uncertainty-Lineage database that is built on top of the traditional DBMS. Queries can be constructed using TriQL, which is a strict extension of SQL.

MayBMS is 'a state-of-the-art probabilistic database management system that leverages the strengths of previous database research for achieving scalability.' [7] It is built as an extension of PostgreSQL and adds functionality for handling probabilistic data.

MayBMS stores probabilistic data with the '*possible worlds*' semantics. 'Assuming a single table, let I be a set of tuples (records) representing that table. A probabilistic database is a discrete probability space *PDB = (W, P),* where $W = I_1,I_2,\ldots,I_n$ is a set of possible instances, called possible worlds, and *P : W*[0, 1] is such that $\sum_{j=1\ldots n} P(I_j) = 1$.' [5]

3. Methodology

This research is comprised of two parts that together create the probabilistic database benchmark. The goal of this benchmark is to assess the performance of the query execution at multiple sample sizes on different hardware platforms proving its validity.

The methodology of these parts is set out in separate sections in this chapter. An overview of the flow of the benchmark is depicted in Figure 1.

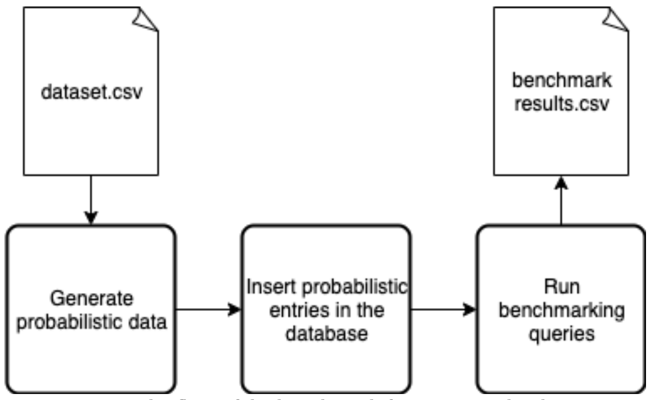
dataset.csv
benchmark results.csv
Generate probabilistic data
Insert probabilistic entries in the database
Run benchmarking queries

Figure 1.; The flow of the benchmark for MayBMS databases

3.1 Data Generator

A data generator has been constructed to obtain representative probabilistic data. The generator had to be easy to use, thus a test dataset had to be added.

The data generator takes a comma separated value file as input. The values are extracted from the file and put into an array. The generator creates a random probability and appends this to the array in a new column named ‘p’. Depending on the generated probability, which is always less than one, it creates two or three new rows with different probabilities for column p. The sum of these probabilities adds up to 1 to abide the rules of MayBMS. The generator puts the new entries into the database after successfully adding probability to each array.

The non-probabilistic dataset that the generator uses to generate probabilistic data has a size of 1 million entries. The generator is able to create up to 3 million entries from this dataset, which can be used for the benchmark.

The sample size for the benchmark is scalable. The number of entries that the benchmark should test can be set to a number compliant with the capabilities of a hardware platform. Though it should not exceed the 3 million entries limit.

The dataset that is used for creating a probabilistic dataset is retrieved from Kaggle [8] [9]. The dataset contains information about United States traffic accidents between February 2016 and December 2019. The precise columns of this dataset are described in Appendix A. This dataset was chosen because of its huge size in columns and rows. Making it ideal to extract useful data from to use for the probability database.

3.2 Benchmark

Performance of the probabilistic database is measured in several ways. Execution time of queries depends on caching operations that happen within a database server. The execution time encompasses various processes: the request going to the database, the query executed in the database, and the retrieval of the result.

The benchmark should be a standalone tool to test the performance of a database with. It should be able to make the connection with a configurated probabilistic database and deliver its own data and queries. The data that is used in the process of running the benchmark is generated on the spot and deleted after the benchmark is done.

To find a way around the initialization delay of running a query, every query that is executed will be run six times. Since the query plan only is generated once and cached, we can ignore the first query result and have five similar query executions from which we take the average execution time.

A second benchmark criterium is the size of the data that is put through the benchmark. We want to know how well query execution scales with different sizes of data. The benchmark will be closely related to the data generator as it will need different sizes of data for different experiments and representative queries for the generated data set.

A third benchmark criterium is testing the functionality of the extensions of MayBMS to the PostgreSQL language in combination with traditional PostgreSQL syntax. The benchmark experiments with conf() and tconf() alongside PostgreSQL expressions WHERE, GROUP BY, ORDER BY, UNION and self-join.

3.3 Experimental setup

To validly measure the performance of the benchmark, two MayBMS instances were used. The database instances were run on two computers with different specifications create a clear overview of the execution time of the queries using different hardware. These specifications are shown in Table 1.

Specifications	**Instance A**	**Instance B**
Model	Acer Aspire 5	Asus X55V
Operating System	Windows OS	Windows OS
Processor	Intel i5-7200U 2.5GHz	Intel i3-2350M 2.3GHz
RAM	8 GB DDR4	4 GB DDR3
Storage	256 GB SSD 1000 GB HDD	320 GB HDD

Table 1.; Specifications of the MayBMS instances

The goal of the experiment is to compare the scalability of probabilistic database system MayBMS on two different hardware platforms.

The benchmark was run on 6 different queries, all different in size and complexity on two different instances. An overview of the queries is available in Appendix B. The sizes of the samples ranged from 1.000 to 1.000.000 entries.

The benchmark is developed with Python, since it poses as an easy-to-use and widely accepted programming language for Data Science research. The proof of concept of the benchmark is available on GitHub[1].

4. Results

The resulting execution times of the queries on different machines at different sample sizes are depicted in Figure 2 and Figure 3. The lines in the graph each represent the execution time of a query. The graph includes a trend for the exponential growth of each of the lines to show the growth of the execution time as the sample size becomes larger.

[1] https://github.com/MarnickvdA/maybms-benchmark

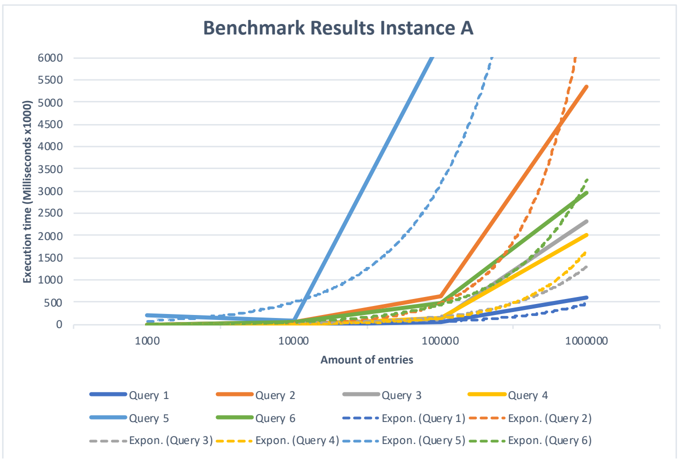

Figure 2.; Benchmark results of Instance A

In Figure 2, the benchmark is run on Instance A. This graph shows a relatively lower execution time for easier queries like 1, 3 and 4. It is remarkable that it shows a higher execution time for query 2, since we labelled that query as relatively easy. Query 5 drops in execution time when the sample size increases from 1.000 to 10.000, but after 10.000 drastically slows down in execution speed. Query 6 is following a similar slope as the easier queries.

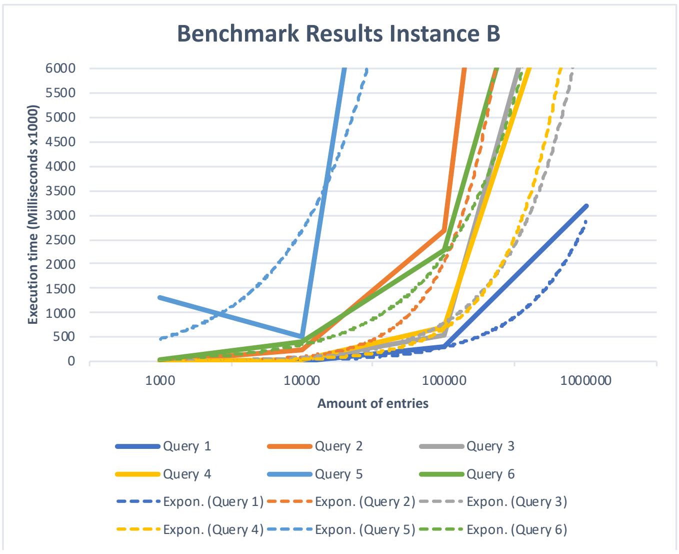

Figure 3.; Benchmark results of Instance B

In Figure 3, the benchmark is run on Instance B. This graph shows a much steeper curve compared to Figure 2. Query 5 decreases more than on Instance A after increasing the sample size from 1.000 to 10.000.

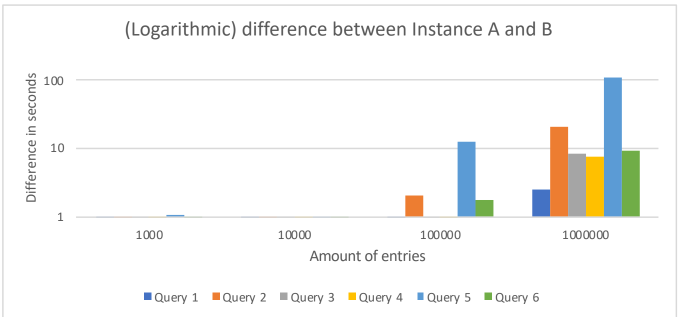

Figure 4.; Difference in execution time between instance A and B

For comparison of the different machines, the bar chart in Figure 4 was constructed.

5. Discussion

The queries are doing what was expected. Running the benchmark with 1.000 entries yield similar results for all the queries. These queries have close to zero matches with this small sample size. Thus, the execution time is nearly the same. The only outlier is query 5. This query takes roughly 100 times longer than the second slowest query on both instances. Before the execution time of the queries is measured, they are running once to cache each of them. An explanation for the slow execution time of query 5 is that the other queries completely fill up the cache when measured before query 5 is measured. This would cause query 5 to miss out on the benefit of the cache. In combination with being a complex query, increases its execution time.

At a sample of 10.000 entries, only query 5 does something very remarkable. It is the only query that decreases in execution time. This occurs on both instances and therefore it is not a fluke. Running the benchmark with this sample size multiple times yielded the same result every time. It is not clear why this event recurs every time.

All the queries increase majorly in execution time when increasing the sample size to 100.000. Query 5 skyrockets for both instances. The big difference in execution time between the queries can be explained through their results. The queries that take the longest are also the queries with the highest number of matches. This shows the impact the number of results makes on the execution time of a query.

Comparing the results of the benchmark on the two hardware platforms does not give surprising results. Instance A is always faster than Instance B. Looking at the bar graph in Figure 4, As Figure 4 depicts, the difference between both instances gets larger as the sample size increases. This was expected because of instance A's faster processor.

The validation of the benchmark was achieved by comparing the execution times of two instances with different hardware. It is clear to see that the hardware has a noticeable impact on the performance of the database. At a million entries, the difference becomes extremely noticeable. Instance A executes most of its queries in 5 or less seconds. Instance B executes most of its queries in 10 seconds. The difference in execution time between both instances for query 5 reaches an amount of 111 seconds. If time is crucial and hundreds of queries must be executed, the benchmark shows that having excellent hardware is a must. Even though instance A is faster for all queries, the ratio between the execution time of the queries is the same on both instances for all samples.

6. Conclusion

Before this research, there were no benchmarks for probabilistic databases that examined the correlation between the hardware platform and query execution time.

This research examined the design of a probabilistic database benchmark with the goal of assessing the performance of the query execution at multiple sample sizes on different hardware platforms. Thus, an experiment was conducted to investigate the behaviour of MayBMS on two different hardware platforms with the purpose of validating the benchmark.

This research shows that hardware becomes more important as the sample size increases. At lower sample sizes, the difference in hardware is barely noticeable. The difference in time is only a few one-ten-thousandths of a second. Taking a sample of a million entries already gives deviations of up to hundreds of seconds.

The amount of results a query returns is of major importance. This is best shown with query 2 and 5. At the sample size of 1.000

entries, query 2 has close to zero results and takes the least amount of time on instance A. At the sample size of 100.000 entries, query 2 and 5 take the most amount of time on both instances. This occurs because these queries have substantially more matches than the other queries. Comparing the other queries with one another shows that the amount of matches mostly affects the execution time of a query. Considering the queries with the most matches take the longest.

The impact of the number of uncertainties was not tested in this project. The generator is not capable of transforming a dataset into a probabilistic dataset with multiple different uncertainties. Therefore, we cannot draw a conclusion on that matter. Testing with actual probabilistic data instead of randomly generated ones could provide an answer to this question.

The benchmark shows the importance of hardware and the number of matches of a query through an experiment. The experiment showed that instance A had a faster execution time than instance B. Though still keeping the same ratio between the execution time of the queries on both instances. The effect of the number of matches of a query can also be derived from the duration of the execution time of a query. This displays that the execution time heavily depends on the amount of matches a query has. Taken together, these conclusions prove the validity of the benchmark.

7. Future work

This research only touches the tip of the iceberg for benchmarking probabilistic databases. The benchmark that is created does not benchmark all functionalities of probabilistic databases. Future work could consider adding more MayBMS specific functionalities.

The current benchmark does not generate data that is used in practice. Real-life datasets with probability data are not easy to find. This research did not invest much time into finding one dataset that contained probability elements. Currently, the data generator has to assign probability to the dataset with a random factor. Future work could investigate the results of benchmarking with real-life queries and real-life datasets. This increases the validity of the benchmark as it is no longer based on fake probabilistic data.

This research only focused on the probabilistic database 'MayBMS'. There are more probabilistic databases of which one is Trio. Future research could investigate the performance of this benchmark on different types of probabilistic databases.

References

[1] F. Sadri and T. Gayatri, “Integration of Probabilistic Information,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Sydney, FL, 2016.

[2] A. Halevy, A. Rajaraman and J. Ordille, "Data integration: The teenage years," in *VLDB 2006 - Proceedings of the 32nd International Conference on Very Large Data Bases*, Seoul, 2006.

[3] J. Kampars and J. Grabis, “Near Real-Time Big-Data Processing for Data Driven Applications,” in *Proceedings - 2017 International Conference on Big Data Innovations and Applications*, Prague, 2017.

[4] M. Nentwig and E. Rahm, “Incremental clustering on linked data,” in *IEEE International Conference on Data Mining Workshops*, Singapore, 2018.

[5] M. van Keulen, “Probabilistic Data Integration,” in *Encyclopedia of Big Data Technologies*, Cham, Springer, 2018, pp. 1-9.

[6] J. Widom, “Trio: a system for integrated management of data, accuracy, and lineage,” in *CIDR*, Stanford, 2005.

[7] Huang, Antova, Koch and Olteanu, “MayBMS: A Probabilistic Database Management System,” in *SIGMOD*, Rhode Island, 2009.

[8] S. Moosavi, M. H. Samavatian, S. Parthasarathy and R. Ramnath, “A Countrywide Traffic Accident Dataset,” Ohio State University, Columbus, 2019.

[9] S. Moosavi, M. H. Samavatian, S. Parthasarathy, R. Teodorescu and R. Ramnath, “Accident Risk Prediction based on Heterogeneous Sparse Data: New Dataset and Insights,” Ohio State University, Columbus, 2019.

[10] S. Moosavi, “Kaggle,” The Ohio State University, 21 May 2019. [Online]. Available: https://www.kaggle.com/sobhanmoosavi/us-accidents. [Accessed 23 January 2020].

Appendices

Appendix A: Columns of Kaggle dataset

On the Kaggle website the dataset that was used for the experiment is described. The columns contain the following information [10]:

- ID: This is a unique identifier of the accident record.
- Source: Indicates source of the accident report (i.e. the API which reported the accident.).
- TMC: A traffic accident may have a Traffic Message Channel (TMC) code which provides more detailed description of the event.
- Severity: Shows the severity of the accident, a number between 1 and 4, where 1 indicates the least impact on traffic (i.e., short delay as a result of the accident) and 4 indicates a significant impact on traffic (i.e., long delay).
- Start_Time: Shows start time of the accident in local time zone.
- End_Time: Shows end time of the accident in local time zone.
- Start_Lat: Shows latitude in GPS coordinate of the start point.
- Start_Lng: Shows longitude in GPS coordinate of the start point.
- End_Lat: Shows latitude in GPS coordinate of the end point.
- End_Lng: Shows longitude in GPS coordinate of the end point.
- Distance(mi): The length of the road extent affected by the accident.
- Description: Shows natural language description of the accident.
- Number: Shows the street number in address field.
- Street: Shows the street name in address field.
- Side: Shows the relative side of the street (Right/Left) in address field.
- City: Shows the city in address field.
- County: Shows the county in address field.
- State: Shows the state in address field.
- Zipcode: Shows the zipcode in address field.
- Country: Shows the country in address field.
- Timezone: Shows timezone based on the location of the accident (eastern, central, etc.).
- Airport_Code: Denotes an airport-based weather station which is the closest one to location of the accident.
- Weather_Timestamp: Shows the timestamp of weather observation record (in local time).
- Temperature(F): Shows the temperature (in Fahrenheit).
- Wind_Chill(F): Shows the wind chill (in Fahrenheit).
- Humidity(%): Shows the humidity (in percentage).
- Pressure(in): Shows the air pressure (in inches).
- Visibility(mi): Shows visibility (in miles).
- Wind_Direction: Shows wind direction.
- Wind_Speed(mph): Shows wind speed (in miles per hour).
- Precipitation(in): Shows precipitation amount in inches, if there is any.
- Weather_Condition: Shows the weather condition (rain, snow, thunderstorm, fog, etc.).
- Amenity: A Point-Of-Interest (POI) annotation which indicates presence of amenity in a nearby location.
- Bump: A POI annotation which indicates presence of speed bump or hump in a nearby location.
- Crossing: A POI annotation which indicates presence of crossing in a nearby location.
- Give_Way: A POI annotation which indicates presence of give_way sign in a nearby location.
- Junction: A POI annotation which indicates presence of junction in a nearby location.
- No_Exit: A POI annotation which indicates presence of no_exit sign in a nearby location.
- Railway: A POI annotation which indicates presence of railway in a nearby location.
- Roundabout: A POI annotation which indicates presence of roundabout in a nearby location.
- Station: A POI annotation which indicates presence of station (bus, train, etc.) in a nearby location.
- Stop: A POI annotation which indicates presence of stop sign in a nearby location.
- Traffic_Calming: A POI annotation which indicates presence of traffic_calming means in a nearby location.
- Traffic_Signal: A POI annotation which indicates presence of traffic_signal in a nearby location.
- Turning_Loop: A POI annotation which indicates presence of turning_loop in a nearby location.
- Sunrise_Sunset: Shows the period of day (i.e. day or night) based on sunrise/sunset.
- Civil_Twilight: Shows the period of day (i.e. day or night) based on civil twilight.
- Nautical_Twilight: Shows the period of day (i.e. day or night) based on nautical twilight.
- Astronomical_Twilight: Shows the period of day (i.e. day or night) based on astronomical twilight.

Appendix B: Overview of the queries

```sql
# Query 1
SELECT ID, tconf()
FROM p_table
WHERE Weight = 1 AND ID = 'A-1';

# Query 2
SELECT ID, Start_Time, End_Time, tconf()
FROM p_table
WHERE End_time < date '2018-07-20 00:00:00'
AND Start_Time > date '2016-03-09 00:00:00';

# Query 3
SELECT conf()
FROM p_table
WHERE Start_Time <= date '2017-04-20 00:00:00'
AND Temperature <= '40'
GROUP BY city;

# Query 4
SELECT City, ID, Severity, tconf() as prob
FROM p_table
WHERE Start_Time >= date '2016-04-09 00:00:00'
AND End_Time <= date '2018-05-12 : 00:00:00'
AND Astronomical_Twilight = 'Night'
AND Source = 'MapQuest'
AND Severity = '2'
GROUP BY ID, City, Severity, prob;

# Query 5
SELECT A.ID as ID1, B.ID as ID2, A.City,
tconf()
FROM P_table A, P_table B
WHERE A.ID <> B.ID
AND A.City = B.city
AND A.Start_Time <= date '2016-08-01 00:00:00'
AND A.Severity = '2'
AND B.Weather_Condition = 'Rain'
ORDER BY A.City;
```

```
# Query 6
SELECT ID, Start_Time, Severity, tconf()
FROM p_table
WHERE Start_time < date '2016-07-20 00:00:00'
UNION
SELECT
ID, Start_Time, Weather_Condition, tconf()
FROM p_table
WHERE Start_time > date '2017-07-20 00:00:00'
ORDER BY Start_Time;
```