Verifying the performance benefits of caching probabilities in probabilistic databases

MATTEO SCHUT, University of Twente, The Netherlands

Additional Key Words and Phrases: Probabilistic Databases, Probability Caching

ABSTRACT

Probability calculation has been observed to dominate performance for some queries in probabilistic databases. Current probability storage structure is suspected to be sub-optimal. This paper focuses on researching possible performance benefits of caching probabilities in probabilistic databases, focused on the probabilistic database management system DuBio. Controlled experiments have been run to measure performance benefits while accounting for different queries, database structures and server delays. This paper presents results showing under which circumstances caching is advisable.

1 INTRODUCTION

In recent years, an increasingly large amount of applications have had to deal with uncertain data. For this reason, a significant effort has been made to research the effectiveness of probabilistic databases, where uncertainty can be included within a parameter of a data value as a probability [7]. Due to the increasing importance of these data management systems, improving performance is of significant interest.

At the University of Twente, an implementation has been made as an extension of PostgreSQL called DuBio. In this extension, probabilities are represented through Binary Decision Diagrams (BDDs, see section 2.2). To calculate the probability of a BDD, a user-defined 'Dictionary' object has to be searched through for each variable in the BDD, which can become quite slow for multiple BDDs. This paper researches the performance benefits of storing probabilities in a column next to the BDDs.

Storing probabilities in this way can be seen as permanent caching. Caching function results is a well-studied subject of research, commonly called memoization. Memoization has been around for quite some time, and as outlined by Maux (2022), can prove to be quite effective [4]. Memoization has also been applied to a different PostgreSQL extension [3] to achieve promising results. Similar memoization techniques have been used when researching probabilistic databases [5], however, these works all focus on temporarily storing function results. If function arguments rarely change, as is the case for this research, cache can be stored more permanently.

TScIT 41, July 5, 2024, Enschede, The Netherlands

© 2024 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Table 1. Example table DuBio

ID	suspect	BDD
1	'Frank'	BDD('car=1 & color=2')
1	'Frank'	BDD('car=2 & color=1')
2	'Amy'	BDD('car=2 & color=1')

2 BACKGROUND

2.1 Probabilistic Databases

Probabilistic databases are designed to store uncertain data, meaning that if one is unsure about the value of a certain variable, all possibilities can be stored next to the probability of it being that value [7]. Probabilistic databases can also store the relation between variables, meaning the probability of a database entry being true can be composed of multiple variables being true or false.

An example could be a witness at a crime scene unsure about their observations. The witness is unsure whether they saw a Honda or a Mazda, but is 90% sure the car was red. You know from police reports that the only people in the area at the time were Frank and Amy. Frank owns a green Honda and a red Mazda and Amy owns a red Mazda.

In DuBio, the probability of each person being a suspect can be stored by including all possibilities in a table, and changing the variables in each BDD to represent that possibility. This is shown in Table 1. Here, 'car=1' is Honda, 'car=2' is Mazda, 'color=1' is red and 'color=2' is green. If, for example, police confirm it to have been a green car, then the probability of 'color=1' becomes 0. This would mean that every BDD containing 'color=1' can be removed from the suspect table, leaving Frank as the only suspect.

2.2 Binary Decision Diagrams

Binary decision diagrams are tree graphs that can represent a boolean expression of any length [2]. They are composed of a root node and several child nodes, each node containing a boolean variable. Traversing the tree from the root node will result in the solution to the boolean expression. An example is visualized in Figure 1. In DuBio, they are represented as seen in Table 1.

2.3 Dictionary

The dictionary is a user-defined PostgreSQL object in the DuBio extension which stores probabilities of boolean variables. Every time a probability needs to be retrieved, the dictionary will execute

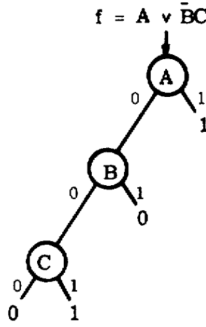

Fig. 1. Example of a boolean expression represented as a binary decision diagram with a depth of 3. Taken from [7].

a binary search operation to find the variable. Variables and probabilities can be added, deleted, or updated with a single query.

3 RESEARCH QUESTION

The research question this research aims to answer is:
Under which circumstances would the caching of probabilities in a BDD type of a probabilistic database result in better performance?
The main question can be answered by first answering these sub-questions:

- SQ1: Which queries are impacted performance-wise by caching probabilities in a BDD type?
- SQ2: Which database parameters are impacted performance-wise by caching probabilities in a BDD type?

4 METHODOLOGY

4.1 Research Objective

The objective of this research is to measure the performance benefits of caching. Here, caching is done by adding a new column to a table, which stores the probability of the binary decision diagram in the same row. It is expected that this will make retrieving probabilities faster, but if the probabilities in the dictionary are updated, then the cache also needs to be updated. An extra column will probably also negatively impact the performance of most regular PostgreSQL queries, such as inserting rows into the table. The goal of the experiments is to determine to what degree these queries are impacted.

4.2 Standardization

To find the performance difference of caching, different queries need to be run on a table with caching and one without caching. To make sure results can easily be compared with one another, the parameters suspected to have a significant impact on query performance have been standardized in Table 2.

Each parameter has a base value and a test range.
Base values are the default value of a parameter, from which the database will be set up. For each test case, unless stated otherwise, assume the database is constructed using the standard parameters listed. Each base value was chosen to be the value within the test range with the fastest average execution time, such that experiments can be run as fast as possible.

Table 2. Base values and test ranges for all database parameters

Variable	Base value	Test range
Amount of possibilities per variable (AOPPV)	10	1 - 10
Dictionary size	2520	2520 - 25200
Row count	2520	2520 - 25200
Column count	2	2 - 20
BDD Size	1	1 - 10
BDD combinator	‘&’	’&’, ’|’, ’!&’, or ’!|’

Table 3. Dictionary with size 6 and AOPPV of 2

aaa=1:0.5	aaa=2:0,5
aab=1:0.5	aab=2:0,5
aac=1:0.5	aac=2:0,5

Test range is the range across which a parameter will be varied to measure the impact of that specific parameter on performance. When varying a parameter, all other parameters are fixed at their respective base values. The upper limits of the ranges are not that high, however, higher upper limits would take longer to execute and would lead to increasingly diminishing results. The expectation is that assumptions can be drawn for larger values by extending observed trends.

BDD size is the number of variables in the binary decision diagram, and BDD combinator is the combinator that provides the relation of all variables between each other within a BDD. An example of a BDD of size 4 with combinator ’&’ (indicating an ’and’ relationship) would be:

$$BDD('aaa = 5 \,\&\, (aab = 5 \,\&\, (aac = 5 \,\&\, aad = 5))')$$

The test range of column count, the number of columns in a table, begins at 2. This is because in addition to the BDD column, an ID column was chosen to also always be present. The table with caching has one more due to the cache being stored in a separate column.

The dictionary can be represented by the number of variables and the amount of possibilities per variable (AOPPV). The AOPPV is consistent across all variables in the dictionary.
To test the size and complexity of the dictionary separately, the two parameters used to create the dictionary will instead be AOPPV and dictionary size, the number of variables multiplied by the AOPPV. An example of a dictionary with size 6 and AOPPV of 2 is shown in Table 3.
The base value of AOPPV is set at 10, because initial testing showed that a higher AOPPV resulted in faster dictionary lookups. The average AOPPV is not expected to exceed 10 in a regular database, so 10 is chosen as the highest AOPPV that will be tested. Dictionary size is set at 2520, as that is the lowest common multiple

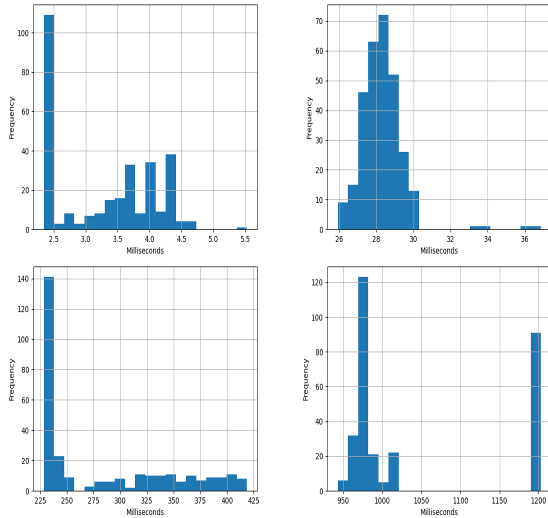

Fig. 2. Show server delay. In the first image, the query was run on a table with 25 rows, the second 252 rows, the third 2520 rows, and the fourth 25200 rows. All histograms are composed of 300 runs and 50 bins.

of 1 through 10. This way AOPPV can be varied freely without having to change the dictionary size.
Row count, the amount of rows in a table, is also set at 2520 for similar reasoning.

4.3 Server delay

During initial testing, it was found that query execution times are significantly affected by delays on the server side. A query that calculates the probability of every BDD in a table was run on several tables, shown in Figure 2.

These histograms show that queries are sometimes impacted by delays. The delays appear random, especially at lower execution times, thus can not be filtered out systematically. To combat this, all query results will be averaged over 100 runs.
Unfortunately, this does not completely get rid of inconsistency, as delays tend to group up. Consecutive runs of a query tend to have the same execution time and delay, and the length of each sequence of delayed runs also seems random.
The histograms in Figure 2 show that a variation of at least 20% needs to be accounted for, and at least 100% at lower average execution times.

4.4 Performance measurements

Measuring the performance of a query is done by analyzing the time the server took to execute the query. This is done by prefixing 'EXPLAIN ANALYSE' to each query, which creates a query plan and shows the execution time. All execution times are measured using this prefix.

From here on out, all query results represent the execution time of a query on a table without cache minus the execution time on a table with cache. This will be called the 'benefit' of a query, representing time gained by caching per execution of that query.

4.5 Sub Question 1

SQ1 is stated as: Which queries are impacted performance-wise by caching probabilities in a BDD type?
The only way the performance of queries can be impacted by caching is if a clear split can be observed between performance on a table with caching and a table without caching, or in other words, if the benefit of a query deviates significantly from 0.
Queries can be split into two different types: DuBio-specific queries and regular PostgreSQL queries.

The only DuBio-specific queries that are impacted by caching are queries that use or calculate probabilities, or queries that update the dictionary. The former involve calculation from BDD and dictionary if there is no cache, and retrieval of cache if there is cache. The latter causes the cache to become outdated, which necessitates updating the cache.
To measure these operations, representative operations have been selected. Retrieving Probabilities on a table with cache means retrieving the 'probability' column, and on a table without cache means calculating the probabilities for each BDD. Updating cache on a table with cache means updating the dictionary and then updating all cache, and on a table without cache means only updating the dictionary.

Most regular PostgreSQL queries are suspected to be negatively impacted by an extra column, but there are too many to test. Most queries run on a table are either retrievals or insertions, therefore representative queries can be chosen for each type. The queries chosen are selecting all columns of a table, and inserting 'x' columns into a table. 'x' is initially set at 1000, as 1000 is deemed high enough to see a relation.

All SQL queries can be found in Appendix B.
If the benefit of a query consistently and significantly differs from 0 then the query will be taken into account for the research question. To test this, all queries will be run while fixing all database parameters to their base values, only varying row count across its test range.

4.6 Sub Question 2

SQ2 is stated as: Which database parameters are impacted performance-wise by caching probabilities in a BDD type?
The six database parameters to be tested are the parameters described in Table 2. To answer the question, the benefit of all impacted queries found in SQ1 will be tested while varying all parameters over the test ranges also outlined in Table 2. Measuring the change in benefit of each parameter individually is done by fixing all other parameters to their base values.

If the benefit of a query consistently and significantly differs between different values for each parameter, the parameter can be considered to influence the benefit of that query. Thus the parameter will be taken into account for the research question.

4.7 Research Question

The research question is stated as: Under which circumstances would the caching of probabilities in a BDD type of a probabilistic database result in better performance?

The results from SQ1 and SQ2 will tell which queries and parameters significantly impact whether caching is advisable.

To answer the research question, the benefit of all selected queries will be compared multiple times, each time with random amounts for each parameter. The amounts for each parameter are randomly chosen between the test ranges shown in Table 2, except for row count and dictionary size.

Row count and dictionary size are instead chosen logarithmically, to test lower amounts more. This is due to the difference between 10 and 13 being magnitudes higher than between 1000 and 1003. Both are also capped at 10000, as execution times would become too large if both parameters randomly were chosen to be a high number.

Row count and dictionary size do need to be adjusted slightly if they are not divisible by AOPPV. Therefore the full process of choosing values for these two parameters is as follows: First a random floating point number x between 1 and 4 is chosen, then the following function is applied:

$$value = 10^x - 10^x \mod AOPPV$$

4.8 Implementation Details

Recreating the experiments can be done by running the code [6]. Some specific implementation details are listed below:

(1) The first column in a table is always the IDs, and the last (or second to last when caching) is always the BDDs. When adding a column, a 'VARCHAR (255)' column is inserted in the middle.
(2) The names of all variables in the dictionary are chosen consecutively from the set:
$$\{\text{'aaa'}, \text{'aab'}, \ldots, \text{'zzz'}\}$$
as seen in Table 3.
(3) To test all possibilities of a variable, BDDs are composed differently according to the row number. A BDD of size 1 is composed as such:
$$BDD(\text{'}aaa = X\text{'}),$$
where
$$X = rowNumber \,\%\, AOPPV$$
(4) The first variable in any BDD is always 'aaa', then 'aab', and so forth, as seen in the example BDD in section 4.2.
(5) To revert the state of the database to where it was before the query was run, some queries in Appendix B are surrounded by 'BEGIN;' and 'ROLLBACK;'.
(6) Updating the dictionary is done by setting the probability of 'aaa=1' to 100% and all other possibilities of 'aaa=x' to 0%, as seen in Appendix B.

5 RESULTS

5.1 Sub Question 1

The results can be found in Figure 3. This shows that the DuBio-specific queries have a clear impact on whether to cache or not, however the other two seem to hover around the 0ms mark. Selecting seems to average -0.025ms, and inserting 1000 rows seems to average 0.4ms per query execution. The benefit of the DuBio-specific queries both range between the hundreds and the thousands of milliseconds. This means that the expected randomness from the DuBio queries significantly overpowers the variation of regular PostgreSQL queries, so should not be taken into account for further testing.

Inserting rows into a table can be influenced by row count in two different ways. Because the number of rows the table already possessed before inserting 1000 rows does not significantly impact the benefit of this query, a new test was done where the number of rows inserted on a table constructed using the base parameters was varied between 2520 and 25200. The results of this are pictured in Figure 4.

This figure has a much higher variance than the previous insertion query, as the query execution times from which the difference is taken is much larger. The maximum observed difference is 10.9 milliseconds when inserting 25200 rows, which still gets significantly overshadowed by the randomness obtained from running the DuBio functions on tables with 25200 rows.

This can be seen by the fact that at a row count of 25200 rows, retrieving probabilities results in a benefit of 2378 milliseconds, and if a variation of 20% needs to be taken into account, 10.9 ms only accounts for 2.2% of that randomness.

For all of these reasons, only the DuBio queries significantly impact the efficacy of caching probabilities in a BDD type, and are the only queries that will be taken into account for the research question.

5.2 Sub Question 2

From the results of SQ1, only the retrieving probabilities and updating cache queries need to be taken into account. These queries are incredibly similar though, as cache updating is done by calculating the probabilities of all BDDs in a table, which is the same thing that happens when retrieving probabilities on a table without cache. If a significant difference can be observed with the retrieving probabilities query, the same will hold for updating cache. Therefore the only query that needs to be tested for all

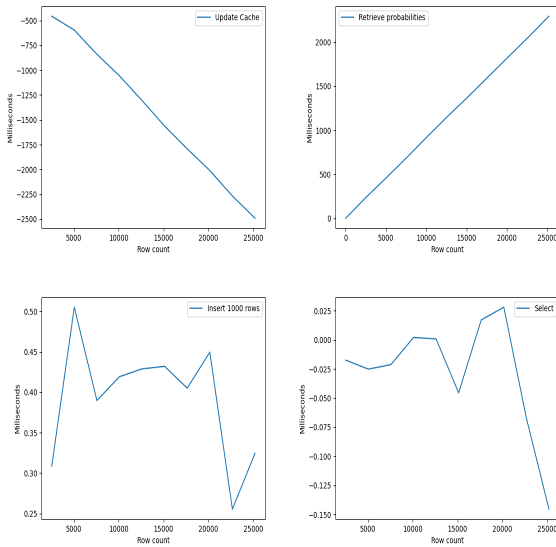

Fig. 3. Representative queries run on a table while varying row count between 2520 and 25200

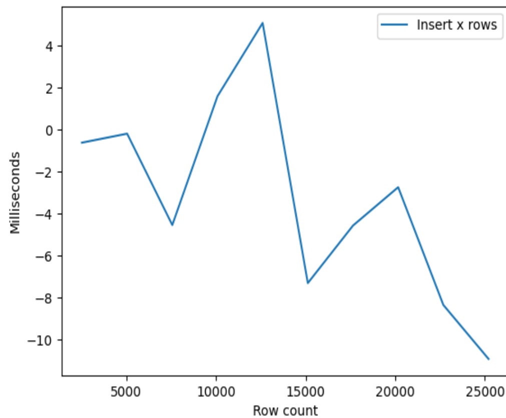

Fig. 4. Inserting 2520 - 25200 rows into a table

parameters is retrieving probabilities.

Row count was already tested in Figure 3, and a clear correlation can be seen.
Dictionary size, as pictured in Figure 5, also shows an extremely clear correlation between dictionary size and execution time. The same goes for AOPPV in Figure 6.

BDD size and combinator have been combined in Figure 7. Conclusions can not yet be drawn though, as all results seem to only significantly differ through a sequence of delayed executions. To test this definitively, a test was run comparing the largest

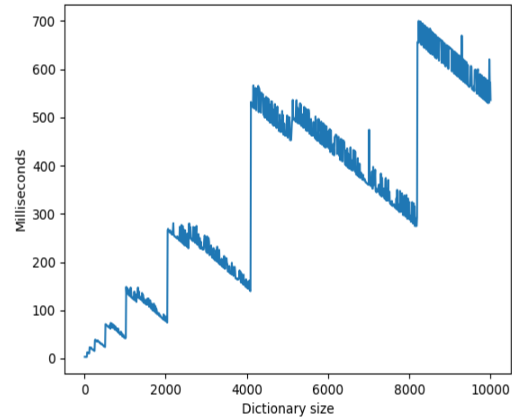

Fig. 5. Execution times when varying dictionary size

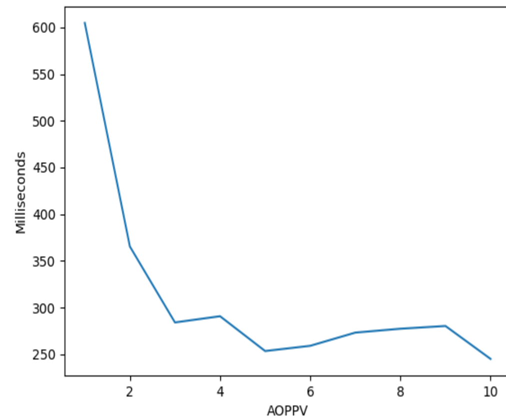

Fig. 6. Execution times when varying AOPPV

difference shown in Figure 7 ('not &' and 'not |' at BDD size 8). This resulted in a normal distribution with a mean difference of 0.03 milliseconds, from which it can be reasonably assumed these two combinators have the same execution times.
BDD size was also rerun, and a new graph was found in which all execution times vary inconsistently between 241 and 254. This is a percentage difference of only 5.4%. A 5.4% percent difference is indistinguishable from randomness when single executions can vary by up to 20%. The same goes for column count, which has a 3.1% percent difference.

For all the reasons listed above, only AOPPV, dictionary size, and row count will be taken into account for the final research question.

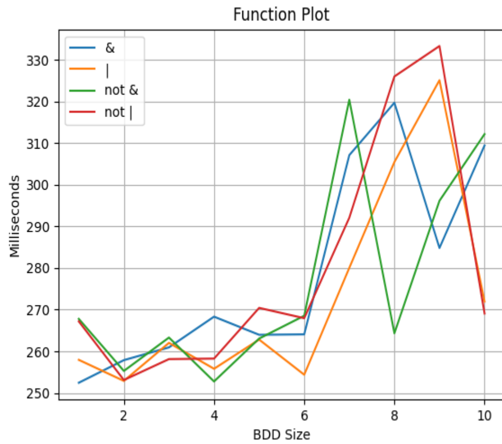

Fig. 7. Execution times when varying BDD size and combinator

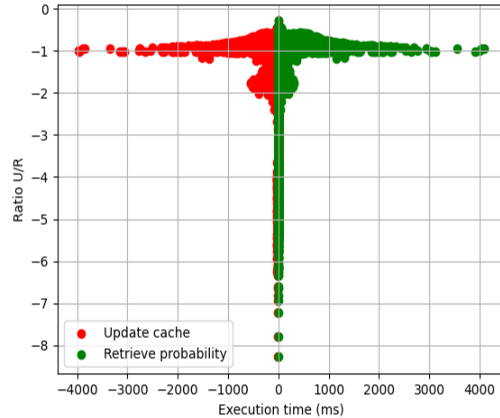

Fig. 9. Comparing the execution times of Updating cache (U) and Retrieving probabilities (R) against the ratio U/R

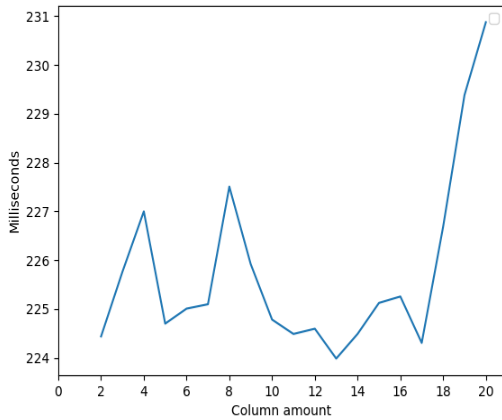

Fig. 8. Execution times when varying column amount

5.3 Research Question

From the results of SQ1, only the retrieving probabilities and updating cache queries need to be taken into account, and from the results of SQ2 only AOPPV, row count and dictionary size need to be taken into account.

Both remaining queries; retrieving probabilities (R) and updating cache (U), were run with random parameters 5000 times. The results of this are shown in Figure 9. This clearly shows a few things.

The first is that at larger execution times the execution times of R and U become roughly equal, as the ratio tends to one. This means that for large tables and dictionaries, caching becomes a good idea if one retrieves more than they update.

The second notable observation is that, with lower execution times, more variation is expected for the ratio between U and R. Indubitably this is at least partly due to delays becoming a more significant factor at lower execution times, as seen in section 4.3.

The scale of Figure 9 makes it hard to observe the variation at lower execution times. Figure 10 shows this more clearly, by multiplying all values of U by -1 and then logging both U and R. In this new figure the ratio between R and U is very visible. The lower the ratio is below -1, the higher the distance between green and red data points. This graph also shows that the ratio between U and R more consistently approaches -1 when U > $10^{1.5}$ and R > 10^{1}, as the ratio then stays between the range $-2 < ratio < -0.5$.

The ratio can also be mapped against the parameters, to see how they influence the ratio, as is done in Figure 11. Here logged dictionary size and row count are shown against the ratio.

Each green dot is linked to a blue dot with the same ratio value. Using this information, it is clear to see that most extreme ratios (lower than -2) are composed with a combination of a dictionary size lower than $10^{1.8}$ and a row count between 10^{2} and $10^{3.7}$.

Mapping AOPPV onto Figure 11 does not show any new information, all extreme ratios are spread out over the whole range of AOPPV.

The ratio could also be logged to show the data points between 0 and 1 better, however, no new trends can be found by doing this. There are also only 24 data points out of 5000 above -0.5, suggesting that they are the products of server delay.

6 CONCLUSION

In this paper, a comprehensive performance analysis of caching probabilities next to binary decision diagrams in the probabilistic database DuBio was conducted. The comparison has been based on query execution time reported from the server while accounting for server delay within those execution times. First was established

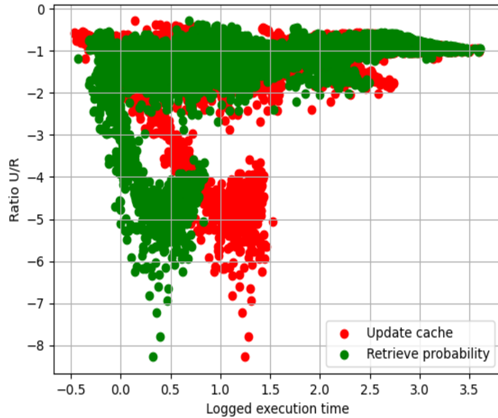

Fig. 10. Comparing the logged execution times of Updating cache (flipped across the y-axis, $-U$) and Retrieving probabilities (R) against the ratio U/R

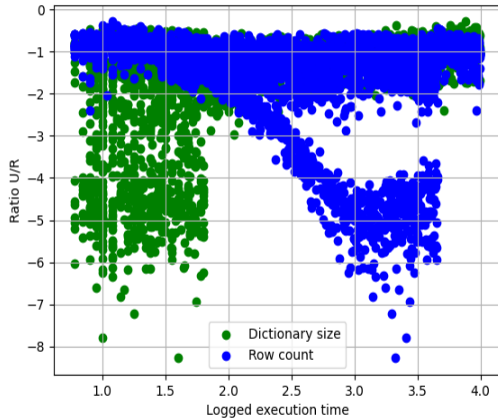

Fig. 11. Comparing the logged values of dictionary size and row count against the ratio U/R

that there are only two queries that make a significant performance difference between caching and not caching, namely retrieving probabilities and updating cache. In most cases, caching tends to improve the retrieval of probabilities to about the same degree that it negatively affects updating cache. This does not hold in specific conditions outlined in section 5.3. Following this observation we recommend database builders to include caching if users are expected to query probabilities more than that they update the dictionary.

7 DISCUSSION

The recommendation from the conclusion only applies to the worst-case scenario.
Assume both updating cache and retrieving probabilities take 1000 milliseconds. If a user were to retrieve slightly before an update, no slowdown would be experienced. If they were to retrieve 800ms after an update was started, they would only experience a slowdown of 200ms. However, they would still be better off with cache as retrieving probabilities is always 1000ms faster with cache. Updating cache is usually done in batches, after new information has been discovered. This means that during usual operation, only positive benefits will be perceived.

The reason for the patterns in some graphs is not currently known, however there is a theory based on the collected data as to why Figure 5 has the shape it does. The dictionary is looked through using binary sort, and BDDs are only created using the first few variables in the dictionary. This creates a situation where, when the dictionary size exceeds 2^x, a huge increase in execution time is accompanied.
This could have been mitigated by distributing variables over all BDDs in the table such that the average amount of lookups is equal to the average amount of lookups for that dictionary size. It is speculated that using this method the graph would take the shape of a linear function. No significantly different outcomes are expected however.

8 FUTURE RESEARCH

Four factors are limiting the scope of this paper which could be researched further.

First off, this research is solely focused on one table, instead of a whole database. This paper does not research whether database structure affects query execution time, so conclusions in this paper cannot automatically be extended to multiple tables. Whether it can is outside the scope of this research.

Second, this research is solely focused on a whole table, without considering what happens when only a specific part is queried. Updating the dictionary could be done on a variable-to-variable basis, so updating all cache is not necessary. The same is probably also true for retrieving probabilities, calculating the probability of a subset of the BDDs in a table is probably faster than calculating the probability for all BDDs.

Third, storing probabilities next to BDDs can be done in a multitude of ways. This research is only concerned with adding a new column next to the BDD column, however storing it in the same PostGreSQL object is also a possibility. Researching the best way to update cache could be especially promising for future research. If cache can be updated concurrently, such that it would not lock all tables during the operation, caching would become purely beneficial.

Lastly, only a few queries were tested, even though a lot more had a small impact on performance. These were currently omitted because the benefit of all of these queries were indistinguishable from delay for queries with larger execution times. The problem of adjusting for uncertainty in query execution time has already been

subject to a great deal of attention (eg. [1], [8]). If such methods can be applied successfully here, other queries could become viable for research.

REFERENCES

[1] M. Akdere et al. “Learning-based query performance modeling and prediction”. In: *ICDE, pages 390–401* (2012).
[2] Sheldon B. Akers. “Binary Decision Diagrams”. In: *IEEE TRANSACTIONS ON COMPUTERS, VOL. c-27, NO. 6* (1978).
[3] Madeleine Mauz. *Using Postgres Hash Table Extension in Compiled Functional-Style SQL UDFs.* 2022.
[4] Stephen E. Richardson. “Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation”. In: *11th Symposium on Computer Arithmetic* (1992).
[5] Anish Das Sarma, Martin Theobald, and Jennifer Widom. “Exploiting Lineage for Confidence Computation in Uncertain and Probabistic Databases”. In: *2008 IEEE 24th International Conference on Data Engineering* (2008).
[6] Matteo Schut. *DuBio-data-generator.* https://github.com/DutchOzz/DuBio-data-generator. 2024.
[7] Dan Suciu. *Probabilistic databases.* Morgan Claypool, 2011.
[8] Wentao Wu et al. “Uncertainty Aware Query Execution Time Prediction”. In: *Proceedings of the VLDB Endowment (PVLDB), Vol. 7(14)* (2014).

A APPENDIX A

An attempt was made to model the performance benefit of caching. This was done for the retrieving probabilities query by extracting functions from Figures 3, 5, and 6, such that they represent the modelled benefit in milliseconds when varying only one variable. Extracting the functions was done loosely, as execution times could vary.
The extracted functions were:

- AOPPV: $f(a) = 10^{-1.28*\log a+2.61} + 229$
- Dictionary size: $g(d) = 2^{\lfloor 2\log d/7.36 \rfloor} - (0.06175*(d-2^{\lfloor 2\log d \rfloor})$
- Row count: $h(r) = 0.091 * r + 3.24$

They all go through one point, the point at which the input variable is at its base value, outlined in Table 2. The execution time at this point was determined to be 232,2.
The functions were then combined such that the combined function represents the rate of change from 232,2, by tracking the percentual change from 232,2. The complete formula was determined to be as follows:

$$R(a, d, r) = 232,2 * f(a)/232,2 * g(d)/232,2 * h(r)/232,2$$

Which can be reduced to:

$$R(a, d, r) = 1/232,2^2 * f(a) * g(d) * h(r)$$

This function was then tested against actual values by choosing random values for parameters in the same way as in section 5.7. The function worked very well at higher execution times. At lower execution times (<200) however, more than half of the predicted execution times were more than 1.5 times off of the actual execution times.

For this reason, this method was not used for answering the research question. The function likely did not work due to high randomness in the amount of server delay present at lower execution times, as demonstrated in Figure 2. If randomness was able to be predicted, or reduced by a different server, this method could likely have been able to accurately predict query benefit.

B APPENDIX B

Table 4. Queries with the SQL queries run to test them.
AOPPV = 3, Column count = 4, and BDD size = 1 for simplicity

Query	Probabilities cached	SQL query
Retrieve probability	yes	SELECT _sentence, probability FROM table
Retrieve probability	no	SELECT t._sentence, prob(d.dict, t._sentence) AS probability FROM table t, _dict d
Updating cache	yes	BEGIN; UPDATE ._dict SET dict = upd(dict, ’aaa=1:1;aaa=2:0;aaa=3:0’); ROLLBACK;
Updating cache	yes	BEGIN; UPDATE ._dict SET dict = upd(dict, ’aaa=1:1;aaa=2:0;aaa=3:0’); UPDATE table SET probability = prob(d.dict, _sentence) FROM ._dict d; ROLLBACK;
SELECT	both	SELECT * FROM table
INSERT INTO x	yes	BEGIN; INSERT INTO table (id, a, b, _sentence) values (1, ’a1’, b1’, BDD(’aaa=1’))::x times; ROLLBACK;
INSERT INTO x	no	BEGIN; INSERT INTO table (id, a, b, _sentence, probability) values (1, ’a1’, b1’, BDD(’aaa=1’), 1/3)::x times; ROLLBACK;