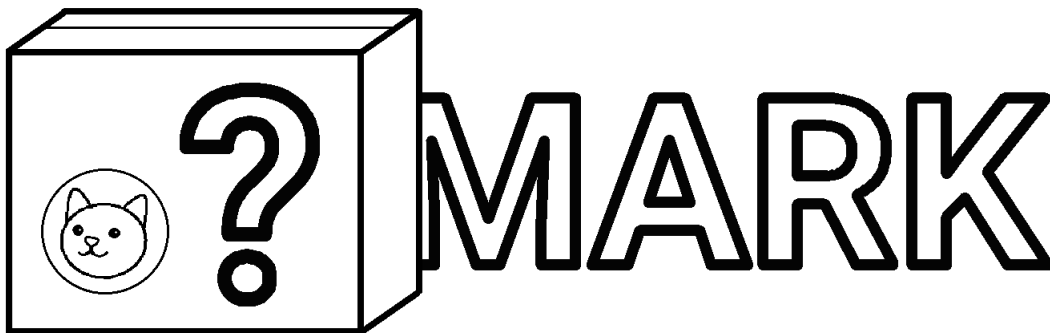# A

The QuestionMark Probabilistic Benchmark

Welcome to the QuestionMark manual. This document provides additional details on the benchmark, as well as a roadmap on how to use it and adapt it for own use. The scientific substantiation of this benchmark can be found in the accompanying thesis[1].

Figure A.1: The QuestionMark logo

As there is no standard available for benchmarking probabilistic databases, Question-Mark aims to cover a wide range of aspects of the tested probabilistic database management system (DBMS). This benchmark provides a convenient way to test various probabilistic database management systems and get insights on their performance. Since the queries provided in this benchmark are written in a pseudocode like language, queries can easily be translated to any probabilistic query dialect. Additionally, it provides clear guidance on how the parameters can be adapted to approach any real-world application as close as possible. QuestionMark © 2023 by the University of Twente is licensed under Attribution 4.0 International

---

[1]Zandbergen, N. (2023) *QuestionMark: Designing a benchmark for probabilistic databases.* M.Sc. Thesis, University of Twente.

For the benchmark, two Python programs have been developed. Both these programs — The Dataset Generator and The Probabilistic Benchmark — need to be run to execute the benchmark. Section A.1 describes the dataset generator. Section A.2 describes the benchmark. The step by step instructions provided in this manual can also be found in `MANUAL.md` in the respective python program. This benchmark natively supports the probabilistic database management systems MayBMS and DuBio. Running the benchmark with any other probabilistic DMBS requires manual adaptation of these programs. More details on how to do this are provided in the QuestionMark Python programs and in Section A.6.

If you want to use this benchmark, allow for a total running time of three to eight hours, depending on the dataset size and included uncertainty. The benchmarking procedure consists of the following phases:

1. (approx. 60 minutes). Reading through this manual and understanding the product.

2. (30 to 180 minutes). Running QuestionMark: The Dataset Generator.

3. (30 to 90 minutes). Running QuestionMark: The Probabilistic Benchmark.

4. (approx. 60 minutes). Digesting the results and drawing conclusions.

When adding a new non-supported DBMS, the implementation of changes require an additional fifteen hours. Including a new non-supported non-PostgreSQL based DBMS, another additional fifteen hours should be taken into account.

A.1. QuestionMark: The Dataset Generator

QuestionMark: The Dataset Generator is a Python program that generates the dataset required for running QuestionMark: The Probabilistic Benchmark. This program can be downloaded from

> https://github.com/utwente-db/QuestionMark/tree/thedatasetgenerator

This dataset generator prepares the dataset required for running the benchmark test. During the dataset generation phase, a dataset is produced that approaches the real-world scenario for which this benchmark is run. For this, parameters can be tuned. This program follows the general product matching workflow, which is as follows.

1. *Data Preparation.* The data is standardised and cleaned. A uniform data structure is applied.

2. *Search Space Reduction.* Since the time needed for evaluating all possible combinations grows exponentially with the dataset size, the search space for possible matches needs to be reduced to allow for efficient matching.

3. *Attribute Value Matching.* The similarity of the remaining data tuples is determined using a syntactic and semantic means, which produces a comparison vector per data attribute.

4. *Classification.* A decision model then determines the similarity score of a data tuple. This score is compared to the set thresholds to determine whether it is a

matching tuple, possibly matching tuple, or non-matching tuple.

5. *Verification.* The performance of the applied product matching algorithm can be verified using standard performance metrics.

A.1.1. The Dataset Generator Roadmap

After having downloaded the program from GitLab, the generation of the probabilistic dataset can begin. When completing the listed steps, the dataset required to run QuestionMark: The Probabilistic Benchmark is obtained.

1. *Downloading the WDC datasets.* First, the base dataset should be downloaded. For additional details on this dataset, see Section A.1.2. To include this dataset in the program, create an empty folder `datasets` in the main project repository. Next, go to the WDC dataset website, scroll to the bottom of the page and download `offers_corpus_english_v2.json.gz` and `all_gs.json.gz`. Include these in the empty datasets folder. If desired, you could also download the samples from the WDC dataset website to get an impression of the dataset.

2. *Preparing the dataset generator.* To create a dataset that approaches a given real-world scenario as much as possible, first the parameters of the generator need to be set. For an explanation on these parameters, see Section A.1.3. When these parameters are set, a database connection should be established. The database of choise should be running and accepting connections. To connect QuestionMark: The Dataset Generator to your database, create a new file called `database.ini` and fill in the credentials according to the defined structure in `database.ini.tmpl`. QuestionMark: The Probabilistic Benchmark is created with PostgreSQL-based Database Management Systems in mind. In case the DBMS you want to benchmark is not PostgreSQL-based, please see Section A.6.2. If you want to benchmark a system other than DuBio or MayBMS, please see Section A.6.1.

3. *Running QuestionMark: The Dataset Generator.* Once the preparations are done, the dataset generation can begin. To run the benchmark, go to *manual.py* and run the script.

4. *What the benchmark does.* During the process of generating the dataset, several phases will be passed. If it is indicated that a smaller dataset will be used, this new dataset is produced first. To do this, a pseudo-random selection of offers is chosen from the dataset. This ensures that the same dataset will be produced each time the benchmark is run on a specific size. Next, this dataset is sorted and a dictionary is created for easy lookup. The offers present in the dataset are then put in blocks. For this, two blocking algorithms are available. First creating blocks reduces the time required to evaluate if offers should be put in the same cluster. More information on this process can be found in Section A.1.4. After the blocks are created, all offers in a block are matched and provided with a probability score. This probability indicates the likelihood that the offer belongs in a cluster, and whether its attributes are likely the correct ones. More information on this process can be found in Section A.1.5. When the clusters are created, a database representation is created and the offers are added to a probabilistic

DBMS. Finally, some preparatory queries are run.

5. *Continue with benchmarking*. The dataset is prepared! Go to QuestionMark: The Probabilistic Benchmark to continue with benchmarking. Optionally, performance tests could be run before continuing the benchmarking process.

6. *Running performance tests*. QuestionMark: The Dataset Generator also comes with a performance evaluator. This evaluator can aid in setting the parameters correctly, such that the produced dataset approaches that of the real-world as closely as possible. Several performance tests have already been run. Open `performance/performance.txt` to get insights into the behaviour of the parameters and the performance of the implemented algorithms. To run the performance tests, set the `performance` parameter to true and run the script in `manual.py` again.

A.1.2. WDC Product Data Corpus and Gold Standard

This program digests the WDC Product Data Corpus and Gold Standard for Large-Scale Product Matching, Version 2.0 to generate a probabilistic dataset from it. The WDC dataset is a large public training dataset for product matching. It is produced by extracting schema.org product descriptions from 79 thousand websites, which provides 26 million product offers. Besides the full dataset, an English language subset is offered. This subset consists of 16 million product offers. This dataset is provided with a clustering. The 16 million product offers in the English subset are categorized in 10 million clusters. Each cluster contains offers of the same product found on different websites. There are roughly 8.5 million clusters with size 1, one million clusters with size 2, and 400.000 clusters with size 3 or 4. Clusters of a size greater than 80 are filtered out of the dataset, as these are likely noise.

For this benchmark, an adaptation of the English subset is used. The dataset was adapted to include a probabilistic clustering. More information on the dataset and details on why this dataset was chosen can also be found in the accompanying thesis.

A.1.3. Dataset Generator Parameters

During the dataset generation phase, there are multiple parameters that can be tweaked. The different parameters and their effect on the resulting dataset are listed below. In order to fit the benchmark to the requirements of a specific application, it should be tailored to represent this real-world scenario as closely as possible. This can be done by tweaking the various parameters of this benchmark. The parameters 'dataset size' and 'upper phi and lower phi' have a high influence in the extent to which the produced dataset resembles the real-world application for which this benchmark test is run.

- *DBMS*. Determines into which database management system the generated dataset should be loaded and what preparatory queries need to be run.

- *Dataset size*. Determines the amount of offers included in the dataset. A percentage of the dataset can be determined to two decimal places. The offers for the new smaller dataset are pseudo-randomly chosen, so that the same dataset is returned for multiple runs. This ensures reproducibility of the results. The

full dataset contains 16 451 499 offers. The smallest dataset that can be generated is 0.01% of the full dataset, which produces an initial dataset of 1653 offers. Choose this value to generate a dataset with a size similar to that of the dataset being digested by the real-world application.

- *Whole clusters.* Determines whether the offers chosen from the larger dataset to include in the new smaller dataset are pulled from entire clusters or not. Including entire clusters increases the uncertainty of the data.

- *Word distance measure.* Determines the manner in which the distance between two words or sentences is calculated. This measure is used during the blocking phase on the attributes determined as Blocking Key Values and on all suitable attributes during the matching phase. The implemented distance measures are Levenshtein, Jaro, Jaro-Winkler, Hamming and Jaccard.

- *Blocking key values.* Determines the attributes that are included to determine the similarity of two offers during the blocking phase. Including more attributes provides a better blocking performance, but at the cost of a higher runtime.

- *Blocking similarity threshold.* Value between 0 and 1 that represents the distance between two offers. Evaluated offers with a distance lower than the threshold are included in the same block.

- *Blocking window size.* Determines the size of the sliding window. Within a window, the distance between the first and last offer is determined. This value influences the runtime.

- *Maximum block size.* Poses a restriction on the block size. Increasing this value improves the performance. As the matching phase includes a calculation with factorial time complexity, this size should not exceed six. Five is advised.

- *Matching attributes.* Determines the attributes that are used to obtain the distance between two offers during the matching phase. Including more attributes improves the performance, but increases the runtime.

- *Matching attribute weights.* Determines the weight of each attribute to calculate the final distance score. This can be tweaked to improve the performance. It has no effect on the runtime.

- *Upper phi and Lower phi.* Determines the upper and lower threshold of the distance measure. If the distance between two offers is greater than the upper phi, the two offers are certainly not the same product. If the distance is smaller than the lower phi, the two offers are certainly the same. Increasing the gap between the values ensures less false matches or non-matches, but increases the computational complexity in later phases and during querying. A smaller gap can be used to artificially reduce the uncertainty in the dataset. This value should be carefully chosen, as this influences to what extent the produced dataset imitates the data being digested by the real-world application.

A.1.4. Blocking Algoritm

To obtain a time-efficient product matching, the search space for matching pairs should be reduced. Disregarding this step results in quadratic time complexity dur-

ing the product matching phase. Having 16 million products in the dataset, this step is thus essential for a time-efficient product matching. For this step, *filtering* or *blocking* can be used. QuestionMark: The Dataset Generator makes use of a blocking algoritm.

A selection of two Rule-Based Blocking techniques was implemented on the dataset to verify which algorithm performed best. These are Incrementally-Adaptive Sorted Neighborhood (ASN) and Improved Suffix Array (ISA) Blocking. Tests executed using the implemented performance evaluator indicate that ASN is the best blocking algorithm for the dataset used. The ISA algoritm is still available for use.

The ASN blocking algoritm works by sliding a window to roughly determine what offers are possible matches. For this, a sorted dataset is required. During each iteration of the algoritm, a block is created. The sliding window is placed at the first offer from the sorted list that is not yet in a block. When the start of the window is set, the enlargement phase is entered. During this phase, the window will iteratively increase in size. This is a fixed increase. After each iteration, the blocking algorithm determines the similarity score of the first and last offer in the window. If the distance between the two offers is smaller than the set threshold, the window is enlarged and a new similarity score is determined. If the distance is higher, the retrenchment phase is entered. During the retrenchment phase, the sliding window will decrease one offer in size and calculate the similarity score between the first offer in the window and the new last offer. Once the similarity score rises above the threshold, the block is created.

A.1.5. Matching Algoritm

During this phase, either an algorithm based approach or a machine learning based approach could be used. When product matching with either approach, the matching can be performed only on the product title or on all available information, i.e. including the product attributes. Only using the product title provides simplicity and speed, but at the cost of a lower precision.

For QuestionMark: The Dataset Generator, the Attribute-Based Entity Resolution approach is used as the foundation of the implementation. For this research, a comparison vector is generated from all attributes of an offer. Within each block, all possible offer combinations are generated and the distance between these offers is then provided by the vector. For simplicity, this vector is combined to a single distance score. The weight of each attribute for this final score is adaptable.

As the benchmark designed in this research is based on probabilistic data, an additional layer had to be build on top of the basic algorithm to include a probabilistic model in the final clustering. The creation of the various probabilistic clusters is based on the possible worlds model. For each block, a matching graph is created and the matching score of each edge is evaluated. Blocks containing only a single offer are always true, so are submitted to a cluster directly. When a block contains multiple offers, their matching score is evaluated. Here exists three possibilities:

- the matching score of their edges all lie above the upper threshold;
- the matching score of their edges all lie below the lower threshold;
- there are one or multiple edges between the two thresholds.

In the first case, the cluster is certain; there is only one possible world. In this case the full block becomes a new cluster. There does exist uncertainty between the correct value of the attributes, as these are likely different. In the second case, the cluster is also certain, as all offers are certainly different. In this case, each offer is put in a separate cluster. In the final case, world graphs should be constructed of the possible worlds. The amount of possible worlds created equals 2^n where n equals the amount of offers connected by an uncertain edge. When these world graphs are created, the inconsistent worlds are removed and the remaining worlds are included as different options for the same cluster. If an offer is certainly present in all worlds, this offer is added later to all generated world graphs. If an offer is certainly not present in all worlds, a separate cluster is created.

For the creation of the possible worlds, a naive implementation is used based on the theory presented in the accompanying thesis. Because of that, the space complexity for the creation of the possible worlds is factorial. This imposes a limit on the block size that can be digested by this algorithm. This imposes a maximum of six offers per block.

A.2. QuestionMark: The Probabilistic Benchmark

QuestionMark: The Probabilistic Benchmark is a Python program that runs a benchmark test for any probabilistic database management system. This program can be downloaded from

https://github.com/utwente-db/QuestionMark/tree/theprobabilisticbenchmark

This benchmark uses the dataset generated using QuestionMark: The Dataset Generator. Again, if a DBMS will be used that is not natively supported, the program needs to be adapted to allow its support. For this, see Section A.6.

A.2.1. The Probabilistic Benchmark Roadmap

After having downloaded the program from GitLab, the benchmark execution can begin. To properly run the benchmark, the following steps need to be followed. It is assumed that the process of QuestionMark: The Dataset Generator has been finished successfully and the dataset is available in a Database Management System that is accepting connections.

1. *Prepare the benchmark.* To connect to the dataset generated by QuestionMark: The Dataset Generator, create a file called `database.ini` and fill enter the credentials according to the defined structure in `database.ini.tmpl`. Then, the parameters for running the benchmark must be set. For an explanation on these parameters, see Section A.2.2. Again, if a new DBMS or a non-PostgreSQL based DBMS will be benchmarked, please follow the steps listed in Section A.6. To test the connection to the database, set the parameter `test` to True. Remember to set this value to False before running the benchmark test.

2. *Run the benchmark.* The benchmark is now fully prepared to be run. To run the benchmark and obtain the results, go to `manual.py` and run the script. For additional details on the queries included in the benchmark, see Section A.3.

3. *Reading the results.* When the benchmark execution is finished, the results can be viewed. The benchmark results are stored in `QM_metric_ results.txt` and `QM_query_results.txt`. Both files provide insights into the performance of the tested benchmark. For more instructions on how to digest and interpret the results, see Section A.4.

A.2.2. Benchmark Parameters

During the benchmarking process, there are also parameters that can be tweaked. The parameters during this phase are mostly related to the DBMS used and the functionality coverage of the benchmark.

- *DBMS.* Determines the database management system that will be used for the execution of the benchmark. Additional systems can be added when support for them is also added to the benchmark program.

- *Iterations.* Denotes the amount of times a query is run to obtain a runtime average from the queries. This is a global variable that is used for all queries. Increasing this number will provide a more precise outcome of the average run time, but at the cost of a longer benchmark execution time. The total amount of iterations is always +1 to create a warm start.

- *Show Query Plan.* Boolean value. If true, the query plan for each query is also provided with the benchmark result. Enabling this variable does not influence the execution time of the queries.

- *Timeout.* Ensures that queries that take too long to return an answer will be aborted. Once a query times out, this will be noted in the benchmark result and the next query is started. The current implementation abruptly stops the benchmark execution.

- *Queries.* A list that contains all queries from the benchmark. Depending on the goal of the benchmark run, queries that are not relevant can be removed from the benchmark run. Removing queries lowers the total time required to run the benchmark and focuses the results to what is important. The benchmark can also be run in several iterations, as to create several smaller, more focused benchmark results.

A.3. Benchmark Queries

This section discusses the queries included in the benchmark. The list of queries included in the QuestionMark benchmark can be found in Appendix A.7.

A.3.1. Queries

QuestionMark: The Probabilistic Benchmark offers a range of queries that can be used to test various types of systems. The queries are selected to cover the diverse possibilities of the dataset, but also include functionalities that are key to some more well-known probabilistic database management systems. The queries are sub-divided into queries that provide insight into the dataset, probabilistic queries that could be run more frequently and insert-update-delete queries. The table below provides a quick

overview of the included queries.

test 1	Simple query to test the connection.
insight 1	Retrieves the full dataset, gain insight in data structure and load handling.
insight 2	Provides insight into the dataset and probability handling.
insight 3	Provides insight into the distribution of cluster volumes.
insight 4	Gets the percentage of certain clusters.
insight 5	Gets the id and probability of the offers with a specific variable value or sentence.
insight 6	Gets the average probability of the dataset.
probabilistic 1	Gets offers with the probability of their occurrence.
probabilistic 2	Gets the expected count of the categories.
probabilistic 3	Gets the expected sum of the product ids per cluster.
probabilistic 4	Gets the sentence and probability per category.
probabilistic 5	Returns the most probable offer that is related to a specified string.
probabilistic 6	Returns all offers containing a specified string with a high uncertainty so these can be classified by human inspection.
insert update delete 1	Inserting a single row.
insert update delete 2	Inserting bulk.
insert update delete 3	Updates uncertainty.
insert update delete 4	Removes uncertainty.
insert update delete 5	Deletes a cluster.

A.3.2. Altering Queries

The queries presented in this benchmark are already translated and included in QuestionMark: The Probabilistic Benchmark in the dialects of DuBio and MayBMS. Please note that these query implementations are written with a dataset of 0.01% size. When producing a dataset of a different size, it could happen that the clusters used in those queries are not present in the produced dataset. It is thus of importance to always check the queries before running the benchmark. The following queries require special attention:

- *Query insight 5.* This query requires a specific variable or sentence to be defined. You could either define one that does not exist in the database, or choose one that does exist.

- *Queries probabilistic 5 and probabilistic 6.* This query uses pattern matching to obtain a selection of offers that satisfy that pattern. It is advised to query for anything that exists in the dataset.

- *Query insert, update, delete 3.* This query requires a specific cluster to be defined. Seek for any cluster of size four. Include its ID in the query and change the probability with variables accordingly.

- *Query insert, update, delete 4.* This query should also be run on any cluster of size four. Include the ID of each offer present in that cluster in one of the four queries. Include the cluster ID in the probability variables.

- *Query insert, update, delete 5.* This query removes a cluster. Search for a cluster with a sufficiently large size and include its ID. With the current limitations, a cluster with the largest size is advised.

- *Queries timing out.* During the benchmarking, it could happen that queries take too long to return an answer. In that case, the query is timed out. To verify whether the functionality of the query is supported, change the query to run on the 'part' table. This part table contains a small portion of the dataset. If the query still times out with this table, it could be worthwhile to decrease the size of this table even further. To do this, go to QuestionMark: The Dataset Generator and open `database_filler_[DBMS].py`. Then reduce the value in `LIMIT FLOOR()` in the first query of `prep_queries`.

- *Queries raising exceptions.* During the benchmarking, it could also happen that queries throw errors. When any query raises the exception `invalid memory alloc request size 1073741824` or `Ran out of memory retrieving query results`, it can also be worthwhile to run the query on the 'part' table. Most likely, reducing the dataset size that the query needs to digest removes this specific error. This verifies whether the functionalities in the query are supported by the system or not. It is worthwhile so remain critical when errors are thrown, sometimes a workaround can be found to still find a fix. Another option is to run the query on a database tool, as it could be that the DBMS cannot handle some requests from `psycopg2`.

A.4. Produced Results

After running QuestionMark: The Probabilistic Benchmark it is time to analyse the produced results. The benchmark provides information about the benchmark through the following metrics:

- The brevity of the query dialect.

- The query functionality coverage.

- The runtime of the queries.

- The probabilistic data overhead.

- The user friendliness of the system.

The benchmark produces two files when run: `QM_metrics_results.txt` and `QM_query_results.txt`. These two files contain the raw metric values that can be digested to obtain valuable information from. The metrics provide information on the effectiveness, efficiency and appeal of the tested software. `QM_metrics_results.txt` provides the raw data of four metrics in the main results part and provides an overview of all errors thrown while running the benchmark. If no errors were thrown, nothing will be printed and only the main part is visible. In `QM_query_results.txt`, all queries, their results, and their execution time are shown.

A.4.1. Metrics

The benchmark thus produces data for five metrics. This section provides additional details on each of these metrics.

Brevity of the query dialect. This metric is determined by the total amount of characters needed for all queries and gives insights into the succinctness of the query language. A more succinct query dialect often requires less time to write queries with and is often easier to understand. This metric value is obtained by iterating over all queries and adding their character count. Spaces are removed from the calculation. Optionally, characters can be removed from specific queries. For example in query `IUD_1_rollback` offers are added to the database. As the data that represents the offer is not indicative of the complexity of the query language, the amount of characters used for that representation is subtracted from the total character count for that query.

Query functionality coverage. This metric provides insight into the functionality coverage of the database system and is determined by multiple sub-metrics. When running the queries to obtain their results and runtime, it can happen that a specific functionality is not supported or the database system cannot handle the load required to execute the query. In these cases, the system returns an error. The error raised during execution are stored and printed as the query result. After the benchmark execution has finished, an overview table is created that indicates what queries finished execution and which threw an error. The percentage of successful queries is then also determined. For each query that threw an error, it also indicates what query functionality might be lacking. In each case, a critical look is needed to verify whether the error is thrown due to an actual lack of functionality support or due to another reason, for example a typo. With the gathered knowledge, the functionality coverage table can be manually filled in. In this table, a distinction is made between functionality that is natively supported and functionality that can be implemented with a workaround method

Runtime of queries. This metric provides insight into the speed of query execution. A lower runtime is required to obtain higher query throughput rates and improves the flow of business processes relying on the query results. This metric is also obtained by a combination of sub-metrics. To obtain the runtime of a query, the PostgreSQL `EXPLAIN ANALYSE` statement is used. This statement returns the execution plan of various queries or statements and tracks its runtime. When available, it differentiates between the planning time and execution time of a query. In this distinction is not supported by the DBMS, only a total runtime is returned. For each query, the average runtime over the specified iterations is printed. Each query is run with a warm start. After all benchmark queries have run, a total average planning time and execution time, or total average runtime is calculated. This is the sum of all time averages of all queries. The total time provides a quick idea of the speed of the tested DBMS. For each application scenario, the acceptable runtime of a query differs. It is thus advised to verify the significance of the queries and per query determine the acceptable runtime.

Probabilistic data overhead. This metric represents the additional storage space required to store the probabilistic representation of the data. When processing large

volumes of data, needing additional storage space to store the probabilistic representation of the data could get costly. As each probabilistic DBMS stores their probabilistic representation in a unique way, the probabilistic data overload is calculated for each DBMS differently. For both systems, the storage space used is determined by the `pg_size_pretty` statement of PostgreSQL. For DuBio, the overhead percentage is determined using the following calculation:

$$\frac{sentence + dictionary}{offers + dictionary} \times 100$$

Here, *sentence* is the size `_sentence` column in the `offers` table, *dictionary* is the size of the `_dict` table, and *offers* is the size of the `offers` table.

For MayBMS, the following calculation is used to determine the overhead percentage:

$$\frac{setup \times (1 - \frac{distinct\ ids\ count}{ids\ count}) + (offers - setup)}{offers} \times 100$$

Here, *setup* is the size of the `offers_setup` table, *distinct_ids_count* is the count of all distinct values of the `id` column in the `offers` table, *ids_count* is the count of all values of the `id` column in the `offers` table, and *offers* is the size of the `offers` table.

The calculation for MayBMS is a bit more complex, as MayBMS does not create a compact representation of the probability space over a single offer. Because of that, data duplication is created in the offers table. The overhead that this duplication creates is determined by counting the `id` values.

User friendliness. User friendliness is another metric that is composed from several sub-metrics. As user friendliness is something of a more personal taste and cannot be measured from a benchmark run, all sub-metrics are in the form of statements that should be rated on a scale from 1 to 5, 1 meaning that the statement is not true, an 5 meaning that it is very much true. The following aspects should be evaluated to determine a final user friendliness score of the system:

[1, 2, 3, 4, 5]	The software is well documented.
[1, 2, 3, 4, 5]	The software was easy to work with.
[1, 2, 3, 4, 5]	We have sufficient in-house expertise to work well with the software.
[1, 2, 3, 4, 5]	I am satisfied with the monetary expenses that need to be made for running the software.
[1, 2, 3, 4, 5]	The software has a support service.

A.5. Digesting the Results

To digest the raw metrics provided by the benchmark and obtain useful information from them, the benchmark performance is categorised in terms of its effectiveness, efficiency and appeal. please follow the instructions below to digest the raw results and gain insights into the performance of the system.

A.5.1. Effectiveness

The effectiveness of the software relates to the quality of fulfilling the purpose. To obtain a complete picture about the effectiveness of the tested software, both generated files should be considered. To obtain a global picture on the effectiveness of the software, open `QM_metrics_results.txt`. Here, the metric 'percentage of successful queries' is of importance. Ideally, this value will be 100%. If this is not the case, errors have been thrown during the benchmarking process. These errors are displayed at the bottom of the file. For each error, verify if it is thrown due to a lack of functionality support, or due to other reasons, such as programming or memory errors. If the error is due to any of the other reasons, try to eliminate these and run the query again. Keep track of the following information when altering queries that have thrown an error:

Fix log #	
Query that raised an exception:	
Prior adaptations done on the query:	
Exception raised by the query:	
Suspected cause of the exception:	
Implemented fix:	

If any error cannot be fixed, the functionality that is required is lacking in the tested software. In the errors overview, a list of possible functionality gaps is listed below each error. Verify if the error is caused by any of these and note the missing functionality. With this, the coverage of query functionality can be identified. Below is a list of the functionalities that are identified. This list can be expanded when different functionalities are of importance.

As a final step, open `QM_query_results.txt` and verify the results returned by the queries. This step is optional, as there is no truth table provided with the benchmark. If anything strange is shown, verify if the tested software is performing badly.

#	Native	Possible	Functionality
1	[ ]	[ ]	Support of most recent deterministic DBMS queries
2	[ ]	[ ]	Offering a compact representation of the present uncertainty
3	[ ]	[ ]	Get the probability of an offer
4	[ ]	[ ]	Get the probability of a composed result
5	[ ]	[ ]	Apply aggregate functions on probabilities
6	[ ]	[ ]	Filtering on probability
7	[ ]	[ ]	Get the expected count
8	[ ]	[ ]	Get the expected sum
9	[ ]	[ ]	Get the most probable answer
10	[ ]	[ ]	Verify if a specific possible world exists
11	[ ]	[ ]	Verify if a record is certain
12	[ ]	[ ]	Updating the uncertainty of an offer
13	[ ]	[ ]	Repair the probability space after addition, update or deletion of offers
14			Any anomalies discovered during benchmarking

The following table can be used as a guide to see what queries require what functionality:

#	Functionality	Queries
1	Support of most recent deterministic DBMS queries	Any
2	Offering a compact representation of the present uncertainty	Insight 2
3	Get the probability of an offer	Probabilistic 1
4	Get the probability of a composed result	Insight 5; Insight 6; Probabilistic 4
5	Apply aggregate functions on probabilities	Insight 4; Probabilistic 4
6	Filtering on probability	Probabilistic 6
7	Get the expected count	Probabilistic 2
8	Get the expected sum	Probabilistic 3
9	Get the most probable answer	Probabilistic 5
10	Verify if a specific possible world exists	Insight 5
11	Verify if a record is certain	Insight 4
12	Updating the uncertainty of an offer	Insert, Update, Delete 3
13	Repair the probability space after addition, update or deletion of offers	Insert, Update, Delete 1; Insert, Update, Delete 4
14	Any anomalies discovered during benchmarking	Any

A.5.2. Efficiency

The efficiency of the software relates to its use of resources and execution speed. To obtain a complete picture of the efficiency of the testes software, several metrics should be evaluated. The most prominent efficiency metric is the speed of the tested software. The speed of the software can be collected by both the total average execution time and the time per query. The desired speed is fully dependent on your require-

ments. If specific types of operations are most important for the real-world software, open `QM_query_results.txt` and verify if the queries containing that functionality have an acceptable execution time. A visual representation of the execution times of the queries can be found in `results/graphs/QM_graph_runtime.png`. To ensure a clear presentation of the visual results, it might be beneficial to run the benchmark over several subsets of queries. One could, for example, make a separate fun for all insert, update and delete statements, or remove all queries with a significantly higher execution time.

Another indication of efficiency is the amount of characters needed for the queries. It is your decision if this metric is of importance. This metric is also related to the appeal of the software. Query dialects that require more characters are possibly more difficult to understand and possibly require more time to define queries with. For this sub-metric, also a visual representation is included. This can be found in `results/graphs/QM_graph_runtime.png`. Again, if the produced graph is hard to read it might be beneficial to run the benchmark over several query subsets.

A final indication of efficiency is the additional storage space required to store the probabilistic representation of the data. If storage is sparse, having a system that requires less storage for the probabilistic representation is better. Please verify what overhead is acceptable. This sub-metric is only indicated as a single percentage in `QM_metrics_results.txt`

A.5.3. Appeal
The appeal of the software relates to the human element, including the satisfaction of use. Whether the tested software appeals to the company is thus more about personal preference. To guide with answering the question if the tested software is appealing, a list of statements is defined. The score given to these statements define the appeal score of the software. Rate each of the statements below with a score from 1 to 5. A 1 means that the statement does not align with your personal opinion on the software, so that you strongly disagree with the statement, whereas 5 means that the statement is very much true, so you strongly agree. Scoring a 3 provides a neutral opinion.

[1, 2, 3, 4, 5]	The software is well documented.
[1, 2, 3, 4, 5]	The software was easy to work with.
[1, 2, 3, 4, 5]	We have sufficient in-house expertise to work well with the software.
[1, 2, 3, 4, 5]	I am satisfied with the monetary expenses that need to be made for running the software.
[1, 2, 3, 4, 5]	The software has a support service.

A.5.4. Drawing Conclusions
After information on each metric is collected, conclusions can be drawn from this newly acquired information. As a first step, verify if the software supports all functionalities required. A software that cannot run your key processes is practically useless. If this is satisfied, the importance of each metric should be identified. When all metrics are ordered by their importance, a better picture of the suitability of the software can be drawn. Be critical of your requirements and if the tested software fits these well

enough. If two systems are benchmarked, compare their results.

A.6. Including Other Database Management Systems

The QuestionMark benchmark for probabilistic databases was designed with generalisability in mind. Hence why all benchmark queries are also provided in a pseudocode like language. To test the system, two promising probabilistic database management systems are already supported in the system; MayBMS and DuBio. Both these systems are based on PostgreSQL, hence why the QuestionMark Python programs are also written with PostgreSQL based systems in mind. If you want to benchmark another probabilistic database system, both Python programs need to be adapted to fit the new system. For this, additional changes are required when implementing a new non-PostgreSQL based system.

A.6.1. Including any new Probabilistic DBMS

When including a new PostgreSQL based DMBS, no alterations need to be made to the existing codebase. However, new functions should be defined based on the existing codebase.

As each probabilistic DBMS has its own unique structure when it comes to representing the probabilities and/or sentences of the possible worlds, the dataset generation should be adapted to fit the requirements of the new systems. For this, new functions should be defined. For ease, the placeholder `NAME` will be used, which denotes the name of the newly added DBMS. The following additions should be made to QuestionMark: The Dataset Generator.

1. `database_filler_NAME.py`. As the dataset needs to be properly prepared for the new DBMS, functions need to be designed to tailor the produced dataset to the needs of the DBMS. The structure should be similar to that defined in `database_filler_dubio.py` and `database_filler_maybms.py`.

A new probabilistic DBMS also has its own SQL dialect, so in QuestionMark: The Probabilistic Benchmark additions should also be made to the code.

1. `queries_NAME.py`. To include a new DBMS, the first step is to include the queries in the corresponding dialect. To do this, create `queries_NAME.py`. To see what queries should be included, `queries_pseudo_code`, `queries_MayBMS.py` and `queries_DuBio.py` can be used as a translation guide. Please stick to the structure used in these files. When the proficiency level of the to be included query dialect is not sufficiently high, it is advised to first test the queries in a database tool of preference. This makes debugging queries easier.

2. `execute_queries.py`. This file is responsible for sending the queries to the DBMS. In this file, include `from queries_NAME import NAME_QUERIES_DICT`. In `execute_query()`, also include the DBMS in the first if-statement. Finally, check if the execution time returned by the DBMS follows the pattern from MayBMS or from DuBio. When the DBMS uses PostgreSQL 10 or higher, the default can be used.

3. `output_tui.py`. This file prints the benchmark output. In `create_result_`

file(), add the new DBMS in the if-statement.

4. `parameters.py`. Include the new DBMS as an option of the DBMS variable.

5. `metrics.py`. To obtain the metric values from the new system, some metrics should be tailored to the system. In `char_count()`, add the DBMS in the if-statement and verify if specific queries should get a discount in character count. This is done, since characters required for raw record information do not count towards the complexity of the dialect. Also the calculation in `prob_size()` should be adapted and tailored to the manner in which the DBMS stores the probabilistic data.

A.6.2. Including a new Non-PostgreSQL Based DBMS

To include a new non-PostgreSQL based DBMS system, additional steps need to be taken. The following adaptations should be made to QuestionMark: The Dataset Generator.

1. `database.ini.tmpl`. Needs to be adjusted to support a different system.

2. `database_filler.py`. Most of this file should be adapted to generate a connection to the DBMS. Right now, the program uses psycopg2 to establish a database connection. This library only works with PostgreSQL based database systems. To provide support for other systems, please read through all methods in this file and make adaptations where required.

3. `insert_query.py`. The `create()` method also uses Psycopg2. This method should thus also be changed.

The following additional adaptations should be made to QuestionMark: The Probabilistic Benchmark.

1. `execute_query.py`. This file establishes the connection with the database and runs the benchmark. Please read through all functions and change the code where needed.

2. `connect_db.py`. This file also establishes a connection with the database and is used for metric queries. Please read through all functions and change the code where needed. Most of these alterations can be copied from those implemented in `execute_query`.

3. `database.ini.tmpl`. Needs to be adjusted to support a different system.

4. `run_benchmark.py`. This is the main file to run the benchmark. Also here, psycopg2 is used. The present function thus needs to be adapted.

A.7. Query Implementations

This Appendix contains the benchmark queries in pseudocode SQL and provides examples of its implementations into the dialects of DuBio and MayBMS.

A.7.1. Queries in Pseudocode

The queries below are included in the benchmark. For each query, additional information is provided on its functioning and why it is included in the benchmark.

Test 1: Testing the connection. The first query is mainly included to have a low strain query that can be used to test the connection. This query consists of basic SQL-functionalities and all systems should be able to run this.

```
01 | select attribute 'id'
02 | from entity 'offers'
03 | return the first 10 records;
```

Insight 1: Retrieve the full dataset, gain insight in data structure. This query is a real strain tester of the system. The query itself is simple, but it requires the DBMS to return all its data.

```
01 | select all attributes
02 | from 'offers';
03 |
04 | if present select all remaining data;
```

Insight 2: Provide insight into the concentration of offers. This query can be used to verify to what extent the DBMS can concentrate the uncertainty of an offer. It also provides insights into the number of clusters that have been formed.

```
01 | select the count of all attributes alias 'records',
    ↪ the count of all distinct values of attribute 'id' alias 'offers',
    ↪ the count of all distinct values of attribute 'cluster_id' alias '
    ↪ clusters'
02 | from entity 'offers';
```

Insight 3: Provide insight into the distribution of cluster volumes. This query is included as lower strain deterministic query and also includes useful insight into the dataset. As larger clusters put more strain on probability calculations, it is useful to gain insight into the distribution of cluster volumes.

```
01 | select attribute 'cluster_size',
    ↪ the count of all values of attribute 'cluster_size' alias 'amount'
02 | from subquery (
03 |     select the count of all distinct values of attribute 'id' alias '
           ↪ cluster_size'
04 |     from entity 'offers'
05 |     grouped by attribute 'cluster_id'
06 | ) alias 'cluster_sizes'
07 | grouped by attribute 'cluster_size'
08 | ascendingly ordered by 'cluster_size';
```

Insight 4: Gets the percentage of certain clusters. This query provides insight into the uncertainty of the generated dataset. A new dataset can be generated when the result of this query does not match the uncertainty of the real-world dataset. It also verifies if probability calculations can be done on aggregated data.

```
01 | select the count of all certain records divided by the count of all
    ↪ attributes times 100 rounded to four decimal places alias 'certain
    ↪ percentage'
02 | from entity offers;
```

Insight 5: Get the id and probability of the offers from a specific possible world. In some situations it might turn out useful to make a selection based on the probability space of a record. This query returns any record satisfying a specific sentence or probability space declaration.

```
01 |   select attribute 'id',
           ↪ the probability attribute,
           ↪ the variable or sentence attribute
02 |   from entity 'offers'
03 |   satisfying a specific variable or sentence statement;
```

Insight 6: Get the average probability of the dataset. This is another query that tests the strain of the system. It performs a probability calculation over the entire dataset. It also provides insights into the uncertainty of the dataset.

```
01 |   select the average of the probability attribute rounded to four decimal
           ↪ places alias 'certainty_of_the_dataset'
02 |   from entity 'offers';
```

Probabilistic 1: Get offers with the probability of their occurrence. This query contains the most basic added functionality of any probabilistic DBMS, which is the presentation of the probability. It also evaluates the speed of ordering based on the probability attribute.

```
01 |   select the probability attribute rounded to four decimal places alias '
           ↪ probability',
           ↪ all attributes
02 |   from entity 'offers'
03 |   descendingly ordered by 'probability';
```

Probabilistic 2: Gets the expected count of the categories. One more advanced operation on probabilistic data is to obtain the expected count of an attribute. This query evaluates if that operation is supported.

```
01 |   select the attribute 'category',
           ↪ the expected count per attribute 'category' alias 'expected_count'
02 |   from entity 'offers'
03 |   grouped by attribute 'category'
04 |   descendingly ordered by 'expected_count';
```

Probabilistic 3: Gets the expected sum of the product ids per cluster. Another closely related operation is the expected sum. This query evaluates if that operation is supported.

```
01 |   select attribute 'cluster_id',
           ↪ the expected sum per attribute 'id' alias 'number_of_offers'
02 |   from entity 'offers'
03 |   grouped by attribute 'cluster_id'
04 |   descendingly ordered by 'number_of_offers';
```

Probablistic 4: Gets the variables/sentence and probability for the categories. This query is again focused on strain testing. This query produces large aggregations of probabilities, which need to be evaluated to return the query result. This query tests if the DBMS can digest these large aggregations.

```
01 |  select attribute 'category',
          ↪ the compound variable/sentence attribute ,
          ↪ the compound probability attribute rounded to four demical places
          ↪ alias 'probability'
02 |  from entity 'offers'
03 |  grouped by attribute 'category'
04 |  descendingly ordered by 'probability';
```

Probabilistic 5: Returns the most probable offer that is related to a specified string . This query represents the behaviour of a search engine, where the most probable offer satisfying a search condition should be returned. An example string is 'card'. The pseudocode provided contains a workaround method to obtain the most probable answer. It can be shortened to represent native support of this functionality. This query contains hard-coded information and may require an adaptation when having generated a fitting dataset. See Section 5.4.1 for more information.

```
01 |  select all attributes ,
          ↪ the probability attribute rounded to four decimal places alias '
          ↪ probability'
02 |  from entity 'offers'
03 |  satisfying that the attribute value 'cluster_id' exists in subquery (
04 |          select attribute 'cluster_id'
05 |          from entity 'offers'
06 |          satisfying that attribute 'title' a specified string
07 |                  or that attribute 'description' contains a specified
                          ↪ string
08 |  )
09 |  descindingly ordered by 'probability'
10 |  return the first 1 records;
```

Probabilistic 6: Returns all offers containing a specified string with a high uncertainty. When a dataset contains large volumes of highly uncertain data, it can be useful to let a selection of data pass human inspection. This query returns the most uncertain offers so these can be manually classified. This query contains hard-coded information and may require an adaptation when having generated a fitting dataset. See Section 5.4.1 for more information.

```
01 |  select attribute 'id',
          ↪ attribute 'cluster_id',
          ↪ attribute 'brand',
          ↪ attribute 'category',
          ↪ attribute 'identifiers'
02 |  from entity 'offers'
03 |  satisfying that attribute 'title' contains a specified string
04 |          or that attribute 'description' contains a specified string
05 |          and the value of the probability attribute is higher than 0.45
06 |          and the value of the probability attribute is lower than 0.55;
```

Insert, Update, Delete 1: Inserting a new probabilistic cluster. When dealing with probabilistic databases, new data can be added regularly. This query verifies the speed at which new clusters can be added to the database.

```
01 |  insert into entity 'offers'
02 |  the values (a copy of a cluster with size five, with negative id values.);
```

```
03 |
04 | if required add the new probabilities to the corresponding entity;
05 | if required manually repair the probability space;
```

Insert, Update, Delete 2: Inserting bulk. When large volumes of data are constantly added to the database, they are likely added in bulk. This query strain tests the DBMS on large additions of probabilistic data. The current table 'bulk insert' contains 1000 offers and their corresponding probabilities.

```
01 | insert into entity 'offers'
02 | the results of subquery (
03 |       select all attributes
04 |       from entity 'bulk_insert'
05 | );
06 |
07 | if required add the new probabilities to the corresponding entity;
08 | if required manually repair the probability space;
```

Insert, Update, Delete 3: Update uncertainty. This query updates the uncertainty of a specific cluster. As the location of the probability greatly determines the form of this query, its pseudocode is more abstract. This query contains hard-coded information and may require an adaptation when having generated a fitting dataset. See Section 5.4.1 for more information.

```
01 | update the entity containing the probabilities.
02 | alter half of the probabilities of a cluster with four offers;
03 |
04 | if required manually repair the probability space;
```

Insert, Update, Delete 4: Remove uncertainty. When working with probabilistic data, chances are that new evidence will be found and the database should be updated accordingly. In this query, a cluster of size 4 will be split into three clusters. It is currently run on the cluster with cluster_id 162. This query contains hard-coded information and may require an adaptation when having generated a fitting dataset. See Section 5.4.1 for more information.

```
01 | update entity 'offers'
02 | set attribute 'cluster_id' with the maximum value of attribute 'cluster_id
       ↪ ' + 1,
03 |    the variable/sentence/probability attribute to certain
04 | satisfying that attribute 'id' has the value of the first offer in the
       ↪ cluster;
05 |
06 | update entity 'offers'
07 | set attribute 'cluster_id' with the maximum value of attribute 'cluster_id
       ↪ ' + 1,
08 |    the variable/sentence/probability attribute to certain
09 | satisfying that attribute 'id' has the value of the third offer in the
       ↪ cluster;
10 |
11 | update entity 'offers'
12 | set the variable/sentence/probability attribute to a new normalized value
13 | satisfying that attribute 'id' has the value of the second offer in the
       ↪ cluster;
14 |
15 | update entity 'offers'
16 | set the variable/sentence/probability attribute to a new normalized value
17 | satisfying that attribute 'id' has the value of the fourth offer in the
       ↪ cluster;
18 |
19 | if required update the probabilities in the corresponding entity;
20 | if required update the probability space;
```

Insert, Update, Delete 5: Delete a full cluster. Any probabilistic data should also not slow down the deletion of data significantly. This query tests the speed of the DBMS when deleting probabilistic data. This query contains hard-coded information and may require an adaptation when having generated a fitting dataset. See Section 5.4.1 for more information.

```
01 | delete all records from entity 'offers'
02 | satisfying that attribute 'cluster_id' has the value of the specified
       ↪ cluster;
03 |
04 | if required delete the probabilities in the corresponding entity;
05 | if required manually repair the probability space;
```

A.7.2. Queries in DuBio

```
01 |   -- Test 1:
02 |   SELECT id
03 |   FROM offers
04 |   LIMIT 10;
05 |
06 |   -- Insight 1:
07 |   SELECT *
08 |   FROM offers;
09 |
10 |   SELECT print(dict) FROM _dict WHERE name='mydict';
11 |
12 |   -- Insight 2:
13 |   SELECT COUNT(*) as records,
14 |          COUNT(DISTINCT(id)) as offers,
15 |          COUNT(DISTINCT(cluster_id)) as clusters
16 |   FROM offers;
17 |
18 |   -- Insight 3:
19 |   SELECT cluster_size, COUNT(cluster_size) as amount
20 |   FROM (
21 |       SELECT COUNT(DISTINCT(id)) as cluster_size
22 |       FROM offers
23 |       GROUP BY cluster_id
24 |   ) as cluster_sizes
25 |   GROUP BY cluster_size
26 |   ORDER BY cluster_size ASC;
27 |
28 |   -- Insight 4:
29 |   SELECT ROUND(COUNT(CASE WHEN istrue(_sentence) THEN 1 END)::decimal /
          ↪ COUNT(*)::decimal, 4) * 100 AS certain_percentage
30 |   FROM offers;
31 |
32 |   -- Insight 5:
33 |   WITH prob AS (
34 |          SELECT prob(dict, 'w43=1') AS probability
35 |          FROM _dict
36 |          WHERE name = 'mydict'
37 |   )
38 |   SELECT offers.id, prob.probability, hasrva(_sentence, 'w43=1')
39 |   FROM offers, prob
40 |   WHERE hasrva(_sentence, 'w43=1');
41 |
42 |   -- Insight 6:
43 |   SELECT AVG(probability) AS certainty_of_the_dataset
44 |   FROM (
45 |          SELECT round(prob(d.dict, o._sentence)::NUMERIC, 4) AS probability
46 |          FROM offers o, _dict d
47 |          WHERE d.name = 'mydict'
48 |   ) AS probabilities;
49 |
50 |   -- Probabilistic 1:
51 |   SELECT round(prob(d.dict, p._sentence)::NUMERIC, 4) AS probability, o.*
52 |   FROM offers o, _dict d
53 |   WHERE d.name = 'mydict'
54 |   ORDER BY probability DESC;
```

```
55 |
56 | -- Probabilistic 2:
57 | SELECT category, SUM(prob(d.dict, o._sentence)) AS expected_count
58 | FROM offers o, _dict d
59 | WHERE d.name = 'mydict'
60 | GROUP BY category
61 | ORDER BY expected_count DESC;
62 |
63 | -- Probabilistic 3:
64 | SELECT cluster_id, ROUND(SUM(id * prob(d.dict, o._sentence))::NUMERIC, 2)
   |     ↪ AS expected_sum, COUNT(id) AS number_of_offers
65 | FROM offers o, _dict d
66 | WHERE d.name = 'mydict'
67 | GROUP BY cluster_id
68 | ORDER BY number_of_offers DESC;
69 |
70 | -- Probablistic 4:
71 | WITH category_sentence AS (
72 |     SELECT category, AGG_OR(_sentence) AS sentence
73 |     FROM part
74 |     GROUP BY category
75 | )
76 | SELECT cs.*, round(prob(d.dict, cs.sentence)::NUMERIC, 4) AS probability
77 | FROM category_sentence cs, _dict d
78 | WHERE d.name = 'mydict'
79 | ORDER BY probability ASC;
80 |
81 | -- Probabilistic 5:
82 | Returns the most probable offer that is related to 'ford'.
83 | SELECT p.*, round(prob(d.dict, _sentence)::NUMERIC, 4) AS probability
84 | FROM part p, _dict d
85 | WHERE cluster_id IN (
86 |         SELECT cluster_id
87 |         FROM part
88 |         WHERE title LIKE '%ford%'
89 |         OR description LIKE '%ford%'
90 | )
91 | ORDER BY probability DESC
92 | LIMIT 1;
93 |
94 | -- Probabilistic 6:
95 | SELECT o.*
96 | FROM offers o, _dict d
97 | WHERE title LIKE '%card%'
98 | OR description LIKE '%card%'
99 | AND prob(d.dict, _sentence) > 0.45
100 | AND prob(d.dict, _sentence) < 0.55;
101 |
102 | -- Insert, Update, Delete 1:
103 | INSERT INTO offers (id, cluster_id, title, brand, category, description,
    |     ↪ price, identifiers, keyvaluepairs, spectablecontent, "_sentence")
104 |     VALUES(-464, 77, ..., Bdd('b77x1=1&v77=1') ),
105 |           (-466, 77, ..., Bdd('b78x1=0&v78=1') ),
106 |           (-468, 77, ..., Bdd('b77x1=2&v77=1') ),
107 |           (-469, 77, ..., Bdd('b78x1=1&v78=1') ),
108 |           (-471, 77, ..., Bdd('b77x1=0&v77=1') );
```

```
109 |
110 | UPDATE _dict
111 | SET dict = add(dict, 'b77x1=0:0.24454, ..., v77=3:0.246')
112 | WHERE name='mydict';
113 |
114 | -- Insert, Update, Delete 2:
115 | INSERT INTO offers(id, cluster_id, title, brand, category, description,
       ↪ price, identifiers, keyvaluepairs, spectablecontent, _sentence)
116 |    SELECT * FROM bulk_insert;
117 |
118 | UPDATE _dict
119 | SET dict = add(dict, 'b000x1=0:0.500000, ... v966=2:0.203147')
120 | WHERE name='mydict';
121 |
122 | -- Insert, Update, Delete 3:
123 | UPDATE _dict
124 | SET dict = upd(dict, 'a7x1=0:0.3992, ..., w8=3:0.184')
125 | WHERE name='mydict';
126 |
127 | -- Insert, Update, Delete 4:
128 | WITH max_cluster AS (
129 |     SELECT (max(cluster_id) + 1) AS max_id
130 |     FROM offers
131 | )
132 | UPDATE offers
133 | SET cluster_id = max_cluster.max_id,
134 |     _sentence = Bdd('1')
135 | FROM max_cluster
136 | WHERE id = 2689021;
137 |
138 | WITH max_cluster AS (
139 |     SELECT max(cluster_id) + 1 AS max_id
140 |     FROM offers
141 | )
142 | UPDATE offers
143 | SET cluster_id = max_cluster.max_id,
144 |     _sentence = Bdd('1')
145 | FROM max_cluster
146 | WHERE id = 7257664;
147 |
148 | UPDATE offers
149 | SET _sentence = Bdd('a162x5=0&w162=0')
150 | WHERE id = 10198975;
151 |
152 | UPDATE offers
153 | SET _sentence = Bdd('a162x5=1&w162=0')
154 | WHERE id = 2668263;
155 |
156 | UPDATE _dict
157 | SET dict = add(dict, 'w162=0:0.83')
158 | WHERE name='mydict';
159 |
160 | UPDATE _dict
161 | SET dict = del(dict, 'a162x5=2')
162 | WHERE name='mydict';
163 |
```

```
164 |  -- Insert, Update, Delete 5:
165 |  DELETE FROM offers
166 |  WHERE cluster_id = 41;
167 |
168 |  UPDATE _dict
169 |  SET dict = del(dict, 'a41x1=0, ..., w44=5')
170 |  WHERE name='mydict';
```

A.7.3. Queries in MayBMS

```
01 |
02 |  -- Test 1:
03 |  SELECT id
04 |  FROM offers
05 |  LIMIT 10;
06 |
07 |  -- Insight 1:
08 |  SELECT *
09 |  FROM offers;
10 |
11 |  -- Insight 2:
12 |  SELECT COUNT(*) as records,
13 |         COUNT(DISTINCT(id)) as offers,
14 |         COUNT(DISTINCT(cluster_id)) as clusters
15 |  FROM offers_setup;
16 |
17 |  -- Insight 3:
18 |  SELECT cluster_size, COUNT(cluster_size) as amount
19 |  FROM (
20 |      SELECT COUNT(DISTINCT(id)) as cluster_size
21 |      FROM offers_setup
22 |      GROUP BY cluster_id
23 |  ) as cluster_sizes
24 |  GROUP BY cluster_size
25 |  ORDER BY cluster_size ASC;
26 |
27 |  -- Insight 4:
28 |  SELECT ROUND(all_certain::decimal / all_offers::decimal, 4) * 100 AS
       ↪ certain_percentage
29 |  FROM (
30 |         SELECT COUNT(id) AS all_offers
31 |         FROM offers_setup
32 |  ) AS count_all, (
33 |         SELECT COUNT(id) AS all_certain
34 |         FROM (
35 |                 SELECT id, tconf() AS confidence
36 |                 FROM offers
37 |         ) AS confidences
38 |         WHERE confidence = 1
39 |  ) AS count_cert;
40 |
41 |  -- Insight 5:
42 |  SELECT id, tconf(*), _v0
43 |  FROM offers
44 |  WHERE _v1 = 52379
45 |  AND _d1 = 548185;
```

```
46 |
47 |  -- Insight 6:
48 |  SELECT round((AVG(tconf()) * 100)::NUMERIC, 4) AS certainty_of_the_dataset
49 |  FROM offers;
50 |
51 |  -- Probabilistic 1:
52 |  SELECT round(tconf()::decimal, 4) AS probability, *
53 |  FROM offers
54 |  ORDER BY probability DESC;
55 |
56 |  -- Probabilistic 2:
57 |  SELECT category, ECOUNT() AS expected_count
58 |  FROM offers
59 |  GROUP BY category
60 |  ORDER BY expected_count DESC;
61 |
62 |  -- Probabilistic 3:
63 |  SELECT cluster_id, esum(id), COUNT(id) AS number_of_offers
64 |  FROM offers
65 |  GROUP BY cluster_id
66 |  ORDER BY number_of_offers DESC;
67 |
68 |  -- Probablistic 4:
69 |  SELECT category, conf() AS probability
70 |  FROM offers
71 |  GROUP BY category
72 |  ORDER BY probability DESC;
73 |
74 |  -- Probabilistic 5:
75 |  SELECT *, round(tconf()::NUMERIC, 4) AS probability
76 |  FROM offers
77 |  WHERE cluster_id IN (
78 |      SELECT cluster_id
79 |      FROM offers_setup
80 |      WHERE title LIKE '%card%'
81 |      OR description LIKE '%card%'
82 |  )
83 |  ORDER BY probability DESC
84 |  LIMIT 1;
85 |
86 |  -- Probabilistic 6:
87 |  SELECT id, cluster_id, brand, category, identifiers
88 |  FROM offers
89 |  WHERE title LIKE '%card%'
90 |  OR description LIKE '%card%'
91 |  AND tconf() > 0.45
92 |  AND tconf() < 0.55;
93 |
94 |  -- Insert, Update, Delete 1:
95 |  INSERT INTO offers_setup (id, cluster_id, title, brand, category,
        ↪ description, price, identifiers, keyvaluepairs, spectablecontent,
        ↪ world_prob, attribute_prob)
96 |     VALUES(-464, 77, ..., 0.42911, 0.629),
97 |           (-466, 77, ..., 0.5, 0.246),
98 |           (-468, 77, ..., 0.32635, 0.629),
99 |           (-469, 77, ..., 0.5, 0.125),
```

```
100 |           (-471, 77, ..., 0.24454, 0.629);
101 |
102 | DROP TABLE IF EXISTS offers_rk_world CASCADE;
103 | DROP TABLE IF EXISTS offers_rk_attrs CASCADE;
104 | DROP TABLE IF EXISTS offers CASCADE;
105 |
106 | CREATE TABLE offers_rk_world AS REPAIR KEY cluster_id IN offers_setup
        ↪ WEIGHT BY world_prob;
107 | CREATE TABLE offers_rk_attrs AS REPAIR KEY id IN offers_setup WEIGHT BY
        ↪ attribute_prob;
108 |
109 | CREATE TABLE offers AS (
110 |     SELECT attrs.*
111 |     FROM offers_rk_attrs AS attrs, offers_rk_world AS world
112 |     WHERE attrs.id = world.id
113 | );
114 |
115 | -- Insert, Update, Delete 2:
116 | INSERT INTO offers_setup (id, cluster_id, title, brand, category,
        ↪ description, price, identifiers, keyvaluepairs, spectablecontent,
        ↪ world_prob, attribute_prob)
117 | SELECT * FROM bulk_insert;
118 |
119 | DROP TABLE IF EXISTS offers_rk_world CASCADE;
120 | DROP TABLE IF EXISTS offers_rk_attrs CASCADE;
121 | DROP TABLE IF EXISTS offers CASCADE;
122 |
123 | CREATE TABLE offers_rk_world AS REPAIR KEY cluster_id IN offers_setup
        ↪ WEIGHT BY world_prob;
124 | CREATE TABLE offers_rk_attrs AS REPAIR KEY id IN offers_setup WEIGHT BY
        ↪ attribute_prob;
125 |
126 | CREATE TABLE offers AS (
127 |     SELECT attrs.*
128 |     FROM offers_rk_attrs AS attrs, offers_rk_world AS world
129 |     WHERE attrs.id = world.id
130 | );
131 |
132 | -- Insert, Update, Delete 3:
133 | UPDATE offers
134 | SET world_prob = 0.345,
135 |     attribute_prob = 0.3992
136 | WHERE _d0 = 615777
137 | AND _d1 = 613619;
138 |
139 | UPDATE offers
140 | SET world_prob = 0.345,
141 |     attribute_prob = 0.6008
142 | WHERE _d0 = 615777
143 | and _d1 = 613841;
144 |
145 | UPDATE offers
146 | SET world_prob = 0.1254,
147 |     attribute_prob = 0.5
148 | WHERE _d0 = 615999
149 | AND _d1 = 613619;
```

```
150 |
151 | UPDATE offers
152 | SET world_prob = 0.1254,
153 |     attribute_prob = 0.5
154 | WHERE _d0 = 615999
155 | and _d1 = 613841;
156 |
157 | UPDATE offers
158 | SET world_prob = 0.487,
159 |     attribute_prob = 0.4300
160 | WHERE _d0 = 615850
161 | and _d1 = 613395;
162 |
163 | UPDATE offers
164 | SET world_prob = 0.487,
165 |     attribute_prob = 0.5700
166 | WHERE _d0 = 615850
167 | AND _d1 = 613692;
168 |
169 | UPDATE offers
170 | SET world_prob = 0.487,
171 |     attribute_prob = 0.4300
172 | WHERE _d0 = 615999
173 | and _d1 = 613841;
174 |
175 | UPDATE offers
176 | SET world_prob = 0.487
177 | WHERE _d0 = 615553
178 | AND _d1 = 613692;
179 |
180 | UPDATE offers
181 | SET world_prob = 0.487
182 | WHERE _d0 = 615553
183 | and _d1 = 613395;
184 |
185 | UPDATE offers
186 | SET world_prob = 0.329
187 | WHERE _d0 = 614032
188 | AND _d1 = 612733;
189 |
190 | UPDATE offers
191 | SET world_prob = 0.329
192 | WHERE _d0 = 614032
193 | and _d1 = 611874;
194 |
195 | UPDATE offers
196 | SET world_prob = 0.184
197 | WHERE _d0 = 615197
198 | AND _d1 = 612733;
199 |
200 | UPDATE offers
201 | SET world_prob = 0.184
202 | WHERE _d0 = 615197
203 | and _d1 = 613039;
204 |
205 | DROP TABLE IF EXISTS offers_rk_world CASCADE;
```

```
206 | DROP TABLE IF EXISTS offers_rk_attrs CASCADE;
207 | DROP TABLE IF EXISTS offers CASCADE;
208 |
209 | CREATE TABLE offers_rk_world AS REPAIR KEY cluster_id IN offers_setup
        ↪ WEIGHT BY world_prob;
210 | CREATE TABLE offers_rk_attrs AS REPAIR KEY id IN offers_setup WEIGHT BY
        ↪ attribute_prob;
211 |
212 | CREATE TABLE offers AS (
213 |     SELECT attrs.*
214 |     FROM offers_rk_attrs AS attrs, offers_rk_world AS world
215 |     WHERE attrs.id = world.id
216 | );
217 |
218 | -- Insert, Update, Delete 4:
219 | UPDATE offers_setup
220 | SET cluster_id = max_cluster.max_id,
221 |     world_prob = 1,
222 |     attribute_prob = 1
223 | FROM (
224 |     SELECT max(cluster_id) + 1 AS max_id
225 |     FROM offers_setup
226 | ) as max_cluster
227 | WHERE id = 12071001;
228 |
229 | UPDATE offers_setup
230 | SET cluster_id = max_cluster.max_id,
231 |     world_prob = 1,
232 |     attribute_prob = 1
233 | FROM (
234 |     SELECT max(cluster_id) + 1 AS max_id
235 |     FROM offers_setup
236 | ) as max_cluster
237 | WHERE id = 16457529;
238 |
239 | UPDATE offers_setup
240 | SET world_prob = 0.63,
241 |     attribute_prob = 0.5
242 | WHERE id = 7339350;
243 |
244 | UPDATE offers_setup
245 | SET world_prob = 0.63,
246 |     attribute_prob = 0.5
247 | WHERE id = 12326926;
248 |
249 | DROP TABLE IF EXISTS offers_rk_world CASCADE;
250 | DROP TABLE IF EXISTS offers_rk_attrs CASCADE;
251 | DROP TABLE IF EXISTS offers CASCADE;
252 |
253 | CREATE TABLE offers_rk_world AS REPAIR KEY cluster_id IN offers_setup
        ↪ WEIGHT BY world_prob;
254 | CREATE TABLE offers_rk_attrs AS REPAIR KEY id IN offers_setup WEIGHT BY
        ↪ attribute_prob;
255 |
256 | CREATE TABLE offers AS (
257 |     SELECT attrs.*
```

```
258 |      FROM offers_rk_attrs AS attrs, offers_rk_world AS world
259 |      WHERE attrs.id = world.id
260 | );
261 |
262 | -- Insert, Update, Delete 5:
263 | DELETE FROM offers_setup
264 | WHERE cluster_id = 41;
265 |
266 | DROP TABLE IF EXISTS offers_rk_world CASCADE;
267 | DROP TABLE IF EXISTS offers_rk_attrs CASCADE;
268 | DROP TABLE IF EXISTS offers CASCADE;
269 |
270 | CREATE TABLE offers_rk_world AS REPAIR KEY cluster_id IN offers_setup
         ↪ WEIGHT BY world_prob;
271 | CREATE TABLE offers_rk_attrs AS REPAIR KEY id IN offers_setup WEIGHT BY
         ↪ attribute_prob;
272 |
273 | CREATE TABLE offers AS (
274 |      SELECT attrs.*
275 |      FROM offers_rk_attrs AS attrs, offers_rk_world AS world
276 |      WHERE attrs.id = world.id
277 | );
```