DuoSQL: A High-Level Query Language for Probabilistic Databases

MARTIN DEMIREV, University of Twente, The Netherlands

ABSTRACT

Uncertainty management is a fundamental challenge in data quality problems and imperfect data integration tasks. Traditional relational databases rely on expert validations or manual workarounds to address data issues; however, this approach is inefficient and does not scale well for large or dynamic datasets. Probabilistic databases extend classic database semantics by associating data points with probabilities that indicate their correctness. One such system is DuBio, developed at the University of Twente, which provides a framework for representing and querying uncertain data in PostgreSQL. However, the current DuBio query interface requires users to explicitly deal with the complexity of uncertainty structures, thus degrading usability.

This research builds on DuBio by introducing DuoSQL, a high-level query language, and its corresponding compiler algorithm. The goal is to simplify querying over uncertain data while preserving the expressiveness of DuBio. Experiments show that DuoSQL reduces user effort by significantly decreasing query length and complexity compared to manually written DuBio SQL queries. Although syntactic overhead from translation is introduced, runtime performance remains similar to manual queries. These results demonstrate that DuoSQL enhances usability without compromising efficiency or correctness, making probabilistic querying more accessible.

KEYWORDS

Probabilistic Databases, Uncertain Data, Data Quality Management, Structured Query Language, Domain-Specific Language

1 INTRODUCTION

In many real-world applications, data is rarely clean or complete. Data integration tasks often involve inconsistencies, missing values, duplicates, and conflicting information from multiple sources. These problems are typically grouped under the broader challenge of data quality. Addressing data quality issues often relies on expert rules and validations, data cleaning pipelines, or discarding uncertain records - strategies that become unmanageable as datasets grow in size and complexity [1]. Moreover, such practices hold the risk of removing vital data, leading to negative impacts such as business losses and introductions of bias.

An alternative approach is to explicitly model uncertainty rather than resolving or ignoring it. This is the core of probabilistic databases, which extend the relational model by associating data entries with probabilities. Probabilistic databases allow users to store, query, and reason over data that is uncertain, incomplete, or conflicting without being forced to resolve all issues immediately [2]. Moreover, probabilistic databases have been seen as a way to "turn dirt into diamonds" by treating uncertainty as a core feature rather than a problem to be eliminated [3].

DuBio [7] is a probabilistic extension of PostgreSQL [8] developed at the University of Twente. Although DuBio provides the foundations for managing uncertain data, its current query interface is not user-friendly. Users must explicitly manipulate low-level probabilistic structures (such as 'sentences' and 'dictionaries') using DuBio-specific functions like `agg_or()`, `array_agg()`.

To illustrate a real-world case, let us consider the example of semantic duplicates given by Van Keulen [2]. We have two columns - "Car brand" and "Sales" (see Appendix A for visualization). We can see that a single real-world object - the car brand "BMW" - is represented in 3 different forms - "BMW", "B.M.W.", and "Bayerische Motoren Werke". Consequently, when implemented in DuBio, each representation is assigned a distinct random variable value (for example, `bmw=1`, `bmw=2`, and `bmw=3`, respectively). These random variables form the entries in the `_sentence` column. For instance, the row with "BMW" would be assigned `Bdd(bmw=1)`. Each sentence is linked to a probability value in the `_dict` table; thus, `bmw=1` would be the key linking to its value of `0.6`, forming the entry `bmw=1:0.6` in the dictionary.

Furthermore, when querying uncertain data, the `_sentence` columns in the relevant tables must be traversed and logically combined to capture all meaningful combinations of possible records. Probabilistic databases operate under the "possible worlds" semantics, treating data as a probability space over all these potential worlds [2]. The final probabilities are derived from the constructed logical sentences (BDDs), as demonstrated in Appendix B.

This research addresses the gap between expressiveness and usability in probabilistic data querying. The project introduces two key contributions: a high-level language and a translation algorithm. These artifacts aim to make probabilistic data systems more accessible to practitioners without sacrificing expressive power and by eliminating the need to be familiar with the underlying theory and functions. The proposed domain-specific language DuoSQL [10], its automatic mapping to DuBio SQL, and its manually written alternative are compared and analyzed in several experiments. The translation process is formalized as a generalized algorithm designed to be implementable in any programming language.

Lastly, the name 'DuoSQL' reflects both the language's dual applicability to certain and uncertain data and its connection to DuBio, the underlying system.

2 PROBLEM STATEMENT

This research builds on DuBio to address the challenge of usability in querying uncertain data. A new abstraction layer is needed to make DuBio more accessible and intuitive to users. Hence, the aim is to design a user-friendly language that allows interaction with uncertain data without requiring users to manually manage underlying probabilistic structures. Therefore, this research addresses the following questions:

TScIT 43, July 4, 2025, Enschede, The Netherlands

- **RQ1:** How can a high-level abstraction layer be designed to improve the usability of querying uncertain data while preserving DuBio's expressiveness of its translated queries?
- **RQ2:** To what extent does the proposed language reduce query technicality compared to manually written DuBio SQL queries?
- **RQ3:** What is the syntactic and computational overhead introduced by automated translation from the high-level language to executable DuBio queries compared to manually written DuBio SQL queries?
- **RQ4:** How does the performance of automatically translated DuoSQL queries compare to their manually written counterparts in terms of execution time?

3 RELATED WORK

Dalvi et al. [3] provided a broad overview of probabilistic databases. Their research highlighted key theoretical challenges in query evaluation and addressed the potential of probabilistic databases to manage "dirt" - uncertainty, inconsistency, and incompleteness - in large-scale data. However, they also pointed out the lack of intuitive query interfaces as a critical barrier to real-world applications.

Van Keulen [2] introduced a probabilistic approach to data integration, where uncertainty is explicitly modeled instead of being resolved upfront. Magnani and Montesi [1] further surveyed the uncertainty in integration systems. They noted that while many representation models exist, there is a gap in user-facing tools that abstract away probabilistic complexity.

Antova et al. [4] addressed the usability challenge of querying uncertain data without requiring users to understand the underlying probabilistic semantics. They introduced MayBMS, a system based on the U-relations model, which enabled SQL-style querying over uncertain data using compact relational representations. By extending SQL with minimal constructs, MayBMS allowed users to express queries intuitively while the underlying system handled probabilistic reasoning. This demonstrates that language-level abstractions can improve usability and efficiency - a principle central to DuoSQL's design.

Other influential systems include Trio[6] and MCDB-R[5]. The Trio system addressed data uncertainty and lineage by extending standard SQL and introducing its own data model (ULDB), query language (TriQL), query API, and graphical user interface. MCDB-R applied the Monte Carlo-based approach of MCDB to uncertain data by supporting "efficient risk analysis" through repeated simulations over parameterized databases. Although these systems are powerful, they still require explicit reasoning about uncertainty, as they expose the user directly to their technical details, which limits usability for broader audiences.

A closely related predecessor to this work is inSQeLto [11], a domain-specific language designed for DuBio. inSQeLto focused on syntactic familiarity and conducted user testing to validate its design. However, it only supported basic SELECT queries with optional probability display. It did not implement more complex SQL functionality and did not evaluate the cost of translation or simplification.

Algorithm 1: Translation of DuoSQL to DuBio SQL

Input: A DuoSQL query Q over a PostgreSQL database schema
Output: Translated DuBio SQL query Q'

Sanitize Q (e.g., strip spaces, trailing semicolons)
Define SQL clause-bounding and clause-capturing regular expressions
Extract SELECT and FROM clauses from Q
Parse WHERE, ORDER BY, LIMIT, GROUP BY, HAVING, SHOW clauses (if present)
Extract all tables and aliases used in Q
Determine whether DISTINCT, aggregates, sentences, or probability are involved
Define sentence expression *sentence_expr*:
if *data is entirely certain* **then**
    sentence_expr := 'certain'
else
    sentence_expr := combination of all `_sentence` columns using logical AND on all probabilistic tables
end

if *sentence_expr == 'certain'* **then**
    if *'probability' is present in the* `HAVING` *clause* **then**
        return Error informing the user that queried data is entirely certain
    return deterministic query Q' without probabilistic constructs and special clauses
else if *Q is an aggregate query over all '*' rows* **then**
    Build view `agg_all_view` using probabilistic BDD logic (see Section 5.6)
else if *Q is a grouped aggregate query* **then**
    Build view `agg_view` using probabilistic BDD logic (see Section 5.7)
else
    Build `join_view` using FROM, WHERE, and optional DISTINCT
end
Optionally build `prob_view` if probability is needed
Compose final SELECT query over the relevant view, applying filters, order, and limits
return Q'

4 THE ALGORITHM

This section outlines Algorithm 1, which forms the basis of DuoSQL. The translation process is modular and language-agnostic, meaning that it can be implemented in any programming language that can connect to PostgreSQL and DuBio. The algorithm treats a DuoSQL query as a string, utilizing regular expressions to identify standard SQL clauses and any probabilistic extensions. It constructs a logical sentence expression (`_sentence`) by combining the sentence columns of all involved probabilistic tables.

If all data is deterministic, the query is returned as-is without including any probabilistic constructs, even if requested to be shown. However, there is an exception: if the user attempts to filter entirely certain data by probability, an error message is shown, informing the user to reconsider their query logic. That is crucial when a probability less than `1.00` is requested. Otherwise, the translator generates modular intermediate views that separate join logic, probabilistic reasoning, and aggregation. This modular view-based structure makes the translation both explainable and composable, while preserving the underlying semantics of DuBio. Probabilistic evaluation occurs only when explicitly requested or when queries include probabilistic filters.

5 THE LANGUAGE

This section presents the DuoSQL language documented like PostgreSQL's synopsis [9]. DuoSQL is a high-level query language for probabilistic databases, built as an abstraction layer over DuBio. The implementation is primarily in Python [12], with supporting aggregate functions written in DuBio SQL. Its syntax closely resembles standard SQL but extends it with probabilistic constructs like `SHOW PROBABILITY`, `SHOW SENTENCE`, and support for probabilistic filtering. DuoSQL is **case-insensitive** and can also be applied to **entirely certain** data (see Appendix C.2 for a certain table example). Its implementation is accessible here [10].

5.1 Language Grammar

```
SELECT [ DISTINCT ] field [ [ AS ] field_alias ] [, ...]
       [, AGG(x) [ [ AS ] agg_alias ] ]
FROM table [ table_alias ] [, ...]
[ JOIN table [ table_alias ] ON conditions ] [...]
[ WHERE conditions ]
[ GROUP BY field [, ...] ]
[ HAVING conditions ]
[ ORDER BY field [ ASC | DESC ] [, ...] ]
[ LIMIT count ]
[ SHOW [ SENTENCE | PROBABILITY ] [, ...] ];
```

5.2 Example DuoSQL Query

```
SELECT w.witness, p.companion AS suspect,
   c.caretaker, count(w.color) AS color_count
FROM witnessed w
JOIN plays p ON w.cat_name = p.cat_name
JOIN cares c ON w.cat_name = c.cat_name
WHERE w.cat_name = 'max'
GROUP BY w.witness, p.companion, c.caretaker
HAVING probability >= 0.5 AND color_count > 1
ORDER BY w.witness DESC, probability ASC
LIMIT 10
SHOW SENTENCE, PROBABILITY
```

5.3 Description

The following subsections list specifications for some of the clauses. If a clause is not present, then the Grammar in Section 5.1 is sufficient for understanding its functionality.

5.3.1 SELECT

- Currently, the aggregation field must appear **last** in the `SELECT` clause for correct parsing.
- It supports all aggregation functions (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) on single fields. For example:

```
AVG(age)
```

- Additionally, `COUNT(*)` is supported, which counts rows. It still requires one `GROUP BY` field, which is not included in the result, but is required when querying multiple probabilistic tables due to overlapping column names (see Appendix C.1). The specific grammar would be as follows:

```
SELECT field, COUNT(*)
...
GROUP BY field
```

- The following functionalities are currently **unsupported** within a single query:
  - Combination of `DISTINCT` and aggregation.
  - Multiple aggregation functions.

5.3.2 FROM

- As many tables as needed can be included and later joined with `WHERE` conditions.
- Aliases are optional even if there is more than 1 table present.

5.3.3 JOIN

- All `JOIN` combinations using these keywords - `FULL`, `LEFT`, `RIGHT`, `OUTER`, `INNER`, `CROSS` - are supported.
- Multiple joins are possible, and aliases are optional.

5.3.4 HAVING

- It is used not only for filtering by the used aggregation, but also to filter by the probability value, which is in the range `[0, 1]`. For example:

```
HAVING avg > 3 AND probability <= 0.75
```

5.3.5 SHOW

- **`SHOW PROBABILITY`**: Appends a computed probability column to the result.
- **`SHOW SENTENCE`**: Appends the underlying logical sentence (BDD) used for probabilistic computation.
- Sentence and probability computations are based solely on uncertain data. Certain data does not have any influence.
- Users can include either or both options in the `SHOW` clause. For example:

```
SHOW SENTENCE, PROBABILITY
```

5.4 Mapping

Appendix F presents example DuoSQL queries alongside their corresponding translated DuBio SQL queries. Each example highlights a different language feature, stated in the comments on the first lines of the DuoSQL queries.

5.5 DuBio Aggregation Functions

This section focuses on the DuBio functions utilized in all aggregation queries. The functions `prob_Bdd` and `prob_Bdd_count` are presented in Appendix D.1. The additional functions `prob_Bdd_sum`, `prob_Bdd_avg`, `prob_Bdd_min`, and `prob_Bdd_max` are not shown, as they follow the same pattern as `prob_Bdd_count`.

Together, `prob_Bdd` and the aggregate-specific functions allow DuBio to compute aggregations across the possible worlds defined by the uncertainty in the sentence BDDs.

5.5.1 prob_Bdd: World Construction

The function `prob_Bdd` is responsible for synthesizing a Boolean sentence (BDD) that represents a possible world under evaluation. It takes an array of BDD expressions and a bitmask as input. Each bit in the mask indicates whether the corresponding BDD in the array should be taken as is (if the bit is 1) or negated (if the bit is 0). The function iteratively builds a conjunction of these terms, resulting in a single BDD representing the logical sentence for a specific possible subset of tuples. This function is central to all probabilistic aggregation as it defines the precise world under which an aggregate is evaluated.

5.5.2 prob_Bdd_count: Aggregation over Masked Values

The function `prob_Bdd_count` operates over an array of values (for example, the cat colors in the column `color` of type `text`) and a bitmask, returning the number of selected elements. It iterates over the array and checks the corresponding bit in the mask, where for every bit set to 1, the associated value is counted. This function computes the size of a subset (i.e., how many tuples are active) in a given world. It also serves as the structural template for the other aggregate functions, which differ only in how they reduce the masked values (e.g., summing them or averaging them instead of counting).

5.6 DuBio Aggregation Queries

This section describes how probabilistic aggregation is implemented using DuBio SQL. The underlying logic combines per-group value arrays and sentence arrays, iterates over all possible subset masks, and computes aggregate results using the DuBio aggregate functions introduced earlier in Section 5.5.

A representative example is shown in Appendix D.2. For each group (`cat_name`), arrays of values (`age`) and BDDs (`_sentence`) are constructed. The `generate_series` function generates all bit masks for subset enumeration. The relevant aggregate function `prob_Bdd_avg` is applied per mask and combined using `agg_or` over all worlds.

5.7 DuBio COUNT(*) Aggregation Queries

`COUNT(*)` queries perform aggregation over the entire dataset. This is why other aggregate functions are not applicable in this context, mainly because of data type mismatches and more specifically, having data types different than numerical. Although the query counts all rows regardless of specific columns, the `GROUP BY` is required due to the overlapping column names across tables. Hence, the given column is actually counted rather than grouped by.

The corresponding implementation is shown in Appendix D.3. Instead of grouping, a single set of values and sentence expressions is aggregated across all rows. The result is computed by applying `prob_Bdd_count` over all possible worlds, with the resulting probability derived from the union of contributing sentences using `agg_or`.

5.8 DuBio DISTINCT Queries

When a DuoSQL query uses the `DISTINCT` keyword, deduplication is performed probabilistically using sentence logic. Each distinct value (e.g., a unique color) is associated with a disjunction of all possible worlds in which it occurs.

Appendix D.4 shows how this is implemented using a `GROUP BY` on the distinct field and an `agg_or` over all associated sentence values. The resulting probability reflects the likelihood that the value appears in at least one possible world.

6 EXPERIMENTAL SETUP

To evaluate the cost and benefits of DuoSQL (also referred to as "High-level"), two automated experiment pipelines were implemented in Python (accessible here [10]) - one for Experiments 7.1 and 7.2, and one for Experiment 7.3. This setup ensures that results are reproducible and extensible.

6.1 Setup for Experiments 7.1 and 7.2

6.1.1 Metrics

- **Code Lines (CL)**: Represents the **count** of (non-empty) lines of code in a given query.
- **Characters**: Represents the character **count** in a given query. Counts more than one subsequent semicolon or whitespace as one character.
- **Level of Complexity (LoC)**: Evaluates the complexity of a given query using a method that fits this project:
  - Base complexity: 1;
  - Joins: +1 for each `JOIN` clause. It is unable to count joins through the `FROM` clause as it is much more complex to automate. **All** test queries use joins **only** through `JOIN` clauses;
  - Where: +1 for each `WHERE` clause;
  - View: +1 for each `CREATE OR REPLACE VIEW`;
  - Nested from-selects: +1 for each "`FROM ( SELECT`" (with flexible spaces).
- **Probabilistic Constructs**: Represents the **count** of probabilistic constructs in a given query. Only DuBio-specific keywords are counted, as they require a more advanced understanding of the underlying concepts and structures. Those are, namely: ‘`prob`‘, ‘`agg_or`‘, ‘`prob_Bdd`‘, ‘**`&`**‘, ‘`_sentence`‘, ‘`_dict`‘. Where ‘`prob`‘ is bounded, hence ‘`probability`‘ that can only be queried when using DuoSQL is **not** counted.

6.1.2 Setup

- 8 test types of 2 queries each (the second more complex than the first) are observed. The same queries are used in both experiments. The test types shown in Table 1 highlight the technical focus of each query they contain. The `MIXED DATA`

Table 1. Reduction Summary: Manual vs High-level

Type	CL %	Chars %	LoC %	Prob Constr. #
1. SIMPLE	0%	0%	0%	0
2. JOIN + PROB	23%	40%	25%	22
3. MIXED DATA	50%	53%	55%	11
4. DISTINCT	64%	69%	73%	14
5. AGGREGATION	70%	80%	76%	35
6. COUNT(*)	75%	78%	76%	37
7. FILTERS	62%	66%	71%	25
8. LARGE	56%	62%	52%	36
Overall	**50%**	**56%**	**54%**	**23**
Overall excl. 1-2	**63%**	**68%**	**67%**	**26**

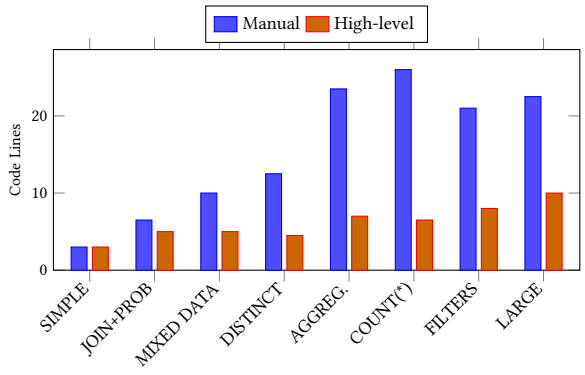


Fig. 1. Code Lines (CL) Comparison: Manual vs High-level

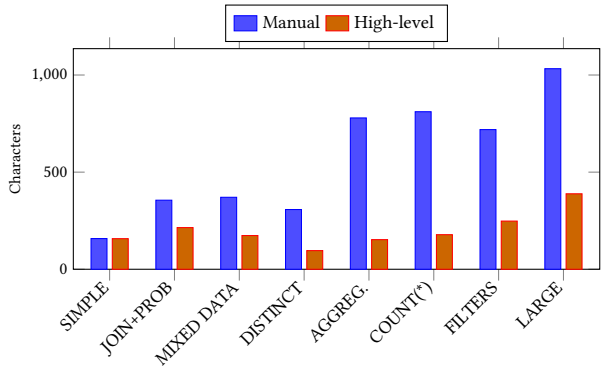


Fig. 2. Characters Comparison: Manual vs High-level

queries combine data from both certain (see Appendix C.2) and uncertain tables (see Appendix C.1).

- The manual queries were written as short as possible. For each of them, only one query (with subqueries when needed) was used.
- There are summary tables with relative reduction data, shown per experiment (for example, Table 1). The results are expressed as **percentages** for *code lines (CL)*, *character count*, and *level of complexity (LoC)*, and as **counts** for *probabilistic constructs*.

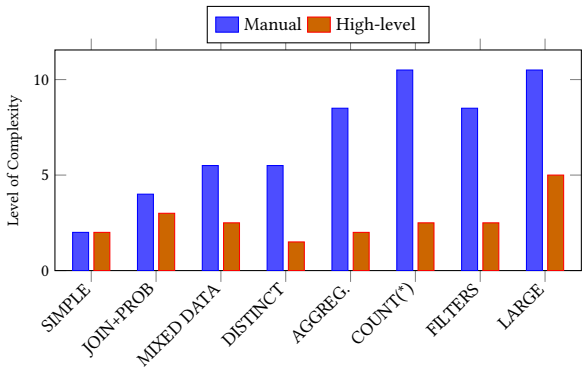


Fig. 3. Level of Complexity (LoC) Comparison: Manual vs High-level

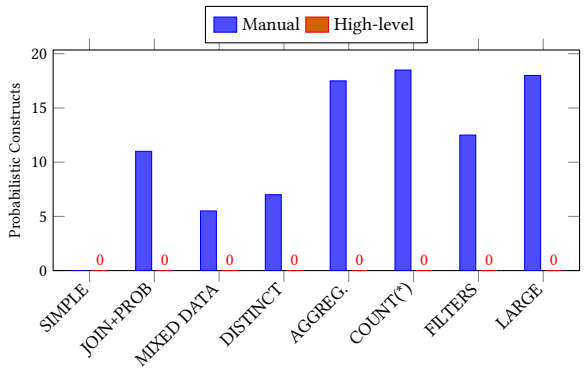


Fig. 4. Probabilistic Constructs Comparison: Manual vs High-level

  - The applied **percentage** measurement is based on the difference between the manual `M` and the high-level `H` or automatic `A` data, divided by the manual `M` and multiplied by `100` for any criterion: `(M-H)/M*100` for Experiment 7.1 and `(M-A)/M*100` for Experiment 7.2. The result may be `0` or negative, implying no reduction or additional overhead, which will be observed in Experiment 7.2.
  - The *probabilistic constructs* use summary **counts** instead of percentages, since there are none in the high-level DuoSQL queries (for instance, in Figure 4), as this is DuoSQL's goal.
  - The **Overall** rolls are calculated by taking the average of all values within a column. The **Overall excl. 1-2** takes all rows except for the first and the second ones that contain the simplest queries.
- Regarding the implementation, the only constraint is that all compared test sets (e.g., Manual vs. Automatic, or Manual vs. High-level) must share the same test types (also referred to as "query types") and number of queries per type.

6.2 Setup for Experiment 7.3

6.2.1 Metrics

- **Limit**: This metric addresses data volume. It shows the number of queried rows in the `LIMIT` clause in the innermost

Table 2. Reduction Summary: Manual vs Automatic

Type	CL %	Chars %	LoC %	Prob Constr. #
1. SIMPLE	-167%	-115%	-50%	0
2. JOIN + PROB	-100%	-71%	-50%	3
3. MIXED DATA	-15%	-25%	9%	1
4. DISTINCT	-16%	-46%	0%	0
5. AGGREGATION	-21%	-24%	-12%	0
6. COUNT(*)	-8%	-17%	0%	2
7. FILTERS	-10%	-16%	0%	2
8. LARGE	-9%	-9%	0%	2
Overall	**-43%**	**-40%**	**-13%**	**1**
Overall excl. 1-2	**-13%**	**-23%**	**-1%**	**1**

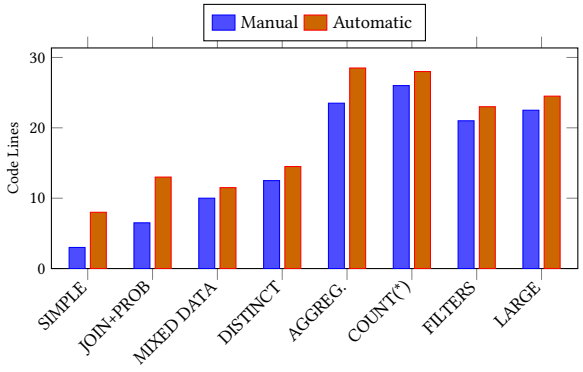


Fig. 5. Code Lines (CL) Comparison: Manual vs Automatic

subquery of the complex query. Appendix E shows a complex query. However, in this example, the `LIMIT` clause is in the main query, whereas for Experiment 7.3, the `LIMIT` clause is placed in the innermost subquery.

- **Tables Joined**: The count of queried tables in the innermost subquery of the complex query. In this experiment, each query involves one to four joined tables to address complexity.
- **Execution Time (sec)**: Measures the time in seconds from sending the query to PostgreSQL to retrieving a result.

6.2.2 Setup

To evaluate the runtime performance of DuoSQL, we compared the execution times of 4 automatically translated queries against 4 of their manual equivalents. Each query is of type `COUNT(*)` (with 1 table), `Aggregation` (with 2 tables), `Filter` (with 3 tables), or `Large` (with 4 tables), following almost the same format of the four most complex query types shown in Appendix F. We executed each query using `LIMIT` values of 5, 10, and stopped at 20, as the execution time increased significantly and took too long to measure after `LIMIT 20`.

7 RESULTS

7.1 High-level Technicality Reduction

This experiment compares the manual DuBio SQL queries with DuoSQL queries. The goal is to quantify the reduction of technical

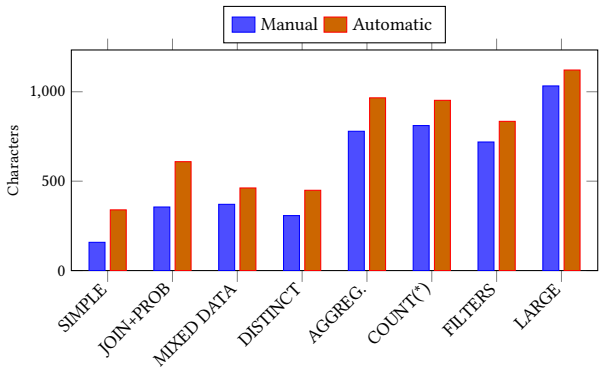


Fig. 6. Characters Comparison: Manual vs Automatic

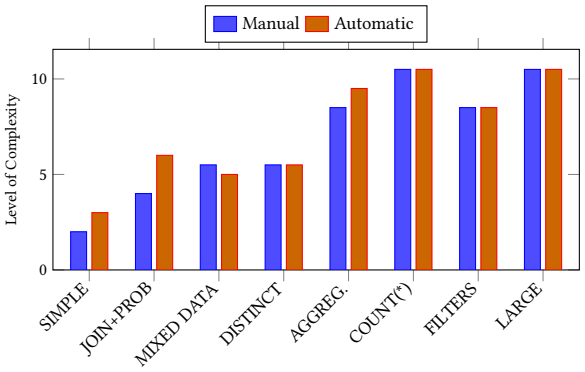


Fig. 7. Level of Complexity (LoC) Comparison: Manual vs Automatic

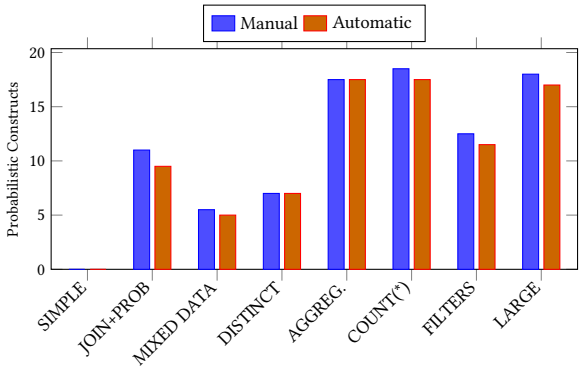


Fig. 8. Probabilistic Constructs Comparison: Manual vs Automatic

burden, measured in terms of code lines (CL), character count, level of complexity (LoC), and probabilistic constructs count.

The bar charts in Figures 1, 2, 3, and 4 visualize the comparative metrics for each query type. As expected, the results of the first two simplest query types do not differ much. However, as queries become more complex and technical, manual results surge, reaching approximately three times the high-level measurements. As we mentioned earlier, DuoSQL is designed to abstract low-level

(probabilistic) structures, explaining all the `0` measurements for the probabilistic constructs in Figure 4.

On average, DuoSQL reduces code lines by **50%**, character count by **56%**, complexity by **54%**, and probabilistic constructs by an average count of **23**. It can be observed that the more complex the query becomes, the higher the reduction is. Simple queries, points 1 and 2 in Table 1, showcase minimal reduction, implying that DuoSQL variants are directly in valid DuBio format and as short as possible; thus, they are taken literally in the manually translated queries.

Furthermore, when we exclude these basic cases, the reduction reaches **67%** on average for the three percentage criteria, as evaluated from the bar charts earlier. Hence, these measurements demonstrate DuoSQL's capabilities in offering more effortless probabilistic data management compared to writing manual queries.

7.2 Automatic Translation Overhead

This experiment highlights the syntactic overhead introduced by the translation logic. To assess the amount of generated overhead by the algorithm, we compare manual DuBio queries with automatically generated ones.

Figures 5, 6, 7, and 8 illustrate the results of this experiment. We can observe that manual queries perform better in keeping lower code lines and character counts. Furthermore, the LoC and the probabilistic construct measurements display fluctuations from both sides with simpler queries, but equalize with lengthier and more complex queries.

Table 2 shows the summary with relative values. The negative values indicate negative reduction, in other words, overhead generation. Although automatic translations introduce code lines and characters of **40%** more on average, they become almost similar in length to manual queries as they become more complex. On the other hand, the automatic level of complexity is much closer to the manual's, differing only by **13%** and averaging to **0%-1%** if we exclude the first 2 simple test types.

As mentioned in Section 6.1.2, the aim when creating manual queries was to make them as short as possible. However, shorter code does not always imply improved clarity and reduced complexity. By comparing a manual alternative, shown in Appendix E, of the last query in Appendix F, we can argue that the manually written code is more cluttered and less readable because it is more nested and not as clear as the automatically translated code which provides more modularity using dedicated views.

7.3 Runtime Performance Evaluation

Table 3 presents all measured execution times. We can observe that for most configurations, automatic queries perform similarly to manual ones, with only marginal differences. A notable exception is the aggregation, where both manual and automatic queries exhibit a large jump in execution time at `LIMIT = 20`. This suggests that data volume, rather than translation, is the primary contributor to execution time.

Furthermore, the surge is likely based on the number of joins. Since the queries with 3 and 4 tables connect only on the chosen overlapping fields, having 2 and 3 `ON` conditions, respectively, leads to fewer matching rows. However, the aggregation has 2 tables, with only 1 `ON` condition, implying easier pairing and more rows. For `COUNT(*)`, although it involves just one table, the complex structure of possible worlds means the `sentence` computations are more involved.

Table 3. Performance Testing Results: Manual vs Automatic

Type	Tables Joined	Limit	Execution Time (sec)
Manual COUNT(*)	1	5	3.1062
Manual Aggregation	2	5	0.0472
Manual Filters	3	5	0.0442
Manual Large	4	5	0.0431
Auto COUNT(*)	1	5	4.1126
Auto Aggregation	2	5	0.0491
Auto Filters	3	5	0.0456
Auto Large	4	5	0.0458
Manual COUNT(*)	1	10	70.4552
Manual Aggregation	2	10	0.4706
Manual Filters	3	10	0.0914
Manual Large	4	10	0.0441
Auto COUNT(*)	1	10	67.7769
Auto Aggregation	2	10	0.2345
Auto Filters	3	10	0.0750
Auto Large	4	10	0.0521
Manual COUNT(*)	1	20	69.3214
Manual Aggregation	2	20	413.9993
Manual Filters	3	20	0.0964
Manual Large	4	20	0.0509
Auto COUNT(*)	1	20	65.4577
Auto Aggregation	2	20	413.7770
Auto Filters	3	20	0.0889
Auto Large	4	20	0.0565

It should be noted that DuBio applies internal query optimizations, which may have influenced these results. Additionally, due to long runtimes and time constraints, we could not run the planned tests with `LIMIT = 50` and `LIMIT = 100`.

This evaluation demonstrates that the translation algorithm introduces negligible to no performance overhead.

8 ANSWERING THE RESEARCH QUESTIONS

RQ1: High-Level Design for Usability and Expressiveness. DuoSQL was successfully designed to preserve the expressiveness of DuBio and offer a more concise and user-friendly query interface. DuoSQL supports all core probabilistic features and uses natural SQL-style syntax. The translation process is fully automated and produces executable DuBio SQL without requiring manual intervention. This shows that expressiveness is preserved while usability is significantly improved.

RQ2: Reduction in Query Technicality. As observed in Section 7.1, DuoSQL provides a significant reduction in technical complexity compared to manually written queries in DuBio SQL. The reduction increases as the queries become more complex, proving DuoSQL's capabilities in simplifying sophisticated query logic. These results provide clear evidence that DuoSQL reduces query technicality.

RQ3: Translation Overhead. As described in Section 7.2, automatic translation from DuoSQL to DuBio SQL introduces some syntactic

overhead due to additional view definitions, explicit aliasing, and modularization. This overhead results in approximately **40%** more code lines and characters on average in simpler queries. However, as query complexity increases, this difference decreases substantially. The level of complexity remains within a small margin on average and converges to (almost) zero in more complex queries. The overhead is thus considered acceptable, as it trades minor verbosity for a more structured and explainable query format. Therefore, while overhead exists, it does not compromise clarity or correctness.

RQ4: Performance Evaluation. In all experiments, DuoSQL translations perform similarly to manual DuBio SQL. Execution time scales primarily with complexity and data size, rather than with the use of automatic translation.

9 CONCLUSION

This research introduced the user-friendly DuoSQL, a domain-specific query language, and a corresponding compiler algorithm that can improve the usability of probabilistic databases. By abstracting away the low-level structures of DuBio, DuoSQL serves users in writing queries over uncertain data without requiring deep knowledge of probabilistic semantics or internal constructs.

Through a series of experiments, the DuoSQL language demonstrated substantial technical simplification, reducing verbosity and complexity in comparison to manually written DuBio SQL. At the same time, the compiler algorithm preserves semantic correctness and modularity, while introducing only minor syntactic overhead, which does not negatively impact performance. The translation process was formalized into a generic and reusable algorithm that can be embedded in any language with PostgreSQL and DuBio connectivity.

Together, these contributions offer a step forward in making uncertain data management accessible to a broader range of users.

10 FUTURE WORK AND DISCUSSION

10.1 Support for Advanced SQL Constructs

While DuoSQL currently supports a broad subset of SQL constructs, several advanced features remain to be implemented or improved:

- **Expression support in `SELECT`:** The current SELECT clause parsing relies on simple comma-separated fields and does not support function-based expressions such as `CONCAT(color, breed)`. Refactoring the underlying regular expressions will allow support for such expressions and other nested function calls.
- **User-defined views:** Currently, DuoSQL-generated queries rely on compiler-defined intermediate views (e.g., `join_view`, `agg_view`). Supporting user-defined views could enhance modularity and allow users to express reusable logic. The implementation of this feature would be analogous to the handling of ordinary queries.
- **Subqueries:** Full support for subqueries would likely be one of the most complex challenges. However, introducing user-defined views may reduce the need for inline subqueries and serve as a practical alternative.
- **Multiple aggregations:** Currently, DuoSQL supports only a single aggregate function per query. Support for multiple aggregates (e.g., `SELECT cat_name, COUNT(color), AVG(age)`) would require complex nested logic. Perhaps a better alternative could be a rewriting strategy that separates aggregations into individual views and joins the results afterwards by a common (group-by) key.
- **Combination of `DISTINCT` and aggregation:** Supporting queries that combine `DISTINCT` with aggregate functions is another challenge that has not been addressed in this project.
- **Data modification support:** Supporting `INSERT`, `DELETE`, `UPDATE`, and conditioning (incorporating new evidence) in probabilistic tables, as well as the creation of probabilistic tables. These capabilities would broaden DuoSQL's applicability from querying to full data lifecycle management.
- **Advanced probabilistic data integration tasks:** Introducing direct commands for probabilistic deduplication of a table, for joining or merging tables based on a similarity function, and others would support common data integration tasks and facilitate more intelligent data cleaning pipelines.

10.2 Built-in DuoSQL Execution Mode in DuBio

Currently, DuoSQL queries must be translated externally into DuBio SQL before execution in PostgreSQL. A promising direction for future development is to integrate DuoSQL natively into DuBio as a preprocessing layer.

This could be implemented by introducing a new query clause, such as `MODE`, that allows users to specify the intended syntax:

```
MODE DuoSQL;
-- or
MODE DuBio;
```

Under this model, the database engine would detect the mode and automatically preprocess DuoSQL queries into their DuBio SQL equivalents before execution. This would eliminate the need for external translation scripts.

11 ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof.Dr.Ir. Maurice van Keulen, for his guidance, commitment, and contagious excitement during this research. I am also grateful to Mihail Stavrev for providing ideas on more extensive real-world uncertain data cases.

12 AI STATEMENT

During this research project, I used ChatGPT to assist in transforming notes and sentences to an academic style or to check their correctness and provide feedback. I also used ChatGPT to find a correct and efficient DuoSQL grammar extraction through code, brainstorm programming principles and practices for the realization of such a project, and to find and resolve bugs in my code.

All ideas, structures, and decisions in this research paper and code were, at the bare minimum, noted in advance and carefully redacted, reorganized, and reviewed after any AI suggestions and before any incorporation into the project. I take full responsibility for the content of this work.

REFERENCES

[1] Maurizio Magnani and Danilo Montesi. 2010. A survey on uncertainty management in data integration. *Journal of Data and Information Quality (JDIQ)* 2, 1 (2010), 5. https://doi.org/10.1145/1805286.1805291

[2] Maurice van Keulen. 2018. Probabilistic data integration. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Zomaya (Eds.). Springer. https://doi.org/10.1007/978-3-319-63962-8_18-1

[3] Nilesh Dalvi, Christopher Ré, and Dan Suciu. 2009. Probabilistic databases: Diamonds in the dirt. *Communications of the ACM* 52, 7 (2009), 86–94. https://doi.org/10.1145/1538788.1538810

[4] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. 2008. Fast and simple relational processing of uncertain data. In *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE)*. IEEE. https://doi.org/10.1109/ICDE.2008.4497507

[5] Subi Arumugam, Fei Xu, Ravi Jampani, Christopher Jermaine, Luis L. Perez, and Peter J. Haas. 2010. MCDB-R: Risk analysis in the database. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 782–793. https://doi.org/10.14778/1920841.1920941

[6] Jennifer Widom. 2006. Trio: A System for Data, Uncertainty, and Lineage. In *Managing and Mining Uncertain Data*, Charu C. Aggarwal (Ed.). Springer, 1–35. https://doi.org/10.1007/978-0-387-09690-2_5

[7] Maurice van Keulen. DuBio Wiki. https://github.com/utwente-db/DuBio/wiki
Accessed: 10-05-2025

[8] The PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/about/
Accessed: 16-06-2025

[9] The PostgreSQL Global Development Group. PostgreSQL Documentation. SELECT Synopsis. https://www.postgresql.org/docs/current/sql-select.html
Accessed: 16-06-2025

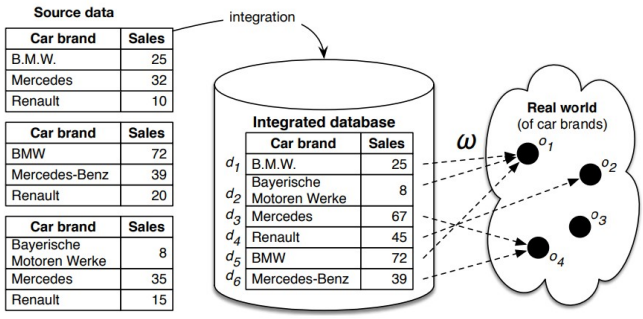
[10] Martin Demirev. DuoSQL. https://github.com/DemirevMartin/DuoSQL
Accessed: 22-06-2025

[11] Jochem Groot Roessink. 2021. inSQeLto: a Query Language for Probabilistic Databases. In *35th Twente Student Conference on IT*, University of Twente. https://essay.utwente.nl/86906/

[12] Python Software Foundation. Python Programming Language. https://www.python.org/.
Accessed: 10-05-2025

APPENDIX

A SEMANTIC DUPLICATES EXAMPLE [2]



B PROBABILISTIC QUERY RESULT

companion	witness	color	breed	probability	_sentence
henry	cathy	white	persian	1.0000	Bdd((w3=1 & p3=1))
frank	alice	gray	tabby	0.2500	Bdd((w1=2 & p1=2))
frank	alice	white	siamese	0.2500	Bdd((w1=1 & p1=1))
grace	ben	black	mainecoon	0.2800	Bdd((w2=1 & p2=1))
grace	ben	gray	mainecoon	0.1800	Bdd((w2=2 & p2=2))

C TABLE DESIGNS

C.1 Uncertain Tables

This section shows the design of the 4 uncertain tables. Their structures are almost identical except for one field - the person in different scenarios. The { } notation matches the tables with their corresponding field name - witnessed with witness, plays with companion, cares with caretaker, owns with owner.

```
CREATE TABLE { witnessed | plays | cares | owns } (
  id integer,
  { witness | companion | caretaker | owner } text,
  cat_name text,
  breed text,
  color text,
  age integer,
  _sentence Bdd
);
```

C.2 Certain Table

```
CREATE TABLE profile_certain (
  cat_id integer,
  cat_name text
);
```

D AGGREGATION

This section presents the underlying DuBio logic for performing aggregation queries over uncertain data using BDDs. It includes the custom-defined aggregate functions, standard aggregation queries with GROUP BY, support for COUNT(*), and distinct value computation.

D.1 DuBio Helper Functions

```
CREATE or REPLACE FUNCTION prob_Bdd(a_s Bdd[], mask bit)
RETURNS Bdd
LANGUAGE plpgsql AS $$
DECLARE
  result    Bdd = Bdd('1');
  mask_len  integer = array_length(a_s, 1);
  mask_start integer = length(mask) - mask_len - 1;
  n_bdd     Bdd;
BEGIN
  FOR i IN 1 .. mask_len LOOP
    IF get_bit(mask,mask_start+i) = 1 THEN
      n_bdd = a_s[i];
    ELSE
      n_bdd = !a_s[i];
    END IF;
    result := result & n_bdd;
  END LOOP;
  RETURN result;
END;
$$;

CREATE or REPLACE FUNCTION prob_Bdd_count(a_b anyarray, mask bit)
RETURNS int
LANGUAGE plpgsql AS $$
DECLARE
  result    integer = 0;
  mask_len  integer = array_length(a_b, 1);
  mask_start integer = length(mask) - mask_len - 1;
BEGIN
  FOR i IN 1 .. mask_len LOOP
    IF get_bit(mask,mask_start+i) = 1 THEN
      result := result + 1;
    END IF;
  END LOOP;
  RETURN result;
END;
$$;
```

D.2 DuBio Aggregation Query

```
DROP VIEW IF EXISTS prob_view CASCADE;
DROP VIEW IF EXISTS agg_view CASCADE;

CREATE OR REPLACE VIEW agg_view AS
SELECT cat_name, avg, agg_or(_sentence) AS _sentence
FROM (
  SELECT cat_name, prob_Bdd_avg(arr,mask) AS avg, prob_Bdd(
       arr_sentence,mask) AS _sentence, arr, arr_sentence, mask
  FROM (
    SELECT cat_name, arr, arr_sentence,
      generate_series(0, (pow(2, array_length(arr_sentence,1))-1)::
           bigint)::bit(64) AS mask
    FROM (
      SELECT cat_name, array_agg(age) arr, array_agg(_sentence)
           arr_sentence
      FROM (
        SELECT cat_name, age, agg_or(_sentence) AS _sentence
        FROM witnessed
        GROUP BY cat_name, age
      ) AS first
      GROUP BY cat_name
    ) AS second
  ) AS third
) AS forth
GROUP BY cat_name, avg;

CREATE OR REPLACE VIEW prob_view AS
SELECT v.*, round(prob(d.dict, v._sentence)::numeric, 4) AS
      probability
FROM agg_view v
JOIN _dict d ON d.name = 'cats_short';

SELECT cat_name, avg, probability, _sentence
```

```
FROM prob_view
WHERE avg > 2
ORDER BY cat_name
LIMIT 10;
```

D.3 DuBio Aggregation All * Query

```
DROP VIEW IF EXISTS prob_view CASCADE;
DROP VIEW IF EXISTS agg_all_view CASCADE;

CREATE OR REPLACE VIEW agg_all_view AS
SELECT count_rows, agg_or(_sentence) AS _sentence
FROM (
  SELECT prob_Bdd_count(arr,mask) AS count_rows, prob_Bdd(
       arr_sentence,mask) AS _sentence, arr, arr_sentence, mask
  FROM (
    SELECT arr, arr_sentence,
      generate_series(0, (pow(2, array_length(arr_sentence,1))-1)::
           bigint)::bit(64) AS mask
    FROM (
      SELECT array_agg(cat_name) arr, array_agg(_sentence) AS
           arr_sentence
      FROM (
        SELECT cat_name, agg_or(_sentence) AS _sentence
        FROM witnessed
        WHERE cat_name = 'max'
        GROUP BY cat_name
      ) AS first
      GROUP BY TRUE
    ) AS second
  ) AS third
) AS forth
GROUP BY count_rows;

CREATE OR REPLACE VIEW prob_view AS
SELECT v.*, round(prob(d.dict, v._sentence)::numeric, 4) AS
     probability
FROM agg_all_view v
JOIN _dict d ON d.name = 'cats_short';

SELECT count_rows, probability, _sentence
FROM prob_view
WHERE probability > 0 AND count_rows > 0;
```

D.4 DuBio DISTINCT Query

```
DROP VIEW IF EXISTS prob_view CASCADE;
DROP VIEW IF EXISTS join_view CASCADE;

CREATE OR REPLACE VIEW join_view AS
SELECT color, agg_or(plays._sentence) AS _sentence
FROM plays
GROUP BY color;

CREATE OR REPLACE VIEW prob_view AS
SELECT v.*, round(prob(d.dict, v._sentence)::numeric, 4) AS
     probability
FROM join_view v
JOIN _dict d ON d.name = 'cats_short';

SELECT color
FROM prob_view
WHERE probability > 0.5
ORDER BY color;
```

E MANUAL QUERY EXAMPLE

The given query is the manual alternative to the last query in Appendix F.

```
SELECT witness, companion, caretaker, owner, cat_name, color_count,
     probability, _sentence
FROM (
  SELECT witness, companion, caretaker, owner, cat_name, color_count,
       round(prob(d.dict, _sentence)::numeric, 4) AS probability,
       _sentence
  FROM (
    SELECT witness, companion, caretaker, owner, cat_name,
         color_count, agg_or(_sentence) AS _sentence
    FROM (
      SELECT witness, companion, caretaker, owner, cat_name,
           prob_Bdd_count(arr,mask) AS color_count, prob_Bdd(
           arr_sentence,mask) AS _sentence, arr, arr_sentence, mask
      FROM (
        SELECT witness, companion, caretaker, owner, cat_name, arr,
             arr_sentence, generate_series(0,(pow(2,array_length(
             arr_sentence,1))-1)::bigint)::bit(64) AS mask
        FROM (
          SELECT witness, companion, caretaker, owner, cat_name,
               array_agg(color) arr, array_agg(_sentence)
               arr_sentence
          FROM (
            SELECT w.witness, p.companion, c.caretaker, o.owner, w.
                 cat_name, w.color, w._sentence & p._sentence & c.
                 _sentence & o._sentence AS _sentence
            FROM witnessed w
            JOIN plays p ON w.cat_name = p.cat_name
            JOIN cares c ON w.cat_name = c.cat_name
            JOIN owns o ON w.cat_name = o.cat_name
            WHERE w.cat_name = 'max'
          ) AS first
          GROUP BY witness, companion, caretaker, owner, cat_name
        ) AS second
      ) AS third
    ) AS forth
    GROUP BY witness, companion, caretaker, owner, cat_name,
         color_count
  ) AS fifth
  JOIN _dict d ON d.name = 'cats_short'
) AS sixth
WHERE color_count > 0 AND probability > 0
ORDER BY witness DESC, probability ASC
LIMIT 10;
```

F TRANSLATION MAPPING OF DUOSQL TO DUBIO SQL

This section shows 1 query for each test type. There are comments, noting the test name and number, at the beginning of every DuoSQL query.

DuoSQL Query	Translated DuBio SQL

`-- 1. Simple Query`
`SELECT p.companion, w.witness, w.cat_name, w.color, w.breed, w.age`
`FROM witnessed w`
`JOIN plays p ON w.cat_name = p.cat_name AND w.color = p.color`
	`DROP VIEW IF EXISTS prob_view CASCADE;`
	`DROP VIEW IF EXISTS join_view CASCADE;`
	
	`CREATE OR REPLACE VIEW join_view AS`
	`SELECT p.companion, w.witness, w.cat_name, w.color, w.breed, w.age`
	`FROM witnessed w`
	`JOIN plays p ON w.cat_name = p.cat_name AND w.color = p.color;`
	
	`SELECT companion, witness, cat_name, color, breed, age`
	`FROM join_view;`

`-- 2. JOIN + Probability`
`SELECT w.witness, p.companion AS player, c.caretaker, o.owner, w.cat_name`
`FROM witnessed w`
`JOIN plays p ON w.cat_name = p.cat_name`
`JOIN cares c ON w.cat_name = c.cat_name`
`JOIN owns o ON w.cat_name = o.cat_name`
`SHOW SENTENCE, PROBABILITY`
	`DROP VIEW IF EXISTS prob_view CASCADE;`
	`DROP VIEW IF EXISTS join_view CASCADE;`
	
	`CREATE OR REPLACE VIEW join_view AS`
	`SELECT w.witness, p.companion AS player, c.caretaker, o.owner, w.cat_name, w._sentence & p._sentence & c._sentence & o._sentence AS _sentence`
	`FROM witnessed w`
	`JOIN plays p ON w.cat_name = p.cat_name`
	`JOIN cares c ON w.cat_name = c.cat_name`
	`JOIN owns o ON w.cat_name = o.cat_name;`
	
	`CREATE OR REPLACE VIEW prob_view AS`
	`SELECT v.*, round(prob(d.dict, v._sentence)::numeric, 4) AS probability`
	`FROM join_view v`
	`JOIN _dict d ON d.name = 'cats_short';`
	
	`SELECT witness, player, caretaker, owner, cat_name, probability, _sentence`
	`FROM prob_view;`

`-- 3. Mixed Data`
`SELECT c.caretaker, pc.cat_id, c.cat_name, c.breed, c.age`
`FROM cares c`
`JOIN profile_certain pc ON c.cat_name = pc.cat_name`
`JOIN owns o ON c.cat_name = o.cat_name`
`SHOW SENTENCE;`
	`DROP VIEW IF EXISTS prob_view CASCADE;`
	`DROP VIEW IF EXISTS join_view CASCADE;`
	
	`CREATE OR REPLACE VIEW join_view AS`
	`SELECT c.caretaker, pc.cat_id, c.cat_name, c.breed, c.age, c._sentence & o._sentence AS _sentence`
	`FROM cares c`
	`JOIN profile_certain pc ON c.cat_name = pc.cat_name`
	`JOIN owns o ON c.cat_name = o.cat_name;`
	
	`SELECT caretaker, cat_id, cat_name, breed, age, _sentence`
	`FROM join_view;`

DuoSQL Query	Translated DuBio SQL

`-- 4. DISTINCT`
`SELECT DISTINCT p.age`
`FROM plays p`
`JOIN witnessed w ON w.cat_name = p.cat_name`
`HAVING probability > 0.5`
`ORDER BY p.age;`
	`DROP VIEW IF EXISTS prob_view CASCADE;`
	`DROP VIEW IF EXISTS join_view CASCADE;`
	
	`CREATE OR REPLACE VIEW join_view AS`
	`SELECT p.age, agg_or(p._sentence & w._sentence) AS _sentence`
	`FROM plays p`
	`JOIN witnessed w ON w.cat_name = p.cat_name`
	`GROUP BY p.age;`
	
	`CREATE OR REPLACE VIEW prob_view AS`
	`SELECT v.*, round(prob(d.dict, v._sentence)::numeric, 4) AS probability`
	`FROM join_view v`
	`JOIN _dict d ON d.name = 'cats_short';`
	
	`SELECT age`
	`FROM prob_view`
	`WHERE probability > 0.5`
	`ORDER BY age;`

`-- 5. Aggregation`
`SELECT w.cat_name, count(companion)`
`FROM plays p`
`JOIN witnessed w ON w.cat_name = p.cat_name`
`WHERE w.color = p.color`
`GROUP BY w.cat_name`
`ORDER BY w.cat_name`
`SHOW SENTENCE, PROBABILITY`
	`DROP VIEW IF EXISTS prob_view CASCADE;`
	`DROP VIEW IF EXISTS agg_view CASCADE;`
	
	`CREATE OR REPLACE VIEW agg_view AS`
	`SELECT cat_name, count, agg_or(_sentence) AS _sentence`
	`FROM (`
	`  SELECT cat_name, prob_Bdd_count(arr,mask) AS count, prob_Bdd(arr_sentence,mask) AS`
	`  _sentence, arr, arr_sentence, mask`
	`  FROM (`
	`  SELECT cat_name, arr, arr_sentence, generate_series(0,(pow(2,array_length(arr_sentence`
	`  ,1))-1)::bigint)::bit(64) AS mask`
	`  FROM (`
	`  SELECT cat_name, array_agg(companion) arr, array_agg(_sentence) arr_sentence`
	`  FROM (`
	`  SELECT w.cat_name, companion, p._sentence & w._sentence AS _sentence`
	`  FROM plays p`
	`  JOIN witnessed w ON w.cat_name = p.cat_name`
	`  WHERE w.color = p.color`
	`  ) AS first`
	`  GROUP BY cat_name`
	`  ) AS second`
	`  ) AS third`
	`) AS forth`
	`GROUP BY cat_name, count;`
	
	`CREATE OR REPLACE VIEW prob_view AS`
	`SELECT v.*, round(prob(d.dict, v._sentence)::numeric, 4) AS probability`
	`FROM agg_view v`
	`JOIN _dict d ON d.name = 'cats_short';`
	
	`SELECT cat_name, count, probability, _sentence`
	`FROM prob_view`
	`ORDER BY cat_name;`

DuoSQL Query

```
-- 6. COUNT(*)
SELECT w.cat_name, COUNT(*) as
      count_rows
FROM witnessed w
JOIN plays p ON w.cat_name = p.
      cat_name
WHERE p.color IN ('gray', 'black')
GROUP BY w.cat_name
HAVING count_rows > 0
SHOW PROBABILITY
```

Translated DuBio SQL

```
DROP VIEW IF EXISTS prob_view CASCADE;
DROP VIEW IF EXISTS agg_all_view CASCADE;

CREATE OR REPLACE VIEW agg_all_view AS
SELECT count_rows, agg_or(_sentence) AS _sentence
FROM (
  SELECT prob_Bdd_count(arr,mask) AS count_rows, prob_Bdd(arr_sentence,mask) AS _sentence, arr,
       arr_sentence, mask
  FROM (
    SELECT arr, arr_sentence, generate_series(0,(pow(2,array_length(arr_sentence,1))-1)::bigint)::bit
         (64) AS mask
    FROM (
      SELECT array_agg(cat_name) arr, array_agg(_sentence) AS arr_sentence
      FROM (
        SELECT w.cat_name, w._sentence & p._sentence AS _sentence
        FROM witnessed w
        JOIN plays p ON w.cat_name = p.cat_name
        WHERE p.color IN ('gray', 'black')
      ) AS first
      GROUP BY TRUE
    ) AS second
  ) AS third
) AS forth
GROUP BY count_rows;

CREATE OR REPLACE VIEW prob_view AS
SELECT v.*, round(prob(d.dict, v._sentence)::numeric, 4) AS probability
FROM agg_all_view v
JOIN _dict d ON d.name = 'cats_short';

SELECT count_rows, probability, _sentence
FROM prob_view
WHERE count_rows > 0;
```

```
-- 7. Filters
SELECT cat_name, COUNT(color)
FROM witnessed
WHERE color IN ('white', 'black')
GROUP BY cat_name
HAVING COUNT(color) > 0 AND
      probability > 0
ORDER BY probability ASC
LIMIT 10
SHOW SENTENCE, PROBABILITY
```

```
DROP VIEW IF EXISTS prob_view CASCADE;
DROP VIEW IF EXISTS agg_view CASCADE;

CREATE OR REPLACE VIEW agg_view AS
SELECT cat_name, COUNT, agg_or(_sentence) AS _sentence
FROM (
  SELECT cat_name, prob_Bdd_count(arr,mask) AS COUNT, prob_Bdd(arr_sentence,mask) AS _sentence, arr,
       arr_sentence, mask
  FROM (
    SELECT cat_name, arr, arr_sentence, generate_series(0,(pow(2,array_length(arr_sentence,1))-1)::
         bigint)::bit(64) AS mask
    FROM (
      SELECT cat_name, array_agg(color) arr, array_agg(_sentence) arr_sentence
      FROM (
        SELECT cat_name, color, agg_or(_sentence) AS _sentence
        FROM witnessed
        WHERE color IN ('white', 'black')
        GROUP BY cat_name, color
      ) AS first
      GROUP BY cat_name
    ) AS second
  ) AS third
) AS forth
GROUP BY cat_name, COUNT;

CREATE OR REPLACE VIEW prob_view AS
SELECT v.*, round(prob(d.dict, v._sentence)::numeric, 4) AS probability
FROM agg_view v
JOIN _dict d ON d.name = 'cats_short';

SELECT cat_name, COUNT, probability, _sentence
FROM prob_view
WHERE COUNT > 0 AND probability > 0
ORDER BY probability ASC
LIMIT 10;
```

DuoSQL Query

```
-- 8. Large Query
SELECT w.witness, p.companion AS player, c.
      caretaker, o.owner, count(w.color) AS
      color_count
FROM witnessed w
JOIN plays p ON w.cat_name = p.cat_name
JOIN cares c ON w.cat_name = c.cat_name
JOIN owns o ON w.cat_name = o.cat_name
WHERE w.cat_name = 'max'
GROUP BY w.witness, p.companion, c.
      caretaker, o.owner, w.cat_name
HAVING probability > 0 AND color_count > 0
ORDER BY w.witness DESC, probability ASC
LIMIT 10
SHOW SENTENCE, PROBABILITY
```

Translated DuBio SQL

```
DROP VIEW IF EXISTS prob_view CASCADE;
DROP VIEW IF EXISTS agg_view CASCADE;

CREATE OR REPLACE VIEW agg_view AS
SELECT witness, companion, caretaker, owner, cat_name, color_count, agg_or(_sentence) AS
     _sentence
FROM (
  SELECT witness, companion, caretaker, owner, cat_name, prob_Bdd_count(arr,mask) AS color_count
       , prob_Bdd(arr_sentence,mask) AS _sentence, arr, arr_sentence, mask
  FROM (
    SELECT witness, companion, caretaker, owner, cat_name, arr, arr_sentence, generate_series
         (0,(pow(2,array_length(arr_sentence,1))-1)::bigint)::bit(64) AS mask
    FROM (
      SELECT witness, companion, caretaker, owner, cat_name, array_agg(color) arr, array_agg(
           _sentence) arr_sentence
      FROM (
        SELECT w.witness, p.companion, c.caretaker, o.owner, w.cat_name, w.color, w._sentence & p
             ._sentence & c._sentence & o._sentence AS _sentence
        FROM witnessed w
        JOIN plays p ON w.cat_name = p.cat_name
        JOIN cares c ON w.cat_name = c.cat_name
        JOIN owns o ON w.cat_name = o.cat_name
        WHERE w.cat_name = 'max'
      ) AS first
      GROUP BY witness, companion, caretaker, owner, cat_name
    ) AS second
  ) AS third
) AS forth
GROUP BY witness, companion, caretaker, owner, cat_name, color_count;

CREATE OR REPLACE VIEW prob_view AS
SELECT v.*, round(prob(d.dict, v._sentence)::numeric, 4) AS probability
FROM agg_view v
JOIN _dict d ON d.name = 'cats_short';

SELECT witness, companion, caretaker, owner, cat_name, color_count, probability, _sentence
FROM prob_view
WHERE probability > 0 AND color_count > 0
ORDER BY witness DESC, probability ASC
LIMIT 10;
```