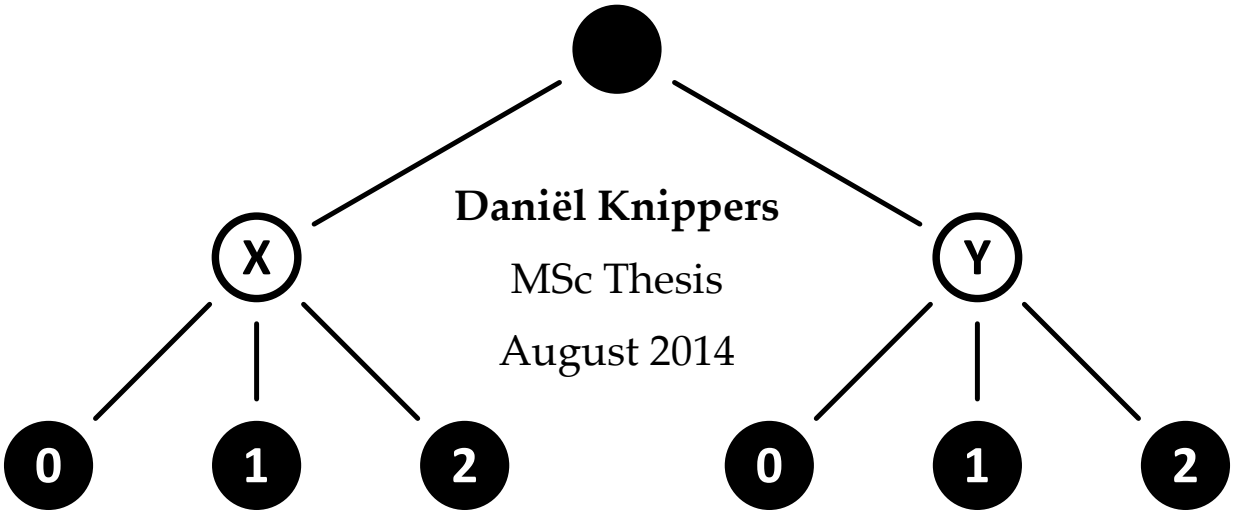
UNIVERSITY OF TWENTE.

Querying Uncertain Data in XML



Daniël Knippers

MSc Thesis

August 2014

Graduation committee

Dr. ir. Maurice van Keulen

Dr. Mena Badieh Habib Morgan

Abstract

This thesis describes the design and implementation of an extension for an XML DBMS which enables the execution of XPath queries over uncertain data. Uncertain data is different from regular data in that in addition to a value there is an associated probability for each item. An implication is that an uncertain dataset represents many different states; one for each combination of alternatives for all uncertain data items. Each state is referred to as a *possible world*. Each possible world has an associated probability itself but contains no uncertain values since an alternative was chosen for each uncertain value. The probabilities of chosen alternatives determine the probability of the possible world. A major problem is the exponential growth of the number of possible worlds with respect to the number of uncertain values.

We describe a way to query the uncertain data directly; without possible world expansion. An XML data format for uncertain data is defined which supports local independence and mutual exclusion relations among different values through random variable annotations. Correct query evaluation over uncertain data is achieved by transforming an input XPath query to an XQuery which keeps track of the random variable annotations that are used to select only consistent values and to compute the probabilities of resulting values. The transformed query is executed by the XML DBMS using its native – i.e., unchanged by our extension – query evaluation implementation.

The implementation can handle the aggregation functions Count, Sum, Min, and Max in addition to regular XPath queries. For these aggregation functions we yield a summary of the results, which describes the distribution of the resulting values. That is, we provide the minimum value, expected value, maximum value, variance, and standard deviation for each aggregation function. For the aggregates Min and Max we additionally compute the top-k result values. The result of a non-aggregation query is set of distinct result values, each with an associated probability. The probability is the sum of probabilities of all possible worlds represented by the uncertain data that yield the value as a result to the query.

Benchmarks indicate the execution time of our implementation scales roughly linearly with respect to the size of the document containing the uncertain data for various queries. There are some cases where this does not hold; in particular when using multiple consecutive nonselective predicates on the same context node. A predicate is nonselective when it is satisfied by many elements. A conjunction of predicates is evaluated in the context of uncertain data by generating the Cartesian product of all predicates which causes performance issues when each predicate generates a large set of matching values.

Contents

1 Introduction

Databases are used to store information to be retrieved at a later time. In most cases the information stored in a database is certain; there is only one option for each data item. For example, passenger information stored by an airline company or student grades stored in the university database are all certain information. When retrieving a student's grade for a specific course the answer will always be a single possible value. In contrast, uncertain data is characterized by having multiple options for each data item; the value of the item is uncertain. Each of the possible choices will have an associated probability, indicating the likelihood that it will be "selected" as the value of the item. The term "uncertain dataset" might give the idea that all of the data it describes is uncertain. This is not the case, as all data items in an uncertain dataset without a probability are certain, just like in a regular database. This does not necessarily mean such "certain" items are always "selected", since they might depend on an uncertain option (that is, one of their ancestors is uncertain) which by definition is not always picked.

There are many application scenarios for uncertain databases, all of which involve a degree of uncertainty associated with the data that is being processed. For example, any scenario involving predictions about future events deals with uncertainty since predictions are inherently uncertain. A company might have made various predictions about the unit sales of its products across the different countries it operates in, based on market research and other means. Based on the credibility of the bureau carrying out the market research, or on historic sales data the company has associated different levels of confidence with each prediction. When creating strategies for production, marketing and logistics the company is interested in the expected number of total sales for each product, or the expected number of combined sales in a specific country. A traditional database that stores the predictions cannot answer those queries as it cannot handle the probabilities that are associated with the different data items required to produce the answer. A probabilistic database will be able to deal with the uncertainty and provide answers to such queries, generally consisting of multiple possibilities each with an associated probability.

Another scenario involving uncertainty is merging multiple datasets that contain information about a similar topic into a single unified dataset. For instance, consider a scenario where we are merging datasets containing metadata of scientific publications such as the author names, the journal of publication, the title of the research and so on. There might be slight differences between the various sources regarding the same publication such as a different spelling of an author name or a different publication year. Instead of picking one of the possible options and throwing the other ones away based on the confidence we have in a specific source, it is possible to store all options with their probabilities in a probabilistic database. This is especially valuable when there is a difference between sources that have a similar level of confidence, in which case either option could be the "right" option. When we execute a query over the merged dataset we are presented with a query answer that consists of multiple possibilities and their probabilities. Being able to store various possible options for a single data item with different probabilities is an essential difference between probabilistic databases and traditional databases.

1.1 Possible Worlds

Uncertainty gives rise to the concept of possible worlds. A dataset consisting entirely of certain values will represent a single possible world; there is no chance of any other representation of the data than that which is there. An uncertain dataset, which by definition has different options for at least one data item it contains, will represent multiple possible worlds. Each combination of all possible options in the entire dataset is a distinct possible world, each with an associated probability which is obtained by multiplying the individual probabilities of the options that are selected for the possible world.

The possible world concept will be illustrated using a small example of an uncertain dataset; weather forecasts for a number of days. Like most predictions about future events that are not fully deterministic, weather forecasts are inherently uncertain. A typical weather forecast contains predictions of weather-related properties such as temperature, rainfall, and wind speed. Figure 1.1 shows an XML representation of a simple dataset with a two-day weather forecast, only containing temperature values to keep it concise. The temperature of each day has two possible values with different probabilities.

This uncertain dataset represents a total of 4 possible worlds; one for each of the 4 possible combinations

```
<forecasts>
    <forecast day="1">
        <temperature probability="0.7">16</temperature>
        <temperature probability="0.3">20</temperature>
    </forecast>
    <forecast day="2">
        <temperature probability="0.4">12</temperature>
        <temperature probability="0.6">18</temperature>
    </forecast>
</forecasts>
```

Figure 1.1: Uncertain dataset representing weather forecasts

of temperature values of day 1 and day 2. For instance, one possible world is generated by selecting the first temperature value for both days, displayed Figure 1.2 below. The probability of this possible world is the product of individual probabilities of the selected temperature values, thus $0.7 \cdot 0.4 = 0.28$. The other 3 possible worlds are created in a similar way. The sum of probabilities of all possible worlds is exactly 1.

```
<forecasts>
    <forecast day="1">
        <temperature>16</temperature>
    </forecast>
    <forecast day="2">
        <temperature>12</temperature>
    </forecast>
</forecasts>
```

Figure 1.2: One possible world, with $p = 0.28$

Creating a possible world from an uncertain dataset is referred to as instantiation. During instantiation an option is selected for all uncertain values. The result of the instantiation, therefore, is a dataset without any uncertain values. As a result, the probability attributes are removed in Figure 1.2. A key property to notice is that the number of possible worlds increases exponentially with respect to the number of uncertain values in the dataset. For instance, if a third forecast element would be added with again two choices for each of its temperature values the number of possible worlds would be doubled compared to the situation with just two forecasts. Similarly, adding a third forecast with four possibilities for each temperature value instead of two would quadruple the number of possible worlds.

The naive way of answering a query over uncertain data is to instantiate all possible worlds and execute the query in each world. The query answer will then consist of these individual answers and their probabilities. However, due to the exponential growth of the number of possible worlds this quickly becomes impossible. The main goal of this research is therefore to answer queries over uncertain data without explicitly enumerating all possible worlds due to the obvious scalability problems it poses.

1.2 Probabilistic XML

The concept probabilistic XML refers to a probability distribution defined over a set of ordinary documents. Typically, probabilistic XML models define the distribution using two types of nodes; distributional nodes which specify the type of distribution and ordinary nodes which are regular XML nodes that appear in a resulting document. The distributional nodes do not appear in a document resulting from the probabilistic process. Such a document is referred to as a random document [1]. The previous section described the notion of a random document as a possible world. In their work, Van Keulen et al.

[2] introduce a probabilistic XML model which defines two types of distributional nodes; (1) probability nodes which represent an uncertain value (or more specifically; an uncertain subtree) and (2) possibility nodes that define the possible values, each with an associated probability. The probabilities of possibility nodes belonging to the same probability node sum to exactly 1. In order to generate a random document, the tree represented by the probabilistic document is traversed, and exactly 1 possibility child of every probability node is selected. This probability / possibility node scheme allows for more structure than just annotating XML nodes with a probability attribute like in Figure 1.1. In that document we implicitly assumed only one temperature value could exist at the same time, but it was not necessarily defined as such. If multiple uncertain values were defined with the same parent (i.e., the forecast element) it would similarly be undefined which values can exist simultaneously and which cannot. Using the probability and possibility nodes, this ambiguity is removed since it is well-defined that possibility children of the same probability node are mutually exclusive.

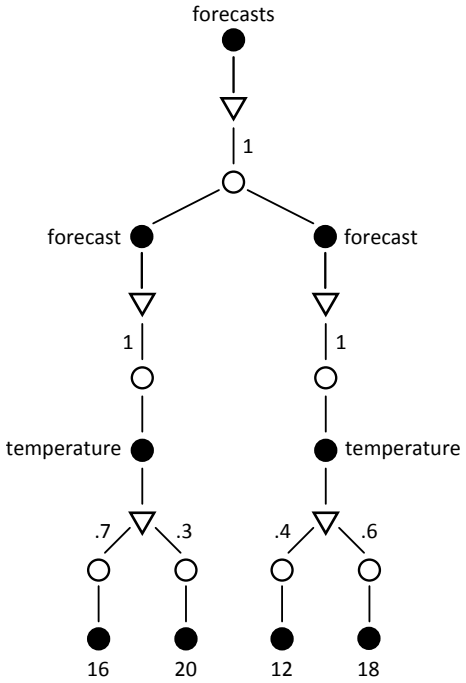


Figure 1.3: Probabilistic XML prob / poss format

The tree structure of the document in Figure 1.1 represented using the described model results in the tree depicted in Figure 1.3, where ▽ represent probability nodes, ○ represent possibility nodes, and ● represent normal XML nodes. A downside of this format is the relatively strict requirement that normal XML nodes can only have probability nodes as children, rather than other XML nodes. This requirement results in many repetitions of a probability node combined with a single possibility node with probability 1 when an XML element is certain. This scenario appears in Figure 1.3 three times; between the root and the two certain forecast elements and between each forecast element and its certain temperature element. In this research we will propose a different probabilistic XML format which does not utilize any distributional nodes but does allow the expression of the same type of relationships as the probability / possibility format, i.e., mutual exclusion and independence among the various uncertain data items. This format is described in Section 3.2.

1.3 Research Objectives

1.3.1 Problem Statement

The most important difference between certain data and uncertain data is the exponential growth in the number of possible worlds represented by uncertain data. In the case of certain data, there is only a single possible world as there are no variations possible for individual data items. Adding new (certain)

data to a certain database does not increase this number. While it will take longer to perform typical database operations on a larger dataset compared to a smaller one, there is no exponential growth for certain data like there is for uncertain data. Figure 1.4 displays the function $f(x) = 2^x$, illustrating the behavior of exponential growth. In terms of an uncertain dataset such a function describes the number of possible worlds, where x is the number of uncertain elements, each having only two possibilities. It is clear from the curve that the number of possible worlds represented by an uncertain dataset reaches numbers that prevent instantiation of each individual one very swiftly.

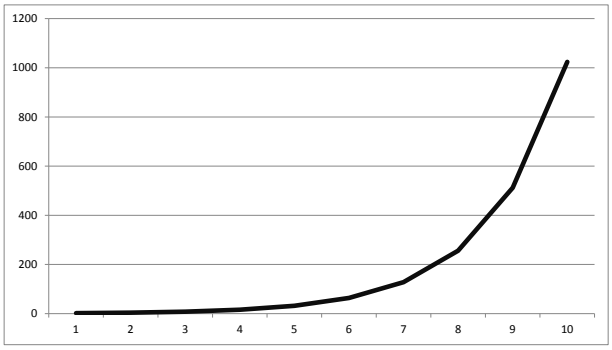


Figure 1.4: Function $f(x) = 2^x$ displaying exponential growth

Handling the exponential growth of the number of possible worlds represented by an uncertain dataset is the main problem we face in this research. Given the exponential growth of the number of possible worlds in the context of uncertain data, the naive approach of instantiating all possible worlds and executing the query on every single one quickly becomes inefficient and even impossible. Instead, we are looking for ways to directly query the uncertain data without instantiating all possible worlds. That is, a query should be evaluated over a single document in the XML database; the document describing the uncertain data using probability annotations. We will typically refer to this "master document" as the uncertain document. This document is the blueprint for all possible worlds; it contains the set of all possible XML elements, values, and attributes that can be present in any possible world along with the random variable string annotations and probabilities associated with them. Mathematically, the uncertain document can be described as a superset of every possible world, making every possible world consequently a subset of the uncertain document.

The result of a query executed directly on the uncertain data should be semantically equivalent to running the query in each of the possible worlds and combining the answers based on the probability of each possible world. Because existing XML databases are not built for handling uncertainty in the data they store, we have to extend the XML DBMS in order to add the probabilistic awareness. The extension must handle the possible world explosion in a way that does not require it to iterate each world. More specifically, we explicitly *avoid* instantiating all possible worlds and will execute a query only on the compact representation of all the possible worlds.

We need to devote special attention to aggregation queries, which map a collection of values to a single value which is possibly not contained within the input sequence of values – and thus not in the uncertain document. An example of an aggregation function is Sum, with obvious semantics. The challenge of computing the result of an aggregation function in the context of uncertain data is that the number of unique values over all possible worlds can be as high as the number of possible worlds. This is different from the values of individual uncertain elements. In that case, the set of all unique possible values are present in the uncertain document since every possible world is a subset of the uncertain document as pointed out earlier. Certain aggregation functions share this property, such as Min and Max. Because those functions select the minimum and maximum value of a set of values V, respectively, the number of unique values over all possible worlds is upper bounded by the length of the input set, $|V|$, rather than by the number of possible worlds. The goal of this research is to compute correct answers to regular queries and aggregation queries over uncertain data, through a plugin for an existing non-probabilistic XML DBMS.

1.3.2 Research Questions

Based on the previous section, we can now formulate our research questions as follows.

- Can we query uncertain data without generating all possible worlds?
- Can the answer to aggregation queries be computed efficiently?
- Are the obtained query results semantically equivalent to the actual results?
- How does the solution scale with respect to different documents and queries?

The rest of this document is organized as follows. Section 2 will present related work on the topic of uncertain databases by other researchers. In Section 3 we will describe the general approach we have taken to answer the posed research questions. It includes the presentation of the XML data representation we use for the uncertain document, and it will cover probabilistic query evaluation. Section 4 is entirely devoted to aggregate queries.

Following that, Section 5 talks about the validity of the implementation in terms of its correctness, while Section 6 tests its performance and scalability. We discuss some of the discovered shortcomings of our implementation in Section 7. Finally, this research is concluded by providing answers to our research questions in Section 8 and discussing open topics for future work.

2 Related Work

Various probabilistic XML models have been proposed in the literature [2][3][4][5][6][7]. Kimelfeld et al. [1] have generalized such known types of probabilistic XML models into different abstract p-document *families* that consist of distributional nodes and regular nodes. The distributional nodes determine the probabilistic distribution of their child nodes in the possible worlds. We use this classification in Section 3.4 to describe the types of probabilistic documents that are supported by our implementation.

There exist not many implementations of the proposed models as an XML database system. One of the few is ProTDB, a probabilistic XML database system resulting from research by Nierman and Jagadish [8]. Applying the p-document family classification, the ProDTB database system is classified as $\text{PrXML}^{\{\text{mux,ind}\}}$; it supports independent and mutually exclusive distributional nodes. An interesting observation made by the authors is that XML does not allow multiple attribute values. Therefore, in order to support uncertain attribute values ProTDB converts all attribute values to regular elements. ProTDB was created by modifying the query parser and query evaluator of the native (non-probabilistic) XML database TIMBER [9]. Li et al. [10] have also created a probabilistic XML database system called PEPX and claim it substantially outperforms ProTDB especially with queries involving descendant axes.

A number of relational database systems supporting uncertainty have also been proposed. An example is Trio, a relational database management system in which data uncertainty and lineage are first-class citizens, introduced by Widom et al. of the University of Stanford [11][12]. The system is built on top of the RDBMS PostgreSQL and implements support for uncertainty and lineage through a translation-based approach. That is, since regular relational tables are used for storage of the uncertain data, queries have to be translated in order to use the probability and lineage metadata. Their own query language, TriQL, allows the user to incorporate specific uncertainty or lineage related expressions in their queries which enables queries such as "select values with a confidence of 98% or higher". Lineage describes where the data came from, for example which original data sources were merged in order to create a resulting value. As such, lineage can be considered a type of metadata which is stored alongside the real data in Trio. Systems similar to Trio are MayBMS, developed by Antova et. al [13], and MystiQ [14], introduced by Boulos et al.

Widom also teamed up with Agrawal [15] to describe a generalized uncertain database which is capable of handling uncertain data even in cases when the exact confidence values or probabilities are not known. Existing uncertain databases require such information on the uncertainty to be present, but Agrawal and Widom present a data-model and semantics that do not break down under such conditions, although no prototype implementing the ideas was created.

Koch and Olteanu [16] discuss a new approach of computing confidence values for the existence of tuples in the result of queries on probabilistic databases involving conditioning the database. This principle entails removing sets of possible worlds which do not satisfy a given condition, resulting in follow-up query operations being applied to a reduced database. The authors additionally introduce the concept of world-set descriptors and give algorithms to store a set of such descriptors, called ws-sets, in a ws-tree which allows for efficient probability computation. The ws-tree is in many ways similar to the tree data structure we utilize for aggregate queries, discussed in Section 4. It also contains two types of nodes; $\oplus$-nodes containing mutually exclusive child nodes, and $\otimes$-nodes containing independent child nodes. These correspond to the RVar and Node nodes that we use in our aggregation tree, respectively.

Aggregation queries have previously been studied by Murthy et al. [17], in the context of the relational probabilistic database Trio. They describe algorithms to obtain the minimum value, expected value, and maximum value for Count, Sum, Min, Max, and Avg aggregates. Their computation of the expected value for the Min and Max aggregates inspired our algorithms for those aggregate functions. In particular, sorting leaf nodes in order to determine the expected value of those aggregate functions is a technique we apply as well. Our implementation additionally calculates the variance, unlike the work of Murthy et al. However, minimum and maximum values for the Avg aggregate function are provided by Murthy et al., but not by our work. Similarly, Chen and Dobra [18] described ways to compute confidence intervals regarding Sum-based aggregate queries over probabilistic relational databases through query rewriting and statistical analysis, relying heavily on the linearity of expectation. They compute the first and second moments of the aggregate function, the expected value and the variance, and use those to compute the confidence intervals. In [19], Abiteboul et al. look at aggregate queries in the context of both discrete

and continuous probabilistic models, and present algorithms to compute the probabilistic moments of the distribution of the aggregation values. Moreover, approximation techniques are explored.

In their work [20], Buneman et al. show the effectiveness of querying a compact representation of an XML document directly from main memory. The compression is based on shared subtrees in the XML document, which is a concept that resembles the relationship between the uncertain document and all possible worlds it represents. The shared subtrees among possible worlds are also "compressed" as a single path in the uncertain document. Additionally, the authors show that succinct compressed data structures of very large XML documents fit in main memory, allowing for faster query evaluation. Storing the uncertain document in main memory might also be an interesting topic for future work on uncertain XML databases.

3 Data Representation and Query Evaluation

This section describes the main parts of our solution in terms of its architecture and important concepts that are utilized in order to obtain correct query results. We begin by introducing the general architecture of our implementation, discuss the data format and show the query result representation that is used. Lastly, we will explain the way we transform an XQuery, which is a bottom-up approach starting at the leaf nodes of the expression tree.

3.1 XML Database Plugin

The solution to the posed research objective will be implemented as a plugin for an existing XML database management system. The main advantage of this approach is that we can leverage integral parts of any XML DBMS such as an XQuery parser, knowing they have been thoroughly tested and proven to be robust. This allows us to focus on our main objective instead.

One of the prerequisites for the XML database is that it should be possible to execute custom queries and access the parsed input query, since our plugin does not provide an XQuery parser of its own. We leverage any suitable built-in functionality as much as possible under the assumption these core methods are implemented very efficiently and refined numerous times over the course of the project's lifetime. The purpose of the plugin is to rewrite a query issued by the user, in such a way that the answer it returns will be semantically equivalent to running the query in every possible world represented by the probabilistic document. Since the XML database does not know the data it stores represents uncertain data, rewriting the input query is necessary to introduce the required probabilistic awareness.

The result of a query over probabilistic data is generally a set of results, each with a certain probability. An example of a query result over uncertain data was given in Section 3.3. Each result in that set of results corresponds to a possible world or set of possible worlds which yield an answer for the given query with a probability higher than 0. The plugin operates on the compiled query, i.e. a tree structure of expressions that make up the query. A similar tree structure will then be created using our own classes representing the various XQuery expressions. Based on transformation rules for each supported expression a new XQuery is then created. This transformation generates a snippet of XQuery for every expression in the query which are combined into a new query and executed. On a higher level of abstraction, Figure 3.1 shows the process described above.

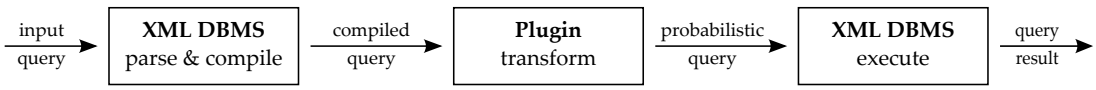


Figure 3.1: Transformation of a query to a probabilistic query

Our implementation creates the new query as an XQuery string, resulting in a second parse and compile step for the XML DBMS before executing the probabilistic query. This is a matter of implementation; an additional parse step can be skipped if the new query is created as a tree of parsed expression objects and directly executed. However, it is important to note that the time it takes to parse an input query, transform it to its probabilistic counterpart and parse it a second time is negligible compared to the execution time of the query itself. We do not support the entire XQuery language, but have rather focused on basic path queries with simple predicates such as `//forecast[temperature > 5]`. The exact list of supported expressions is described in Section 3.6. Even with this small set of tools fairly sophisticated queries can be created – primarily by nesting these expressions – although it should be noted that it does not come close to the expressiveness of XQuery's FLWOR expression.

The XML database that was used for the implementation in this research is BaseX, an open source XML database developed at the University of Konstanz [21]. The database is written in Java and supports plugins which can be used to add user defined functions to BaseX. These functions can then be called from a query issued to BaseX. In that way, plugins can manipulate XQuery values during query execution,

which is functionality we extensively use. Additionally, the plugin can access core BaseX classes such as the query parser and compiler, which can be invoked on any String representing a query resulting in a compiled query object which can subsequently be executed. It thus suits the needs for this research project perfectly. However, the general concepts introduced are not exclusively applicable to BaseX, but rather to any XML DBMS that supports extensions, given that input queries and the query results can be manipulated through user-defined functions.

3.2 Uncertain Data Representation

We use discrete random variables to represent the uncertain values in the dataset, introducing a new random variable for each uncertain value. A random variable is a variable that maps from a sample space Ω to some set of real values, which is referred to as the range of random variable. The set of all possible worlds is the sample space of the random variables that are contained in the uncertain document, which is the compact representation of all possible worlds. We utilize discrete random variables; every random variable has a range containing integer values v, where $0 \leq v < n$, with n being the number of available options for the uncertain data item. The assignment of a value v to a random variable from its range thus corresponds to a subset of Ω; the subset of all events that are mapped to the value v. For a random variable X and every value v of its range R, the probability $P(X = v)$ is defined, denoting the probability that the element annotated with that random variable assignment exists. The probabilities associated with each assignment are stored in a separate probabilities element in the uncertain document. For example, the XML document in Figure 3.2 below describes the uncertain dataset given in Figure 1.1 using random variables.

```
<forecasts>
    <probabilities>
        <probability rv="X=0">0.7</probability>
        <probability rv="X=1">0.3</probability>
        <probability rv="Y=0">0.4</probability>
        <probability rv="Y=0">0.6</probability>
    </probabilities>
    <forecast day="1">
        <temperature rv="X=0">16</temperature>
        <temperature rv="X=1">20</temperature>
    </forecast>
    <forecast day="2">
        <temperature rv="Y=0">10</temperature>
        <temperature rv="Y=1">15</temperature>
    </forecast>
</forecasts>
```

Figure 3.2: Uncertain data annotated with random variables

Random variables X and Y are introduced for the uncertain temperature values in day 1 and day 2. Different random variables are independent with each other; the value assigned to one random variable does not influence the probabilities of the possible values of the other random variable. In this example, the temperature of day 1 does not influence the probabilities of the temperatures on day 2. Because the temperature values are mutually exclusive with each other on a specific day, a single random variable with 2 possible values represents the uncertain value on each day. Both variables have the range $\{0, 1\}$. A possible world is instantiated by assigning all random variables in the uncertain dataset a value from their range and selecting only elements with that particular assignment. That is, with random variables X and Y both having the range $\{0, 1\}$, there are two possible assignments for both variables. The assignment of value 0 to X is denoted by $X \leftarrow 0$. Multiple assignments are enclosed in brackets, i.e., $\{X \leftarrow 0, Y \leftarrow 1\}$ denotes the assignment of 0 and 1 to X and Y respectively, identifying a possible world of Figure 3.2. An alternative way of expressing random variable assignments uses = to connect random variable and value and displays the assignment as a String value, i.e. "X=0" denotes random variable X being assigned the

value 0. Multiple assignments can simply be separated by spaces, as such: "X=0 Y=1". This notation will generally be used since it corresponds one-to-one with the way the implementation processes the random variable strings of database nodes during query execution.

3.2.1 Probability Computation

Random variables X and Y are independent random variables which allows for straightforward probability computation for intersection and union of events belonging to those random variables, listed in Figure 3.3 below.

$$P(X = x \cap Y = y) = P(X = x) \cdot P(Y = y)$$

$$P(X = x \cup Y = y) = 1 - ((1 - P(X = x)) \cdot (1 - P(Y = y)))$$

Figure 3.3: Probability computation of intersection and union of events belonging to assignments of independent random variables

We calculate the union using multiple complements rather than the more common $P(X \cup Y) = P(X) + P(Y) - P(X \cap Y)$ since such a computation becomes inefficient with multiple operands. That is, $P(X \cup Y \cup Z)$ would lead to $P(X) + P(Y) + P(Z) - P(X \cap Y) - P(X \cap Z) - P(Y \cap Z) + P(X \cap Y \cap Z)$. Instead, with our approach, it becomes $1 - ((1 - P(X)) \cdot (1 - P(Y)) \cdot (1 - P(Z)))$. Each additional set S introduces a single additional term $1 - P(S)$ to the computation.

The equations apply to any number of events. Generalized forms are thus formalized as follows, where E is a set of events (each event is the assignment of a random variable, for example $X \leftarrow 0$) and $P(e)$ is the probability of that event e. An event e can also be interpreted as a subset of the sample space Ω, i.e., a subset of all possible worlds represented by the uncertain document.

$$P(\bigcap E) = \prod_{e \in E} P(e)$$

$$P(\bigcup E) = 1 - \prod_{e \in E} 1 - P(e)$$

Figure 3.4: Probability computation for intersection and union of set of independent events E

An illustration of the generalized formula for the union is given in Figure 3.5, where three sets of independent events are displayed. There, the intersection of the complements of the sets X, Y, and Z is equal to the gray area. The union of the sets is then equal to the complement of that area. The union and intersection of events associated with random variable assignments is applied when calculating the probabilities of query results after query evaluation.

3.2.2 Consistency

In the previous section, we introduced a random variable X which takes a value from the set $\{0, 1\}$, where $P(X = 0) = 0.7$ and $P(X = 1) = 0.3$. This random variable can be either 0 or 1, but not both at the same time since the temperature values represented by the random variable are mutually exclusive. This is the case for all random variables in the uncertain dataset; they can only be assigned one of their values at the same time. "At the same time" in this case refers to "in the same set of possible worlds". Combining all possible worlds, every random variable will have been assigned each of its possible values in at least one possible world. During query execution it is continuously checked whether a mutually exclusive pair of values is being processed, in which case query processing of the current path will stop. This will be illustrated by means of an example. Consider an uncertain forecast shown Figure 3.6 for a

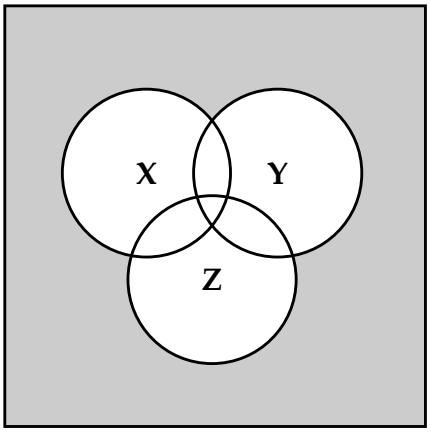


Figure 3.5: Venn diagram of the union of sets X, Y, and Z

single day which describes two possible worlds; one with a temperature of 10 °C and a wind speed of 4 Beaufort, and one with a temperature of 15 °C and a wind speed of 2 Beaufort.

```
<forecast>
    <temperature rv="X=0">10</temperature>
    <temperature rv="X=1">15</temperature>
    <windspeed rv="X=0">4</windspeed>
    <windspeed rv="X=1">2</windspeed>
</forecast>
```

Figure 3.6: Forecast with temperature and windspeed

If mutual exclusion is not taken into account, a query such as `forecast[temperature=10 and windspeed=2]` would yield the forecast element depicted in Figure 3.6 since it contains child elements temperature and windspeed that satisfy the predicate. However, there would not be a single possible world where both those values occur simultaneously as is obvious from the random variable assignments belonging to the hypothetical situation in which that would be the case; "X=0 X=1". Those random variable assignments are inconsistent due to assigning both 0 and 1 to X simultaneously. Trivially, any answer to the probabilistic query is correct if and only if it occurs in a possible world, thus any inconsistent answers cannot be a correct answer. During query processing, we have to verify this "hidden predicate" – that the random variable assignment of the result must be consistent. In the above case, when the temperature element X=0 is combined with the windspeed element X=1 in order to check if they satisfy the query predicate, the consistency check will yield false which will cause the query processing to proceed with the next combination of elements rather than continue with the inconsistent (partial) result. This pruning of the search space is especially beneficial in larger documents with nested random variables, where detecting inconsistencies as soon as they arise reduces the amount of possibilities by a large amount. The implementation of the consistency function will be discussed in Section 3.5.2 section.

3.3 Uncertain Query Results

Because an uncertain dataset represents various different worlds, a query that is applied to it will generally not yield a single answer. The only situation in which a single answer results from a query is when all possible worlds would return the same answer to the given query, which can only occur when the query does not depend on any of the uncertain values, or all the possibilities of the uncertain values yield the same query result. In the general case, however, a query over uncertain data will not get a unified answer but rather multiple answers with different probabilities. Consider again the weather forecasts depicted in Figure 1.1 and a query expressed in natural language as "which days will have a temperature higher than 16 degrees?", i.e., the XQuery `//forecast[temperature > 16]`. It is clear from looking at

the data that neither day 1 nor day 2 is always the answer to this query due to possible temperature values of 16 and 12, respectively. The exact answer can be computed easily when every possible world is instantiated. Table 3.1 below displays the temperature values in every possible world, the query result, and the probability of that result.

	Day 1	**Day 2**	**Query result**	**Probability**
World 1	16 °C	12 °C	empty	0.28
World 2	16 °C	18 °C	day 2	0.42
World 3	20 °C	12 °C	day 1	0.12
World 4	20 °C	18 °C	day 1 and day 2	0.18

Table 3.1: All possible worlds and query results

However, generating all possible worlds like this is quickly becomes unfeasible when processing uncertain data with a more realistic size, considering the number of possible worlds represented by a probabilistic document grows exponentially with respect to the number of random variables in the document. Without possible world expansion, the result of a query over uncertain data cannot be displayed for each individual possible world. Instead, there are two main types of result representation that we use in the created prototype: (1) group by value and (2) group by random variable string. We will discuss each variation below.

3.3.1 Group by Value

This representation scheme yields, for each unique value over all possible worlds, the probability of that value. That is, the computed probability of a unique value v given the set of all possible worlds W is $\sum \{\, p(w) \mid w \in W \wedge v \in w \,\}$, where $p(w)$ computes the probability of a single possible world w.

Applying this representation scheme to the query results displayed in Table 3.1 yields the result displayed in Table 3.2. We have additionally sorted the output by descending probability.

Value	**Probability**
day 2	0.60
day 1	0.30
empty	0.28

Table 3.2: Query results grouped by value

This scheme has both advantages and disadvantages. An advantage is that the number of possible results has a clear upper bound in the number of unique values present in all possible worlds combined, which is equal to the unique values in the uncertain document. This is generally much lower than the number of possible worlds and will therefore be less likely to produce a large amount of results, each with an extremely low probability. Similarly, grouping on unique values eliminates any value duplication which can happen when predicates over uncertain elements are involved. In cases like that, there are many combinations of worlds that satisfy the predicate and each could yield the same context node with a different probability. In this scheme those would all be merged since they yield the same value, and their probabilities would be properly added.

A second advantage is the ranked results (i.e., the values sorted in descending order by their probability) quickly reveal the most probable query result values. When we do not group on unique values it is still possible to sort the results by descending probability, but the same value can occur many times with different probabilities thus the first result (i.e., with the highest probability) does not necessarily correspond to the most likely value – the most likely value could occupy positions 4, 5, and 6 with a combined probability that is higher than the element at position 1 of the sorted sequence.

An important downside of this scheme is the fact it loses information about the query results. In particular, information regarding which values can occur together in the same (set of) possible worlds is entirely lost. To illustrate this, consider Table 3.1 again. We can see day 1 and day 2 occur *together* in the set of query results with a probability of 0.18, and day 1 and day 2 are the *only* answer to the query with probabilities 0.12 and 0.42, respectively. This information is lost in Table 3.2, which only shows the total probability of each value.

3.3.2 Group by Random Variable String

This representation scheme provides a result per unique random variable string. In some cases it is exactly equivalent to the result given in Table 3.1, but in the general case all possible worlds are not generated but rather various subsets of all worlds. When the same value occurs in multiple different world sets, it will thus be duplicated in the query results with possibly different probabilities. This mainly occurs when a predicate addressing uncertain elements is applied to some context node. That context node will then be associated with all possibilities of the predicate that evaluate to `true`. As a result, this representation will usually yield more distinct results than the style discussed earlier which displays probabilities per unique value, but it will retain information regarding query results that occur in the exact same set of possible worlds – since results occurring in the exact same set of possible worlds have the same random variable string, which is used to group results in this representation. Any other overlap of query results is not visible however, since computing all overlapping worlds of the answers boils down to computing all possible worlds which we actively try to avoid.

```
<forecast>
    <temperature model="X" pxml:rv="X=0">16</temperature>
    <windspeed   model="X" pxml:rv="X=0">5</windspeed>

    <temperature model="Y" pxml:rv="X=1">14</temperature>
    <windspeed   model="Y" pxml:rv="X=1">4</windspeed>
</forecast>
```

Figure 3.7: Example document illustrating results grouped by random variable string

As an example of a scenario where this scheme will be useful, consider the document in Figure 3.7. This document integrates data from two different weather prediction models. We assume that only one of the models is right at the same time, thus the values associated with one model are mutually exclusive with the values associated with the other model. When we run a query that asks for all predicted values of a forecast, i.e., `/forecast/*`, it is expected and convenient to group these result per model and thus obtain only 2 possible results (each with temperature and windspeed elements). The representation scheme discussed earlier which groups results by value, however, would provide 4 different results since there are 4 distinct result values. When we apply the current scheme which groups values by their random variable string instead, it yields only two results; "X=0" containing the entries of model X and "X=1" containing the entries of model Y. This corresponds exactly to the possible worlds represented by this document.

When used in conjunction with queries that involve predicates over uncertain elements, however, this representation scheme results in unintuitive answers. A slightly altered version of the document in Figure 3.7 will show this, depicted in Figure 3.8. If we execute the XQuery `/forecast[temperature > 10]` in the depicted document, it is obvious to us that in each of 3 possible worlds the forecast of day 1 is the only result. A reasonable result to this query, then, would be this forecast element with a probability of 1. However, this representation groups the answers by their random variable string, which is different for each of the temperature values that are part of the predicate. Because of that, the answer will consist of 3 different results; each associating the forecast of day 1 with a different random variable string and thus a different probability. In this trivial example we would identify that the 3 results yield the same element and their probabilities sum up to 1, but in more realistic documents we cannot identify this and are left with a large number of results pointing to the same element (the context node of a predicate), each with a

small probability. In such cases, this representation scheme is inferior to the previously discussed scheme where we group the results by unique value and display the total probability of each value. Would that scheme be applied in this case, we would obtain the expected result; the day 1 forecast element with a probability of 1.

```
<forecast day="1">
    <temperature model="X" pxml:rv="X=0">14</temperature>
    <temperature model="Y" pxml:rv="X=1">15</temperature>
    <temperature model="Z" pxml:rv="X=2">16</temperature>
</forecast>
```

Figure 3.8: Example document to show a weakness of the per world set representation

3.3.3 Default Representation Scheme

Because we believe queries involving predicates over uncertain elements are a common occurrence, we favor the representation scheme that groups results per unique value over the representation scheme that groups results by their random variable string. The latter would result in a lot of duplicate values caused by the many possible worlds that satisfy a predicate for any given context node. The approach that groups results per unique value eliminates those duplicates with its grouping operation, and presents each unique value and its total probability instead. Additionally, the results can be easily ranked in order to identify the most likely result. However, the per world set representation scheme can be enabled through a configuration option, all of which are detailed in Section A.1. Note that these representation schemes are only relevant for non-aggregate queries. When an aggregate query is issued, the result will always consist of the summary values that are applicable to the specific aggregation function. The implementation and representation of aggregation queries is described in Section 4.

3.4 Supported P-Document Families

Kimelfeld et al. introduced different families of probabilistic documents in their work [1]. The families are classified based on the distributional nodes that are used in the document. Important to note is that since our implementation does not support any distributional nodes at all, any probabilistic document containing them should first be translated to the format described in the previous section which uses random variable assignment annotations. Before discussing which families of p-documents are supported, we first describe the various families introduced in [1] below, in order of increasing complexity and expressiveness.

det Probabilistic documents belonging to this family contain distributional nodes which are *deterministic*; *all* child nodes are selected when an XML document is generated from the probabilistic document. Thus, the child nodes implicitly have a probability of 1.

mux The *mutual exclusion* distributional node will yield at most 1 of its child nodes when an instance is created from the probabilistic document, since the children are mutually exclusive with each other. We say *at most* instead of *exactly* since the sum of probabilities may be less than 1, in which case it is not guarenteed that a child node is selected.

ind Child nodes of an *independent* distributional node are all independently included in a possible instance with a certain probability. Including child n has no influence on the inclusion of child $n + 1$, and so on.

exp *Explicit* distributional nodes define probabilities per distinct subset of child nodes and chooses exactly one of those subsets to be included in the generated XML document. Not all subsets have to occur in the definition, and one of the subsets can be the empty set $\varnothing$.

cie A *cie* distributional node selects children based on the truth value of a *conjunction of independent events*. Given independent boolean variables $e_1, \ldots e_n$ with associated probabilities, each child

node is associated with a conjunction of the form $a_1 \wedge \ldots \wedge a_n$ where each a_i corresponds to a boolean event e_i or its negation $\neg e_i$. The child node is selected when the conjunction is true. Each child node can have different number of terms in its conjunction, and the used boolean events can overlap between children *and* between different *cie* nodes in the document. This is different from the other families, where no such interdependence exists; the previous distributional nodes are independent from other distributional nodes, but *cie* nodes are not since they can share the boolean events of other *cie* nodes.

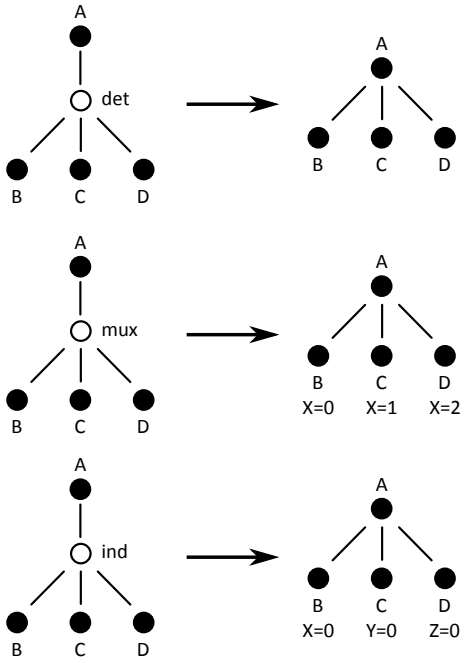


Figure 3.9: Transformation of P-Document families to random variable assignment format

A single p-document can belong to multiple families. That is, a document which both mutual exclusion distributional nodes as well as explicit distributional nodes belongs to both the *mux* and *exp* families. The notation used by Kimelfeld et al. for such a document is PrXML$^{\{\text{mux,exp}\}}$. Our implementation supports documents belonging to the class PrXML$^{\{\text{mux,det,ind}\}}$. However, since the documents can only contain random variable annotations and no distributional nodes these documents have to be transformed to the proper format first. Figure 3.9 shows examples of the required transformations for each of the supported families.

These transformations are straightforward. In all cases, the distributional node is removed and the children are attached to first regular node in the ancestor chain. Depending on the type of distributional node, we create random variables with assignments and probabilities corresponding to the semantics of the distributional node. In the case of a deterministic distributional node we do not need to introduce any random variables since the child nodes are always selected. For a mutual exclusion node, we introduce a single random variable with as many assignments as the *mux* node has children. The different assignments of a random variable are also mutually exclusive, so this corresponds exactly to the *mux* node semantics. The children of an *ind* node are independent, thus we introduce a new random variable for each of the children. Important to note in that case is the generated random variables are boolean random variables; they have exactly 2 assignments. One assignment corresponds to the child being selected with a certain probability p, and the second assignment corresponds to the child node *not* being selected, with a probability $1 - p$. This latter assignment is not visible in Figure 3.9 since it is an empty value but is present in the document's metadata section which describes the random variable assignments and their associated probabilities. That is, it would contain entries for both "X=0" and "X=1", whereas "X=1" would represent an empty element. The same holds for variables Y and Z in the referenced figure.

3.5 Random Variable String Manipulation Primitives

Section 3.2 detailed how random variables are used to represent possibilities in an uncertain document. It was mentioned that the random variable string is continuously checked for consistency. The implementation of this functionality in the plugin will be explained in this section. The combine and consistent functions perform very basic operations on a random variable string, which is essential for the transformation process. These primitive functions are utilized in the transformations of other XQuery expressions, which are discussed in Section 3.6. A transformed query will in turn consist of those transformed expressions. This bottom-up transformation process, starting with random variable string primitive functions that are incorporated in transformed expressions, which are joined together to form the transformed XQuery is the logical result of evaluating the expression tree that represents the compiled original query in a depth-first manner, where each node first transforms all its children (i.e., its sub-expressions). An example of a transformed probabilistic query is presented in Section 3.8.

3.5.1 Combine

The combine function is used to build up the random variable string. It simply combines two random variable strings to create their concatenation without duplicates. The implementation is fairly straightforward. A Set instance is utilized to make sure there will be no duplicate random variable assignments in the resulting combined string. This proofed to be quicker than checking for existence in the resulting string using the contains method of Java's `String` class which yields true if the string contains another `String`. The `LinkedHashSet` is chosen as the implementation of `Set` in order to retain the insertion order. This is important when the random variable string is split up and inserted in a tree. If the hierarchy is different due to a different unpredictable order the resulting tree might not reflect the structure of the uncertain XML document. The implementation is shown in Figure 3.10. Some example inputs and outputs of the function are listed below.

- `combine("X Y", "X Y")` → `"X Y"`
- `combine("X Y", "X Z")` → `"X Y Z"`
- `combine("X X", "Y Y")` → `"X Y"`
- `combine("X Y", "A B")` → `"X Y A B"`

```
1  String combine(String s1, String s2) {
2      if(s1.isEmpty()) return s2;
3      if(s2.isEmpty()) return s1;
4
5      Set<String> set = new LinkedHashSet<>();
6      for(String s : s1.split(" ")) set.add(s);
7      for(String s : s2.split(" ")) set.add(s);
8
9      Iterator<String> it = set.iterator();
10     String combined = it.next();
11     while(it.hasNext()) combined += " " + it.next();
12
13     return combined;
14 }
```

Figure 3.10: Combine function

3.5.2 Consistent

The consistent function is used to check whether or not the random variable string parameter contains any inconsistencies. An inconsistency occurs when the same random variable is present in the string but has two different assigned values. Thus, a string such as "X=0 Y=0 X=1" is inconsistent due to

having two different assignments for the random variable X. Since a single possible world contains a single value of each random variable, this random variable string does not correspond to any possible world and is therefore not valid. The implementation splits the string on the space character and compares every element to all following elements of the resulting list. The comparison searches for equivalent random variable identifiers and different values, in which case it immediately returns false. If no such combination can be found, the loop will end normally which means the input string was consistent, hence true is returned.

```
1  public boolean consistent(String rvs) {
2      if(rvs.isEmpty()) return true;
3      String[] parts = rvs.split(" ");
4      if(parts.length == 1) return true;
5
6      for(int p1 = 0; p1 < parts.length; p1++) {
7          String[] vv1 = parts[p1].split("=");
8          for(int p2 = p1 + 1; p2 < parts.length; p2++) {
9              String[] vv2 = parts[p2].split("=");
10             if(vv1[0].equals(vv2[0]) && !vv1[1].equals(vv2[1]))
11                 return false;
12         }
13     }
14
15     return true;
16 }
```

Figure 3.11: Consistent function

3.6 Representation of Intermediate Results

The probabilistic query – that is, the transformed input query – needs to keep track of all possibilities for each expression used in the input query. For example, the path expression `//forecast[temperature > 5]` results in two possibilities when evaluated in the document seen in Figure 3.6 which have to be stored in a variable somehow along with their probabilities. Every expression in a probabilistic query potentially has multiple possibilities with different probabilities. We call such expressions probabilistic expressions. XQuery provides a map datatype which can be used to store information as key => value pairs. We use such a map to represent a single possibility of a probabilistic expression. The map contains two keys; 'rv' which points to the random variable string, and 'v' which points to the value of the expression. A probabilistic expression is represented by a sequence of these maps. Thus, the result of the path expression `//forecast[temperature > 5]` applied to the document Figure 3.6 would be represented in the probabilistic query like in Figure 3.12, where the windspeed element was left out for brevity.

```
(
  { 'rv' : "X=0",
     'v' : <forecast><temperature>10</temperature></forecast> },
  { 'rv' : "X=1",
     'v' : <forecast><temperature>15</temperature></forecast> }
)
```

Figure 3.12: Probabilistic expression; a sequence of maps

This uniform representation of all probabilistic expressions as sequences of maps enables us to define how any probabilistic expression has to be handled when used as a sub expression in other expressions. As mentioned before, our implementation supports only simple path expressions which can contain predicates with And, Or, and Comparison expressions, as well as atomic values like strings and numbers. We will

now show how these expressions are created in more detail. We will discuss each of the supported XQuery expressions listed in Table 3.3.

Expression	Example
Atomic values	"Saturday", 42, xs:date("2014-02-28")
Path	/forecast/windspeed
And / Or	(temperature = 5 and windspeed = 3) or rainfall < 50
Sequence	(5, "Monday", 10)
Comparison	$e_1 > e_2$, $e_1 \geq e_2$, $e_1 = e_2$, $e_1 \leq e_2$, $e_1 < e_2$
Arithmetic	$e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$, e_1 div e_2

Table 3.3: Supported XQuery expressions

3.6.1 Empty Value

XQuery uses the empty sequence as their empty, or NULL, value. Since some expressions can yield an empty value, such as a path expression which does not match any elements, we need to be able to represent an empty value as a probabilistic expression. That is done in the following way:

```
{ 'rv' : '', 'v' : () }
```

This expression makes sure our approach which regularly uses nested for loops does not break down when it encounters an empty value. This empty value – implicitly a sequence of length 1 –, will be iterated once. A regular XQuery empty sequence `()` has length 0 and would not be iterated at all, thus loops nested within it are never reached.

3.6.2 Atomic Value

Simple atomic values like strings and numbers are translated to a probabilistic expression in a very straightforward way. These expressions are static in the sense that they do not change depending on the possible world they are evaluated in, thus have only a single possible value and their random variable string is empty – signifying a probability of 1. For example, a number like 42 is transformed to a probabilistic expression like this:

```
{ 'rv' : '', 'v' : 42 }
```

The string and number types are transformed in this natural way. Other types, such as xs:date, are represented using their constructor instead in order to keep their type intact. For instance, the date type denoting 14 April 2014 transformed to a probabilistic is shown below.

```
{ 'rv' : '', 'v' : xs:date("2014-04-14") }
```

A full list of the XQuery atomic values and a comprehensive definition of all other types can be found in [22].

3.6.3 Path Expression

A path expression is the standard way to navigate an XML document making it the single most important expression to support, given any non-trivial query will contain at least one path expression. A path consists of a number of steps, which in turns consist of an axis (e.g., child, descendant, parent), a node test (e.g., "forecast" to select all `<forecast>` elements on the specified axis) and a list of predicates (e.g., "temperature > 10") to filter the selected elements. We need access to each element matched by every step of the path in order to apply the combine and consistent functions introduced earlier. This is what XQuery's `for` loop does, which allows us to insert the mentioned functions and check for consistency at each individual matching element. If the random variable string is inconsistent, we do not search any

further (i.e., we do not apply any following steps but immediately consider the next matching element). In order to transform the path expression, we string together the for loops of the steps and finally return a map element containing both an rv element to store the random variable string, as well as a v element which holds the value of the expression (i.e., the XML node(s) resulting from the path expression). The transformation of a path expression `/forecast[temperature > 10]` yields the XQuery snippet displayed in Figure 3.13.

```
1  for $v1 in /forecast
2    let $r1 := $v1/@rv
3    where consistent($r1)
4
5    let $v2 := ... (: transform temperature > 10 :)
6
7    for $v3 in $v2
8      let $r2 := combine($r1, $v3('rv'))
9      where consistent($r2)
10     where $v3('v')
11     return { 'rv' : $r2, 'v' : $v1 }
```

Figure 3.13: Transformation of a Path expression

The predicate generation was left out for space considerations. The predicate is transformed to a probabilistic expression, meaning it will be a sequence with possibly more than 1 value, hence we apply the for loop to iterate every value it contains and return the context element (`$v1` in Figure 3.13) to which the predicate was applied for each value which yields true, checked by `where $v3('v')`. The predicate in this case is a comparison (`temperature > 10`) which will be discussed next.

3.6.4 Binary Expression

And, Or, Arithmetic, and Comparison expressions are mentioned separately but are roughly equivalent on a higher level; each of the expressions applies an operator to two sub expressions to produce a single output expression. Any binary expression is of the form $e_1 \diamond e_2$, where e_1 and e_2 are expressions and $\diamond$ is some binary operator. The transformation of an original expression e_1 `and` e_2, which applies the binary operator And to the input expressions yields the snippet of XQuery displayed in Figure 3.14. Binary expressions can act as sub expressions for other binary expressions, which means this simple structure allows an arbitrarily deep nesting of such expressions, providing the ability to create complex expressions and predicates.

Technically, an expression such as e_1 or e_2 or e_3 is modeled as a single Or expression with 3 operands in BaseX. Our Binary Expression follows suit, thus the terminology *binary* is not entirely accurate since our binary expression supports 2 *or more* operands.

```
1  let $e1 := ... (: transform e1 :)
2  let $e2 := ... (: transform e2 :)
3
4  for $x in $e1
5    for $y in $e2
6      let $r1 := combine($x('rv'), $y('rv'))
7      where consistent($rv)
8      let $v1 := $x('v') and $y('v')
9      return { 'rv' : $r1, 'v' : $v1 }
```

Figure 3.14: Transformation of a Binary expression

The code displayed in Figure 3.14 is like this for the general case. However, we have defined special cases for the And and Or expressions. For an Or expression we do not build the Cartesian product like we do

for other binary expressions, since we do not have to combine their random variable strings or check the consistency of each combination. We do not have to do this because of the definition of an Or; it yields true when any of its operands is true. Thus, if Figure 3.14 would display the XQuery snippet of an Or expression, it would immediately return a sequence of `$e1` and `$e2`, mapping each value to its boolean value after their declarations using our `bool()` function, discussed in Section 3.7.

In the case of an And expression, it is not always needed to evaluate all its operands in order to determine the value of the expression. That is, we know that the value of an And expression such as $e_1 \wedge \cdots \wedge e_n$ is `false` whenever any of its operands is `false`. To that end, we add code that handles that case in two different positions in the generated XQuery snippet; between each operand expression and within the nested `for` loops. For an And expression with 2 operands e_1 and e_2, the generated snippet in Figure 3.15 reveals the additional code. Notice that the optimization steps belonging to e_2 are omitted for clarity, but are equivalent to the ones for e_1.

```
1  let $e1 := ... (: transform e1 :)
2
3  return if(every $x in $e1 satisfies not($x('v')))
4  then $e1 else
5
6  let $e2 := ... (: transform e2 :)
7  (: ... :)
8
9  for $v1 in $e1
10   let $r1 := $v1('rv')
11
12   return if(not($v1('v')))
13   then { 'rv' : $r1, 'v' : false() } else
14
15   for $v2 in $e2
16     (: ... :)
```

Figure 3.15: And expression with added optimizations

The first `if` expression checks if all the values contained in the probabilistic expression e_1 evaluate to `false` – i.e., `not($x('v')`. If that is the case, we can return e_1 as the result to the entire And expression; the value of the expression will be `false` in all worlds described by e_1. The same holds for any of the other operands of the And expression.

The second optimization is performed inside the `for` loops that would build the Cartesian product of the operands. If any single value evaluates to `false` we return the current random variable string alongside the value `false`; i.e. we do not continue to loop through any following operands. The reasoning is similar to the earlier case, with the difference that we now look at individual values rather than an entire sequence.

When none of the added `if` expressions evaluated to `true`, it is clear that the Cartesian product of the operands is created, which will cause performance problems when the sequences that serve as operands contain many elements.

3.6.5 Sequence Expression

A sequence is an ordered list of zero or more items. Essentially every datatype in XQuery is a sequence. That is, even single values are sequences of length 1. For example, the single value 42 responds to an indexing operation just like any other sequence; 42[1] simply yields 42, and any index not equal to 1 yields the empty sequence. Further, a sequence of length 1 containing an item is considered equivalent to the item on its own [23]. Note the indices of XQuery sequences are 1-based, rather than the more common 0-based indices that other programming languages use for their container types. The sequence expressions referred to in this section are sequences of at least 2 items. This is akin to how BaseX implements the

class they use to represent such a "true" sequence. The sequence type is supported mainly since the BaseX compiler creates them in certain situations.

For example, BaseX will simplify an `Or` expression such as `temperature = 5 or temperature = 10` to `temperature = (5, 10)` when possible. To that end, we have to support the sequence expression. And obviously, we now allow the usage of the sequence expression directly in the input query as well. The sequence cannot only contain atomic values such as numbers or strings, but rather any other type. For instance, we could specify a predicate that yields only those forecast elements which have a temperature equivalent to the temperature on any of the following 2 days using a sequence expression: `//forecast[temperature = (fs::forecast[1]/temperature, fs:forecast[2]/temperature)`. The notation `fs::` is shorthand for `following-sibling::`, and is not part of official XQuery but was added to the plugin. Section A.1.2 describes that shorthand and other shorthands that were added.

```
1 let $e1 := ... (: transform e1 :)
2 let $e2 := ... (: transform e2 :)
3
4 return ($e1, $e2)
```

Figure 3.16: Transformation of a Sequence expression

The implementation of the sequence expression is very straightforward; we transform all items and yield a sequence containing the results of the transformations. We do not have to create a Cartesian product and check the consistency of the various elements when combined with each other, since the sequence is more similar to an Or expression rather than an And. Notice that the transformation described in Figure 3.16 is performed on a sequence with exactly 2 elements, but the approach is applied to sequences of arbitrary size.

3.6.6 Function Expression

Apart from aggregation functions which are discussed in Section 4, other functions are supported as well when their output is not influenced by uncertain elements. For example, the `position()` function will not work properly in the context of uncertain elements when applied as-is since the position of elements surrounded by uncertain elements is uncertain as well. If one or more uncertain preceding siblings of an element do not exist, for example, the position of the node itself is moved up a few indices compared to the case where they do exist. Therefore, a function like `position()` which is dependent on the context generally will not work properly. However, a function like `data()` that extracts atomic values from an element will have no issues with uncertain elements since it just transforms an uncertain element to its data value. There are more functions similar to `data()`, like `string()` to convert the argument to a string and `number()` to convert the argument to a number. In such cases, we can apply the function without any other modifications in a straightforward manner, depicted in Figure 3.17 for the `data()` function which has a single argument e_1.

```
1 let $e1 := ... (: transform e1 :)
2
3 for $v1 in $e1
4   let $r1 := $v1('rv')
5   let $v2 := fn:data($v1('v'))
6   where p:consistent($r1)
7   return { 'rv' : $r1, 'v' : $v2 }
```

Figure 3.17: Transformation of the fn:data() expression

In general, functions that are independent of their context and apply a simple transformation to their arguments are supported by our implementation. However, we did not extensively test all built-in XQuery functions due to their sheer number [24].

3.7 Intermediate Result Manipulation Functions

We have defined a number of functions that operate on the intermediate results, which are all sequences of maps. Their application can be seen in the probabilistic query that is presented in Section 3.8. Their use cases and implementations will be described below.

3.7.1 Empty

The `empty()` function replaces any empty sequence with a sequence that contains a single element; an empty map value. The reasoning for this is provided in Section 3.6.1. We apply the function to all probabilistic expressions, since each yields a sequence of maps which can possibly be empty.

3.7.2 Boolean

For the Or expression we have created a utility function that takes a sequence of maps as input, and outputs a sequence containing the maps with all values replaced by the boolean representation of the map's original value. We added this since our Or expression does not explicitly apply the `or` operation; the result of an Or expression is simply a sequence of operand values. This does not make a difference when using the Or expression as a predicate, since the values will then be implicitly converted to a boolean when applied in a Where clause. However, when the Or expression is used as the root expression of a query we want to return their boolean values thus we do the transformation. If the value is any of `false`, `0`, "", or `()` it will be `false`, otherwise `true`.

3.7.3 Group

We apply grouping using a custom `group()` function. It is used to group unique elements together. We choose for a custom method instead of XQuery's built-in `group by` operation due to the latter grouping different database nodes together when they have the same atomic value. For example, the two distinct database nodes `<x>1</x>` and `<y>1</y>` would be grouped together when using the native group by. It turns out the native group by function compares contents of database nodes, thus if their content is the same they are grouped together. In the custom group function we first check if we are dealing with a database node and if so, we compare their unique identifiers. Otherwise, we use the `hash()` function as well. We do not want to group distinct database nodes with the same values together, since that would produce incorrect results for the Count aggregation function. We count every grouped value as 1, which is not correct when n distinct nodes are all grouped together; their count is obviously n.

3.7.4 XML

A sequence of maps cannot be output as-is, the XML DBMS does not know how to serialize its contents. We have therefore defined a `xml()` function that will transform the sequence of maps into valid XML. Additionally, the function will calculate the probability belonging to each value and include it as a `prob` attribute. The random variable strings belonging to each value are used to compute the probability and are then omitted from the result – although it is possible to include them via a configuration option discussed in Section A.1.

The XML format we use is straightforward; each map is transformed to an element containing the map's value as well as the computed probability as an attribute. The resulting elements are wrapped in a parent element and returned. The format is displayed below in Figure 3.18.

3.8 Probabilistic Query

When the transformation rules described earlier are applied to an input query which does not include any probabilistic awareness, a new query is generated which does take the probabilistic nature of the

```
<results>
    <result prob="...">...</result>
    <result prob="...">...</result>
    <result prob="...">...</result>
</results>
```

Figure 3.18: Output XML format

uncertain data into account. The plugin rewrites the input query quite substantially to introduce this probabilistic awareness. An example of such a probabilistic query is displayed in Figure 3.19 below. The query was created by transforming the input query `count(/forecasts/forecast[temperature > fs::forecast[1]/temperature])` which selects only those forecast elements which have a temperature that is higher than the temperature of the following day's temperature. The syntax contains a custom axis shorthand, `fs::`, which is short for `following-sibling::`. A number of custom syntax shortcuts are listed in Syntax Shorthands. The query in Figure 3.19 applies the transformations described earlier recursively, yielding a nested structure which traverses the document using the path specified and applies the predicates given by the user. While traversing, the random variable string is built up, stored in an `$rX` variable, and is continuously checked for consistency. Values belonging to matching, consistent elements are stored in a map alongside their random variable string, referenced as `$vX` values in XQuery.

```
declare namespace pxml = 'db.ewi.utwente.nl';
import module namespace p = 'org.basex.modules.pxml.PXML';

let $doc := db:open('pxml', 'forecasts.xml')

let $v1 := (
  let $v2 := (
    for $v3 in $doc/child::forecasts
      let $r1 := string($v3/@pxml:rv)
      for $v4 in $v3/child::forecast
        let $r2 := p:combine($r1, string($v4/@pxml:rv))
        where p:consistent($r2)
        let $v5 := (
          let $v6 := (
            for $v7 in $v4/child::temperature
              let $r3 := p:combine($r2, string($v7/@pxml:rv))
              where p:consistent($r3)
              return { 'rv' : $r3, 'v' : $v7 }
          )
          let $v7 := (
            for $v8 at $v8_p in $v4/following-sibling::forecast
              let $r3 := p:combine($r2, string($v8/@pxml:rv))
              where p:consistent($r3)
              where $v8_p = 1
              for $v9 in $v8/child::temperature
                let $r4 := p:combine($r3, string($v9/@pxml:rv))
                where p:consistent($r4)
                return { 'rv' : $r4, 'v' : $v9 }
          )
          for $v8 in p:empty($v6)
            let $r3 := p:combine($r2, $v8('rv'))
            where p:consistent($r3)
            for $v9 in p:empty($v7)
              let $r4 := p:combine($r3, $v9('rv'))
              where p:consistent($r4)
              let $v10 := $v8('v') > $v9('v')
              return { 'rv' : $r4, 'v' : $v10 }
        )
        for $v6 in p:empty($v5)
          let $r3 := p:combine($r2, $v6('rv'))
          where p:consistent($r3)
          where $v6('v')
          return { 'rv' : $r3, 'v' : $v4 }
  )
  for $v3 in p:empty($v2)
    let $r1 := $v3('rv')
    let $v4 := $v3('v')
    return { 'rv' : $r1, 'v' : $v3 }
)
return p:xml(p:group($v1))
```

Figure 3.19: Probabilistic query resulting from transforming the input query `/forecasts/forecast[temperature > fs::forecast[1]/temperature]`

4 Aggregate Queries

4.1 Motivation and General Approach

In contrast to regular queries on probabilistic data where the result consists of unmodified elements from the input document, aggregate queries by definition apply an additional function to the set of resulting elements which performs the aggregation – the aggregation function. XQuery supports [24] the five most common aggregation functions; Count, Sum, Min, Max, and Avg, all with well-known semantics. These functions map a sequence of values to a single value. This sounds trivial, but when the input sequence consists of uncertain data it is usually impossible to define a single output value. In terms of the possible worlds concept; each world yields an aggregate value which is potentially different from values yielded in all other possible worlds. While this is similar to non-aggregate queries, there is one important difference. The possible results of a non-aggregate query are limited to elements in the input document. For example, given an input document like the one in Figure 4.1 and a query to select all values (i.e., `//value`) any possible world can only contain some combination of the 10 possible `<value>` nodes.

```
<values>
    <value rv="A=0">5</value>
    <value rv="A=1">39</value>
    <value rv="B=0">15</value>
    <value rv="B=1">1</value>
    <value rv="C=0">7</value>
    <value rv="C=1">83</value>
    <value rv="D=0">56</value>
    <value rv="D=1">123</value>
    <value rv="E=0">30</value>
    <value rv="E=1">6</value>
</values>
```

Figure 4.1: Values representing 2^5 possible worlds

More precisely, any of the 32 possible worlds contains exactly 5 `<value>` nodes; one for each of the 5 random variables $A \ldots E$. Moreover, we can accurately predict the proportion of possible worlds a value is present in by just looking at its probability. That is, a `<value>` node with probability p will be part of a subset W' of all possible worlds W whose sum of probabilities is exactly p:

$$p = \sum_{w \in W'} prob(w)$$

Thus, an answer to the query `//value` in this case would be all `<value>` nodes and their probability, since that corresponds very closely to the result of applying the query to all possible worlds. Per the equation above, summing the probabilities of possible worlds containing a specific `<value>` node always yields the probability of the `<value>` node we return as a response to the query. The limited possibilities for the resulting values can be illustrated by looking at the possible distinct values of the query result. Consider all 32 possible worlds represented by the document in Figure 4.1. The number of distinct values over all worlds is just 10; equivalent to the number of distinct values in the input document. In contrast, the aggregate query `sum(//value)` over the same document will not yield just 10 distinct values, but 32; one for each possible world. Yielding a semantically equivalent result for a Sum query would require calculating all possible values for the Sum function.

There exists no efficient algorithm to yield all possible values without explicitly performing the Sum operation in all possible worlds which is no option given the scalability problems described earlier, due to exponential growth of the number of possible worlds. In case all values could be efficiently generated from the input values, there is the issue of representing the resulting values. As described above, an aggregation function applied to an uncertain input sequence representing n possible worlds can yield up

to n distinct values. In cases like that, where sense has to be made of a large set of numbers, it can be useful to instead yield a summary of the numbers. That is, one can describe a large sequence of numbers using properties which can be extracted from the sequence such as:

- Minimum value
- Expected value
- Maximum value
- Variance
- Standard deviation

These values describe an arbitrarily large set of numbers in a concise way fairly accurately. It is such a summary that we aim to deliver as an answer to an aggregate query on probabilistic data. A hard requirement is to be able to yield such a summary without having to iterate all possible worlds.

It is important to note that the upper bound of n distinct values for an aggregate query over n possible worlds is only true for Sum/Count/Avg aggregate functions. Min and Max behave differently, in that they map a sequence of values to some element that is *part of* the input sequence. As a result, the upper bound of a Min/Max aggregate function is defined in terms of the number of distinct values in the input document rather than in the number of possible worlds. This property makes that the implementation of Min/Max is very different than Count/Sum/Avg. The rest of this section will describe the implementations in detail.

4.2 Tree Data Structure

We introduce a data structure to store the intermediate query results (i.e., the results to the query before applying the aggregation function). It is a simple tree data structure which has 2 node types; RVars representing random variables, and Nodes which represent an XML element containing a value. An RVar has 1 or more Node children and has exactly 1 Node parent. A Node has 0 or more RVar children and 0 or 1 RVar parent. When a Node has no RVar parent, it is the root of the tree. Additionally, a Node stores the probability of itself given its ancestors exist, and it contains the values associated with it. The root of the tree will have probability 1, since it does not belong to any RVar. The tree hierarchically stores intermediate query results based on their random variable string. The two types of nodes are required since the semantics are different between them. As defined earlier, the different valuations of one random variable are mutually exclusive. This translates to only 1 of the Node children of an RVar being true at the same time. On the other hand, all RVar children of any Node do exist at the same time. Therefore, it was necessary to keep track of the random variables associated with the values to know which values are mutually exclusive and which can exist simultaneously.

Consider an example of a simple probabilistic document with nested random variables in Figure 4.2 and a query which calculates the sum of all the `<leaf>` values: `sum(//leaf)`. In order to construct the tree, the non-aggregate part of the query is executed first to yield all `<leaf>` nodes and their random variable strings. The random variable string of a `<leaf>` element is its random variable attribute combined with the random variable attribute of all its ancestors. The tree is then built from these <random variable string, value> tuples. The random variable string (e.g., "X=0 Y=0") is processed from left to right to determine a value's destination node. The resulting tree is displayed in Figure 4.3, where ○ represent RVars and ● represent Nodes.

Such a tree is constructed for each aggregation function, with which the various summary values are calculated. Each aggregation function uses a slightly different implementation of the tree, which was inevitable since different operations are applied to a sequence of values at each node depending on the aggregate function (addition for Count/Sum and obvious operations for Min, Max, and Avg). In addition, the summary values are not the same for all functions. In the Min/Max case we can go beyond just summary values and are able to yield the top-k result values as well. That is, we return the actual result values to the aggregation function, ordered by descending probability and limited to an arbitrary number k. For Count/Sum/Avg yielding the top-k result values was not possible due to the characteristics of those aggregate functions, specifically the fact the number of distinct result values for those functions can

```
<root>
    <branch rv="X=0">
        <leaf>2</leaf>
        <leaf>12</leaf>
        <leaf rv="Y=0">18</leaf>
        <leaf rv="Y=1">22</leaf>
        <leaf rv="Z=0">16</leaf>
        <leaf rv="Z=1">25</leaf>
    </branch>
    <branch rv="X=1">
        <leaf>5</leaf>
        <leaf>17</leaf>
        <leaf rv="A=0">23</leaf>
        <leaf rv="B=0">1</leaf>
        <leaf rv="B=0">7</leaf>
        <leaf rv="B=0">20</leaf>
    </branch>
</root>
```

Figure 4.2: Document with nested random variables

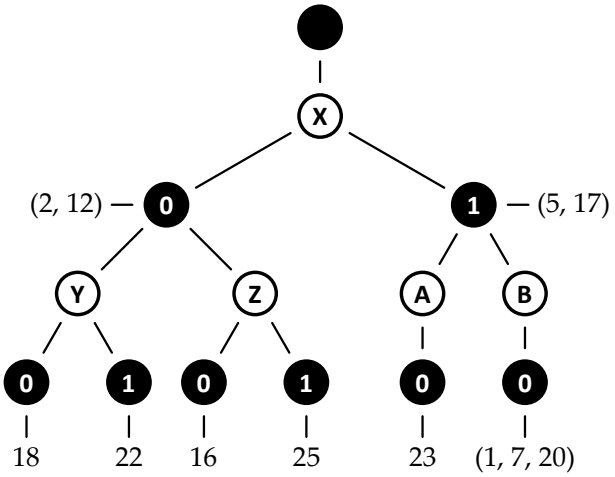


Figure 4.3: Tree structure of values before aggregation

grow as large as the number of possible worlds and there is no way to efficiently produce each possible value, or just the k most likely ones.

4.2.1 Tree Confidence

A property of the tree that is shared among all implementations is the computation of the confidence of the tree. The confidence corresponds to the probability of the union of the random variable strings associated with the leaf nodes in the tree. The computation of this probability is defined recursively in our tree structure. In order to efficiently compute it, we use the property $A \cup B = (A^c \cap B^c)^c$, i.e. the union of two independent sets is equal to the complement of the intersection of their complements. We have discussed this property in Section 3.2.1.

We use this principle to compute the confidence value of any {Node, RVar} tree by defining a recursive *confidence* function on either type of node. The confidence of an RVar node is defined as the sum of confidences of its Node children. Taking the sum is allowed since the child nodes are mutually exclusive. For a Node we apply the union property described above. That is, the confidence of a Node is computed

by taking the complement of the multiplication of the confidence complements of all its RVar children, multiplied with the probability of the Node itself – the probability associated with the single random variable assignment that the Node represents, for example "X=0". If a Node has no RVar children, the confidence is equal to this probability. Formally, the confidence is thus calculated as below. In this equation, n denotes a Node, r denotes an RVar, $prob(n)$ denotes the probability of a Node n, $n.R$ denotes the set of RVars belonging to Node n, and $r.N$ denotes the set of Nodes belonging to RVar r. The formulas are based on the work of Koch and Olteanu [16], who have defined a similar tree structure – defined as a *ws-tree* – in their research on conditioning probabilistic databases. When defining our tree structure we were not aware of their work yet.

$$conf(n) = \begin{cases} prob(n) & if\ n.R = \varnothing \\ prob(n) \cdot (1 - \prod_{r \in n.R} 1 - conf(r)) & otherwise \end{cases}$$

$$conf(r) = \sum_{n \in r.N} conf(n)$$

Figure 4.4: The recursive computation of the confidence in a {Node, RVar} tree

The confidence is used in the summary computation of the Min and Max aggregation functions, as well in non-aggregate queries to calculate the probability of each unique value. In such a non-aggregate case, a unique value can have multiple associated random variable strings which each yield the specific value. In order to calculate the overall probability of the value – i.e. the confidence of the set of possible worlds that contain that value – we build a tree structure filled with all the random variable assignments and compute its confidence. The resulting probability is then equal to the sum of probabilities of possible worlds the value exists in. The remainder of this section will describe the procedures we use to yield the summary values for each of the aggregation functions.

4.3 Count and Sum

The aggregate functions Count and Sum are similar, in that both use the addition operation to aggregate their input values. The only relevant difference is Count will first convert any input value to 1, unless it is the empty sequence `()`, which will count as 0. Thus, Count is essentially equivalent to Sum with all input values being 1. As a result, we can use the same approach to calculate the summary for either aggregate function. The algorithms to deliver the various values of the aggregate summary will be described below.

4.3.1 Extreme Values

The extreme values (i.e., the minimum value and the maximum value of the aggregate function over all possible worlds) can be computed from the tree using a straight-forward algorithm which is recursively defined and ends up traversing the tree bottom-up, as the extreme value of any Node or RVar depends on the extreme value of its children. The algorithm for the minimum value will be discussed next, note that the algorithm for the maximum value is analogous when replacing all occurrences of min with max. The minimum value of a Node consists of the sum of the Node's own values and the sum of the minimum values for each of its RVar children. The minimum value of an RVar is obtained by selecting the lowest minimum value from all its Node children. A "missing" Node child, which is a possible valuation of the random variable not present in the tree (it did not match the query), counts as 0. This is required since both Count and Sum applied to the empty sequence yield 0 as well.

An example of a missing Node in the tree of Figure 4.3 is the `<value>` element with random variable string "X=1 A=1". An RVar without missing children is considered complete. In the tree above, A and B are incomplete whereas X, Y, and Z are complete. More formally, let $f_{op}(n)$ and $f_{op}(r)$ be the functions yielding the minimum or maximum value for a Node and RVar, respectively, thus $op \in \{min, max\}$. Let

$n.V$ be the set of all values associated with a Node n, $n.R$ the set of RVar children of that Node, $r.N$ the set of Node children of an RVar r and op be the function that applies the specified operation to a sequence of numbers. The extreme values are then computed as follows.

$$f_{op}(n) = \sum_{v \in n.V} v + \sum_{r \in n.R} f_{op}(r)$$

$$f_{op}(r) = \begin{cases} op(\{\ f_{op}(n) \mid n \in r.N\ \}) & if\ RVar\ is\ complete \\ op(\{0\} \cup \{\ f_{op}(n) \mid n \in r.N\ \} & otherwise \end{cases}$$

Figure 4.5: Minimum and maximum value computation for Count and Sum

4.3.2 Expected Value

In order to calculate the expected value, the tree structure is not required. The hierarchical organization of the random variables and values is irrelevant; we only need the probabilities and values of the results of the query before the aggregation is applied. First, consider the definition of the expected value in the context of possible worlds. Let W be the set of all possible worlds represented by the document to which an aggregation query q is applied, with the aggregation function being either Count or Sum. The value resulting from applying q to a possible world w is denoted by $q(w)$. The probability of a world is denoted by $prob(w)$. Then, the expected value of query q is defined as follows:

$$E(q) = \sum_{w \in W} prob(w) \cdot q(w)$$

Figure 4.6: The expected value of a Count/Sum query q

While our tree does not contain the query result for all possible worlds, it does contain the probability and value of all individual elements which together make up all possible worlds. For each value, the probability of the value is exactly equal to the sum of probabilities of all possible worlds the value is present in. In each of the worlds w that contain the value, it contributes $prob(w) \cdot value$ to the expected value of that world, like all values of w do. That is, since $q(w)$ is the sum of all values existing in world w, the expected value of a single world w, denoted by $E(w)$ can be written as in Figure 4.7 below.

$$\begin{aligned} E(q, w) &= prob(w) \cdot q(w) \\ &= prob(w) \cdot (v_1 + v_2 + ... + v_n) \\ &= prob(w) \cdot v_1 + prob(w) \cdot v_2 + ... + prob(w) \cdot v_n \end{aligned}$$

Figure 4.7: The expected value of query q for a single possible world w

On the last line of the equation, it is shown that each value v present in world w is weighted by the probability of w and summed to get the expected value of w. The probability which is stored in our tree is equal to the sum of probabilities of possible worlds the value appears in. We showed each value contributes $prob(w) \cdot value$ to the expected value of a possible world w. Since the probability in our tree represents the sum of n possible world probabilities, the multiplication of probability and value equals the total contribution of the value to the expected values in all worlds it is present in. Since the tree stores all values and probabilities which together make up all possible worlds, we can obtain the expected value $E(q)$ of all worlds for query q by taking the sum of all these products. As mentioned earlier, this sum of products can be computed directly from the results of the non-aggregate part of the query. It can

also be computed when the tree has been built, in which case the expected value can be defined on Node and RVar as follows, where $n.V$ are values of Node n, $n.R$ are RVars of Node n, and $r.N$ are Nodes of RVar r:

$$E(n) = \sum_{v \in n.V} v \; + \sum_{r \in n.R} E(r)$$

$$E(r) = \sum_{n \in r.N} prob(n) \cdot E(n)$$

Figure 4.8: The expected value of Node and RVar

4.3.3 Variance and Standard Deviation

The expected value combined with the extreme values does not characterize a sequence of numbers properly, as it does not accurately describe the *distribution* of the values in its domain. We cannot infer from the given three properties alone if the values are grouped at either the minimum or maximum value but rarely at the expected value itself, spread evenly along the entire range between the minimum and maximum value, or if most values are spread around the expected value with only few values at the minimum and maximum. All three such series will have similar minimum, maximum and expected values. Figure 4.9 illustrates this using histograms of arbitrary sequences in the order described. Note that for the illustration we assume a simple case of unweighted items and thus the expected value is the same as the average of all items.

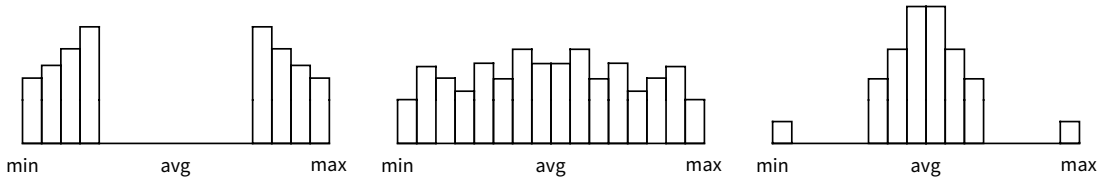


Figure 4.9: Histograms of 3 sequences with the same minimum, average, and maximum values

It is clear from the histograms these sequences are distributed in very different ways, yet yield the exact same minimum, average, and maximum values. In order to differentiate between these, it was necessary to introduce additional properties which describe the spread among the values; the variance and its square root; the standard deviation. The variance would help distinguish between the cases shown in Figure 4.9, as the variance would be the highest for the left case, average for the middle case, and the lowest for the right case. Given the query `sum(//value)`, we are interested in calculating the variance using the tree data structure which should be equivalent to the variance of the set of query results of all possible worlds. In the definition of the variance $var(q)$ of query result q below, the result of applying query q to world w is denoted by $q(w)$, with W being the set of all possible worlds. The probability of world w is denoted by $prob(w)$.

In the tree structure, the variance and expected value are defined recursively for Node and RVar. In the definitions, $r.N$ is the set of Nodes belonging to RVar r, $n.R$ is the set of RVars belonging to Node n, and $n.V$ is the set of values associated with Node n.

4.3.4 Shannon Expansion

For Count/Sum/Avg aggregates, we perform Shannon expansion on the random variable strings associated with each unique value before we build the aggregation tree of all values. This was necessary for the correct computation of summary values in cases where a query result consisted of independent random variable

$$
\begin{aligned}
var(q) &= \sum_{w \in W} prob(w) \cdot (q(w) - E(q))^2 \\
&= E[(q - E(q))^2] \\
&= E[q^2 - 2qE(q) + E(q)^2] \\
&= E(q^2) - 2E(q)E(q) + E(q^2) \\
&= E(q^2) - E(q)^2
\end{aligned}
$$

Figure 4.10: Variance of a query q over possible worlds W

$$
\begin{aligned}
var(n) &= \sum_{r \in n.R} var(r) \\
var(r) &= E(r^2) - E(r)^2 \\
E(r^2) &= \sum_{n \in r.N} prob(n) \cdot (E(n)^2 + var(n)) \\
E(n) &= \sum_{v \in n.V} v + \sum_{r \in n.R} E(r) \\
E(r) &= \sum_{n \in r.N} prob(n) \cdot E(n)
\end{aligned}
$$

Figure 4.11: Recursive computation of variance in the tree for Node n and RVar r

assignments pointing to the same result element. We will provide a small example that illustrates this scenario, and follow it up with an explanation of Shannon expansion and how it solves the problem.

Consider the XML document in Figure 4.12 and a simple aggregation query that counts the number of forecast elements which contain a temperature child with a value greater than 10 – `count(//forecast[temperature > 10])`. The forecast element contains two independent temperature child nodes, both of which satisfy the predicate if they exist.

```
<forecast>
    <temperature pxml:rv="X=0">11</temperature>
    <temperature pxml:rv="Y=0">12</temperature>
</forecast>
```

Figure 4.12: Shannon expansion requirement example

The child nodes are independent since they are annotated with different random variables, X and Y. Both child nodes have a probability < 1 thus there exist random variable assignments "X=1" and "Y=1", which correspond to empty values. That is, the number of possible temperature children for the forecast element ranges from 0 to 2. Without these alternatives, the temperature nodes would not be uncertain values since they would always exist.

Executing the given simple aggregation query – `count(//forecast[temperature > 10])` – will yield the following intermediate query result, where we have grouped on unique values which is the default for our implementation. Notice it provides the forecast element, with both associated random variable strings

"X=0" and "Y=0".

```
{ 'rv' : ("X=0", "Y=0"), 'v' : <forecast> ... </forecast> }
```

Figure 4.13: Intermediate query result

When we compute the aggregation values and omit Shannon expansion, the aggregation tree is created by inserting the random variable strings of each unique value into the tree, associated with a value. In case of a Count aggregate, this value is always 1 (we do not insert the empty value if it is part of the intermediate query result, which would have value 0). The given example query yields a very simplistic tree which represents only a single value, displayed in Figure 4.14.

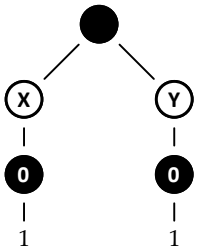


Figure 4.14: Aggregation tree when no Shannon expansion is performed

In this specific case, the wrong values are computed because the aggregation tree contains the independent leaf nodes "X=0" and "Y=0" which satisfy the predicate of the given query (i.e., `temperature > 10`). These leaf nodes both have the value 1 associated with them, the value obtained from transforming the `<forecast>` element to the number 1 since we are dealing with the Count aggregation function. Then, using algorithms discussed in Section 4.3, the summary values are calculated and since the leaf nodes are independent they will both be counted towards a maximum of 2, even though with 1 forecast element in total the maximum cannot be higher than 1. Similarly, errors are introduced in the computation of the other summary values.

In general, this problem arises whenever an expression used in a predicate yields multiple nodes with independent random variable strings for a single context node. A special case that usually results in multiple independent nodes is any predicate that uses a wildcard path, i.e., `/path[* > 5]`, where any child element of `/path` that satisfies the comparison will be an intermediate result value pointing to the same `path` element.

Shannon Expansion Theorem We can avoid the erroneous aggregation values by using Shannon expansion, which decomposes the independent random variable strings into mutually exclusive strings. Shannon expansion states that for any Boolean function f consisting of variables $X_1 \dots X_n$, the following holds, where X_1' denotes the complement of X_1:

$$f(X_1, X_2, \dots, X_n) = X_1 \cdot f(1, X_2, \dots, X_n) + X_1' \cdot f(0, X_2, \dots, X_n)$$

Figure 4.15: Shannon expansion theorem

That is, any boolean expression can be partitioned into disjoint sub-expressions by extracting one of its variables and assigning it a truth and false value, each combined with the original function where the variable is set to 1 and 0, respectively. This creates two mutually exclusive sub-expressions of which the union is equal to the original expression. We will show an example using the random variable strings from earlier in this section; "X=0" and "Y=0". The corresponding boolean expression will be denoted as $X_0 \cup Y_0$. Applying Shannon expansion with respect to X_0 yields $(X_0 \cap (1 \cup Y_0)) \cup (\neg X_0 \cap (0 \cup Y_0))$.

This expression can be simplified further. We will look at both operands of the second $\cup$ operator, thus the following two sub-expressions:

1. $X_0 \cap (1 \cup Y_0)$
2. $\neg X_0 \cap (0 \cup Y_0)$

Looking at (1), we identify that $1 \cup Y_0 \Rightarrow 1$ and $X_0 \cap 1 \Rightarrow X_0$. That is, part (1) of the expression is simplified to X_0. The second part can also be simplified in a similar way, using $0 \cup Y_0 \Rightarrow Y_0$, to become $\neg X_0 \cap Y_0$. Thus, the resulting expression of the Shannon expansion is a union of disjoint sub-expressions; $X_0 \cup (\neg X_0 \cap Y_0)$. Translating that back to random variable strings, we obtain "X=0" and "X=1 Y=0", given that the range of X is {0, 1} and thus $\neg X_0 = X_1$. This leads to the following aggregation tree, where all leaf nodes are mutually exclusive:

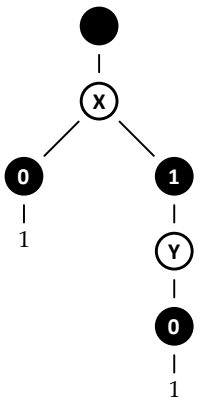


Figure 4.16: Aggregation tree when Shannon expansion is performed

Notice that if the size of X's range is more than 2 this becomes more complex. Consider the case when X has a range of {0, 1, 2} instead. $\neg X_0$ then corresponds to $X_1 \cup X_2$. Then, $X_0 \cup (\neg X_0 \cap Y_0)$ expands to $X_0 \cup ((X_1 \cup X_2) \cap Y_0)$, or $X_0 \cup (X_1 \cap Y_0) \cup (X_2 \cap Y_0)$, i.e. {"X=0","X=1 Y=0","X=2 Y=0"}.

These strings are mutually exclusive, thereby making it impossible for the aggregation algorithm to select multiple leaf nodes simultaneously for the same result value which yielded the incorrect values. We have implemented Shannon expansion and apply it to the random variable strings of each unique value that results from the non-aggregate part of a query and insert the set of resulting strings into our aggregation tree. Each leaf will be associated with the value that we grouped on, i.e., each leaf will have the same value. Since the leaves are mutually exclusive, only one of them can be selected at any time. The process of Shannon expansion will also fix any corruption in the tree, something that is discussed in Section 5.3.1. Because we perform Shannon expansion on each unique value but not on the entire aggregation tree, it is theoretically possible that a corrupted subtree is created within the aggregation tree, due to some overlap in the random variable strings of different values. If that happens all summary values except for the expected value can possibly yield the wrong value. However, in our experience corrupted nodes are almost always created with random variable strings belonging to the same value, which will be prevented by performing Shannon expansion.

4.4 Min and Max

In case of the aggregate functions Min and Max all summary values can be computed during a single iteration over a sorted set of leaf nodes, sorted on their total value. This approach is based on the work of Murthy et al. [17] who designed the algorithm to compute EMIN and EMAX in the relational probabilistic database system Trio [11]. The total value of a leaf node is defined as the aggregated value along the path from the root to the leaf. For example, for a query applying a Min function in the tree of Figure 4.3 the leaf node "X=0 Y=0" will have a total value of 2. Its own value is 18, but his ancestor "X=0" contains a smaller value and thus the minimum value of all possible worlds containing node "X=0 Y=0" will be at most 2. It is at most 2 since there could be other nodes present which bring a smaller

value, which ultimately will then be the minimum value of said world. When operating on a sorted set of all leaf nodes – non-descending for Min, non-ascending for Max – we are actually iterating through all distinct values resulting from applying Min or Max in all possible worlds.

This is not hard to proof. We have established earlier our tree contains all values which make up the domain of any possible world in the context of non-aggregation queries. That is, when performing a query such as `//value` on the input document given in Figure 4.1, the set of distinct values over all possible worlds will be equal to the distinct values of the input document. The aggregation functions Min and Max map a sequence of values to one of the values of that sequence (i.e., the minimum and maximum value, respectively). That is, the resulting value of Min or Max in a possible world will always be an element of the set of all values of that possible world. This is different from Count or Sum, where the result of the addition is not necessarily an element of the set of all input values. The tree contains all values of the input sequence (that is, values matching a query), thus when iterating all values in the tree we are guaranteed to have iterated all possible results of Min or Max in all possible worlds.

The values are iterated in sorted order to calculate the expected value. This will be explained in the context of the Min function, thus the values are sorted in non-descending order. The first and thus smallest leaf value will be guaranteed to be the lowest value in all worlds it is present in, since it is the lowest overall value. The sum of probabilities of all possible worlds where this leaf node exists is equal to the total probability of the leaf node, which represents the probability of the leaf node and all its ancestors being "selected" at the same time. This probability is simply the product of the probabilities of all ancestors and the Node itself. If A is the set of ancestors of Node n, the total probability $prob_t$ of n is calculated as below, where $prob(n)$ denotes the probability of a Node n.

$$prob_t(n) = prob(n) \cdot \prod_{a \in A} prob(a)$$

For the first Node, the total probability equals the sum of probabilities of possible worlds where Node represents the minimum value of the possible world. However, for all subsequent Nodes this is not necessarily the case. The total value of any next Node in the non-descending set only is equal to the minimum value if none of the Nodes before it exist in the same possible world. If any of the previous Nodes would exist in the same world as the current Node, the current Node would not be the minimum due to the sort order. The probability of a Node's total value being the minimum value, which we will call the effective probability of Node n, denoted by $prob_e$, is thus defined as the probability of the intersection of the complement of previously processed Nodes and the Node itself. Let P denote the set of all Nodes having been processed before Node n. The effective probability of Node n is then given by:

$$prob_e(n) = prob(P^C \cap n)$$

In case of the first node this will be equal to $prob(n)$ since P is empty, thus P^C is the entire universe which intersected with n yields n. In practice, the effective probability is calculated by subtracting the confidence before processing n from the confidence after processing n. While iterating the sorted set of nodes, we build a new tree structure which calculates the confidence of the entire tree again upon insertion of a new node. Only the confidence of affected nodes is re-calculated, i.e., ancestors of the inserted node. This makes the continuous confidence calculation fairly efficient. Section 4.2.1 details the algorithm for the calculation of the confidence in our {Node, RVar} tree.

4.4.1 Extreme Values

Since we process every possible value of all worlds in the iteration, determining the extreme values is trivial. The minimum and maximum values are initialized to ∞ and $-\infty$, respectively. When iterating all values, it is checked if the current value is lower than the minimum or higher than the maximum, in which case the minimum or maximum value is set to the current value. Otherwise, the minimum and maximum are left unchanged. When the iteration has ended, the minimum and maximum values are equal to the minimum and maximum value of the query result over all possible worlds.

4.4.2 Expected Value

In order to compute the expected value during the iteration loop, we use an incremental calculation method for the weighted expected value, taken from the work of Finch [25]. He refers to the expected value as the mean value, and defines the incremental mean value as follows:

$$\mu_n = \mu_{n-1} + \frac{w_n}{W_n}(x_n - \mu_{n-1})$$

Figure 4.17: Incremental weighted mean

In the definition, W_n represents the sum of all weights $w_1, \ldots, w_n$, with w_n being the weight of the current value, x_n. Using that definition, we are able to iteratively build up the mean value using the incremental mean value of the previous iteration and the sum of weights of all processed values. The key difference between this approach and the standard formula for the weighted mean is that is it not necessary to know the sum of all weights in advance. Additionally, the incremental weighted mean will be used simultaneously for the calculation of the incremental weighted variance, explained in the next section.

4.4.3 Variance and Standard Deviation

Similar to the mean value, the variance is computed using an incremental algorithm which was defined by Finch in his work [25]. The incremental weighted variance is defined below, where W_n, w_n, and x_n have the same semantics as in the case of the mean value.

$$\sigma_n^2 = \frac{S_n}{W_n}$$

$$S_n = S_{n-1} + w_n(x_n - \mu_{n-1})(x_n - \mu_n)$$

Figure 4.18: Incremental weighted variance

The incremental variance uses the current and previous incremental mean values (μ_n and μ_{n-1}), which are being computed during the same iteration. The standard deviation follows from the variance by taking its square root.

4.4.4 Algorithm

The pseudo-code of the algorithm which computes all the properties of the Min and Max aggregation functions, all of which were described above, is listed in Figure 4.19. The actual algorithm is implemented in Java and contains a few more operations which were left out for readability. Most variables in the algorithm have obvious semantics with the exception of `node.value`, which denotes the result of the aggregation of all values from the root node to the leaf node, i.e. the minimum or maximum value of the leaf node and all its ancestors. We have described this value previously as the *total value* of a leaf node.

4.5 Avg

The Avg aggregate is hard. Neither approach discussed before applies to Avg. Unlike in the Count/Sum case, where it was possible to determine the amount a value contributed to the expected value of the aggregate over all values, this is impossible for Avg. This is due to the definition of the Avg operation; a value will have more influence on the outcome of the function when there are few other values present

```
min          = infinity
max          = -infinity
expected     = 0
variance     = 0

weightSum    = 0

prevExpected = 0     // Previous incremental expected value
prevConf     = 0     // Previous confidence of tree

root         = new Node // Computes confidence of processed nodes

for node in sortedLeafNodes do
    root.insert(node)

    curConf   =  root.confidence()
    weight    =  curConf - prevConf
    weightSum += weight

    expected += (weight / weightSum) * (node.value - expected)
    variance += weight * (node.value - prevExpected) * (node.value - expected)

    if(node.value < min) min = node.value
    if(node.value > max) max = node.value

    // Stop loop when we covered all possible worlds
    if(curConf == 1.0) break

    prevConf     = curConf
    prevExpected = expected
end

variance = variance / weightSum
```

Figure 4.19: Min / Max summary algorithm

(or none at all, in which case the value will be equal to the average value), compared to when there are many other values present. It is not possible to say with certainty the amount a value will contribute to the overall average thus we cannot yield the exact value for the mean value of Avg from the input values. Similarly, the minimum and maximum values for Avg are not as trivially computed as in the Count/Sum case. Because again the value of the average function depends not only on the values of the sequence but also the length of the sequence, an algorithm which selects the minimum average value at each RVar and simply combines these at each Node might not end up with the actual minimum value for Avg. In certain situations choosing a sequence of values with a higher average might actually lead to a smaller overall Avg value when combined with results from other parts of the tree, something that cannot be foreseen.

For example, when a subtree B holds values which are very high compared to a subtree A, subtree A should prefer longer sequences of his values over shorter sequences, even if the average of the longer sequence is higher. For instance, imagine the algorithm searching the minimum values for Avg yields (1000,1000,1000) for subtree B as minimum values. For subtree A the possibilities were (1,1) and (50,50,50). Clearly, the algorithm applied in the Count/Sum case would pick (1,1) for subtree A here as it has the lowest average value. However, the overall average would then become $(3 \cdot 1000 + 2)/5 = 600.4$. When (50,50,50) would have been selected instead, the overall average would have been much lower: $(3 \cdot (1000 + 50))/6 = 525$. This is not possible since subtree A has no knowledge of subtree B thus this algorithm can yield incorrect

results and is therefore unusable to compute the minimum and maximum values for the Avg aggregate function. To summarize an Avg aggregate query, we can only yield an estimate of the expected value efficiently. The expected value is estimated by dividing the expected value of the sum by the expected value of the count;

$$E(Avg) = \frac{E(Sum)}{E(Count)}$$

This simple approximation was used based on the work of Murthy et al. [17], where the authors argue the estimate has a guaranteed error of 0 in some cases, and in other cases the error is "quite small in practice". This latter statement was based on a number of experiments performed across a wide variety of distributions with reasonable data sizes. This approximation can be computed efficiently by looping once through all values in the tree, as explained in the section describing Count and Sum.

5 Correctness Validation

In this section we will discuss the validation of the implementation in terms of its correctness. The performance and scalability of the implementation, which are important aspects of validation as well, are discussed in Section 6.

Correctness is of the utmost importance to any information system, and our implementation is no exception. This is reflected in one of the posed Research Questions; are the results of our implementation semantically equivalent to the actual result? Before discussing the correctness of our implementation, it is necessary to define *correct* in the context of our implementation. Semantic equivalence goes beyond just correctness, in that it entails both correctness and having the same meaning as the entity referred to – in this case, the query result in all possible worlds which will be referenced as the *actual query result* from this point. In Section 3.3, we showed two types of query result representation that are used in the implementation, (1) results grouped by value and (2) results grouped by random variable string. Neither of those representation schemes is always semantically equivalent to the actual result. A third type of result representation is the aggregation summary representation we yield for aggregation queries, which by definition is not semantically equivalent to the actual result since is describes the distribution of the result values rather than the result values themselves; information such as the top-k results that can be extracted from the actual result are not present there, for example, because actual result values are (usually) not part of the summary values.

5.1 Correctness and Semantic Equivalence

We consider a query result to be correct when it can be obtained from – or, is contained in – the actual query result. However, the query results yielded by our implementation are generally also incomplete; the actual result contains more information than our implementation provides such as certain relationships between result values. A short example will be given below, where X and Y are arbitrary query results such as distinct XML elements. The actual query result in Table 5.1 shows 4 different result values; each corresponding to a possible world with their probabilities summing to 1.

Result	**Probability**
X, Y	0.45
X	0.30
Y	0.15
empty	0.10

Table 5.1: Actual query results

The implementation query results in Table 5.2, however, show only 3 results; one for each distinct value. That result is correct in the sense that these results follow from the actual result in a straightforward way. The value X is indeed part of the query result with a confidence value of 0.75 (0.45 + 0.30 from Table 5.1). Similarly, the probability of Y is in fact 0.60 (0.45 + 0.15), and the result is empty with a probability of 0.10. Thus while correct, this answer is not semantically equivalent due to the loss of information regarding the simultaneous occurrence of X and Y and the probability associated with that event. The probability of that event – 0.45 – is contained in the confidences associated with both X and Y in Table 5.2 and as such is "lost", hence the implementation results are incomplete when compared to the actual query results.

Notice that in this case with only two distinct non-empty elements we can infer that X and Y must occur together in some set of possible worlds because the sum of their probabilities is more than 1, but with more than 2 distinct elements we cannot infer which elements occur together anymore. We will note that the lack of semantic equivalence does not imply incorrectness. On the contrary, incorrect results *do* imply a lack of semantic equivalence since the latter is essentially a stronger version of correctness. In the remainder of this section we will discuss the validity of our implementation in terms of correctness. This

is split up in two parts; selecting the correct *elements* from the document, and computing the correct *probabilities* of those elements, which together make up the query results.

Result	Probability
X	0.75
Y	0.60
empty	0.10

Table 5.2: Implementation query results

5.2 Correct Elements

An important way our implementation prevents incorrect result elements is through the constant application of the `consistent` function which was discussed in Section 3.2.2 and Section 3.5.2. It ensures no inconsistent path in the XML document is traversed and consequently none of the returned results can be inconsistent. Ensuring consistency of the results on its own does not guarantee semantic equivalence with the actual results or correctness, however. Consistency is a prerequisite for correctness, which follows trivially from the fact that an inconsistent result does not appear in any possible world and thus not in the actual result, thereby making inconsistent results also incorrect.

Another method to make sure we select the correct elements from the document is to apply only the bare minimum of required transformations to the original query to add probabilistic awareness while preserving all parts of the original query. The expressions from the input XQuery reappear in the probabilistic query practically unchanged; they have only been broken up and appear in different parts of the probabilistic query but their order and context in which they were originally applied are no different. This is illustrated in Figure 5.1. It displays the general transformation applied to each input in a simplified form. Operations related to handling the random variable string were left out to emphasize the resemblance between the original query and the resulting probabilistic query.

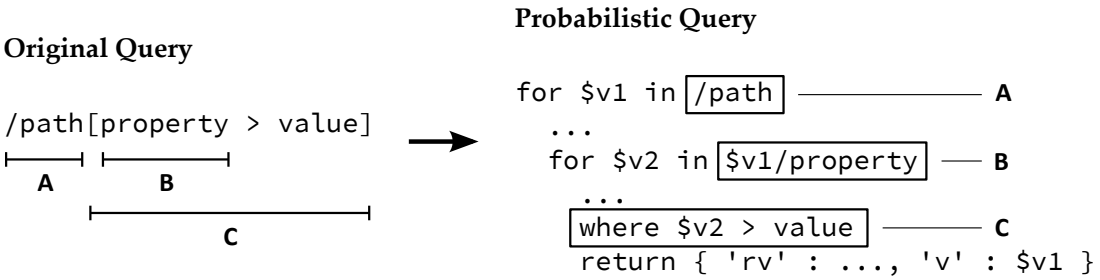


Figure 5.1: Similarities between an original query and a probabilistic query

The essential difference between an original query and the created probabilistic query is that the probabilistic query takes the uncertain nature of the data into account. That is, it has to account for the fact it is querying multiple possible worlds simultaneously and many combinations of elements of the uncertain document cannot exist in any possible world; they are inconsistent. To identify these elements, the probabilistic query has to access each element in a collection and check its consistency when it is combined with the set of elements that make up the current query context. To that end, it will apply `for` loops to every collection and access each individual element. This process has been detailed for every supported expression in Section 3.6. Any operations defined in the input query will be applied directly to the original element; nothing has been changed during query transformation in that regard.

It is apparent that the results of both queries are very similar. The main difference is the probabilistic query wraps the original results inside a map to associate a random variable string, and it duplicates each `path` element for each of its `property` sub-elements that match the given predicate – `property > value`, but those duplicates will be removed when grouping on unique values when creating the result representation. Because inconsistent results are not included, and in other cases matched elements are selected in the

same way as in the original query, the results of the probabilistic query are also correct in the sense that they must appear in some subset of the actual result. Note that this simple reasoning does not hold for expressions that are dependent on their context, such as a `position()` function where the value of the function depends on the existence of its siblings. However, all supported expressions listed in Section 3.6 do not have such dependencies and either traverse the document or apply predicates to values, optionally combining those with And or Or expressions. In those cases, showing that the document is traversed in the same way as in the original query while filtering out inconsistent elements is sufficient to establish the correctness of our implementation, noting again that some information is lost compared to the actual query result which makes the implementation result not semantically equivalent.

5.3 Correct Probabilities

The intermediate result of a probabilistic query is a sequence of maps consisting of a random variable string and a value. Depending on the used representation scheme, the two of which are discussed in Section 3.3, we have to compute the probability of (1) a single random variable string belonging to 1 or more values, or (2) a set of random variable strings belonging to a single value. Those correspond to the representation schemes that either group the results by random variable string or by value, respectively.

That is, when we group by random variable string each unique random variable string will have 1 or more values that belong to the set of worlds described by that random variable string. In that case, the correct probability computation is straightforward; it is the product of the probabilities of the random variable assignments in the random variable string. For example, the probability of the random variable string "X=0 Y=1 Z=0" is equal to $P(X = 0) \cdot P(Y = 1) \cdot P(Z = 0)$. We can multiply the individual probabilities since different random variables are independent thus for each pair of random variables X and Y it holds that $P(X|Y) = P(X)$. Following that, the formal definition of intersection $P(X \cap Y) = P(X|Y) \cdot P(Y)$ can be rewritten as $P(X) \cdot P(Y)$ by substituting $P(X|Y)$ with $P(X)$. This is similar for $P(Y \cap X)$ due to commutativity of intersection.

When results are grouped by value, we associate each unique value with 1 or more random variable strings. Thus, to obtain the probability of that value we have to compute the probability of a set of random variable assignments. Each random variable assignment describes a set of possible worlds in which the value exists. We are effectively looking for the probability of the union of these sets of worlds, since the value exists in every set separately and thus the set of all worlds that contains the specific value is the union of all individual sets.

The probability of this union corresponds exactly to the confidence of the aggregation tree that was introduced in Section 4.2. We construct a similar tree in order to compute the probability belonging to the set of random variable assignments. However, since we are not interested in computing any summary values we omit all values but instead insert only the random variable strings into the tree. When the set of random variable strings belonging to a single value is {"X=0 Y=0", "X=0 Z=1", "X=1"}, the constructed tree in Figure 5.2 would be the result.

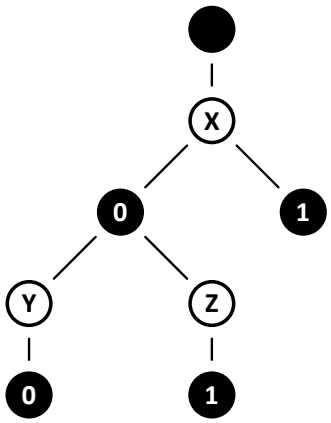


Figure 5.2: Simple tree created to compute the probability of a set of random variable assignments

In order to compute the probability of this tree, we make use of two simple properties.

1. For any set of mutually exclusive events E, it holds that $P(\bigcup E) = \sum_{e \in E} P(e)$.
2. For any set of independent events E, it holds that $P(\bigcup E) = 1 - \prod_{e \in E} P(e^{\mathsf{c}})$, where e^{c} denotes the complement of event e.

In the created tree we apply these properties recursively to compute the probability of RVars (○) and Nodes (●), respectively. That is, since the children of RVars are mutually exclusive events we sum their probabilities to obtain the probability of the RVar, whereas the children of a Node are independent events thus we apply the second property to obtain its probability. The described recursive computation of the probability was given earlier, in Section 4.2.1. By creating a tree for each unique value, we can thus accurately compute the probability belonging to every value.

5.3.1 Corrupt Trees

Without any post-processing on the tree after inserting the random variable strings there are some situations where the probabilities yielded by the application of the described algorithms for independent and mutually exclusive events will be incorrect. This happens when the created tree is *corrupt*, which is a state we detect and fix. We are not referring to tree data structures that contain a cycle, which are sometimes called corrupt as well. Rather, a tree used to compute the probabilities is corrupt when different branches of a Node that are assumed to be independent turn out not to be.

An example of such a scenario involves the set of random variable strings {"X=0 Y=0", "Y=0"}. Denoting "X=0" and "Y=0" by X_0 and Y_0 respectively, the probability would then be computed as $1 - ((1 - P(X_0 \cap Y_0) \cdot (1 - P(Y_0)))$. However, $X_0 \cap Y_0$ and Y_0 are not independent events and thus neither are their complements. This means we cannot simply multiply the probabilities of their complements to obtain the probability of the intersection of their complements. The independence does not hold since the set of worlds described by Y_0 contains all worlds described by $X_0 \cap Y_0$. Therefore, $P(X_0 \cap Y_0 \mid Y_0)$ is equal to $P(X_0)$, rather than $P(X_0 \cap Y_0)$ like we assume for independent events. This is more apparent when displayed in a Venn diagram, given in Figure 5.3.

Looking at the figure and generalizing, we can see that for any two sets A and B where $B \subset A$ (i.e., B is contained in A) it holds that:

$$A \cup (B \cap A) = A \tag{1}$$

$$A^{\mathsf{c}} \cap (B \cap A)^{\mathsf{c}} = A^{\mathsf{c}} \tag{2}$$

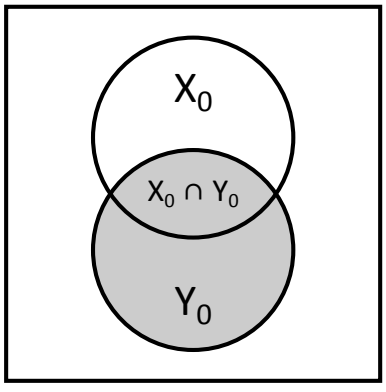


Figure 5.3: Venn diagram illustrating that Y_0 contains $X_0 \cap Y_0$ and thus $Y_0 \cup (X_0 \cap Y_0) = Y_0$

As a consequence, when we construct a tree in order to compute the probability of a set of random variable strings we must make sure that there are no leaf nodes that contain other leaf nodes present, which leads to incorrect probabilities.

Another scenario yielding incorrect probabilities is similar to the one we just described and also involves unaccounted dependence between different branches of the same Node, which we have assumed are actually independent. However, in this case there is no complete containment of one set of worlds by another set of worlds. A minimal example illustrating this scenario is the set of random variable strings {“X=0 Z=0”,“Y=0 Z=0”}. The corresponding tree is displayed in Figure 5.4.

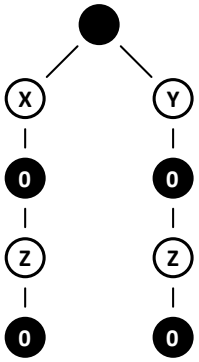


Figure 5.4: Tree where branches of root node are not independent, but are assumed to be

The leaf nodes share the assignment “Z=0” and are thus not independent. However, we treat them as such since the leaf nodes belong to different branches of the root node and the branches of a ●-node are assumed to be independent. The example case could be transformed into a proper tree relatively easily by making Z the only child of root, and making X and Y the child nodes of Z=0. This would reflect the situation correctly, i.e., X and Y being independent from each other but sharing the variable Z and its assignment Z=0. This is displayed in Figure 5.5.

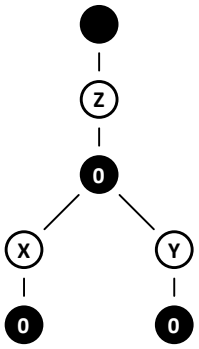


Figure 5.5: Resulting tree after performing partial Shannon expansion on corrupt tree of Figure 5.4

In order to fix the corrupt trees we apply a modified version of Shannon expansion. Shannon expansion yields mutually exclusive leaf nodes when applied to a tree of arbitrary assignments, as was discussed in Section 4.3.4 in the context of Count/Sum aggregates. We will only Shannon expand with respect to random variables that occur at least twice in the whole subtree that was created from the corrupt nodes. The result will thus not necessarily consist entirely of mutually exclusive leaf nodes like is the case for full Shannon expansion.

The Shannon expansion will make sure there will be no more corrupt nodes in the tree. After the expansion we insert the tree back into the original tree, as a subtree of the node that discovered the corrupt nodes – i.e., the closest common ancestor of the corrupt nodes. For the tree depicted in Figure 5.4 the corruption was detected in the root node when it encountered the assignment “Z=0” in two of its branches. The result of Shannon expanding the two branches of the example tree with respect to the random variable “Z” will be equivalent to the optimal solution displayed in Figure 5.5. We only expand with respect to Z since it is the only variable that occurs twice; X and Y only occur once and thus do not require expansion. The result is inserted back into the node that detected the corruption, which was the root node in this case. Thus, if such a corrupt composition of nodes is found much deeper in the tree we will remove the nodes there and insert them back at that position. The change is therefore applied only locally; unrelated

parts of the tree are left untouched.

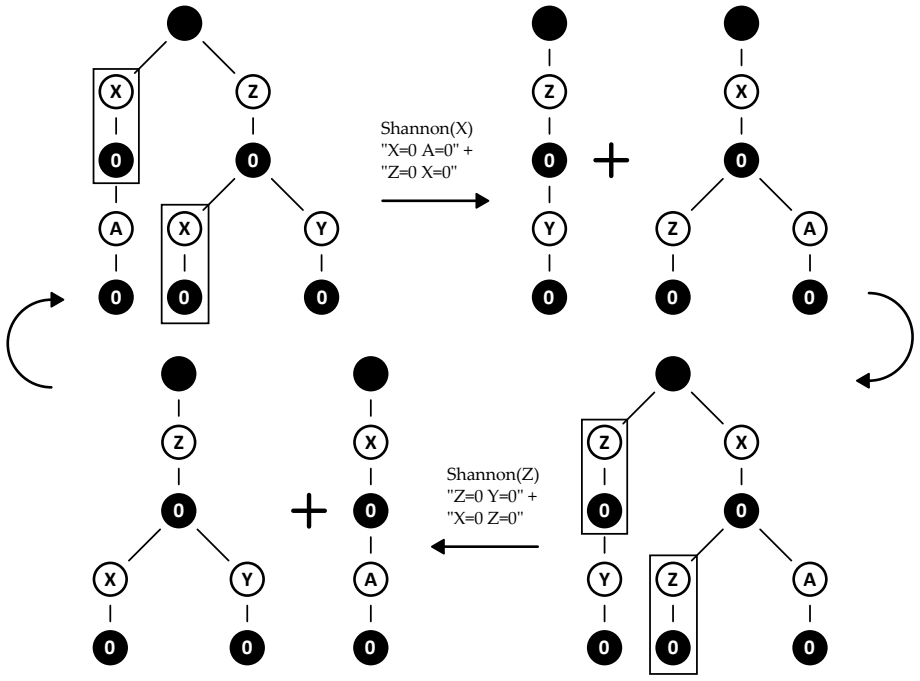


Figure 5.6: Example showing the potential of an infinite loop when related nodes are not included in the partial Shannon expansion of corrupt nodes

In addition to the corrupt nodes and their ancestor nodes upto the first common ancestor of all corrupt nodes we include any leaf nodes of the common ancestor that share at least one random variable with the corrupt nodes' random variables. This is done to prevent situations where the "fixed" trees introduce a new dependency with an existing node. By including all leaf nodes that share at least one random variable with one of the corrupt subtrees we make sure that such a new dependency will not be created. An example illustrating the need to include any related nodes is displayed in Figure 5.6, which shows that failing to include related nodes leads to an infinite loop.

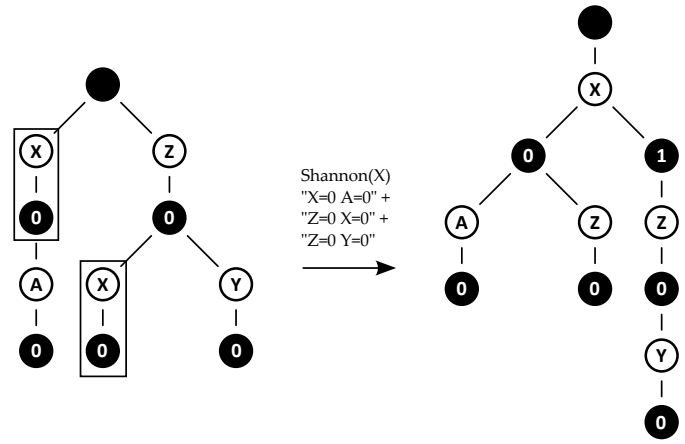


Figure 5.7: Including related node "Z=0 Y=0" prevents an infinite loop when fixing corrupted nodes

The solution is to always include nodes that share a random variable with the set of corrupted nodes, and also share the closest common ancestor of the corrupted nodes. This is illustrated in Figure 5.7. When fixing that tree, the node "Z=0 Y=0" in included in the partial Shannon expansion, since it shares the random variable Z with the corrupt node "Z=0 X=0", and it resides in the subtree rooted by the corrupt nodes' closest ancestor, the root node. This will then produce a tree that is guaranteed free of any other corrupted nodes, as can be seen in that figure. While the tree contains the assignment "Z=0" twice,

those nodes are not assumed to be independent since they occur in different branches of the X random variable, which are mutually exclusive. Because of this mechanism we can guarantee that any created tree will conform to the assumptions of independence for ●-nodes and mutual exclusion for ○-nodes, which in turn guarantees that the algorithms used for the computation of mutually exclusive events and independent events can be applied and will yield the correct probability.

6 Performance & Scalability

The performance and scalability of the implementation is tested by running benchmarks of different queries on different input documents. Multiple types of benchmarks were run, where either a variable of the input documents was being changed or the queries were varied but run on the same set of documents. The former measures the impact of the changed variable of the document on the execution time, whereas the latter approach tests the difference in performance between different types of queries. In particular, the performance difference between the 2 main types of aggregation queries supported by our implementation – Count/Sum and Min/Max.

6.1 Benchmark Method

The benchmarks are performed on a warmed-up database. Before a query is run on a set of documents, the query is run 4 times on a document that is not used in the benchmark. Each query is then run 4 times on each document, with the result averaged over those 4 runs. The execution time is split up in at most 3 categories:

- **Metadata:** Reading and storing the probabilities described in the document into a Java data structure for quick access.
- **XQuery:** Execution time of BaseX database operations for the generated query. That is, selecting matching elements and checking random variable consistency. Aggregation summary is not included.
- **Aggregation:** Time required for computing the aggregation summary values. This includes the time to generate the tree which was discussed in Section 4.2, Tree Data Structure.

We say *at most* since non-aggregation queries will not construct a tree and perform any aggregation, thus will not report an execution time in the last category. When the category of the displayed execution time is not mentioned, it is simply the sum of the separate categories – the total execution time.

6.2 Benchmark Results

The benchmark results will now be presented for each category of benchmarks.

6.2.1 Document Size

The document size was being varied to test the influence on query performance in this category of benchmarks. The document size is defined as the number of `<forecast>` elements present in a document.

Documents Each input document contains a varying number of forecast elements, where each forecast element consists of 4 weather properties with 4 possible options. An example forecast element is depicted in Figure 6.1. The document size is expressed in terms of the number of these forecast elements. The benchmark is run on 8 different documents with an exponentially increasing size. The first document contains 1024 forecast elements, whereas the last document has 131072. That is, document n has 2^{10+n} forecast elements, where n ranges from 0 to 7. The actual document size in megabytes can be expressed as a function of n as follows: $size(n) = (n+1)^2$. That is, the largest document has a size of roughly 256MB.

Queries A total of four queries were used in this benchmark. The queries have different characteristics and behave differently. Generally, the fewer predicates a query has the faster it will be processed by BaseX. In case of an aggregation query, an additional layer of processing is added to produce the summary values discussed in the Aggregate Queries section. This step depends entirely on the number of elements resulting from the non-aggregation part of the query produced by BaseX. As such, more selective queries tend to reduce the aggregation time since the number of elements is reduced compared to queries selecting everything.

```
<forecast wday="Wednesday" year="2014" month="1" day="1">
    <temperature rv="A=0">21.5</temperature>
    <temperature rv="A=1">22.5</temperature>
    <!-- 2 more temperature elements -->
    <sunshine rv="B=0">10.4</sunshine>
    <sunshine rv="B=1">9.6</sunshine>
    <!-- 2 more sunshine elements -->
    <rainfall rv="C=0">53.0</rainfall>
    <rainfall rv="C=1">51.8</rainfall>
    <!-- 2 more rainfall elements -->
    <windspeed rv="D=0">2.6</windspeed>
    <windspeed rv="D=1">1.8</windspeed>
    <!-- 2 more windspeed elements -->
</forecast>
```

Figure 6.1: Forecast element structure used in this benchmark

Type	Query
Simple aggregation	`max(/forecasts/forecast/temperature)`
Selective aggregation	`count(/forecasts/forecast[@wday="Saturday"][temperature > 18][windspeed < 3])`
Selective path	`/forecasts/forecast[temperature > 25]`
Complex aggregation	`count(/forecasts/forecast[temperature > 18][fs::forecast{1..2}/temperature > 18])`

Table 6.1: Benchmark queries

The different queries were chosen to measure the impact the document size has in various contexts. We will now discuss the results of this benchmark.

Results The simple aggregation of Figure 6.2 shows predictable behavior; the scalability is roughly linear and XQuery makes up a relatively small part of the total execution time. This is not surprising considering the query does not contain any predicates, and BaseX can thus yield the results using a straight-forward linear scan of all temperature nodes. Aggregation makes up a larger part of the whole since the generated tree contains all temperature values present in the document. Each document contains 4 temperature values per forecast, thus the aggregation tree will contain more than half a million elements for the last document. Furthermore, because the query applies the Max function, it has to first sort all those values in non-descending order which is very time consuming. The performance of Count and Sum aggregate functions is better but shows similar scalability. The results of the performance benchmark which compares the performance of the various aggregation functions are shown in Section 6.2.3.

In contrast to that simple aggregation, the selective aggregation in Figure 6.3 shows distinctly different behavior. It scales roughly linearly again, yet the execution time is divided very differently. Due to the selectiveness of this query with its 3 predicates that each select a small subset of the matching elements, the resulting set of forecast elements is extremely small compared to the size of the input document. This makes the aggregation step extremely quick; only a few values result from the XQuery step which makes building the aggregation tree and computing the summary values trivial.

The execution time of the aggregation step is so small that it cannot be distinguished in the result graph. Each bar is seemingly only divided between Metadata (i.e., reading the probabilities and storing them in memory) and XQuery (traversing the uncertain document and selecting the matching elements). To illustrate this; the time required to compute the aggregation values for the largest document was only 32 milliseconds.

The selective path query of Figure 6.4 is similar to the previously discussed selective aggregation in terms of behavior. Here too the XQuery part makes up the majority of the execution time, albeit slightly more

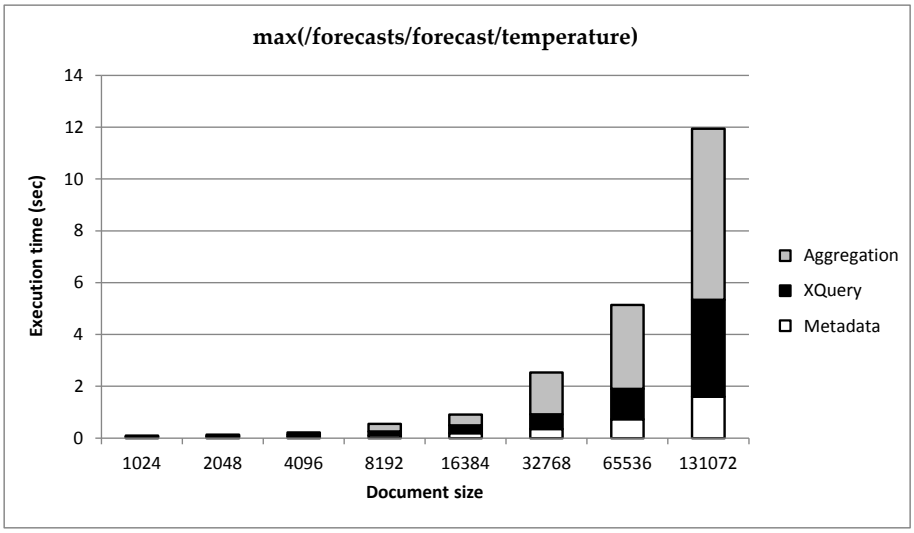


Figure 6.2: Simple aggregation

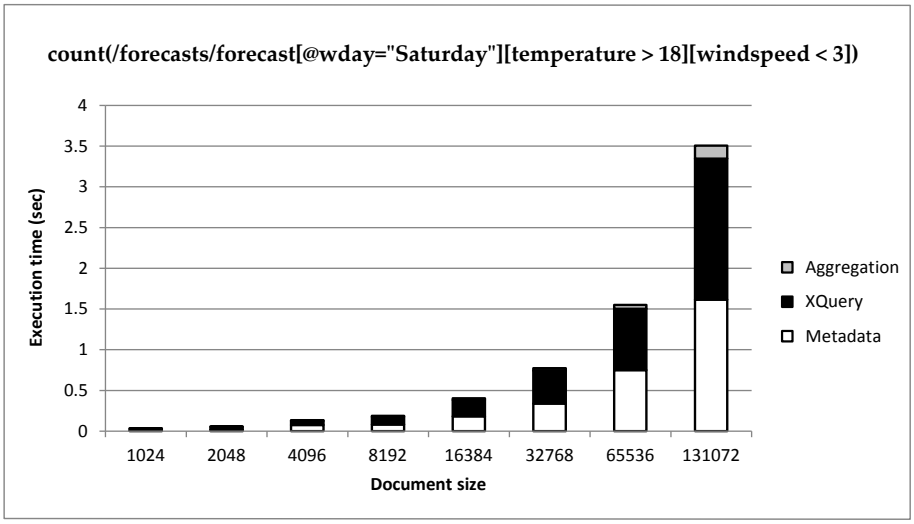


Figure 6.3: Selective aggregation

due to an overall (slightly) higher execution time. This similarity is expected; the execution time of a selective aggregation is determined primarily by the XQuery execution time as explained earlier. This is due to the definition of a *selective* query; it does not yield many results but selecting the right results might take a lot of I/O on the database side. Aggregating the resulting small set of answers, on the other hand, will not take long at all. Consequently, the performance and scalability of an aggregation query and a non-aggregation query are essentially the same as they are both bound by the BaseX execution time. The difference between these specific two queries can be explained by the fact the selective aggregation in Figure 6.3 starts with a predicate which selects only the Saturdays which reduces the set of forecasts to only $\frac{1}{7}$ of the total set. Since the `wday` attribute only has 1 possibility (instead of 4) BaseX can evaluate it quicker than the other predicates which makes the query overall slightly faster than the selective path query which applies a single `temperature` predicate.

The last query, complex aggregation shown in Figure 6.5, does not scale exactly linearly with respect to the size of the input documents. Rather, the execution time increases by a factor of about 3 when the document size increases by a factor 2. This is true for the larger documents. For the smaller documents the increase is about linear. Notice this predicate uses shorthands discussed in Section A.1.2; `fs::forecast{1..2}/temperature > 18`, which translates to

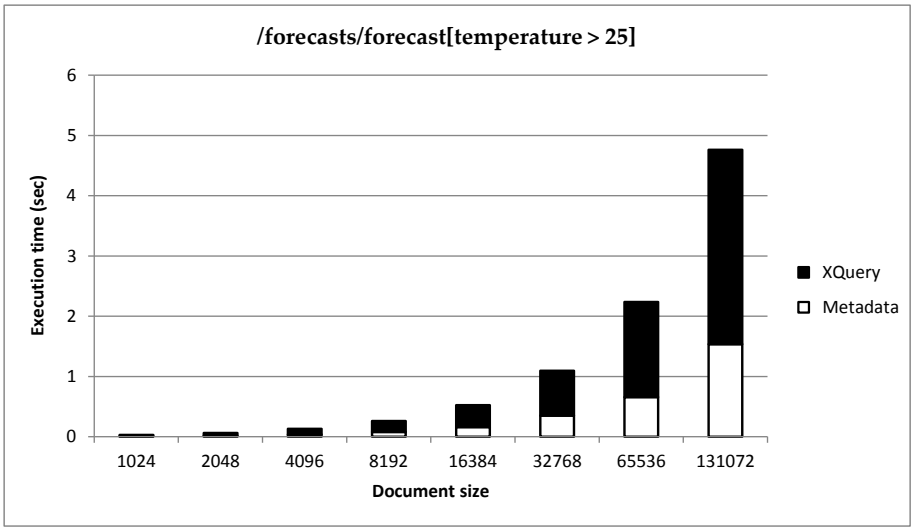


Figure 6.4: Selective path

```
following-sibling::forecast[1]/temperature > 18 and
following-sibling::forecast[2]/temperature > 18
```

BaseX generates all possible combinations of the predicates, which contributes to the higher execution time. It combines the 3 predicates; the temperature elements of the context forecast, the temperature elements of the first following sibling forecast and the temperature elements of the second following sibling forecast. Each forecast contains 4 temperature elements, thus for each forecast element considered, a set of $4^3 = 64$ combinations is generated. That is not very large number, but it is done at every single forecast element which is reasonably large with the highest number for this specific query being 131072, thus requiring a total of $2^{17} \cdot 2^6 = 2^{23} = 8388608$ combinations to be generated. While we can apply optimizations which should reduce this number, as discussed in Section 3.6.4, this is not enough to push the execution time down to the extent that it increases by the same factor as the document size. The execution time of the aggregation in particular is increased due to the large number of results that are being generated from all combinations.

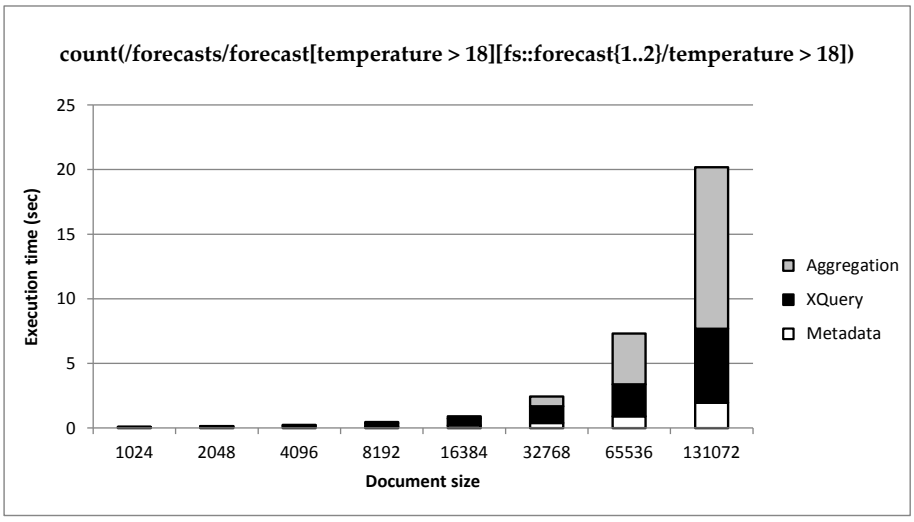


Figure 6.5: Complex aggregation

6.2.2 Document Uncertainty

Documents The documents used in this benchmark have a varying degree of uncertainty, but are otherwise similar in structure and size. The document structure displayed in Figure 6.1 is used again, albeit with a different number of random variables. We have used documents with 0%, 25%, 50%, 75%, and 100% uncertainty in this benchmark, each with 2^{16} `<forecast>` elements. The uncertainty is implemented per `<forecast>` element; either all of its child elements – temperature, windspeed, sunshine, rainfall – are uncertain or all of them are certain. For example, in case of 50% uncertainty and forecast elements $f_0, f_1, \ldots, f_i$, all forecast elements with an even i are uncertain and the forecast elements with an odd i contain only certain child elements. Consequently, the document size is also dependent on the uncertainty, since an uncertain element has (in this case) 4 different alternatives to choose from whereas a certain element only has 1.

Alternatively, we could have chosen to add 4 certain elements to replace the single uncertain element. This would keep the document size roughly equal – we still omit the random variable assignment attributes, thus the size would decrease slightly – but would also increase the number of paths that are traversed during query evaluation. To elaborate; only a single alternative for an uncertain element can be considered simultaneously and the query execution will prune any inconsistent path (i.e., where two or more alternatives of the same choice are selected). This pruning would not occur when we add certain elements without random variable assignments, thus making query execution more complex when we decrease uncertainty. It was decided it makes more sense for the certain case to be more similar to selecting a single possible option, rather than selecting all of them. The decreased document size is then a logical result of it, and is *not* considered a flaw in this benchmark.

Queries We used a subset of the queries used in the document size benchmark to test the influence of the document uncertainty in various scenarios. The complex aggregation query was left out due to performance issues identified in the previous benchmark. The queries are listed in Table 6.2.

Identifier	Query
Query 1	`max(/forecasts/forecast/temperature)`
Query 2	`count(/forecasts/forecast[@wday="Saturday"][temperature > 18][windspeed < 3])`
Query 3	`/forecasts/forecast[temperature > 25]`

Table 6.2: Document uncertainty benchmark queries

Results The results of the document uncertainty benchmark are displayed below in Figure 6.6. The results of query 1 in particular stand out due to fluctuating performance between the various levels of uncertainty. Most notable is the highest execution time for the document with 0% uncertainty. It is hard to explain why this happens, considering this document has the smallest amount of elements and does not require any random variable string operations since it does not contain any. It is possibly related to the tree data structure, where all 2^{16} `temperature` values are stored in a single list at the tree's root node. There are no other nodes in the tree due to the lack of uncertainty. It is possible doubling the internal `Array` used to hold the items of Java's `List` each time the limit is reached is causing performance issues.

The other queries show a linear increase in execution time with respect to the uncertainty of the input document. As discussed in the Documents paragraph, an increase in uncertainty comes with an increase of the number of XML elements in the document. Considering the queries used in this benchmark scale linearly with respect to the document size as shown in Section 6.2.1, this document size increase on its own explains the increase in execution time. Thus, the increase in uncertainty on its own seems to have little or no influence on query execution time.

6.2.3 Aggregation Functions

This benchmark tests the performance and scalability differences between the two classes of supported aggregation functions – Count/Sum and Min/Max. Within a class the implementation is equivalent and

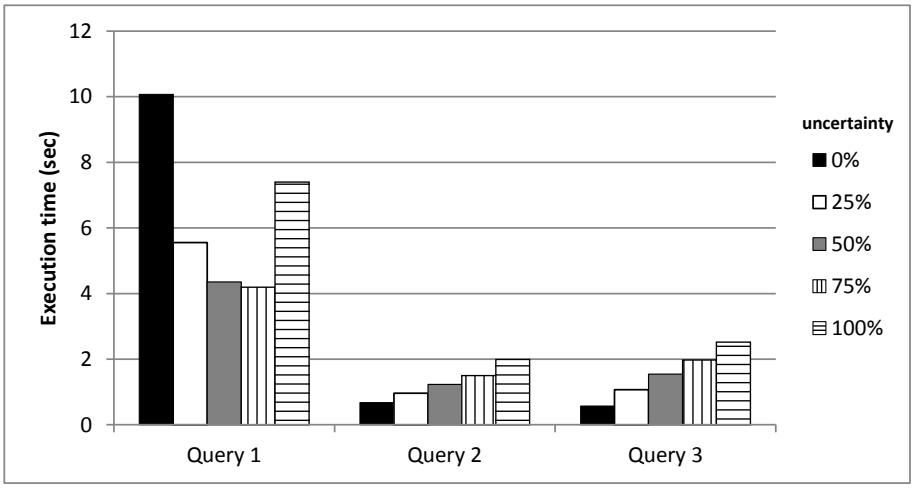


Figure 6.6: Document uncertainty benchmark results

thus performance and scalability are as well, which is why we performed this benchmark with one member of each class; Count and Max. The Avg aggregation function is not considered a separate class since it only yields the expected value which is obtained by taking dividing the expected value from Sum by the expected value from Count and will thus show similar performance and scalability, at most a factor 2 slower since it has to calculate both. Moreover, since Avg only computes the expected value and not the variance or extreme values, comparing its performance and scalability with the other aggregation functions (which do compute all summary values) is fairly meaningless.

Documents The documents used in this benchmark have the same structure as the documents used in the Document Size benchmark. Instead of 8 different document sizes we chose just 5 in this benchmark; the number of `<forecast>` elements is again set to 2^{10+n} but n now ranges from 3 to 7 inclusive.

Queries Because we are testing only the aggregation scalability which is applied after query processing on the set of query results, the queries themselves do not influence the aggregation at all. Therefore, we used a single "base" query in this benchmark and applied the Count and Max aggregation functions to it. The base query is the non-aggregation part of the Simple aggregation query listed in Table 6.1; `/forecasts/forecast/temperature`.

Results The result of the benchmark in Figure 6.7 shows similar performance for the Count function and the Max function. The latter takes around 1.5 times longer than the former for the largest document. In terms of scalability both functions appear to scale roughly linearly with respect to the number of aggregated values; when the number of aggregated values is doubled the execution time follows suit. It is important to note that the displayed execution time is *only* the aggregation part of the query, i.e. the Metadata and XQuery categories are not counted. The horizontal axis displays the number of values that were aggregated, i.e., the number of `<temperature>` values in each document.

The small difference in performance can be explained by the fact that the Max aggregate function has to first sort all leaf values of the generated tree structure in order to yield its summary values, whereas Count operates directly on the unsorted values. Sorting the leaf values is required for Min and Max, the rationale for which was provided in Section 4.4.

However, it should be noted that there also exist cases where the performance of Count/Sum will be worse than that of Min/Max, because Count/Sum applies Shannon expansion before building the aggregation tree. Thus, depending on the random variable strings resulting from the query the Shannon expansion step can take a long time. In this benchmark we performed a query without any predicates, thus the resulting random variable strings per unique temperature element did not need to be Shannon expanded, thus the performance here is a best case scenario for Count/Sum. This was intended, as we planned

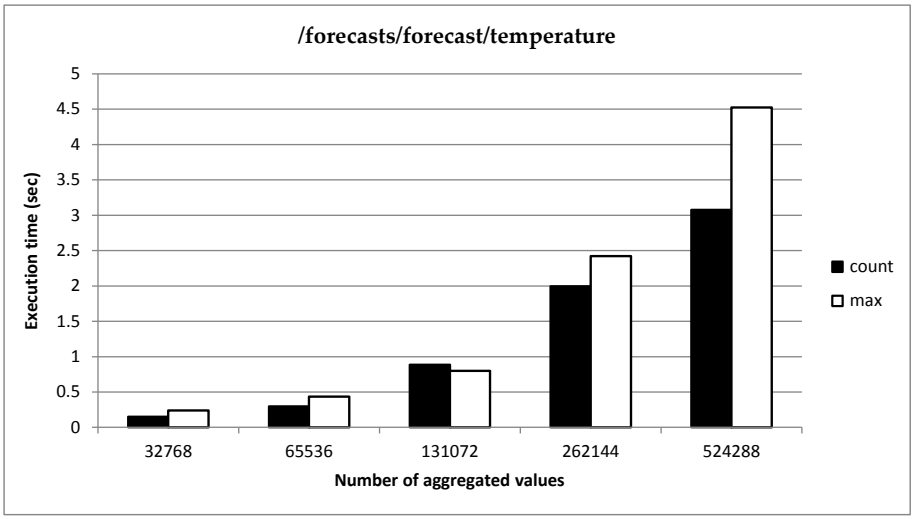


Figure 6.7: Aggregation benchmark result

to measure only the scalability of the aggregation tree and the summary functions. It is, however, an important issue to note.

6.2.4 Predicate Size

In this benchmark we measure the performance and scalability of an increasing number of predicate expressions in the form of (nested) And expressions used in a query predicate. As detailed in Section 3.6.4, an And expression results in an XQuery snippet that computes the Cartesian product of the operands. The same occurs for multiple predicates, i.e., `/path[pred_1][pred_2][pred_3]`, since that is equivalent to `/path[pred_1 and pred_2 and pred_3]`. The size of this product grows exponentially with respect to the number of operands, thus it is expected that the performance of a query that contains many such expressions is poor.

Documents For this benchmark we have used a single document with $2^{13} = 8192$ forecast elements, each containing 4 random variables with 4 possibilities like in the other benchmarks. The document size was deliberately chosen to be fairly small, since we expect the runtime of the queries to scale exponentially with respect to the number of conjunctions and disjunctions in the query.

Queries We have created three types of queries for this benchmark: (1) selective queries that apply predicates that are satisfied by almost no values, (2) nonselective queries which apply predicates that are satisfied by almost all values present in the document, and (3) standard queries which apply predicates that are each satisfied by roughly half of the values. The queries we used are listed in Table 6.3. Each of the queries in the table below will be executed with the first 2, 3, and 4 predicates, thus a total of 9 queries are executed in total.

Type	Query
Selective	`//forecast[temperature > 25 and windspeed < 2 and sunshine > 16 and rainfall < 10]`
Nonselective	`//forecast[temperature <= 25 and windspeed >= 2 and sunshine <= 16 and rainfall >= 10]`
Standard	`//forecast[temperature > 7.5 and windspeed > 6 and sunshine > 11 and rainfall > 100]`

Table 6.3: Predicate size benchmark queries

To explain why these queries are selective or nonselective, it is important to know the domains of the various weather properties. We have listed the domains in Table 6.4.

Property	Domain
Temperature	$\{-12.5 \dots 27.5\}$
Sunshine	$\{4.0 \dots 18\}$
Rainfall	$\{0 \dots 200\}$
Wind speed	$\{0 \dots 12\}$

Table 6.4: Domains of weather property values

Results The results are listed in Figure 6.8. It is apparent that the nonselective queries, which apply predicates that are satisfied by almost all values in the document, show roughly exponential scalability. In contrast, selective queries and "standard" queries that apply predicates that are satisfied by a much smaller subset of values show only slight increases in execution time when the number of operands increases.

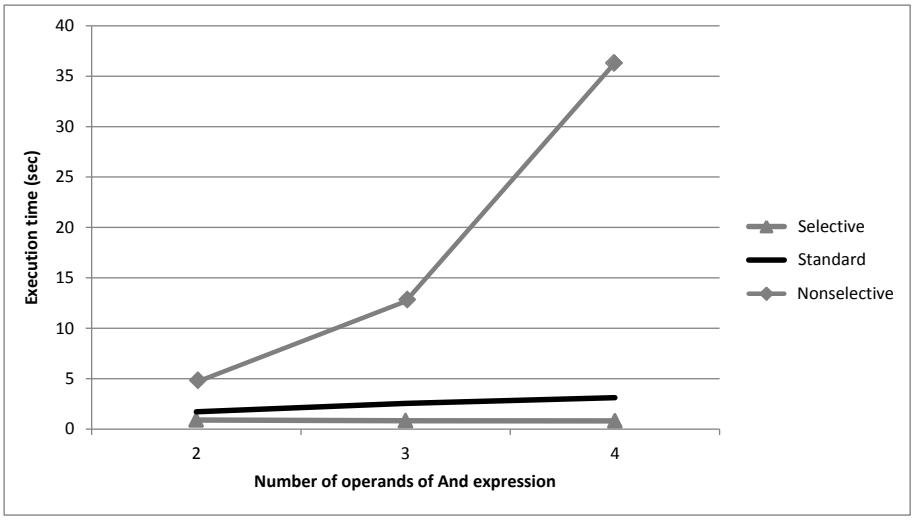


Figure 6.8: Results of the predicate size benchmark

The major performance difference can be explained when we look at possible optimizations for an And expression. It has been discussed earlier in Section 3.6.4, but we will reiterate it here. When evaluating an And expression, i.e. $e_1 \wedge \dots \wedge e_n$, we can return `false` whenever any of the operands $e_1 \dots e_n$ evaluates to `false`. Thus, when applying selective or standard predicates, a fairly large number of values of the operands will evaluate to `false`, thus we do not have to compute the Cartesian product since we already know the And expression will not be satisfied when any of its operands is `false`.

However, in the case of highly nonselective queries such as the one we have used in this benchmark, almost all values of the evaluated operands will be `true`. As a result, we have to compute a Cartesian product of the operands for each `<forecast` element. Afterwards, each resulting value has to be inserted in a tree structure in order to compute the probability of each value, as well as the probability of all values combined in order to yield the probability of the empty result. The size of the Cartesian product grows exponentially with respect to the size of the operands, explaining the very poor performance displayed by the nonselective query in Figure 6.8.

7 Discussion

In this section we describe some of the limitations of the current implementation, and if known their possible solutions.

7.1 Scalability of And Expressions

And expressions show exponential scalability with respect to the number of operands, due to computing the Cartesian product of the operands. Each operand is a sequence of maps, and since it is an And expression we have to check all combinations for consistency and combine their random variable strings. In order to do this, we compute the Cartesian product over all operands, which scales exponentially. The And expression is optimized when possible, i.e., we can return `false` when any of the operands is `false` and do not have to create the full Cartesian product in every situation. However, we have shown in Section 6.2.4 that in cases where queries apply predicates that are satisfied by almost all nodes in the document these optimizations will hardly ever be applied, since almost all values of the operands evaluate to `true`. The same issue occurs for multiple subsequent predicates, which is equivalent to an And expression. That is, `/path[p1][p2][p3]` is equivalent to `/path[p1 and p2 and p3]` and shows exponential scalability as well with respect to the number of predicates.

We have not found a solution to the problem of having to determine consistency for an And expression efficiently. In many cases, creating the Cartesian product per matching context node in the document does not adversely affect performance, in part due to optimizations that we discussed above. This is a major performance concern of our implementation, and effectively makes it impossible to issue certain queries on certain documents.

7.2 Memory Usage

The memory used by BaseX Server when running our implementation on large documents can get very high, > 1GB is normal. What makes matters worse is the memory does not seem to be freed up once a query completes. Executing consecutive queries will drive up memory usage further, even if the same query is executed all the time. We have observed this issue on stock BaseX as well – i.e., BaseX without running our plugin – although the overall memory usage was much lower in comparison and the increase in memory usage in that case is not an issue.

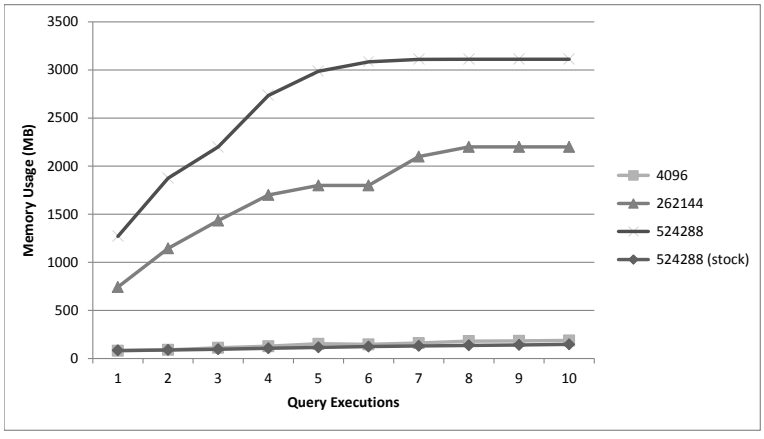


Figure 7.1: Memory usage for the query `/forecasts/forecast[temperature > 25]`

The graph in Figure 7.1 shows the memory usage of the simple query `/forecasts/forecast[temperature > 25]` on three different documents. Their structure is equivalent to the documents used in the benchmark of Section 6.2.1, although their structure is not particularly important for the memory usage depicted here. The graph contains two large documents with 262144 and 524288 random variables each, and a small document with only 4096 random variables. Additionally, the memory usage for the same query executed

on BaseX without calling the plugin's `query` function is displayed. This does not take the probabilities into account and does not return a correct answer but is included to contrast the enormous amount of memory that is used to execute the probabilistic query.

The memory usage keeps increasing until the same query has been executed roughly 8 times, then it remains stable. For the probabilistic queries issued on the large documents, the difference in memory usage of the BaseX Server after the first and last execution of the exact same query is very large, a factor of ±3 and ±2.5 for the second largest and largest document, respectively. The small document and the large document without uncertainty support show a difference of about a factor 2.

```
1  let $int := function() {
2      let $values := (1 to 1000000)
3      return fold-left($values, 0, function($sum, $value) {
4          $sum + $value
5      })
6  }
7
8  let $map := function() {
9      let $maps := for $i in (1 to 1000000) return { 'value' : $i }
10     return fold-left($maps, 0, function($sum, $m) {
11         $sum + $m('value')
12     })
13 }
```

Figure 7.2: Memory comparison functions

Investigating the issue, it appears BaseX requires a lot of memory when maps are used, the key-value data type that is heavily used in our implementation. We ran a query on stock BaseX which calculates the sum of all numbers from 1 to 1,000,000 inclusive, without making use of the obvious $\frac{n(n+1)}{2}$ but rather by summing the individual numbers. We used two methods; one of them uses a sequence of integers and sums those directly. The second method uses a sequence of maps, each associating the key "value" with an integer value to be summed. The definitions of both functions used in the test are displayed in Figure 7.2.

We tested the memory usage by manually running both the $int and $map functions 10 consecutive times on a cold BaseX server. BaseX showed significantly higher memory usage when using the map variant: the maximum memory usage was reached after 8 executions and peaked at roughly 648MB. This is much higher than the memory usage in the plain integer case, which peaked at ±118MB which was reached after only 4 executions. The graph is shown in Figure 7.3.

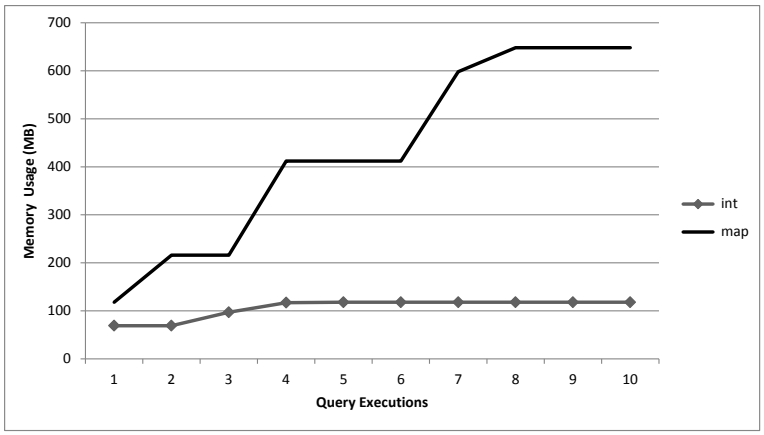


Figure 7.3: Memory usage comparison between sequence of integers and sequence of maps

While it is not surprising a key-value data type uses more memory than an integer, the fact that the memory usage is increased by a factor 5.5 between the first and last query execution as opposed to about

1.70 in the integer case shows there are possible issues with the implementation of the map data type or the caching of variables of that type. It was considered out of scope to investigate the root cause of the high memory usage within the source of BaseX. We have shown that an unmodified BaseX shows similar high memory usage when a lot of map variables are created and thus the high memory usage for probabilistic queries is most likely a direct result of using the map data type. The maps we use in probabilistic queries are more complex than the one in the test we ran, since the values that are stored within it can be any arbitrary XML subtree contained within the document or any other XQuery data type.

8 Conclusions

We have presented an approach to evaluate probabilistic queries over uncertain XML data, using an existing XML DBMS. An uncertain data representation format which uses random variables to annotate uncertain elements in the XML document is described. The annotations are used to describe mutually exclusive or independent elements, as well as to attach a probability to every uncertain element. Two or more elements are mutually exclusive when they are annotated with different assignments of the same random variable, or independent when different random variables are assigned to the items. Each random variable assignment has an associated probability, indicating the probability of an element given that its ancestors exist.

We utilize the annotations during query evaluation in order to select the correct elements for a query and compute their probabilities. This functionality is added as a plugin to the existing XML DBMS. We leverage all core parts of the DBMS; primarily its XQuery parser and query evaluator. In order to introduce the probabilistic awareness that the DBMS lacks we transform an input XQuery using transformation rules defined for all supported XQuery expressions. The implementation supports basic XPath queries with simple predicates, as well as most aggregation functions supported by XQuery; Count, Sum, Min, and Max. The result of an aggregation query over uncertain data is a set of values that describe the distribution of the result values in terms of their expected value, minimum value, maximum value, variance, and standard deviation. For the Min and Max aggregates the top-k results and their probabilities are also computed. The result of regular queries is a set of distinct matching elements, each with an associated probability. The probability is equal to the sum of probabilities of all possible worlds that yield the element as a result to the query.

Benchmarks have been conducted and show generally good performance and scalability when querying documents with as many as 2^{19} different random variables. We have identified certain cases that show exponential growth in execution time with respect to the number of operands in an And expression which can cause performance issues depending on the documents and queries that are involved. These situations occur when a query applies nonselective predicates joined by an And expression. In order to evaluate an And expression, we generate the Cartesian product of the evaluated operands. An evaluated operand is represented as a sequence of individual result values and their random variable annotations. Since the predicates are nonselective, the sequences involved in the Cartesian product tend to contain many `true` values since many elements match the nonselective query. When all involved expressions of a Cartesian product are `true` we cannot apply any optimizations and have to generate it entirely, yielding a much larger sequence of all combined inputs causing the performance and scalability issues.

We will now proceed with formulating answers to the posed Research Questions.

Can we query uncertain data without generating all possible worlds? We have shown that we can successfully query an uncertain document directly without instantiating all possible worlds, and yield a correct result to an XQuery that makes use of a subset of the XQuery language. This is achieved by transforming the original XQuery to a new XQuery that takes into account the probabilistic nature of the data while preserving the XPath steps specified in the original XQuery. By evaluating the transformed XQuery directly on the document which describes all possible worlds, we are essentially querying all possible worlds that match the given query simultaneously without explicit expansion.

Are the obtained query results semantically equivalent to the actual results? The query results generated by our implementation are correct but are usually not semantically equivalent due to the loss of information when compared to the actual result of the probabilistic query, i.e. the result of the query when it is executed in all possible worlds. That is, we provide for each unique result value – generally a distinct XML element from the input document – the sum of the probabilities of all possible worlds that would yield the value as a result to the query. We do not, however, provide these probabilities for combinations of elements that occur together in (a set of) possible worlds. As such we provide a set of results that is not semantically equivalent to the actual result, but the given result values and probabilities are accurate. More specifically, the result we produce is equivalent to the actual result when the latter performs a group by operation on each distinct value and sums the involved probabilities per value.

For aggregate queries we provide values that describe the distribution of the result values. The actual result values and their probabilities cannot be computed for Count and Sum aggregates without expanding all possible worlds, thus the answer to such queries is by definition not semantically equivalent with the actual result. For Min and Max aggregates we do yield the actual result values in addition to values that describe the result distribution. Therefore, the result to a Min or Max aggregate function is semantically equivalent to executing the function in every possible world.

Can the answer to aggregation queries be computed efficiently? We have performed a benchmark which compares the scalability and performance of the Count/Sum aggregate functions to the Min/Max aggregate functions. It was shown that both classes of aggregation functions show roughly linear scalability with respect to the number of elements that were being aggregated. The query on the largest document yielded over half a million values to be aggregated. The execution time of the Min/Max class was shown to be slightly higher – upwards of 50% in the worst case that was tested – than that of the Count/Sum class. We believe this is primarily caused by the sorting of values that is required for Min/Max, but not for Count/Sum due to their different implementations. It is important to note that while there is some overhead in the Min/Max case, the result to those aggregate functions will also include the top-k result values and their probabilities unlike the Count/Sum aggregates.

How does the solution scale with respect to different documents and queries? Benchmarks have shown the execution time of various queries generally scales linearly with respect to the document size. However, we have identified cases involving certain queries and documents that lead to exponential growth of the execution time with respect to the number of operands of an And expression. This is the result of having to compute a Cartesian product of the operands, which are sequences of elements. The Cartesian product is created in order to verify the consistency of each combination of operands, and their truth value. This process takes a lot of time, and additionally this generates a much larger intermediate result which slows down all follow-up operations applied to the intermediate results. In particular, the tree structures that are created from the sequence of maps will also be much larger.

8.1 Future Work

The implementation shows some limitations, among which is the high memory usage for query evaluation discussed in Section 7. We have identified a possible cause for the high memory usage in the application of XQuery's map data type, which appears to lead to high memory usage in BaseX, the XML DBMS we used to implement our prototype. It is possible that this issue will not occur in other available XML DBMSs, this was not tested. However, it is an issue that deserves some more attention. While the map data type is used extensively throughout the transformed XQuery it remains an implementation detail; there are other ways to keep track of intermediate result values and their random variable strings without using maps. If other XML DBMSs show similar memory usage for map data types, it is advisable to experiment with other ways to represent the intermediate query results.

As was briefly mentioned in Section 4.3.4, it is possible that the aggregation tree we create for Count and Sum contains a corrupt subtree which can possibly lead to inaccurate summary values, except for the expected value which will remain correct. In the tree structure we use to compute probabilities these corrupt subtrees will be fixed automatically, which is the same structure we use to perform Shannon expansion on distinct values of Count and Sum. However, the aggregation tree itself does not fix corrupt subtrees since it also contains values, making Shannon expansion more difficult. Thus, if a corrupt subtree is created from the random variable strings belonging to different values this will not be fixed. We mentioned that such a situation is very uncommon, but it is something that can be looked into in future work.

Another topic for future research is finding optimizations for handling And expressions in query predicates. In our implementation, a Cartesian product of the operands of an And expression is created to check the consistency and build the combined random variable string of each combination of operands. As each operand is a sequence of maps, this results in poor scalability and performance in certain situations. When dealing with an And expression consisting of n operands – i.e., $e_1 \wedge \cdots \wedge e_n$ – we can stop the

evaluation when we find a `false` value since that would make the entire conjunction `false`. However, such optimization cannot always be applied and the implementation will then proceed to generate all combinations of the operands which scales exponentially. This situation can occur even for trivial queries with as few as two operands, depending on the size of the sequences that represent each operand expression and the number of Cartesian products that is being generated. Ideally, the Cartesian product computation is entirely avoided as it closely resembles expanding all possible worlds.

In addition to optimizations there is also the topic of extending support to more advanced XQuery constructs, specifically the FLWOR expression – For Let Where Order by Return. Our implementation only supports basic XPath expressions consisting of a path query and predicates. While fairly expressive queries can be constructed using these tools, the FLWOR greatly expands the possibilities. It is also considerably more complex than other expressions, which is the main reason we have not addressed it in this research.

In terms of aggregation functions we only properly support Count/Sum and Min/Max. The Average aggregate function only yields the expected value through an estimation using the expected values of Sum and Count but does not provide the complete set of summary values that we compute for the other aggregation functions. There are likely ways to compute the summary values for Average as well, which can be investigated further. Additionally, extending the aggregation support of Count/Sum to be able to yield the top-k possible values for the aggregation function and their probabilities is an interesting topic that we could not address in this research. We only provide the minimum, expected, and maximum values and cannot compute the actual values and their probabilities, or even just the top-k results for a small value of k. While we have argued that the yielded values combined with the variance and standard deviation represent the actual result very well, the top-k results would further improve this. We do provide top-k for arbitrary values of k for Min/Max aggregates.

Apart from specific improvements to the implementation that we created as a plugin to BaseX, there are larger goals to be achieved in the area of probabilistic databases in general. Thus far, probabilistic databases are generally built on top of existing databases that have no native support for uncertain data. The query planners and query optimizers utilized by such databases cannot properly create a plan for uncertain queries, thus while probabilistic database can rewrite queries in such a way that the correct answer is provided, the way that such a query is executed based on the plan generated by the query planner will likely be suboptimal since properties such as mutual exclusion of various data items are not taken into account. It would thus be interesting to see if a fully native probabilistic database can be developed, where support for data uncertainty is not "patched in" but rather is one of the most important factors in designing the core parts of the database.

Acknowledgements

This thesis would not have been realized without the support of great people I have worked with over the course of this project. In particular I would like to thank my first supervisor Maurice, who offered great insights on multiple occasions which helped solve some of the challenges. This was particularly important at the start of the project, and perhaps even more so during an exploratory project that was done in preparation of this final project that was also supervised by Maurice. During this time I learned he is one of the busiest people of the DB group, making me appreciate the time he found to help me out even more. I would also like to extend my appreciation to my second supervisor Mena. While the number of meetings we have had was fairly limited, it was valuable to hear his thoughts and feedback on the project.

Perhaps my greatest appreciation goes to my mother, Marie-Louise. Having shown amazing support, not only throughout my path as a student at this university but during my whole life, her contributions triumph all others. She always encouraged my curiosity when I was younger, something I will always be grateful for. Considering I could – and perhaps *should* – have finished this chapter of my life faster than I did I suspect she feels slightly relieved that it is coming to a successful close now, although above all else she will undoubtedly be proud I can now call myself a Master of Science.

References

[1] Benny Kimelfeld, Yuri Kosharovsky, and Yehoshua Sagiv. Query efficiency in probabilistic XML models. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 701–714. ACM, 2008.

[2] Maurice van Keulen, Ander de Keijzer, and Wouter Alink. A probabilistic XML approach to data integration. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 459–470. IEEE, 2005.

[3] Serge Abiteboul and Pierre Senellart. Querying and updating probabilistic information in XML. In *Advances in Database Technology-EDBT 2006*, pages 1059–1068. Springer, 2006.

[4] Edward Hung, Lise Getoor, and VS Subrahmanian. Probabilistic interval XML. In *Database Theory–ICDT 2003*, pages 361–377. Springer, 2003.

[5] Edward Hung, Lise Getoor, and VS Subrahmanian. PXML: A probabilistic semistructured data model and algebra. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 467–478. IEEE, 2003.

[6] Benny Kimelfeld and Yehoshua Sagiv. Matching twigs in probabilistic XML. In *Proceedings of the 33rd international conference on Very large data bases*, pages 27–38. VLDB Endowment, 2007.

[7] Pierre Senellart and Serge Abiteboul. On the complexity of managing probabilistic XML data. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 283–292. ACM, 2007.

[8] Andrew Nierman and HV Jagadish. ProTDB: Probabilistic data in XML. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 646–657. VLDB Endowment, 2002.

[9] Hosagrahar V Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks VS Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, et al. TIMBER: A native XML database. *The VLDB Journal–The International Journal on Very Large Data Bases*, 11(4):274–291, 2002.

[10] Te Li, Qihong Shao, and Yi Chen. PEPX: a query-friendly probabilistic XML database. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 848–849. ACM, 2006.

[11] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. *Technical Report*, 2004.

[12] Charu C Aggarwal. Trio A System for Data Uncertainty and Lineage. In *Managing and Mining Uncertain Data*, pages 1–35. Springer, 2009.

[13] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and simple relational processing of uncertain data. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 983–992. IEEE, 2008.

[14] Jihad Boulos, Nilesh Dalvi, Bhushan Mandhani, Shobhit Mathur, Chris Re, and Dan Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 891–893. ACM, 2005.

[15] Parag Agrawal and Jennifer Widom. Generalized Uncertain Databases: First Steps. In *Proceedings of the 4th International VLDB Workshop on Management of Uncertain Data (MUD2010), Singapore*, 2010.

[16] Christoph Koch and Dan Olteanu. Conditioning probabilistic databases. *Proceedings of the VLDB Endowment*, 1(1):313–325, 2008.

[17] Raghotham Murthy, Robert Ikeda, and Jennifer Widom. Making aggregation work in uncertain and probabilistic databases. *Knowledge and Data Engineering, IEEE Transactions on*, 23(8):1261–1273, 2011.

[18] Lixia Chen and Alin Dobra. Efficient Processing of Aggregates in Probabilistic Databases. Technical report, Technical Report REP-2008-454, University of Florida, 2008.

[19] Serge Abiteboul, T-H Hubert Chan, Evgeny Kharlamov, Werner Nutt, and Pierre Senellart. Aggregate queries for discrete and continuous probabilistic XML. In *Proceedings of the 13th international conference on database theory*, pages 50–61. ACM, 2010.

[20] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 141–152. VLDB Endowment, 2003.

[21] BaseX | The XML Database. http://basex.org. Accessed 3 March 2014.

[22] World Wide Web Consortium. XQuery and XPath Data Model 3.0: Type system. http://www.w3.org/TR/xpath-datamodel-3/#types-hierarchy. Accessed 7 May 2014.

[23] World Wide Web Consortium. XQuery and XPath Data Model 3.0: Sequences. http://www.w3.org/TR/xpath-datamodel-3/#sequences. Accessed 7 May 2014.

[24] World Wide Web Consortium. XPath and XQuery Functions and Operators 3.0. http://www.w3.org/TR/xpath-functions-30. Accessed 21 May 2014.

[25] Tony Finch. Incremental calculation of weighted mean and variance. *University of Cambridge*, 2009.

Appendices

A Configuration and Usage

This section will explain how the plugin can be used to execute a query on uncertain data. It is assumed the uncertain data is already stored in the database in the proper format. That is, the format where random variables are used to describe the uncertain values, and those elements and attributes are in the `db.ewi.utwente.nl` namespace. Without it, the plugin will not be able to identify the random variables and probabilities.

A.1 Configuration Options

The valid configuration options and their effects are listed below. Most options are toggles, i.e., valid values are either 0 or 1 where 1 corresponds to 'on' and 0 to 'off'. The set of all configuration options should be passed as an XQuery map to the query function. An example illustrating how the configuration options can be set will be given in the next section.

apply.rounding Controls whether or not probabilities in the output are rounded. The plugin will never apply any intermediate rounding internally.

clean.pxml.attrs Removes all attributes belonging to the `pxml` prefix. More specifically, removes any attributes attached to the `db.ewi.utwente.nl` namespace.

disable.shorthands Disables shorthands entirely, nothing will be searched and replaced.

display.num.worlds Outputs the number of possible worlds in the current document to the standard output of the server.

display.num.rvars Outputs the number of random variables in the current document to the standard output of the server.

display.timings Displays execution time for various parts of the query execution on the server's standard out.

enable.comments Adds comments to the generated probabilistic query, indicating which part of the query processes which part of the original query.

expand.result Expands all possible worlds and applies the query. Yields the correct answer but is not scalable beyond small documents.

group.by.value Whether the results should be grouped by unique value, which is the default. When disabled, results will be grouped by their random variable string. The difference between both result representation styles is discussed in Section 3.3.

order.by The result will be ordered on the specified attributes in the specified order. This has no effect for aggregation queries since those yield summary values.

order.by.attribute Orders on a specific attribute. Possible values: 'p', 'rv', and 'v' for probability, random variable string, or value respectively.

order.by.order The order of the order by clause, either 'ascending' or 'descending'.

output.result If set to false, the result will always be the empty sequence. Used in some benchmarks to skip XML generation of lots of result nodes.

round.precision The number of decimals to round on, only relevant if apply.rounding is enabled.

show.result.rvas Includes the random variable strings belonging to each result element in the result element.

top.k.limit	Sets the number of top-k result values to be generated for Min/Max aggregates.
write.timings	Writes the recorded timings to a timings.txt file.
write.query.to.file	Writes the transformed query to a p-query.txt file.

A.1.1 Query Execution

The plugin defines a `query` function, to be called from within a query issued to BaseX. The query function takes three required arguments; the name of the database and document to run the query on, as well as the query to be executed. An optional third parameter can be used to provide a map of configuration options. These configuration options control various parts of the plugin's behavior. A full list of available options and default values is listed in Section A.1. Before the plugin's query function can be called the namespace containing the plugin should be imported, assigned to an arbitrary prefix. In this example, we use the `pxml` prefix. Then, the `query` function can be called with the described parameters. In Figure A.1, a simple query is executed which selects the days with a temperature higher than 27 degrees Celsius and orders the result on descending probability. Furthermore, the probabilities are rounded to 4 decimal places, and execution times of the different steps are displayed on the standard output of BaseX's server instance.

```
1  import module namespace pxml = 'org.basex.modules.pxml.PXML';
2
3  let $cfg := {
4    'apply.rounding'     : 1,
5    'round.precision'    : 4,
6    'display.timings'    : 1,
7    'order.by'           : 1,
8    'order.by.attribute' : 'p',
9    'order.by.order'     : 'descending'
10 }
11
12 return pxml:query(
13   'pxml', 'forecasts.xml', '//forecast[temperature > 27]', $cfg
14 )
```

Figure A.1: Query Execution

Since the result of the `query` function is a regular XQuery expression – a `<results>` element with `<result>` child elements –, it is possible to manipulate this expression afterwards to suit a specific purpose. For example, if the user is interested in the top-k query results by descending probability this can be easily achieved by manipulating the query result in the following way, where we use the BaseX's pre-defined higher order function `top-k-by`.

```
1  import module namespace pxml = 'org.basex.modules.pxml.PXML';
2
3  (: A flat sequence with result elements :)
4  let $query_results := pxml:query(
5    'pxml', 'forecasts.xml', '//forecast[temperature > 27]'
6  )/result
7
8  let $k := 5 (: Retrieve the top 5 results :)
9
10 return hof:top-k-by($query_results, function($r) { $r/@prob }, $k)
```

Figure A.2: Query Result Manipulation

Note that we left out the configuration options, thus the result is not ordered on probability anymore like it was in Figure A.1. If that were the case, a simple `subsequence($query_result, 1, $k)` would have been sufficient here as well. Generally, being able to easily manipulate the probabilistic query result like this allows more fine-grained control over the query results, which cannot be achieved using configuration options alone.

A.1.2 Syntax Shorthands

A small number of shorthands has been defined that allows for a more compact way to express certain queries which otherwise required a lot of verbosity. Shorthands have been defined for both the axis names and for and/or predicates on a range of elements. Because we are using the built-in parser of BaseX which could not be altered using a plugin, the implementation of these shorthands is done through String replacements on the input query in a pre-processing step. This approach might cause issues in rare cases when queries include a shorthand in some other context, e.g. as an operand of some expression, not meant as a shorthand. To that end, the shorthands can be disabled entirely using the `disable.shorthands` configuration option.

Axes The axis names defined in XQuery can be addressed using the following much shorter variants. Due to matching on the axis name followed by a literal "::", we believe this replacement will generally not interfere with any other query expressions that can contain the same names – like traversing to an `<ancestor>` XML node inside a document.

XQuery Axis	Shorthand
ancestor	a
ancestor-or-self	aos
following	f
following-sibling	fs
preceding	p
preceding-sibling	ps

Table A.1: Axis shorthands

Range Predicates A query that applies a predicate involving some subset of elements of a sequence is difficult to express quickly in XQuery. For example, if we want to find all days with a temperature higher than all of the following 3 days using expressions supported by our implementation we have to write a query such as this:

```
//forecast[temperature > following-sibling::forecast[1]/temperature and
           temperature > following-sibling::forecast[2]/temperature and
           temperature > following-sibling::forecast[3]/temperature]
```

Instead, using both the axis shorthand for following-sibling introduced in the previous section and the range predicate shorthand for an And expression, this can be written much more concisely as below.

```
//forecast[temperature > fs::forecast{1..3}/temperature]
```

The curly brackets denote an `And` of the elements of the range, i.e. elements 1 through 3 of the following-sibling forecast elements in this example. Square brackets are used to create an `Or` expression from the elements. It has to be noted a similar expression is possible using XQuery's quantifiers, but those are not supported by our implementation. That is, we can write the same query in the following way.

```
//forecast[every $f in subsequence(following-sibling::forecast, 1, 3)
           satisfies temperature > $f/temperature]
```

To create an `Or` we would use `some` instead of `every` in the previous query. Even so, this is still a fairly verbose way of expressing the desired query compared to the shortened query we presented using the shorthands.