Data Wrangling - Project

Théo Delemazure

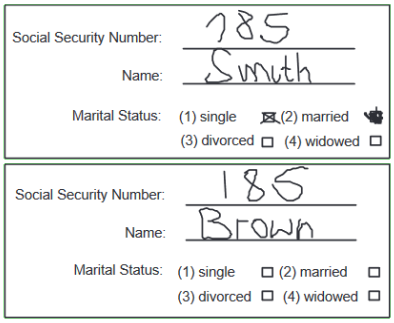
April 2020

Abstract

This short report details my project for the Data Wrangling course. This work is based on the paper " 10^{10^6} *Worlds and Beyond : Efficient Representation and Processing of Incomplete Information"* by **Lyublena Antova, Christoph Koch, and Dan Olteanu**, in which the authors present an efficient way to represent incomplete information into databases, using World-Set Decomposition (WSD). This model being a strong representation of incomplete database for every query language, the result of every query on WSD can be expressed as a WSD. I present here how I implemented algorithms to answer relational algebra queries on it. Finally, I test my algorithms on a dataset with real incompleteness instead of random noise.

Introduction

In a real database, it is not uncommon to deal with a lot of uncertainty and incoherent information. For instance, it is possible to have several possible values for one field, if the information is erroneous or may be interpreted in different ways. In this paper, we are only considering cases with a finite set of possible worlds. A toy example is given in Figure 1.



id	SSN	Name	Status
0	{185, 785}	Smith	{S, M}
1	{185, 186}	Brown	{S, M,D,W}

Figure 1: Approximate form filling and an Or-Set relation representing it

We saw in class that we can represent an uncertain probabilistic database with TID and BID, which are easy to compute, but that are not strong representation for the relational algebra. It is the same problem with *Or-set relations*, as the one shown in Figure 1. For instance, we have no way to enforce the SSN to be unique in all possible worlds.

We can also represent uncertain database as *c-tables*, which is a strong representation system. However, it is not scalable to big databases an even simple operations take exponential time [3].

In the first version of the paper "10^{10^6} *Worlds and Beyond : Efficient Representation and Processing of Incomplete Information"* [4] by *Lyublena Antova, Christoph Koch, and Dan Olteanu* from the University of Saarlandes in Germany, published in 2007, the

id	t1.SSN	t1.Name	t1.Status	t2.SSN	t2.Name	t2.Status
0	185	Smith	S	186	Brown	S
1	185	Smith	S	186	Brown	M
2	785	Smith	M	185	Brown	S
...	...	...	...	...	...	...
27	785	Smith	M	186	Brown	W

Table 1: A World-Set Relation (1-WSD) representing our running example.

authors details a new model to represent uncertain data, which is both **scalable** and **a strong representation**. They published later an extension of their work for probabilistic databases. In this report, I will explain what are World-Set Decompositions (WSD) and the algorithms I used to answer relational algebra queries on these decompositions.

The authors showed the efficiency of their model on a dataset in which they added noise. For my project, I implemented my own version of these algorithms, tried to optimize them, and most importantly, test them on a real world dataset **with real uncertainty**. Moreover, I wanted to see how some factors can change the running time of algorithms, and which part of the algorithm is actually the most costly.

1 World-Set Decompositions

First of all, let's explain the concept of World-Set Decomposition (WSD). I will explain it for a database with only one relation R, but it is generalizable to any number of relation. Let's denote by $\mathcal{W}$ the set of possible worlds and $|R|_{max}$ be the maximum number of tuples in a possible world:

$$|R|_{max} = \max\{R^W | W \in \mathcal{W}\}$$

The first idea is to create a table in which **each row is a possible world**. Formally, we have a relation with one attribute for each tuple between 1 and $|R|_{max}$ and each attribute of R. This is called a **World-Set Relation** and has the following schema:

$$\{t_i.A_j \mid i \in [1, |R|_{max}],\ A_j \in schema(R)\}$$

In which each possible world $W = (t_1, ..., t_n)$ with $n \leq |R|_{max}$ can be represented as the tuple

$$t^W = t_1 \circ t_2 \circ ... \circ t_n \circ \underbrace{(\bot, \cdots, \bot)}_{arity(R)\times(|R|_{max}-n)}$$

For instance, the World-Set Relation of the example on Figure 1 is represented Table 1. However, with a big amount of data, the size of the table can be exponential, and there is a lot of **Multi Valued Dependencies**. That's the reason why the authors introduce World-Set Decomposition.

Definition 1. *Let $\mathcal{W}$ be a world-set and $R_{\mathcal{W}}$ a world-set relation representing $\mathcal{W}$. Then a world-set m-decomposition (m-WSD) of $\mathcal{W}$ is a product m-decomposition of $R_{\mathcal{W}}$.*

For instance, our running example can be represented as the WSD on Table 2. More generally, it is obvious that any finite world-set can be represented as a 1-WSD, and therefore **WSD is a strong representation system for any query language**.

Finally, some components of the WSD contain only one row, which make this information certain. Consequently, we can reduce the number of components by keeping what is called a **Template relation** with all certain information (see for instance Table 3).

t_1.SSN	t_2.SSN
185	186
785	186
785	185

$\times$

t_1.Name
Smith

$\times$

t_1.St
S
M

$\times$

t_2.Name
Brown

$\times$

t_2.St
S
M
D
W

Table 2: A WSD for our running example with the maximum number of components.

id	SSN	Name	St
t_1	?	Smith	?
t_2	?	Brown	?

t_1.SSN	t_2.SSN
185	186
785	186
785	185

$\times$

t_1.St
S
M

$\times$

t_2.St
S
M
D
W

Table 3: A WSDT for our running example.

2 How to implement WSD in practice?

The number of components, and the arity of these components can be arbitrary big. In practice, most DBMS do not allow that. That's why the authors suggest a representation of WSDT with **only 4 relations** of a fixed arity. I modified them a bit in my implementation, but the idea is the same. Let's denote $(C_1, ..., C_m)$ the components of the WSDT, then we have the following tables:

- The **template relation** R^0 is not modified.
- F contains the **components description**. Indeed, it associates each tuple and attribute to one component: $F := \{(t, A, i) \mid t.A \in C_i\}$.
- C contains all **the possible worlds** for every component. We have $(t, A, i, v) \in C$ if attribute $t.A$ has value v in the world number i of the component C_k.
- The authors also introduce a relation W such that $(i, j) \in W$ if there is a world number j in the component C_i. However, this corresponds to the view described by the query below, so I did not keep it for my experiments:

```
SELECT DISTINCT cid,lwid FROM F JOIN C ON F.tid = C.tid AND
                F.attribute = C.attribute
```

  Note that this fourth relation became interesting when **we introduce probabilities** in the WSD.

This is called a **Uniform World Set Decomposition**. The UWSD representing our running example is described in Table 4.

Finally, when creating these tables, it is essential to specify **the primary key** of each table ($(tid, attribute)$ for F and $(tid, attribute, lwid)$ for C) because it enables good indexing and consequently **faster computation**. Let's see now how to answer queries on WSD.

3 Answering queries on WSD

For each relational algebra operation, the authors describe the algorithm to compute the result on WSD. However, most of the time, their algorithm need to be optimized or more

R^0	SSN	Name	St
t_1	?	Smith	?
t_2	?	Brown	?

F	tid	attribute	comp.
	t_1	SSN	1
	t_2	SSN	1
	t_1	Status	2
	t_2	Status	3

C	tid	attr	lwid	val
	t_1	SSN	1	185
	t_2	SSN	1	186
	t_1	SSN	2	785
	t_2	SSN	2	186
	t_1	SSN	3	785
	t_2	SSN	3	185
	..	...	...	...
	t_2	St.	3	D
	t_2	St.	4	W

Table 4: A UWSDT for our running example

detailed **to be used in practice** with UWSD. For the implementation, I used the `MySQL` DBMS because there is a good PYTHON library for querying the database. Moreover, every algorithm I will describe now is implemented as a **SQL query plan** (i.e. there is no complex computation made in python, everything is directly computed through SQL queries). You can check how I implemented them in practice on the GitHub repository [1].

Please not that the algorithms are not done in-place. For each operation, we **output new tables** R^0_{new}, C_{new} and F_{new}. Moreover, as soon as the new relations are created, we add primary key and indexing constraints on them because a good indexing is **the key for a faster computation**. Note also that I use auxiliary tables in a lot of queries because materialized views do not exist in `MySQL` and it enables to avoid computing the same thing several times.

Renaming Let's start with an easy one : $\delta_{name_1 \to name_2}$. Here we simply need to rename attribute $name_1$ as $name_2$ in R^0 and replace every occurrence of $name_1$ by $name_2$ in C and F.

Selection $A\theta c$ For the selection operator, we first consider the case in which only one attribute A is involved, and is compared to a constant c with the operator $\theta \in \{=, \neq, <, \leq, >, \geq\}$. This is done in two steps :

- Add into R^0_{new} every tuple from R^0 which validate the selection condition in at least one world. If the value is `NULL` in R^0, we must look into the table C.
- Copy rows from C and F into C_{new} and F_{new} which corresponds to a tuple in R^0 verifying the condition.
- In the table C_{new}, if the *attribute* is A and the *value* does not verify the condition, replace it by `NULL`:

`REPLACE INTO` C_{new} `SELECT tid,attribute,lwid,NULL as value FROM` C_{new} `WHERE attribute =` A `and NOT(value` θ c`)`

Once this is done, it is very important to **propagate** `NULL`**s**. Indeed, if we add a `NULL` in one possible world, that means the corresponding tuple cannot be selected in this world. Consequently, we need to propagate the `NULL` to all other attribute of the same tuple which are in the same component, in the case we do a projection later and forget the attribute A. This corresponds to the following query:

```
REPLACE INTO C SELECT C.tid,C.attribut,C.lwid,NULL FROM C JOIN C' JOIN
    F JOIN F' ON C.tid = C'.tid AND C.tid = F.tid AND C.attribut =
    F.attribut and C'.tid = F'.tid AND C'.attribut = F'.attribut AND
F.component = F'.component AND C.lwid = C'.lwid WHERE C'.value = NULL;
```

Selection $A\theta B$ This one is trickier. Indeed, we are looking at **two attributes** A and B at the same time. Consequently, we will probably **need to merge** the components of A and B referring to the same tuple of R^0. Let's see in more details how I implemented this algorithm:

1. First of all, get the id of tuples which verify the condition $A\theta B$ in at least one world. For that, we take the union of 3 queries :
   - One for tuples with both A and B not `NULL` in R^0: Just check the condition.
   - One for tuples with one attribute not `NULL` and the other `NULL` in R^0: Check if there exist a world verifying the condition, as in $A\theta c$.
   - One for tuples with both A and B `NULL` in R^0: Check if there is one world in which the condition is verified. Note that we need to look at the `lwid` only if the two attributes are already in the same component.
2. We create R^0_{new}, C_{new} and F_{new} by keeping only those tuples.
3. Now, we need to **merge components** of tuples in the third case of step 1. Indeed, if both attributes are uncertain and there is at least one world in which the condition is verified, then we need to merge the two components to **delete every world**, in which the condition is not verified. Since the merging algorithm is used in several queries, it is explained later.
4. Finally, replace by `NULL` the values in C which do not verify the condition, and we propagate the `NULL`s as in the selection $A\theta c$.

Projection This operation is also harder than it seems. Indeed, look at the example in Table 5. If we directly project over the attribute A, then we lose the information that there can be only one tuple in each possible world. For this reason, the authors suggest to merge all components referring to the same tuple of R^0. However, we do not need to merge the components of **every** tuples. In particular, if a tuple never occurs in a component **with another tuple**, then no merging is necessary. However, if there is a `NULL` in at least one world of at least one attribute not in the projection, we need to add this possibility to every component of this tuple.

t_1.A
0

$\times$

t_2.A
1

$\times$

t_1.B	t_2.B
$\perp$	1
2	$\perp$

t_1.A	t_2.A
$\perp$	1
0	$\perp$

Table 5: A WSD (left) and its projection on the attribute A (right).

component	count
0	256
2	0
2	1
5	0
5	2

Table 6: Example of the output of the aggregation operation on a WSD. Two components have uncertainty on the number of tuple they contains.

The problem of Merging The merging algorithm is not detailed in the original paper. However, it is really hard to merge components with only SQL queries. When we merge two components, we need to reindex the `lwid` of possible worlds. We assume completeness of `lwid`, i.e. if there is N possibility for a component, these possibilities have id $0, \cdots, N-1$. Then, if we merge components c_1 and c_2 with respectively N_1 and N_2 possible worlds (that we need to compute), we perform the following steps:

1. Multiply every `lwid` of c_1 by N_2.
2. Copy every world of c_1 with `lwid` j into N_2 worlds of `lwid` $j, j+1, \cdots, j+N_2-1$ into an auxiliary table.
3. Copy every world of c_2 with `lwid` j into N_1 worlds of `lwid` $j, j+N_2, \cdots, j+(N_1-1)\times N_2$ into C_{new} and insert tuples from the auxiliary table of Step 2.

My first idea was to merge them one by one, but with a big amount of components to merge, it would take forever. In the query 3 of the experiments section, the time would be 16 minutes instead of 50 seconds if we merge them one by one.

We can try to merge them all in the same time, and it works if no component need to be merged to at least 2 other components. Unfortunately, it cannot work if 3 components need to be merged together, because if we do it in one step, we need to create $N_1 \times N_2 \times N_3$ possible worlds. That's why my algorithm, which use a recursive `with` query, works in **several steps**.

At each step, it computes a list of components which need to be merged such that each component appear only once in the list. However, when I tried it on a `join` query, the DBMS return a **memory error** after only 6 steps. But when we look closer, it is one component contains already a million possible worlds. I tried to avoid this issue by applying the selection at the end of each step and removing tuples with only `null`s. It helped a little, but I still got the memory error. I concluded that the results **cannot be represented as a WSD in practice** unless you have a lot of memory.

Cross Product The cross product is not too complicated, but it causes an explosion of the size of C. Indeed, to make the cross product of two WSD (R_1^0, C_1, F_1) and (R_2^0, C_2, F_2), each tuple of R_1^0 need to be duplicated into $|R_2^0|$ tuples and reciprocally each tuple of R_2^0 need to be duplicated into $|R_1^0|$ tuples. However, the most complex task is maybe to reindex every tuple and components. Indeed, if a component has index 42 in F_1 and another component has index 42 in F_2, this two components **should not have the same index** in F_{new}.

1. We create a table containing all pairs of id from R_1^0 and R_2^0 and we create a new column with properties `PRIMARY KEY NOT NULL AUTO_INCREMENT` :

```
CREATE TABLE new_id AS SELECT R1^0.tid id_1,R2^0.tid id_2 FROM R1^0,R1^0;
ALTER TABLE new_id ADD COLUMN id PRIMARY KEY NOT NULL AUTO_INCREMENT;
```

2. We do the same thing for components in F_1 and F_2 to obtain a table `new_components (new_id,old_id,old_R)` with `old_R` $\in \{1, 2\}$ the WSD from which the component come from.
3. We do the cross product of R_1^0 and R_2^0 to obtain $R_{new}^0 = R_1^0 \times R_2^0$ and replace the ids.
4. We add $|R_2^0|$ entries of each row of C_1 into C_{new}, with the corresponding `tid` and reciprocally:

`INSERT INTO` C_{new} `SELECT new_id.id,`C_1`.attribut,`C_1`.lwid,`C_1`.value FROM` C_1 `JOIN new_id ON` C_1`.tid = new_id.id_1`

5. We do the same operations for F_{new}, but this time we also need to specify the new component id.

6. Once everything is done, drop tables `new_id` and `new_components`.

Union The union is very similar: We also need to create new tuple ids and new component ids, but this time **we do not need to duplicate each row** of C_1 and C_2. Consequently, we just copy the old tables into new tables and change the ids. Please note that it corresponds to the `union all` in `MySQL` and not to the `union`, which only return distinct elements.

Difference The author do not really explain their algorithm for difference, and I suspect it to be much harder than it seems. Moreover, none of their queries in the experiments section **uses the difference** operator. Consequently, I did not implement it, in order to concentrate my efforts on other personal contributions.

Normalization An important part of my code is dedicated to normalization functions. Indeed, I implemented some functions to **reduce the size of C my normalizing the WSD**:

- Due to selections, many components are filled with worlds containing only `null` values. This first function only keeps one of them and delete others.

- Since we just delete some worlds, we need to reindex the `lwid` for these worlds (because of the assumption used in the merging algorithm).

- Finally, if an attribute of a tuple has the same value in every worlds, we can replace the `null` in R^0 by this value.

Please note that to obtain an optimal WSD, we need **a lot more work**, by removing duplicates or decomposing components which can be, for instance.

Functions The first operation I added is the possibility to apply functions with one argument to attributes. For instance, if we want the height in centimeter, we do $height \times 100$, if we want the first 3 letters of the borough, we can use the SUBSTR `MySQL` function, etc. This is actually very simple to compute since we just need to apply the function to the non-`NULL` fields of R^0 and the fields of the *value* column of C.

Aggregation Finally, I added a way to use the aggregation operator `count`, `sum` and `average`. The results of such queries cannot be a WSD nor a single value. Here is how I implemented it:

1. Merge every component referring to the same tuple of R^0.

2. Do the sub-aggregation inside each component and report each possibility in a table T. Add to this table the result of the aggregation operator on every tuple of R^0 not appearing in any cluster of F and assign it to component 0.

J	Young	22%
JA	Young Adult	25%
A	Adult	48%
M	Mature	5%

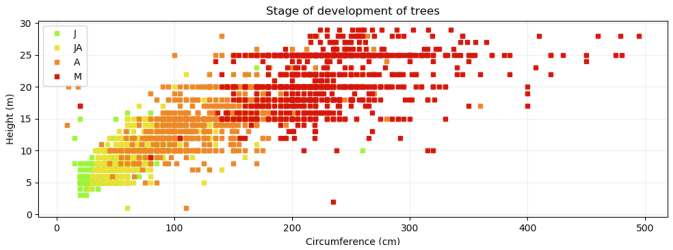


Figure 2: Distribution of stage of development (left) and distribution of height and circumference of trees with different stage of development (right).

3. Aggregate again the results of components with only one result possible and the results of the aggregation on R^0. For instance with `count`, if some component contain exactly two tuples in every possible world, add 2 to the results of the component 0 created in Step 2.

An example of the output of the `count` is shown Table 6.

4 Dataset choice

For their experiments, the authors of the paper used the publicly available 5% extract from 1990 US census IPUMS, consisting in 50 multiple choice questions about the respondent's profile (marital status, citizenship, military status, etc.). It contains **12.5 Millions entries**, but there is no natural uncertainty. They added uncertainty in this database by randomly replacing some values by Or-Set (as in Table **??**) of size at most 8. There most incomplete database contains 0.1% of noise, giving 62449 components, and therefore at least 2^{62449} possible worlds.

I wanted to try algorithms in a context of **real uncertainty**. I analysed the biggest datasets from the *Open Data website of the city of Paris* [2]. The most appropriate one was the *Tree dataset*, containing more than 200 000 entries with interesting attributes such as **specie, height, circumference, borough, stage of development or importance**. I quickly noticed there was **a lot of errors** on this dataset, in which values were probably manually entered.

For instance, this dataset contains 135 trees taller than 100 meters, even if the highest known tree in Paris is 45 m high (In the dataset, there is a *Tilleul* which is 881km high !). Similarly, there are trees with a circumference bigger than 1km, even if the biggest tree of Paris has a circumference of 7 meters. Moreover, 15% of trees are 0 meters high and 11% of trees have a circumference of 0 centimeters. Finally, for 28% of the trees, the stage of development (see Table 2) is not specified. In some cases, we can give a sense to erroneous entries. For example, an height of 1216 m probably corresponds to an height of 1216 **cm**. A circumference of 160170 cm probably corresponds to a circumference **between** 160 and 170 centimeters. With empirical observations like that, I used the rules below to add uncertainty to the database.

This database contains less tuples, but a highest level of incompleteness. In the end, we obtain more than 10^{10^4} possible worlds (the number of possible worlds added by each rule is **W** in the Table below), and some components contain several attributes due to the third rule.

	Attribute	Rule	W
1	height	If $= 0$, select 3 different heights at random using a gaussian distribution.	3^{23103}
2	height	If > 45, add every combination of 1 or 2 consecutive digit giving a value ≤ 45 and if < 100, try to replace the first digit by 1.	$\sim 2^{645}$
3	height important	If $\in [30, 45]$, and important = False, add one world with the same height and important = True and a add worlds by replacing the first digit with 0, 1 or 2.	4^{116}
4	circumference	If $= 0$, select 3 different circumference at random using a gaussian distribution.	3^{31907}
5	circumference	If > 695, add every combination of 2 or 3. consecutive digit giving a value ≤ 695.	$\sim 2^{127}$
6	stage of dev.	If not specified, add a world for each possibility.	4^{57518}

5 Adding constraints to the dataset

To create components with several attributes, the authors added logical constraints to the database, like *"People who participated in the second world war must have completed their military service"*, also written WWII $= 1 \Rightarrow$ MILITARY $\neq 4$. To enforce these rules, the authors use what they call **an adaptation of the Chase**, which is actually an adaptation of the Selection $A\theta B$ algorithm, in which we delete bad tuples instead of replacing them by `NULL`.

I also added constraints to my tree database, all of use attributes with incompleteness. They enable to remove inconsistent worlds added during the creation of the dataset (Section 4). The rules are described below and are based on empirical observation like on Figure 2, which shows evolution of the height and the circumference with the stage of development of trees. For each one, I specify in the table below the number of components merged and the time taken by the chase algorithm.

	Rule			#Merging	Time (s)
1	$development = "JA"$	$\Rightarrow$	$height \leq 25$	147	31.5
2	$development = "J"$	$\Rightarrow$	$height \leq 20$	1718	93.8
3	$development = "JA"$	$\Rightarrow$	$circumference \leq 250$	255	36.4
4	$development = "J"$	$\Rightarrow$	$circumference \leq 200$	2036	102.7

After adding the constraints, I obtain more than $100,000$ components and $4,000$ of them include two or more attributes. For comparison, this gives a higher ratio $\#components/\#tuples$ than in any artificial databases of the paper.

6 Experimental results

Like in the original paper, I selected a few queries to see if my implementation is working, and how fast it is. The queries are such that every algorithm described in Section 3 is tested, and preferably in different contexts. Table 7 describe all the queries and Table 8 details all the results. If you are interested of the `MySQL` query plan for each query, the logs are on the GitHub repository. The first two queries only use **Selection** $A\theta c$ and **Projection** and no merging is needed because no component is referring to several tuples of R^0. The time is still bigger than for classic database because there is a lot of

q_1	*What are the species, heights, circumferences, and public states of* ***mature*** *trees from the* ***5th arrondissement of Paris*** *?*	Selection $A\theta c$ Projection
q_2	*What are the species, heights ,circumferences, and stage of development of trees in* ***woods*** *around Paris* ***larger than 250 cm*** *of circumference ? We want the height in* ***centimeters****.*	Selection $A\theta c$ Projection Math function
q_3	*What are the species, boroughs, heights, ? circumferences, and importance of trees verifying* ***height (cm)*** $<$ ***circumference (cm)***$\times 5$	Selection $A\theta B$ Projection Math function
q_4	*What is the* ***average height*** *of trees in* q_2 *?*	Aggregation
q_5	*What are species and circumferences of trees in* $q_1 \cup q_2$ *?*	Union
q_6	*Take the* ***join*** *of* q_1 *and* q_2 *on pairs of trees with different species but the same circumference. What are the species of the two trees and their circumference (+ some other attributes) ?*	Cross product Rename Selection $A\theta B$ Projection

Table 7: The different queries and the operator they are using

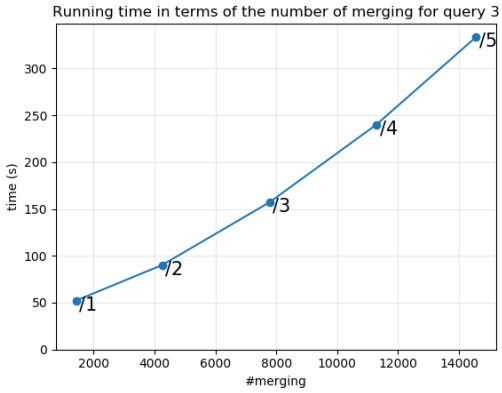


Figure 3: Number of components to merge and running time of the algorithm for different multiplication factors for the circumference in the query 3.

writing: create new databases, add constraints, delete tuples, etc. Moreover, we note that $|R^0| > |R|$, it is simply because we add uncertainty and then, new possibilities!

The third query uses the **Selection** $A\theta B$ and perform 1425 merges. However, it is an easy case of merging, and it is done in only 5 minutes. If we change the multiplication factor, we obtain a different number of components to merge. Figure 3 show that in the case of the query 3, the running time is linear in the number of components to merge.

The query 5 performs a union between results from the first two queries. The difference of time taken between classic DB and WSD is really small for this query. It is probably due to the fact that we keep only one uncertain attribute, and therefore the maximum number of attributes of one component is 1.

Finally, the last query takes a lot of time because we need to merge at least 4 components together, due to the cross product in the `JOIN` condition. Indeed, the algorithm took 3 merging steps and the biggest component contains 68 attributes in the end. Moreover, if we replace *250 cm* by *200 cm*, we obtain a dozen more components to merge, but it is too much for `MySQL` and I get a memory error.

query	DB	WSD	Norm.	$\|C\|$	#c	#c ≥ 2	max $\|c\|$	$\|R\|$	$\|R^0\|$
R^0				465473	109016	4096	3	205199	205199
q_1	0.6 s	6 s	1.3 s	1171	288	1	2	77	786
q_2	0.9 s	9.7 s	1.2 s	431	75	33	3	334	360
q_3	0.6 s	339 s	25.2 s	116199	14951	8	2	9141	19908
q_4	0.8 s	3.4 s							
q_5	1.5 s	4.6s		1876	188	0	1	411	1146
q_6	1.7 s	63.8 s	1.5 s	828	5	3	68	133	301

Table 8: For each query, this table contains the running time to execute queries on classic DB, WSD, time of normalization, size of resulting C, number of components, number of components with at least two attributes, maximum size of a component, and size of the resulting tables R (classic DB) and R^0 (WSD).

Conclusion

We saw that uncertain databases can be represented as World-Set Decomposition, and that it is a strong representation system for any query language. There are efficient algorithms for querying WSD, however, the computation time explodes when a component contains more than a million possible worlds. As a personal contribution, I implemented almost every algorithm, and proposed new algorithms for aggregation and functions on attributes. I tested the algorithms on a dataset in which I added natural and coherent incompleteness using various rules and constraints. All the code, data and query log can be found in the github repository [1].

The next step for this project would have been to add probabilities to WSD to obtain PWSD presented by the authors in a more recent version of the paper (that I discovered too late).

References

[1] Github repository for this project : https://github.com/theodlmz/datawranglingproject.

[2] https://opendata.paris.fr.

[3] Serge Abiteboul, Paris Kanellakis, and Gösta Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78(1):159 – 187, 1991.

[4] Lyublena Antova, Christoph Koch, and Dan Olteanu. 10106 worlds and beyond: Efficient representation and processing of incomplete information. *The VLDB Journal*, 18:606–615, 01 2007.