# A Formal Context for Acyclic Join Dependencies

Jaume Baixeries

Departament de Ciències de la Computació. Universitat Politècnica de Catalunya.
08024 Barcelona. Catalonia. jbaixer@cs.upc.edu

**Abstract.** Acyclic Join Dependencies (AJD) play a crucial role in database design and normalization. In this paper, we use Formal Concept Analysis (FCA) to characterize a set of AJDs that hold in a given dataset. This present work simplifies and generalizes the characterization of Multivalued Dependencies with FCA.

## 1 Introduction and Motivation

In database theory, a **dependency** expresses a relationship between sets of attributes in a dataset. There are numerous types of dependencies: conditional dependencies, sequential dependencies, order dependencies, to name just a few of them (see [13] for a more detailed survey). Dependencies may have different semantics, and may express relationships of different nature: equality, similarity, order, distance, etc.

However, not all of those dependencies have been equally popular. The most common dependencies in the relational database model are **functional dependencies** (FD's), which have been widely studied in the field of database theory ([16]). The reason of this success may be twofold: on the one hand, their semantics is very simple and intuitive, on the other hand, they have been proven to be very versatile, since they can be used for database design, database validation, and, also, data cleaning [9]. They play a key role to explain the normalization of a database scheme in the relational database model.

Another type of dependencies that have been relevant in database theory are **multivalued dependencies** (MVDs) ([16]). These dependencies are a generalization of functional dependencies, and their semantics is capable of expressing how a table can be split into two different tables such that their join is exactly the original table. This procedure is of key importance for database normalization and design.

**Acyclic Join Dependencies** (AJDs) ([17]) are a generalization of multivalued dependencies. AJDs are of critical importance in the decomposition method [10], which is a method for designing database schemes. An AJD specifies a lossless decomposition of a dataset, which is the decomposition into different (smaller) datasets such that their composition restores the original dataset. The interest of this decomposition is that it allows the dataset to be in the so-called 4th-normal form (4NF), which prevents redundancy and update errors. We discuss AJDs in more detail in Section 2.1.

Formal Concept Analysis (FCA) is a simple and elegant lattice-oriented mathematical framework that is strongly connected to lattice theory. The uses of FCA are manyfold, as, for instance, knowledge discovery and machine learning [14], among many others. We discuss FCA in more detail in Section 2.2.

Formal Concept Analysis has been widely used to characterize and compute different types of dependencies. We present the most relevant work on this subject in Section 2.3.

In this paper, we deal with the characterization of a set of AJDs that hold in a given dataset, using the formalism of FCA. The goals of this paper are twofold: on the one hand, extend previous work that dealt with the characterization of functional and multivalued dependencies. On the other hand, propose FCA as a tool to compute sets of AJDs that hold in a dataset, compute the closure of a set of AJDs and explore the possibility of computing minimal bases for this kind of dependencies.

This paper starts with the Notation section, where we explain the basics of AJDs, FCA and also, previous work linking FCA and data dependencies. In the Results section, we present a new formal context for AJDs. We also present an example in a separate section to illustrate the results. Finally, we present the conclusions and future work.

## 2 Notation

The primary objects with which we deal in this paper are a set of **attributes** and a **dataset**. A dataset $T$ is a set of tuples: $T = \{t_1, \dots, t_N\}$ (we use indistinctively dataset and set of tuples as equivalent terms) and a set of attributes $\mathcal{U} = \{a, b, \dots\}$ (commonly known as column names). Each tuple has a value associated to each attribute. We use non capital letters for single elements of the set of attributes, starting with $a, b, c, \dots$, and capital letters $X, Y, Z, \dots$ for subsets of $\mathcal{U}$. We drop the union operator and use juxtaposition to indicate set union. For instance, instead of $X \cup Y$ we write $XY$. If the context allows, we drop the set notation, and write $abc$ instead of $\{a, b, c\}$.

| id | a | b | c | d |
|---|---|---|---|---|
| $t_1$ | 1 | 1 | 1 | 1 |
| $t_2$ | 1 | 2 | 1 | 1 |
| $t_3$ | 1 | 1 | 2 | 2 |
| $t_4$ | 1 | 2 | 2 | 2 |

**Fig. 1.** Example dataset.

For instance, Figure 1 is an example of a dataset $T = \{t_1, t_2, t_3, t_4\}$, with its set of attributes $\mathcal{U} = \{a, b, c, d\}$. We use the notation $t(X)$ to indicate the **restriction** of a tuple $t$ to the set of attributes $X$. In this same example, $t_2(\langle b, c \rangle) = \langle 2, 1 \rangle$. It is necessary to note that when the values of the tuple

are given, some order must be implicit, because a value that appears in a tuple is always related to an attribute. We also use juxtaposition for the composition of tuples. For instance, $t(\langle a, b \rangle)t(\langle c, d \rangle)$ is the tuple $t(\langle a, b, c, d \rangle)$. We also have that $t(\langle a, b \rangle)t(\langle c, d \rangle)$ is equivalent to $t(\langle c, d \rangle)t(\langle a, b \rangle)$, assuming that we have a total order on the attributes set. We use the notation $\Pi_X(S)$, where $X \subseteq \mathcal{U}$ and $S \subseteq T$ as the set $\Pi_X(S) = \{ t(X) \mid t \in S \}$, this is, the set of restrictions of all tuples in $S$ to the set of attributes $X$. In this same example, $\Pi_{\langle a,d \rangle}(T) = \{ \langle 1, 1 \rangle, \langle 1, 2 \rangle \}$.

Given a set $S$, we define also the set of its splits and pairs:

**Definition 1.** *The set* $\text{Split}(S)$ *is the set of partitions of* $S$ *of size 2. For instance, if* $S = \{ a, b, c, d \}$*, then,* $\text{Split}(\mathcal{U}) = \{ [a \mid bcd], [b \mid acd], [c \mid abd], [d \mid abc], [ab \mid bc], [ac \mid bd], [ad \mid cd] \}$.

**Definition 2.** *The set* $\text{Pair}(S)$ *is the set of pairs of elements of* $S$*, modulo reflexivity and commutativity. For instance, if* $S = \{ a, b, c, d \}$*, then,* $\text{Pair}(S) = \{ (a, b), (a, c), (a, d), (b, c), (b, d), (c, d) \}$.

Finally, we define the **join** operator $\bowtie$ on the datasets $R$ and $S$ as follows: $R \bowtie S = \{ r \cup s \mid r \in R \ \wedge \ s \in S \ \wedge \ r(X) = s(X) \}$, where $X$ is the set of attributes that are common to both $R$ and $S$. If they have no common attributes, then, this operation becomes a cartesian product.

### 2.1 Acyclic Join Dependencies

An acyclic join dependency is a special case of a join dependency. We provide first the definition of a join dependency, and then, we describe the restriction that applies to acyclic join dependencies.

**Definition 3.** *Let* $T$ *be a set of tuples and let* $\mathcal{U}$ *be its attribute set. A* ***join dependency*** $R = [R_1, \ldots, R_N]$ *is a set of sets of attributes such that:*

1. $R_i \subseteq \mathcal{U}, \forall i : 1 \leq i \leq N$.
2. $\mathcal{U} = \bigcup_{1 \leq i \leq N} R_i$*, this is, all attributes are present in* $R$.

*A join dependency* $R$ *holds in* $T$ *if and only if:*

$$T = \Pi_{R_1}(T) \bowtie \ldots \bowtie \Pi_{R_N}(T)$$

The intuition behind a join dependency is that the set of tuples $T$ can be decomposed into different smaller (with less attributes and, maybe, tuples) sets of tuples, such that their composition according to $R$ is lossless, this is, no information is lost [7]. **Acyclic join dependencies** are join dependencies that hold in a set of tuples according to the condition in Definition 3. However, they have some syntactical restrictions that make them more tractable than join dependencies, both in terms of axiomatization and computational complexity [12]. A deep discussion on the differences between both join and acyclic join

dependencies is far beyond the scope of this paper, and we will provide only the basic definition and properties of AJDs that are relevant to this paper.

The notion of a join dependency is closely related to that of a **hypergraph** [8]. Let $S$ be a set of vertices, a hypergraph extends the notion of a graph in the sense that an edge in a hypergraph is not limited to two vertices, as in a graph, but to any number of vertices. In a join dependency $R = [R_1, \ldots, R_N]$, the set of vertices would be $\mathcal{U}$, and $R$ would be the set of edges. Also in hypergraphs, there is the notion of **acyclicity**, but this notion is not as intuitive as in the case of graphs. In fact, there are different definitions of acyclicity, some more restrictive than others (in [6] some are discussed). In this case, we use the definition of acyclicity that appears in [6]. But, again, this definition can be enunciated in as many as 12 different equivalent ways, but we just use one of them which is necessary to understand the results in this paper. With this definition, we proceed to define acyclic join dependencies.

**Definition 4 ([6]).** *Let $R = [R_1, \ldots, R_N]$ be a join dependency. A **join tree** $JT$ for $R$ is a tree such that the set of nodes is the same as $R$ and:*

1. *Each edge $(R_i, R_j)$ is labeled with $R_i \cap R_j$.*
2. *For any pair $R_i, R_j \in R$, where $i \neq j$, we take the only path $P = \{R_i, R_{i+1} \ldots R_j\}$ between $R_i$ and $R_j$. For all edges $(R_{i+k}, R_{i+k+1})$ we have that: $R_i \cap R_j \subseteq R_{i+k} \cap R_{i+k+1})$.*

*A join dependency $R$ is an **acyclic join dependency** (AJD) if and only if it has a join tree. If an acyclic join dependency has only two components, then, it is called a **multivalued dependency**.*

*Example 1.* For instance, if we have that $\mathcal{U} = \{ a, b, c, d, e, f, g \}$ and the AJD $R = [abc, ace, abfg, abd]$, a join tree for $R$ is in Figure 2. We can see, for instance, that in the path from node $ace$ to node $abd$, the intersection $ace \cap abd = a$ appears in all the edges of the path.

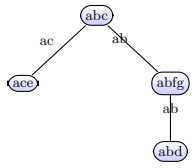


**Fig. 2.** Join tree for $R$.

Therefore, an acyclic join dependency is a join dependency that has a join tree or, equivalently, is an acyclic hypergraph. For a given AJD, there can be more than one possible join trees, since, for instance, the choice of a root is completly arbitrary (although swiching the root is not the only possible way to have different join trees).

One important and useful property of AJDs is that a single AJD is equivalent to a set of multivalued dependencies. We remind the reader that a MVD is a special case of AJD with cardinality 2. Definition 5 is a technicality that helps to understand how to compute the set of MVDs that are equivalent to an AJD in Proposition 2.

**Definition 5.** *Let $JT = \langle R, E \rangle$ be the join tree of an AJD $R$. We define the function* Removal *that returns a partition of the set of attributes $\mathcal{U}$ into three classes according to an edge of $JT$.*

*Let $C_1 = \langle V_1, E_1 \rangle, C_2 = \langle V_2, E_2 \rangle$ be the two connected components that appear after the removal of an edge $(R_i, R_j) \in E$ in $JT$. Let $C_i = \langle V_i, E_i \rangle$, then* $\text{attrib}(C_i)$ *returns the set of attributes that appear in that connected component, this is:* $\text{attrib}(C_i) = \bigcup_{X \in V_i} X$. *The function* $\text{Removal}(\langle R, E \rangle, (R_i, R_j))$ *returns the triple:*

$$\text{Removal}(\langle R, E \rangle, (R_i, R_j)) := \langle R_i \cap R_j,\ \ \text{attrib}(C_1) \setminus (R_i \cap R_j),\ \ \text{attrib}(C_2) \setminus (R_i \cap R_j) \rangle$$

**Proposition 1.** *Let $R$ be an AJD and let $JT = (\mathcal{U}, R)$ be its join tree. The function* Removal *returns a partition of the set $\mathcal{U}$.*

*Proof.* Since $R$ is an AJD, then, $R$ contains all the attributes in $\mathcal{U}$. Lets assume that we are removing the edge $(R_i, R_j)$ After splitting $JT$ we have two connected components: $C_1 = \langle V_1, E_1 \rangle, C_2 = \langle V_2, E_2 \rangle$. Therefore, all the attributes are either in $V_1$ or in $V_2$. This implies that all attributes will appear in one of the classes returned by Removal, since either this attribute will be in $V_1$, or in $V_2$ or in $R_i \cap R_j$. Now we need to prove that an attribute will not appear in more than one class. Because $R_i \cap R_j$ is substracted in $V_1$ and $V_2$, if there is a repeated attribute, it will only be in $V_1$ and $V_2$. Let us suppose, by way of contradiction, that there is an attribute that is in $\text{attrib}(C_1) \setminus (R_i \cap R_j)$ and $\text{attrib}(C_2) \setminus (R_i \cap R_j)$, this is, in $R_k \in E_1$ and $R_j \in E_2$. Since $JT$ is a join tree, it means that this attribute will appear in all the edges in the path from $R_k$ to $R_j$, also in $R_i \cap R_j$ because the edge $(R_i, R_j)$ is necessarily in that path, which yields a contradiction.

*Example 2.* We return to Example 1. We have the AJD $R = [ace, abc, abfg, abd]$ and the corresponding join tree $JT = \langle AJD, (ace, abc), (abc, abfg), (abfg, abd) \rangle$. We have that:

$$\text{Removal}(JT, (abc, abfg)) = \langle ab, ce, dfg \rangle$$

$$\text{Removal}(JT, (ace, abc)) = \langle ac, e, bdfg \rangle$$

We now define the set of MVDs that are equivalent to a single AJD.

**Proposition 2.** *Let $R$ be an AJD, and let $JT = \langle R, E \rangle$ be its join tree. $R$ holds in a table $T$ if and only if $\forall (R_i, R_j) \in E : [XY, XZ]$ holds in $T$, where $\langle X, Y, Z \rangle := \text{Removal}(JT, (R_i, R_j))$. The proof is in Theorem 7.1 in [6].*

### 2.2 Formal Concept Analysis

In this brief account of Formal Concept Analysis (FCA), we use standard definitions from [11]. Let $G$ and $M$ be arbitrary sets and $I \subseteq G \times M$ be a binary relation between $G$ and $M$. The triple $(G, M, I)$ is called a formal context. Each $g \in G$ is interpreted as an object, each $m \in M$ is interpreted as an attribute. The statement $(g, m) \in I$ is interpreted as "$g$ has attribute $m$". The two following derivation operators $(\cdot)'$:

$$A' = \{m \in M \mid \forall g \in A : gIm\} \qquad for\ A \subseteq G,$$
$$B' = \{g \in G \mid \forall m \in B : gIm\} \qquad for\ B \subseteq M$$

define a Galois connection between the powersets of $G$ and $M$. The derivation operators $\{(\cdot)', (\cdot)'\}$ put in relation elements of the lattices $(\wp(G), \subseteq)$ of objects and $(\wp(M), \subseteq)$ of attributes and reciprocally.

### 2.3 Previous Work

The characterization of dependencies (in a generic sense) and FCA can be roughly divided into two different parts: a syntactical characterization and a semantical characterization. In a **syntactical** characterization, a formal context characterizes the closure of a set of dependencies. This is, given a set of dependencies, what is the maximal set of dependencies that can be derived according to the axioms for these dependencies?

The **semantical** characterization of a set of dependencies takes into account the definition of these dependencies with respect to a dataset. In this line of work, a formal context is constructed, such that the set of objects is related to a dataset, and the set of attributes is related to the set of attributes of the dataset, which is a parameter of the formal context. In this paper, we continue this latter like of work.

The definition of a formal context to characterize the functional dependencies that hold in a dataset can be found in [1,18,15]. These dependencies are also characterized with pattern structures (a generalization of FCA) in [5]. Degenerate multivalued dependencies are characterized in [2], multivalued dependencies in [3], and similarity dependencies are characterized in terms of pattern structures in [4].

In all previous cases, the characterization in terms of FCA (or pattern structures) allows the formal context to answer the question: does a specific dependency hold in that dataset? It can also compute the whole set of dependencies that hold in that dataset. As we have mentioned, previously, the defined context takes into account the set of tuples and the set of attributes of a given dataset. As an example, we describe the formal context that was defined in [3] to characterize the set of multivalued dependencies that hold in a dataset $T$. This formal context was defined as $\mathbb{K}_T = (\text{Part}(T), \text{Part}(\mathcal{U}), I)$ where $\text{Part}(T)$ is the set of all partitions that can be formed with the set of tuples $T$, and $\text{Part}(\mathcal{U})$ is the set of all partitions of the set of attributes $\mathcal{U}$. We mention this example because

multivalued dependencies are a special case of acyclic join dependencies, which are treated in this paper. In the next section, we define a new formal context for acyclic join dependencies that will be simpler (in size) than this previous context, and that will characterize acyclic join dependencies, as well as multivalued dependencies, as a special case.

## 3 Results

In this section we present the main result in this paper. Given a dataset, we define a formal context that can be used to check if an AJD holds in that dataset. Eventually, this formal context could also compute the set of all AJDs that hold in that dataset, although this is not discussed in this paper.

Before defining the formal context, we need to define the binary relation it will contain.

**Definition 6.** *Let $T$ be a set of tuples, and $\mathcal{U}$ its set of attributes. Let $S = [X \mid Y] \in \text{Split}(\mathcal{U})$ and let $(t_i, t_j) \in \text{Pair}(T)$. We define the relation $I \subseteq \text{Split}(\mathcal{U}) \times \text{Pair}(T)$ as follows: $[X \mid Y]$ is related to the pair of tuples $(t_i, t_j)$ if and only if the tuples $t_i(X)t_j(Y)$ and $t_j(X)t_i(Y)$ are in $T$. More formally:*

$$[X \mid Y] \quad I \quad (t_i, t_J) \Leftrightarrow t_i(X)t_j(Y), t_j(X)t_i(Y) \in T$$

We are now ready to define the following formal context:

**Definition 7.** *Let $T$ be a set of tuples, and $\mathcal{U}$ its set of attributes. The AJD-formal context for the set of tuples $T$ is:*

$$\mathbb{K}_T = (\text{Split}(\mathcal{U}), \text{Pair}(T), I)$$

We now use this formal context in order to check if an AJD holds in $T$. We have seen that, according to Proposition 2, in order to check if an AJD holds in a dataset, we need to check if the equivalent set of MVDs hold as well. This is the base for our next theorem.

**Theorem 1.** *Let $T$ be a dataset, and let $R = \{ R_1, \ldots, R_N \}$ be an acyclic join dependency, and let $JT = \langle R, E \rangle$ the corresponding join tree of $R$. $R$ holds in $T$ if and only if:*

$$\bigwedge_{e \in E} \{ [X \mid YZ] \}' = \{ [Y \mid XZ], [Z \mid XY] \}'$$

*holds in the formal context $\mathbb{K}_T = (\text{Split}(\mathcal{U}), \text{Pair}(T), I)$, where $\langle X, Y, Z \rangle = \text{Removal}(JT, e)$.*

*Proof.* The base of this proof is to check if this condition holds for all the MVDs that are equivalent to an AJD according to Proposition 2.

($\Rightarrow$) We assume that $R$ holds in $T$, we need to prove that

$$\bigwedge_{e \in E} \{\, [X \mid YZ] \,\}' = \{\, [Y \mid XZ], [Z \mid XY] \,\}'$$

where $\langle\, X, Y, Z \,\rangle = \text{Removal}(JT, e)$. We take an arbitrary $e \in E$ and a pair of tuples $t_i, t_j \in T$. We need to prove now two things:

1. If $(t_i, t_j) \in \{\, [X \mid YZ] \,\}'$, then, we have that $(t_i, t_j) \in \{\, [Y \mid XZ], [Z \mid XY] \,\}'$. Since $(t_i, t_j) \in \{\, [X \mid YZ] \,\}'$ tuples $t_i(X)t_j(YZ)$ and $t_j(X)t_i(YZ)$ are in $T$. Therefore, we have that $T$ contains, at least:

$$\begin{array}{ll} t_i(X)t_i(Y)t_i(Z) & t_j(X)t_j(Y)t_j(Z) \\ t_i(X)t_j(Y)t_j(Z) & t_j(X)t_i(Y)t_i(Z) \end{array}$$

   Now, if $(t_i, t_j) \in \{\, [Y \mid XZ] \,\}'$, we need the following tuples to be in $T$: $t_i(X)t_j(Y)t_i(Z)$ and $t_j(X)t_i(Y)t_j(Z)$ and for $(t_i, t_j) \in \{\, [Z \mid XY] \,\}'$, we need the following tuples: $t_i(X)t_i(Y)t_j(Z)$ and $t_j(X)t_j(Y)t_i(Z)$. By Proposition 2, we have that $[\, XY, XZ \,]$ holds in $T$. This implies that the following tuples are also in $T$:

$$\begin{array}{ll} t_i(X)t_i(Y)t_j(Z) & t_i(X)t_j(Y)t_i(Z) \\ t_j(X)t_j(Y)t_i(Z) & t_j(X)t_i(Y)t_j(Z) \end{array}$$

2. We need to prove the inverse condition: if $(t_i, t_j) \in \{\, [Y \mid XZ], [Z \mid XY] \,\}'$, then, we have that $(t_i, t_j) \in \{\, [X \mid YZ] \,\}'$. Since $(t_i, t_j) \in \{\, [Y \mid XZ], [Z \mid XY] \,\}'$, we have, at least, the following tuples in $T$:

$$\begin{array}{ll} t_i(X)t_i(Y)t_i(Z) & t_j(X)t_j(Y)t_j(Z) \\ t_i(X)t_j(Y)t_i(Z) & t_j(X)t_i(Y)t_j(Z) \\ t_i(X)t_i(Y)t_j(Z) & t_j(X)t_j(Y)t_i(Z) \end{array}$$

   If we want $(t_i, t_j) \in \{\, [X \mid YZ] \,\}'$ we need the following tuples to be in $T$ as well: $t_i(X)t_j(Y)t_j(Z)$ and $t_j(X)t_i(Y)t_i(Z)$. Since $[\, XY, XZ \,]$ holds in $T$, we also have the following tuples in $T$: $t_i(X)t_j(Y)t_j(Z)$ and $t_j(X)t_i(Y)t_i(Z)$.

($\Leftarrow$) We now assume that $\bigwedge_{e \in E} \{\, [X \mid YZ] \,\}' = \{\, [Y \mid XZ], [Z \mid XY] \,\}'$ where $\langle\, X, Y, Z \,\rangle = \text{Removal}(JT, e)$ holds, we need to prove that $R$ holds in $T$. For that, we use Proposition 2, and, therefore, we prove that all the MVDs that are equivalent to $R$ hold in $T$.
We prove that an arbitrary MVD $[\, XY, XZ \,]$, where $\langle\, X, Y, Z \,\rangle = \text{Removal}(JT, e)$, holds if $\{\, [X \mid YZ] \,\}' = \{\, [Y \mid XZ], [Z \mid XY] \,\}'$. We take a pair of tuples $(t_i, t_j)$ such that $t_i(X) = t_j(X)$. In order to prove that $[\, XY, XZ \,]$ holds, we need to prove that the following tuples are in $T$ as well: $t_i(X)t_i(Y)t_j(Z)$ and $t_i(X)t_j(Y)t_i(Z)$. Clearly, $(t_i, t_j) \in \{\, [X \mid YZ] \,\}'$, and by the hypothesis, we have that $(t_i, t_j) \in \{\, [Y \mid XZ], [Z \mid XY] \,\}'$. This means that the following tuples are in $T$: $t_i(X)t_i(Y)t_j(Z)$ and $t_i(X)t_j(Y)t_i(Z)$.

■

## 4 Example

We provide a running example in order to illustrate and clarify the results that are contained in the previous section. From the dataset in Example 1, we define the formal context $\mathbb{K}_T = (\text{Split}(\mathcal{U}), \text{Pair}(T), I)$ in Table 1.

**Table 1.** Formal context $\mathbb{K}_T = (\text{Split}(\mathcal{U}), \text{Pair}(T), I)$

| $\mathbb{K}$ | $(t_1, t_2)$ | $(t_1, t_3)$ | $(t_1, t_4)$ | $(t_2, t_3)$ | $(t_2, t_4)$ | $(t_3, t_4)$ |
|---|---|---|---|---|---|---|
| $[a \mid bcd]$ | × | × | × | × | × | × |
| $[b \mid acd]$ | × | × | × | × | × | × |
| $[c \mid abd]$ | × | | | | | × |
| $[d \mid abc]$ | × | | | | | × |
| $[ab \mid cd]$ | × | × | × | × | × | × |
| $[ac \mid bd]$ | × | | | | | × |
| $[ad \mid bc]$ | × | | | | | × |

The AJD $R = [\, ab, bc, cd \,]$ holds in $T$ because $T = \Pi_{ab}(T) \bowtie \Pi_{bc}(T) \bowtie \Pi_{cd}(T)$. We check that in our context we also have this result. Let $E = \{\, (ab, bc), (bc, cd) \,\}$ and $JT = \langle\, R, E \,\rangle$ be the join tree of $R$. We verify that $\bigwedge_{e \in E} \{\, [X \mid YZ] \,\}' = \{\, [Y \mid XZ], [Z \mid XY] \,\}'$, where $\langle\, X, Y, Z \,\rangle = \text{Removal}(JT, e)$. This is:

$$\{\, [b \mid acd] \,\}' = \{\, [a \mid bcd], [ab \mid cd] \,\}' \wedge \{\, [c \mid abd] \,\}' = \{\, [ab \mid cd], [abc \mid d] \,\}' \Leftrightarrow$$
$$\{\, (t_1, t_2), (t_1, t_3), (t_1, t_4), (t_2, t_3), (t_2, t_4), (t_3, t_4) \,\} =$$
$$\{\, (t_1, t_2), (t_1, t_3), (t_1, t_4), (t_2, t_3), (t_2, t_4), (t_3, t_4) \,\} \cap \{\, (t_1, t_2), (t_1, t_3), (t_1, t_4), (t_2, t_3), (t_2, t_4), (t_3, t_4) \,\}$$
$$\wedge \{\, (t_1, t_2), (t_3, t_4) \,\} = \{\, (t_1, t_2), (t_1, t_3), (t_1, t_4), (t_2, t_3), (t_2, t_4), (t_3, t_4) \,\} \cap \{\, (t_1, t_2), (t_3, t_4) \,\}$$

which is true.

## 5 Conclusions and Future Work

We have presented a new formal context for acyclic join dependencies. This context generalizes a previous approach for multivalued dependencies simply because these dependencies are a special case, and it simplifies it because the formal context has a smaller size. Acyclic join dependencies are of capital importance in database design and validation, among many others.

This result is just a first step towards (1) a more complete characterization of acyclic join dependencies or join dependencies within the formal concept analysis framework, (2) the application of FCA algorithms to compute the set of AJDs that hold in a dataset, (3) the computation of minimal bases for AJDs, and

(4) using other FCA-related formalisms for the same purpose, as, for instance, pattern structures.

**Acknowledgments.** This research work has been supported by the SGR2014-890 (MACDA) project of the Generalitat de Catalunya, and MINECO project APCOM (TIN2014-57226-P).

## References

1. J. Baixeries. A formal concept analysis framework to model functional dependencies. In *Mathematical Methods for Learning*, 2004.
2. J. Baixeries and J. L. Balcázar. Characterization and armstrong relations for degenerate multivalued dependencies using formal concept analysis. In B. Ganter and R. Godin, editors, *ICFCA*, volume 3403 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2005.
3. J. Baixeries and J. L. Balcázar. A lattice representation of relations, multivalued dependencies and armstrong relations. In *ICCS*, pages 13–26, 2005.
4. J. Baixeries, M. Kaytoue, and A. Napoli. Computing similarity dependencies with pattern structures. In M. Ojeda-Aciego and J. Outrata, editors, *CLA*, volume 1062 of *CEUR Workshop Proceedings*, pages 33–44. CEUR-WS.org, 2013.
5. J. Baixeries, M. Kaytoue, and A. Napoli. Characterizing functional dependencies in formal concept analysis with pattern structures. *Annals of Mathematics and Artificial Intelligence*, 72(1-2):129–149, Oct. 2014.
6. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, July 1983.
7. C. Beeri and M. Y. Vardi. Formal systems for join dependencies. *Theoretical Computer Science*, 38:99 – 116, 1985.
8. C. Berge. *Hypergraphs*, volume 45 of *North-Holland Mathematical Library*. North-Holland, 1989. Combinatorics of Finite Sets.
9. P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.
10. E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.
11. B. Ganter and R. Wille. *Formal Concept Analysis*. Springer, Berlin, 1999.
12. M. Gyssens. On the complexity of join dependencies. *ACM Trans. Database Syst.*, 11(1):81–108, Mar. 1986.
13. P. C. Kanellakis. Elements of relational database theory. In J. van Leeuwen, editor, *Handbook of theoretical computer science (vol. B)*, pages 1073–1156. MIT Press, Cambridge, MA, USA, 1990.
14. S. O. Kuznetsov. Machine learning on the basis of formal concept analysis. *Autom. Remote Control*, 62(10):1543–1564, Oct. 2001.
15. S. Lopes, J.-M. Petit, and L. Lakhal. Functional and approximate dependency mining: database and fca points of view. *Journal of Experimental and Theoretical Artificial Intelligence*, 14(2-3):93–114, 2002.
16. D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
17. F. Malvestuto. A complete axiomatization of full acyclic join dependencies. *Information Processing Letters*, 68(3):133 – 139, 1998.
18. R. Medina and L. Nourine. Conditional functional dependencies: An fca point of view. In L. Kwuida and B. Sertkaya, editors, *ICFCA*, volume 5986 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2010.