Discovering Meaningful Keys from Ontologies

Oscar Romero, Alberto Abelló, and Joan-Marc Montesó

Dept. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
Jordi Girona 1-3, 08034 Barcelona, Spain
{oromero|aabello|jmonteso}@lsi.upc.edu

Abstract. Object identification is a crucial step in most information systems. Nowadays, we have many different ways to identify entities such as surrogates, keys and object identifiers. However, not all of them guarantee the entity identity. Many works have been introduced in the literature for discovering meaningful keys, but all of them work at the logical or data level and they share some inherent constraints. Addressing it at the logical level, we may miss some important data dependencies, while the cost to identify data dependencies at the data level may not be affordable. In this paper we propose an approach for discovering meaningful keys from domain ontologies. In our approach, we guide the process at the conceptual level and we introduce a set of pruning rules for improving the performance by reducing the number of key hypotheses generated and to be verified with data. Finally, we also introduce a simulation over a real world case study to show the feasibility of our method.

1 Introduction

Object identification is a crucial step in most information systems. Identification mechanisms guarantee that a real world entity will be uniquely identified in the system and distinguished from the others. Nowadays, we have many different ways to identify entities such as surrogates (i.e., internal identifiers), keys and object identifiers. However, not all of them guarantee the entity identity [20]. As example, the uniqueness of the surrogates (the most common way to implement primary keys in relational databases) are meaningless for data quality. For instance, consider the table depicted in figure 1 and extracted from [13]. The primary key of the relation is the `empId` attribute (a surrogate) and it clearly shows that despite uniqueness of the primary key is enforced by the database, we have two different internal identifiers (i.e., surrogates) for the same real world entity (i.e., the person with name `Abraham Millard`). In this case, the `SSN` would be a much better primary key since every person is supposed to have a unique `SSN`. Meaningful identity keys are crucial for assuring the quality of data in many reengineerning process; e.g., for ensuring the quality when migrating, converting and merging systems, for data cleansing, data integration or for building data warehousing systems free of problems like the one shown above [13].

In this paper we propose an approach for discovering meaningful keys from domain ontologies. The software engineering area has claimed to use conceptual

E_EMPLOYEE_PROFILE						
EmplD	SSN	LastName	FirstName	BirthDate	Gender	CompanyCode
141857	834-51-2788	ACCARDI	LAURENE	9/30/1954	F	B8
235572	770-27-1938	BUBB	CHARLES E	8/4/1942	M	B6
346981	988-65-1411	ABRAHAM	MILLARD	7/19/1943	M	N2
155107	988-65-1411	ABRAHAM	MILLARD	7/19/1943	M	N2
247826	989-85-2689	CHIU	HARLAN	1/1/1922	M	N2

Fig. 1. Example of a non-identity identifying key

representations of the domain on the top of systems to have an up-to-date and accurate formalization of the system domain [14]. This approach has given rise to concepts such as OBDA (Ontology-Based Data Access) [15] and nowadays, ontologies are used in many fields such as data integration [10], conceptual modeling [15] and the semantic web [2]. In our approach, we guide the process at the conceptual level and we introduce a set of pruning rules for improving the performance by reducing the number of key hypotheses generated and to be verified with data. Our algorithm is relevant since, despite the importance of object identification, most description logics (the most extended ontology languages and also the basis of the OWL language -a W3C recommendation-) do not provide identification mechanisms, and only very expressive DL (that are not suitable for real world applications due to their computational complexity) incorporate them [3]. For instance, the only way to specify identification in OWL DL is by means of one-to-one relationships which are clearly not enough. Furthermore, the fact that most description logics do not consider n-ary relationships makes impossible to assert meaningful keys for ontologies. All in all, an algorithm to discover meaningful keys benefiting from the semantics of an ontology is a relevant issue for many real world systems. Finally, we also introduce a simulation over a real world case study to show the feasibility of our method.

2 Foundations

Our approach is guided by knowledge about the domain captured in the ontology. However, the key concept is tightly related to the database theory and it deserves a further discussion of what a key means at the conceptual level. We denote concepts by uppercase letters from the beginning of the alphabet (such as A and B) and sets of concepts by uppercase letters from the end of the alphabet (such as Y and Z).

Definition 1. *We say that a set of concepts* Z *is a key of a concept* A*, if there is an injective function from* A *to* Z *(i.e., a mandatory one-to-one relationship).*

Along the paper we will use a generic conceptual notation but by *concepts* we refer to an ontology concept or datatype and by *relationships* to ontology roles. Notice that in this definition we do not ask for a mandatory participation of Z in A (i.e., a bijective function) since some values of the key could not have a correspondence into the identified concept. This is needed when considering non-reusable keys. For instance if a person has a social security number (SSN) and when s/he dies it is not reused again. This definition is equivalent to other

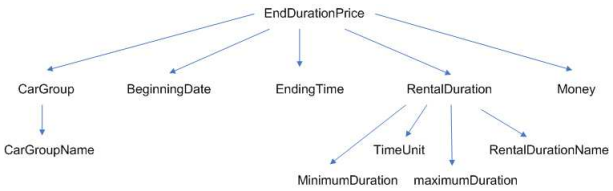

Fig. 2. The fd tree for the `EndDurationPrice` concept

concepts previously introduced that we could consider equivalent. For example, the one-to-one relationships introduced in [4] or the *reference mode* (mandatory one-to-one relationships) in ORM [19]. This definition entails that both, the concept identified functionally depends on the key and the key functionally depends on the concept, and gives rise to the following proposition:

Proposition 1. *A set of concepts* Z *is a key of a given concept* A *iff* Z *uniquely or functionally determines the values of* A *(i.e.,* Z $\rightarrow$ A*) and* A *functionally determines the values of* Z *(i.e.,* A $\rightarrow$ Z*).*

This is sound with previous work presented in the literature to discover keys. As discussed in section 6, previous approaches work at the data level and a key is defined as a specific kind of functional dependency (i.e., a *minimal* set of attributes that uniquely identify the whole *tuple*). Moreover, according to the relational model assumptions, each relation row is supposed to represent a different instance [5], giving rise as a whole to a one-to-one relationship (furthermore, since a candidate key does not allow NULL values, it is also mandatory).

The first step in our approach is to compute the asserted functional dependencies (in short, fd's) in the domain ontology. Due to space limitations, in this paper we will consider that we have already computed the fd's of each ontology concept. The reader may refer to [17] to find an algorithm to identify fd's from an ontology and to [18] to find a deep discussion of what a functional dependency means at the conceptual level and the differences we need to take into account when looking for them. In short, there is no way to express that $A \cup B \rightarrow C$ holds (where A, B and C are concepts) because in most ontology languages[1], roles are binary predicates relating one concept to another concept. Roughly speaking, in a DL TBox we can only assert fd's by means of functional roles (or equivalent assertions) and therefore, of the kind $A \rightarrow B$, where A and B are concepts. Then, by means of transitivity and concept taxonomies we compute the closure of the fd's asserted in the ontology (see [18] for further details). The algorithm discussed in [18] rises a nice computational complexity which happens to be polynomial for real ontologies. With this algorithm, we get a directed tree of fd's for each domain concept as shown in figure 2. This example refers to the `endDurationPrice` concept of a car renting ontology [9]. It represents the final price charged for the car renting to the customer. As shown in the figure, it has 10 fd's: the `car group` (i.e., kind of car rented) and the `car group name`, the

[1] Despite we do not focus on a specific ontology language, ontology languages such as OWL DL and most common DLs only consider binary roles.

`beginning` and `ending` date of the rental agreement, the final price (i.e., `money`) and the rental agreement `duration` (which consists of the `rental duration name` and a `time unit` used to express the `minimum` and `maximum duration` allowed for that rental). From here on, we will refer to this tree as the *fd tree*. We refer as *root concepts* to those concepts in the first level of the tree (in our example: `car group`, `beginning` and `ending` date, `duration` and `money`) and as *fd-concepts* to the rest. We will also use typical tree notation and we will talk about depth levels, ancestors and descendants.

Next, once fd's asserted in the ontology have been computed, for every ontology concept A, we aim to find the set of concepts Z such that Z is a key of A. According to prop. 1, Z is a key of A iff $A \rightarrow Z$. Thus, we only need to generate combinations of concepts among those functionally identified by A. For instance, in our example, it means that all the possible keys of `endDurationPrice` are combinations of concepts between its fd's (i.e., those represented in the figure). Notice the benefits of this proposition. Traditionally, when looking for keys, the searching space we have is conformed of all the attribute combinations up to size N, where N is the number of attributes in the database (i.e., 2^N combinations), but by using ontological knowledge we reduce the searching space to 2^P, where P is the number of concepts functionally dependent on A.

2.1 Necessary Conditions

A naive approach for discovering keys would be to generate all the combinations in our searching space (i.e., 2^P combinations) and sample data to verify them. However, despite we have reduced considerably the searching space, we may have computational problems for concepts having many fd's, since the searching space is still exponential. Furthermore, querying the data may be expensive for large tables. For this reason, we further exploit the conceptual knowledge we have before verifying key hypotheses with data.

Given a set of fd's F, a *minimal cover* [1] of F is a set F' of fd's such that:

- (i) Each dependency in F' has the form $Z \rightarrow C$, where C is a concept,
- (ii) $F' \equiv F$,
- (iii) no proper subset of F' implies F and
- (iv) for each dependency $Z \rightarrow C$ in F' there is no $W \subset Z$ such that $F \models W \rightarrow C$.

In our approach, for every ontology concept A, we define F as the set of fd's of the kind $Z \rightarrow A$ (where Z is a set of concepts). That is, all those fd's determining A and we look for a minimal cover of F because we aim to minimize the number of queries posed to the database. For our purpose, the minimal cover represents the minimum set of fd's from where to derive F. Said in other words, it is the minimum set of fd's to be verified as keys with data. The rest of fd's in F can be generated and verified from F' in polynomial time by the *Armstrong axioms* [16] (i.e., a fd of the kind $Z \rightarrow A$ holds iff fd $\in F'^{+}$). Thus, we do not need to pose new queries to verify them as keys (indeed, as we will discuss later, they will not be keys since they are not minimal).

In our approach, (i) is guaranteed as we compute the fd's from an ontology. As discussed in previous section, the fd's we may find in an ontology are of the kind $A \rightarrow B$, (where A and B are concepts), and in our algorithm we generate combinations of concepts in the left-hand side of the fd (i.e., left-hand side multi-attribute fd's). Regarding (ii), F will be equivalent to the set of fd's determining A that we can infer from knowledge contained in the ontology (from where we compute the initial knowledge that guides the search) and data (from where verify combinations found). Thus, if a key cannot be inferred from the ontology and verified with data, we will not be able to identify it. Section 4 guarantees that our algorithm is complete with regard to knowledge captured in the ontology and data. Finally, (iii) and (iv) guarantee that the set of fd's in F' is minimal. Notice that these two conditions are desirable for our purpose, as they enforce the *minimality* property of keys. Consequently, fd's in the minimal cover are the only candidates to be keys. In our approach, these two items give rise to two necessary conditions a fd must fulfill prior to be verified as a key with data. Prior to introduce them, we present the *Armstrong axioms* used to infer all the fd's that can be computed from a given set F of fd's (i.e., the F closure or F^+) [1], since we will need them to proof our propositions:

- (reflexivity) If $Y \subseteq X$, then $X \rightarrow Y$,
- (augmentation) If $X \rightarrow Y$, then $X \cup Z \rightarrow Y \cup Z$
- (transitivity) If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

We also introduce the pseudo-transitivity rule [1] (that can be easily derived from the three inference rules discussed above), as we will need it later:

- (pseudo-transitivity) If $X \rightarrow Y$ and $YW \rightarrow Z$, then $XW \rightarrow Z$.

Proposition 2. *Let* Z *and* W *be sets of concepts functionally dependent on a given concept* A*. If* $Z \subseteq W$ *and* Z *is a key of* A *then,* W*, despite functionally determining* A*, does not belong to the minimal cover since it is not minimal; there exists a subset of* W *(i.e.,* Z*) functionally determining* A*.*

We will not present a proof for this proposition since it is directly formulated from item (iv) in the minimal cover definition. Intuitively, this proposition enforces the minimality property of a key. Said in other words we must look for fd's ($Z \rightarrow A$) such that every concept in the left-hand side (i.e., in Z) is necessary (i.e., if we drop any concept, Z does not functionally determine A anymore).

Proposition 3. *Let* Z *and* W *be two sets of concepts functionally dependent on a given concept* A*. If* ZW *is a minimal key of* A *then,* Z *and* W *are orthogonal. That is, it does not exist two sets of concepts* Z_1 *and* W_1 *such that* $Z_1 \subseteq$ Z *and* $W_1 \subseteq$ W *and (*$Z_1 \rightarrow W_1$ *or* $W_1 \rightarrow Z_1$*).*

Let $W_2 = W$ - W_1 (respectively $Z_2 = Z$ - Z_1). If $WZ \rightarrow A$ and $Z_1 \rightarrow W_1$ (resp. $W_1 \rightarrow Z_1$) then $Z W_2 \rightarrow A$ (resp. $W Z_2 \rightarrow A$) holds (by the pseudo-transitivity rule). Thus, given a set of fd's F' such that $\{Z_1 \rightarrow W_1$ (resp. $W_1 \rightarrow Z_1$), $ZW \rightarrow A$, $Z W_2 \rightarrow A$ (resp. $W Z_2 \rightarrow A$)$\} \in F'$, F' is not a minimal cover. We can show it by contradiction.

Proof. Let F' be a minimal cover of F and $\{Z_1 \rightarrow W_1$ (resp. $W_1 \rightarrow Z_1)$, $ZW \rightarrow A$, $Z\,W_2 \rightarrow A$ (resp. $W\,Z_2 \rightarrow A)\} \in F'$. Then, there is a proper subset F'' of F' (i.e., $F'' = F'$ - $\{ZW \rightarrow A\}$) such that F'' implies F, which violates item (iii) in the minimal cover definition. We can get $ZW \rightarrow A$ from $Z\,W_2 \rightarrow A$ (resp. $W\,Z_2 \rightarrow A$) by the augmentation rule (i.e., adding W_1 (resp. Z_1) to both sides), and then by the reflexivity and transitivity rules. □

Intuitively, this property says that the combination ZW must not be considered if Z and W are not orthogonal. Otherwise, this set of fd's will not be minimal as demanded in the minimal cover definition. Finally, we introduce a third necessary condition derived from the fd's theory:

Proposition 4. *Let* W *be a set of concepts functionally dependent on a concept* A*, and* C *a concept such that* C ∈ W*. Let* $\mathcal{I}$ *be the set of intermediate concepts giving rise to the many-to-one path between* A *and* C*.* W *functionally determines* A *iff, for each* $\{C_i\} \in \mathcal{I}$*,* $(W - \{C\}) \cup \{C_i\}$*, namely the intermediate sets of* C*, functionally determines* A*.*

Proof. It can be seen by means of the pseudo-transitivity rule. Since each intermediate concept determines C then, if $W \rightarrow A$, for each $\{C_i\} \in \mathcal{I}$, $(W - \{C\}) \cup \{C_i\} \rightarrow A$. □

Intuitively, we are taking advantage of the pseudo-transitivity rule to foresee if a given combination can functionally determine A. In our approach, we only generate and verify with data those combinations that fulfill the three necessary conditions introduced above. In short, let A be a concept and Z a set of concepts. We check whether $Z \rightarrow A$ iff $A \rightarrow Z$ (i.e., for generating Z we only consider combinations of concepts in the fd tree of A). Furthermore, among all the potential combinations we may generate to verify $Z \rightarrow A$, we only look for those that would belong to the minimal cover of the fd's determining A by applying props. 2 and 3. We also use prop. 4 for pruning. If a certain combination does not guarantee any of these conditions we may foresee it will not be a key and thus, it does not have to be generated and verified. Only those combinations satisfying all the conditions are *feasible keys* and have to be verified with data. In this way, we reduce drastically the searching space and the number of combinations to be tested against the database (i.e., minimizing the number of queries posed to the RDBMS).

2.2 Searching Space

In this section we define in detail the searching space our algorithm will explore and prune thanks to the necessary conditions presented in previous section. Our searching space can be characterized as a directed graph as shown in figure 3. The number of combinations in the searching space is 2^P where P is the number of concepts functionally dependent on A. In our example, we build the searching space for a given concept C (see right side of the figure). The first two depth levels of the searching space (we do not show the rest for the sake of comprehension)

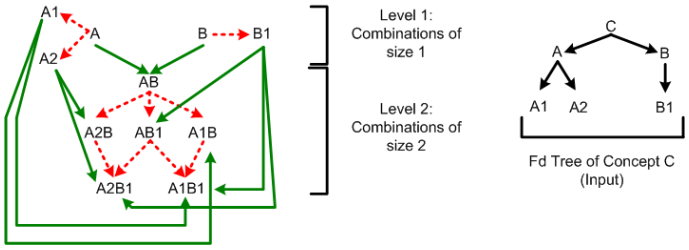

Fig. 3. The searching space of a concept C and its input fd tree.

are shown at the left side of the figure. Two combinations in the searching space can be related by two different kind of edges:

- Subset edges (regular edges): These edges link two nodes of size i and $i+1$ respectively (where i is an integer between 1 and P) such that the first is a subset of the second one. For instance, the edge between A and AB, and we say that AB is a *s-descendant* of A (we denote this as $A \subset AB$). Notice that these arrows link combinations in consecutive depth levels. Therefore, if a 3-sized combination (in the third depth level) such as ABA_1 was depicted in the figure it would be related with a regular arrow with AB, AA_1 and BA_1 but not with A, A_1 or B, since they are not placed in consecutive depth levels.
- Fd edges (dashed edges): These edges are derived from the input fd tree. They link two nodes of the same size such that there is one concept in the first one functionally determining one concept in the second one. For instance, AB is related with AB_1, as $B \rightarrow B_1$. We say that AB_1 is a *fd-descendant* of AB (we denote this as $AB \prec AB_1$). Notice that these edges relate combinations where only one concept changes. For instance, AB is not related with A_1B_1 despite $B \rightarrow B_1$ and $A \rightarrow A_1$, as we must substitute two concepts of AB to get A_1B_1. Indeed, AB is related to A_1B_1 by pseudo-transitivity, as, following the nomenclature of prop. 4, there are two intermediate sets (i.e., AB_1 and A_1B; see figure 3) between them.

Notice talking about depth levels is meaningless in most graphs, but we can still talk about graph depth levels according to how we have defined the edges. On level i we depict those combinations of size i. Between consecutive depth levels we find regular arrows, and dashed arrows between combinations in the same depth level. Moreover, the meaning of the subset and fd edges introduce a *partial order* in the searching space (i.e., $\subset$ for subsets and $\prec$ for fd's). Finally, the reader may notice that subset edges will be those explored (and pruned if needed) by prop. 2 whereas fd edges will be explored (and pruned) by prop. 4.

3 An Algorithm for Discovering Keys

In this section we present an algorithm that generates the minimal cover of fd's of our searching space (i.e., those fd's being a feasible key). This algorithm has two inputs: the concept we are looking keys for (for instance, the

```
function seek_keys (Concept A, FdTree M) returns Set<Key>

1. Set<Concept> Comb; Ordered Set<Comb> Candidates_Sets, Feasible_Keys;
2. int i:=1; Set<Comb> Keys := {};
3. Feasible_Keys := Get_Root_Concepts(A,M);
4. while(Feasible_Keys != Ø)
   (a) Candidates_Sets := {};
   (b) Comb := Get_First_Combination(Feasible_Keys);
   (c) while(Comb != Ø)
         i. if(Determines(Comb,A)) then
              A. Keys += Comb;
              B. if (Has_fd-Descendants(Comb,M)) then
                   Set<Comb> NewCombs := Generate_New_Combinations(Comb, Keys, M);
                   Feasible_Keys += NewCombs;
        ii. else
              A. Candidates_Sets += Comb;
       iii. Feasible_Keys -= Comb;
        iv. Comb := Get_Next_Combination(Feasible_Keys);
   (d) i++;
   (e) Feasible_Keys := Generate_Combinations_by_Size(i, Keys, Candidates_Sets, M);
5. return Keys;
```

Fig. 4. An algorithm to look for **Keys**

`endDurationPrice`), and its fd tree (see figure 2). This algorithm takes advantage of previous generated and tested combinations to decide if we have to explore further alternatives according to the necessary conditions introduced in section 2. This algorithm is devised according to three main sets:

- *Feasible keys*: In the i^{th} iteration, this set represents those combinations of size i, satisfying the three necessary conditions introduced in section 2.
- *Candidate sets*: In the i^{th} iteration, this set contains those *feasible keys* refuted as keys with data.
- *Keys*: In the i^{th} iteration, this set contains those *feasible keys* verified as keys with data.

Given a concept A and its fd tree, our algorithm starts considering each root concept as a *feasible key* (see figure 4, step 3). Thus, they are verified (see step 4ci and section 3.1) to see if, according to data, it is indeed a key (if it is, this combination is added to the *key set*; see step 4ciA) or if it is not (then, it is added to the *candidate sets*; see step 4ciiA).

Notice that we only explore the fd-descendants of a combination Z if Z determines A (see step 4ci and section 3.1). It is a direct application of prop. 4: the fd-descendants of a given combination Z cannot determine A if Z does not determine A. From here on, we will refer to it as the *Intermediate Set Rule* (ISR). A combination will be generated by ISR in the *generate_new_combinations* function iff its direct fd-ancestors determine A (the reason why this function needs the *key set* as parameter). Notice that we only need to check the direct fd-ancestors, as they were generated by ISR and thus, fulfilling prop. 4. The number of new combinations added to the *feasible keys* by applying ISR for each key identified, is, in the worst case, linear regarding the number of direct fd-descendants the key has (see the properties of fd edges in section 2.2). For instance, if {`BeginningDate` × `EndingDate` × `RentalDuration`} is a key, then

function *Generate_Combinations_by_Size* (int i, **Set**<Comb> Keys, **Ordered Set**<Comb> Candidates_Sets, FdTree M) **returns** Set<Comb>

1. **Set**<Comb> Combinations := {};
2. **For**(**int** j = 0; j < **sizeof**(Candidates_Sets); j++)
   (a) CS1 := get_candidate_set(Candidates_Sets, j);
   (b) **For**(**int** z = j+1; z < **sizeof**(Candidates_Sets); z++)
      i. CS2 := get_candidate_set(Candidates_Sets, z);
      ii. **if** (Have_(i-1)_Concepts_In_Common(CS1,CS2) AND (i != 2 OR orthogonal(CS1,CS2)))
         A. **if**(Find_Subsets(CS1,CS2, Keys, Candidates_Sets, z)) **then**
         B. Combinations += Eliminate_Duplicates(CS1 ∪ CS2);
3. return Combinations;

Fig. 5. An algorithm to Generate i-sized Combinations

by ISR we must add to the *feasible keys*: {`BeginningDate` × `EndingDate` × `RentalDurationName`}, {`BeginningDate` × `EndingDate` × `MinimumDuration`}, {`BeginningDate` × `EndingDate` × `MaximumDuration`} and {`BeginningDate` × `EndingDate` × `TimePeriod`}. Each one of the new combinations generated satisfy props. 2 and 3 (see section 4) and for this reason we directly add them to the *feasible keys*. Thus, each new combination is verified as a key. If it is a key then, we apply again ISR and we continue exploring its fd-descendants iteratively. Notice that a combination generated by ISR which is refuted as a key (i.e., the *determines* function returns false and therefore, it is queued in the *candidate sets*) may give rise to keys of size bigger than i which involve other orthogonal concepts.

When the first iteration is done we generate *feasible keys* of size 2 from the *candidate sets* of size 1 (see step 4e and section 3.2 for further details). The algorithm iterates until we are not able to generate *feasible keys* of size *i+1*.

3.1 The *determines* Function

This function is called when the three necessary conditions are guaranteed (i.e., we have identified a *feasible key*). Then, we verify if this combination determines A by querying data. Prior to query the instances, we first introduce a final pruning rule:

Proposition 5. *Let* Z *be a feasible key. We say that* Z *is still a* feasible *key if it is able to identify all instances of* A*. Said in other words, if the cardinality of* A *is lower or equal than the product of the cardinalities of those concepts in* Z *(i.e.,* $\forall Z_i \in Z$*:* $\prod |Z_i| \geq |A|$*)*

Notice that this pruning rule disregards combinations by just querying the RDBMS catalog with the following queries (expressed in Oracle syntax):

```
SELECT NUM_ROWS FROM USER_TABLES WHERE TABLE_NAME = t;
SELECT NUM_DISTINCT FROM USER_TABS_COLS WHERE TABLE_NAME = t AND COLUMN_NAME = c;
```

Where t is the name of a table and c of a column. If the ontology concept maps to a relational table then by means of the first query we get the cardinality of t, and if the ontology concept maps to a relational attribute by means of the

```
function Find_Subsets (Comb CS1, Comb CS2, Set<Comb> Keys, Ordered Set<Comb>
Candidates_Sets, int z) returns Boolean

1. Set<Comb> SubSets := Generate_Subsets(CS1, CS2);
2. For(int i = 0; i < sizeof(Keys); i++)
   (a) KeyAux := get_key(Keys, i);
   (b) if(KeyAux in SubSets) then
         i. return false;
3. For(int w = z+1; w < sizeof(Candidates_Sets); w++)
   (a) CSAux := get_candidate_set(Candidates_Sets, w);
   (b) if(CSAux in SubSets) then
         i. SubSets -= {CSAux};
4. foreach(SubSet in SubSets) do
   (a) if(all_root_concepts(Subset))
         i. return false;
5. return true;
```

Fig. 6. An Algorithm for Validating Subsets of a Given Combination

second query we get the number of different values it has. Those combinations satisfying this rule are still candidates to be a key, and we verify it by the following query (in Oracle syntax):

```
SELECT "key" FROM DUAL WHERE NOT EXISTS(SELECT attrSet FROM tables WHERE joinConds GROUP BY
attrSet HAVING COUNT(*) > 1)
```

Where `DUAL` is the *dummy* table in Oracle and *attrSet* are the attributes conforming the *feasible key* to be verified, *tables* the list of tables containing that attributes and *joinConds* the join clauses needed to join tables involved in the query. If we are able to find two rows with the same values for the key hypotheses then, this combination, according to data, is not a key. Notice that we use a `NOT EXISTS` expression so that if we find a counter example for this combination then, the RDMBS engine could stop the query.

3.2 Generating Combinations of Size (i+1)

In this section we present how combinations of size i are generated in our algorithm (see figure 4, step 4e) by means of the *generate_combinations_by_size* function (see figure 5). This function generates (*i+1*)-sized combinations from the i-sized *candidate sets* got in previous iteration. Combinations generated must be *feasible keys* (as we only verify with data those sets satisfying the three necessary conditions). Thus, we go through the *candidate sets* in such an order that we look for pairs of sets having (i-1) concepts in common (see figure 5, step 2bii) without generating twice the same set (see the configuration of the two main loops; steps 2 and 2b). Each set of size (i+1) is considered a *feasible key* if it satisfies props. 2 and 3 (notice that prop. 4 is satisfied by the *candidate sets* definition -see section 4 for details-). Prop. 3 is guaranteed in step 2bii. A 2-sized set is generated if concepts combined are orthogonal (i.e., if they are not in the fd tree of each other). As we will show in section 4, by the combination of props. 2 and 3, it is enough to check this proposition for 2-sized sets. Prop. 2 is guaranteed in step 2biiA by the *find_subsets* function (properly defined in figure

6), where all the subsets of the current set are verified not to be keys: (1) If all the subsets are in the *candidate sets* then we can generate it (see figure 6, step 3), otherwise, (2) if any of them is in the *key set* (see step 2), this set would not be minimal and it is not considered a (i+1)-sized *feasible key*. There is a third alternative though, (3) if a subset is neither present in the *candidate sets* nor in the *key set*. This may happen due to our pruning rules. In short, if the subset consists of root concepts, it should be refuted as a valid subset (as our algorithm is exhaustive regarding root concepts we can assure it is a s-descendant of a key; see step 4a) and accepted otherwise (as this subset fulfills the three necessary conditions; see step 5). We justify properly this decision in section 4. Subsets of size i-1 are generated in linear time. If we have i concepts, we just have to generate i subsets overlooking one concept in each subset.

4 Algorithm Correctness

Our algorithm is sound and complete with regard to the domain ontology and data, as it generates sets from knowledge captured in the ontology and verifies them with the data. In this section we show that we generate all the sets of the searching space that fulfill the three necessary conditions and we do not generate any set that do not fulfill them. Thus, our proofs of soundness and completeness rely on the three necessary conditions discussed and justified in section 2. Said in other words, in our algorithm we generate all those combinations of the searching space that (i) all its subsets are not keys (i.e., all its s-ancestors are not keys), (ii) its subsets are orthogonal (i.e., we cannot find two s-ancestors such that one is a fd-descendant of the other one) and (iii) its intermediate sets are indeed keys (i.e., all its fd-ancestors are keys). If we are able to generate all the sets meeting the necessary conditions and no other sets, it is easy to see that by verifying them with data (see section 3.1) our algorithm is still sound and complete.

For the sake of comprehension, we divide our proof in two parts. The first part shows that our algorithm is sound and complete for *root combinations* (i.e., those only consisting of root concepts). Then, we show that it also sound and complete for the *fd-combinations* (i.e., those that contain, at least one fd-concept).

4.1 Soundness & Completeness for Root Combinations

In this section we show that we generate all the root combinations satisfying the three necessary conditions (completeness) and no other combination (soundness). We show it by induction:

First Iteration: [`completeness`] In the first iteration of the algorithm, every root concept is considered a *feasible key*, as they are the smaller set to be verified as a key (see figure 4, step 3). [`soundness`] Since all of them are 1-sized sets, they satisfy props. 2 and 3.

Second Iteration: [`soundness`] 2-sized root combinations are generated from combining all the 1-sized *candidate sets* found in the previous iteration. By definition, those 1-sized sets in the *candidate sets* have been refuted as keys

(i.e., fulfilling prop. 2), and only those 2-sized sets consisting of orthogonal concepts are generated (i.e., fulfilling prop. 3. See figure 5, step 2bii). [`completeness`] We generate all the 2-sized combinations between root concepts except for (i) those consisting of a root concept being a key and (ii) those consisting of two root concepts such that one is a fd-descendant of the other.

Induction Step: If we generate all the n-sized root combinations fulfilling the three necessary conditions and only these combinations (induction hypotheses), we will generate all the (n+1)-sized root combinations that fulfill the three necessary conditions and no other sets. [`soundness`] If all the n-sized *feasible keys* are generated, those verified as keys were added to the *key set* and those refuted as keys to the *candidate sets*. According to figure 6 (steps 3 and 4), prop. 2 is fulfilled because root combinations are generated if all their subsets are in the *candidate sets* (according to the induction hypotheses all the n-sized *candidate sets* are generated) and disregarded otherwise. Prop. 3 is guaranteed because 2-sized sets consisting of non-orthogonal concepts were not generated. It can be shown by contradiction. Supposed that a (n+1)-sized set Z, that contains two concepts A and B such that $A \rightarrow B$ is generated by our algorithm. Thus, all the n-sized subsets of Z are in the n-sized *candidate sets* (i.e., there is at least one n-sized set W such that $W \subset Z$ and $A, B \subset W$). But it is against our induction hypotheses, as W is not a n-sized *feasible key* and it is not present in the n-sized *candidate sets*. Intuitively, since $\{A, B\}$ was not generated, all its s-descendants were not generated either (i.e., $\{A, B\}$ will not be in the 2-sized *candidate sets* and according to prop. 2, every 3-sized set containing it will not be generated, neither those 4-sized sets containing the 3-sized sets and so on up to W that will not be generated and thus neither Z). [`completeness`] All the (n+1)-sized root combinations are generated except for those that at least one n-sized subset is not in the *candidate sets*. It may happen because (i) this subset is a key (all the n-sized keys are found by hypotheses) or (ii) it contains a subset that is a key (i.e., it is not a n-sized *feasible key*) and thus not generated by hypotheses.

Notice that we have not justified prop. 4. This is because root combinations do not have intermediate sets by definition. □

4.2 Soundness & Completeness for Fd-combinations

In this section we show that we generate all the fd-combinations satisfying the three necessary conditions (completeness) and no other combination (soundness). Like in previous section, we show it by induction:

First Iteration: [`soundness`] 1-sized fd-combinations are generated applying ISR to combinations verified as keys and thus, fulfilling prop. 4. This new generated combinations are of size 1 and they also fulfill props. 2 and 3. [`completeness`] All the 1-sized fd-combinations are generated because ISR is iteratively applied (i.e., it follows the fd-edges until it finds fd-descendants that are not a key). Therefore, those fd-descendants not explored will not be a *feasible key* since they do not fulfill prop. 4.

Second Iteration: [`soundness`] 2-sized fd-combinations can be generated from the *generate_ combinations_ by_ size* function or by applying ISR over 2-

sized combinations verified as keys. In the first case, they fulfill the necessary conditions by the same proof introduced for 2-sized root combinations. In the second case, ISR guarantees prop. 4 as explained in the first iteration. Furthermore, combinations generated by ISR also guarantee props. 2 and 3. Let W, Z be two 2-sized sets of concepts such that $W \prec Z$, and Z was generated by ISR. By definition of the fd-edges (see section 2.2) there is a concept B such that $B \subset Z$ and $B \subset W$, and two concept A, A_1 such that $A \rightarrow A_1$ and $A \subset W$ and $A_1 \subset Z$. Z satisfies prop.2 because if W is generated, B is not a key (since it is in the *candidate sets*), and if A_1 is a key, then A is a key as well. Thus, A would have been verified as key in the first iteration and by ISR A_1 as well (so that, A_1 would be in the *key set* and Z would not be generated). Prop. 3 is guaranteed because 2-sized sets are generated if concepts on it are orthogonal (see figure 5; step 2bii). [`completeness`] We show that our algorithm finds all the 2-sized fd-combinations by contradiction. Let Z be a 2-sized fd-combination not found by our algorithm. If we have not been able to generate Z, at least one of its concepts (i.e., A_1) must not be in the 1-sized *candidate sets*. Therefore, A_1 is a fd-concept (as every root concept will be present in the *candidate sets* or in the *key set*; see second iteration in previous section). By definition of our searching space, there exists a 2-sized combination W such that W is an intermediate set of Z and it consists of root concepts (see section 2.2). Said in other words, $W \prec Z$ by pseudo-transitivity. Thus, by prop. 4, if Z is a key then W is also a key. Since our algorithm is complete for root combinations, W would have been generated and Z as well by ISR.

Induction Step: If we generate all the n-sized fd-combinations fulfilling the three necessary conditions and only these combinations (induction hypotheses), we will generate all the $(n+1)$-sized fd-combinations that fulfill the three necessary conditions and no other sets. [`soundness`] If all the n-sized *feasible keys* are generated, those verified as keys were added to the *key set* and those refuted as keys to the *candidate sets*. $(N+1)$-sized fd-combinations can be generated by applying ISR over $(n+1)$-sized combinations verified as keys or from the *generate_ combinations_ by_ size* function. In the first case, we can show that the necessary conditions are preserved by a proof analogous to the one introduced for 2-sized sets generated by ISR. In the second case, prop. 2 is guaranteed because the subsets of the $(n+1)$-sized *feasible keys* are minimal (see 6; steps 3 and 5): either we may find them on the n-sized *candidate sets* or they are not in the *candidate sets* nor in the *key set*. The latter raises when two n-sized combinations (where n $\geq$ 2) generated by ISR have been refuted as keys (thus, added to the *candidate sets*) and combined for generating a $(n+1)$-sized set. Let W and Z be the two n-sized sets generated by ISR from W' and Z' (i.e., $W' \prec W$ and $Z' \prec Z$). Let A, B, A_1 and B_1 concepts s.t. $A \rightarrow A_1$, $B \rightarrow B_1$, $A \subset W$, $A_1 \subset W'$, $B \subset Z$ and $B_1 \subset Z'$. If W, Z can be combined (i.e., $Y = W \cup Z$), they share n-1 concepts in common (i.e., there exists a subset X s.t. $W = X \cup A_1$, $Z = X \cup B_1$ and $Y = X \cup B_1 \cup A_1$), and the n-sized subsets of Y where B_1 and A_1 are combined will be missing. These sets, however, can be shown to be minimal because they are of the kind $X' \cup B_1 \cup A_1$, (where $X' \subset X$). By hypotheses, X

is not a key (otherwise, W and Z would have not been minimal sets). Prop. 4 is guaranteed because the fd-ancestors of Y are of the kind: $X \cup A \cup B_1$ (i.e., $W \cup B_1$) and $X \cup A_1 \cup B$ (i.e., $Z \cup A_1$). Since W and Z are keys they fulfill prop. 4 (notice that, in this case, the fd-ancestors are not minimal, but it is sound since they are not generated as *feasible keys* in our algorithm). Finally, prop. 3 can be shown by contradiction. Let X be a $(n+1)$-sized combination containing two concepts A, A_1 such that $A \rightarrow A_1$ and $Y = X$ - $\{A\}$ - $\{A_1\}$. Thus, either A is a key (and then A_1 was generated by ISR) or $A \cup Y'$ (where $Y' \subset Y$) is a key (and then $A_1 \cup Y'$ was generated by ISR). Thus, the set being a key would be missing and according to prop. 2, X would have not been generated (since it is not minimal). □

5 Case Study

In this section we introduce results got after carrying out our algorithm over the EU-Car Rental case study [9]. This ontology refers to a car renting domain [2] and it has 65 concepts and 170 relationships. For each concept, we computed its asserted fd's (from knowledge asserted in the ontology) and later, by means of algorithm introduced in section 3, its *feasible keys*. As result, each concept has an average of 31'83 concepts in the fd-tree (i.e., in our algorithm we have an average searching space of $2^{31'83}$ combinations). Among concepts in the fd-tree, an average of 6'67 are root concepts (i.e., the average value of combinations we start with in the first iteration of our algorithm). When computing keys we get an average of 156'53 *feasible keys* per concept (i.e., we will query the database 156'53 times in average; in front of the 3.817.550.246 times if we would have generated all the combinations in the searching space -i.e., $2^{31'83}$-). In general, considering all the queries posed to the database for all the concepts we have a total of 10.018 queries, of which a 15% are answered by querying the catalog (see section 3.1) and the rest, by queries over data. The reader will notice that this is the minimum number of queries we must pose in order to find all the keys of the domain (see section 2). Furthermore, it is interesting to realize that about the 30% of n-sized *feasible keys* generated are generated by combining (n-1)-sized *candidate sets* whereas the rest are generated by ISR. The execution time of our algorithm is insignificant in front of the cost of querying data. Indeed, all the *feasible keys* for all the concepts were generated in less than 10 seconds and in general, our algorithm behaves much better than traditional approaches.

The computer used in these test was equipped with an Intel Core 2 Duo 1.33 GHz processor, 1'99 GB of RAM. With respect to software, we used the Java SKD v1.4.2 for implementing the algorithm described in this paper and the Protégé-OWL API [8] for loading and handle the domain ontology.

[2] The whole ontology is available in OWL DL notation at www.lsi.upc.edu/∼ oromero/EUCarRental.owl

6 Related Work

To our knowledge, this is the first work addressing this issue at the conceptual level. We may find many works for computing functional dependencies and keys (e.g., [6,7,11,12] among others), but they work at the logical or data level and they share some inherent constraints. A logical schema is tied to the design decisions made when devising the system (for instance, denormalization of data) and these decisions either made to fulfill the system requirements (for instance, improve query answering, avoid update / insertion / deletion anomalies, preserve features inherited from legacy systems, etc.) or naively made by non-expert users have a big impact on the data semantics captured in the schema. Therefore, to avoid missing some important data dependencies, these approaches make some unrealistic assumptions such as completeness of the data structures (i.e., all the constraints of the domain of interest are captured at the logical level). Addressing this issue at the data level though, may imply such an effort that may not be worth. If the logical schema lacks semantics, we may need to sample data at such a low level (for instance, testing any possible combination of attributes) that for large number of attributes or instances it would rise an unaffordable computational complexity. Moreover, these approaches exclusively work over instances and they cannot easily tolerate erroneous data (that may generate fake fd's / keys that do not hold or overlook real ones). Unlike these approaches, we use ontological knowledge to guide the search and we do not exclusively rely on data. It has two main benefits. On one hand, we rely on a clear picture of the domain of interest free of logical / physical design decisions and we are able to considerably improve the performance by reducing drastically the number of hypotheses to be verified with data. On the other hand, this approach may be used for assuring the quality of data, as the feasible keys identified for each concept are extracted from knowledge in the domain ontology. Thus, it opens new perspectives and we may use this knowledge for detecting erroneous data in the database (e.g., those feasible keys refuted as keys that should be verified).

7 Conclusions

In this paper we have proposed an algorithm for generating meaningful keys from domain ontologies. In our approach, we take advantage of knowledge captured in the ontology to generate key hypotheses that are later verified with data. Unlike previous approaches that addressed this task at the data level, we take advantage of ontological knowledge that allows to better depict and prune the searching space. As consequence, our approach does not completely rely on data and it opens new perspectives for data quality processes. We have shown that our algorithm is sound and complete with regard to knowledge captured in the domain ontology and data, and we have presented the feasibility of our method by means of the statistics raised by the implementation of our algorithm over a case study.

As future work, we plan to adapt the algorithm to look for keys for all the domain concepts at the same time instead of performing this task one by one.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5), 2001.
3. D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Path-based identification constraints in description logics. In *Proc. of 11th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 231–241. AAAI Press, 2008.
4. P. P.-S. S. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
5. E. F. Codd. *The Relational Model for Database Management, Version 2*. Addison-Wesley, 1990.
6. J. Demetrovics and V. D. Thi. Some Remarks On Generating Armstrong And Inferring Functional Dependencies Relation. *Acta Cybern.*, 12(2):167–180, 1995.
7. P. A. Flach and I. Savnik. Database Dependency Discovery: A Machine Learning Approach. *AI Commun.*, 12(3):139–160, 1999.
8. S. C. for Biomedical Informatics Research. Protégé-OWL API. http://protege.stanford.edu/plugins/owl/api/.
9. L. Frías, A. Queralt, and A. Olivé. EU-Rent Car Rentals Specification. Technical report, 2003.
10. M. Lenzerini. Data integration: A theoretical perspective. In *Proc. of 21th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 233–246. ACM, 2002.
11. W. M. Lim. Discovery of Constraints from Data for Information System Reverse Engineering. In *Proc. of the 2nd Australian Soft. Eng. Conf.*, pages 39–48. IEEE, 1997.
12. M. Kantola, H. Mannila, K.-J. Räihä, and H. Siirtola. Discovering Functional and Inclusion Dependencies in Relational Databases. *Int. J. of Intelligent Systems 7*, pages 591–607, 1992.
13. A. Maydanchik. *Data Quality Assessment*. Technic Publications, 2007.
14. A. Olivé. On the role of conceptual schemas in information systems development. In *Proc. of 9th Int. Conf. on Reliable Software Technologies (Ada-Europe 2004)*, volume 3063 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004.
15. A. Poggi, D. Lembo, D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *Journal on Data Semantics*, 10:133–173, 2008.
16. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 2003.
17. O. Romero and A. Abelló. Automating Multidimensional Design from Ontologies. In *Proc. of ACM 10th Int. Workshop on Data Warehousing and OLAP*, pages 1–8. ACM, 2007.
18. O. Romero, D. Calvanese, A. Abelló, and M. Rodríguez-Muro. Discovering Functional Dependencies from Ontologies. Technical report, 2009, http://www.lsi.upc.edu/∼techreps/files/R09-10.zip.
19. T. M. T. Halpin. *Information Modeling and Relational Databases*. Morgan Kauffman, 2008.
20. R. Wieringa and W. de Jonge. Object Identifiers, Keys, and Surrogates: Object Identifiers Revisited. *Theory and Practice of Object Systems*, 1(2):101–114, 1995.