Explaining Query Answer Completeness and Correctness with Minimal Pattern Covers

Fatma-Zohra Hannou, Bernd Amann, Mohamed-Amine Baazizi

Explaining Query Answer Completeness and Correctness with Minimal Pattern Covers

Fatma-Zohra Hannou
Sorbonne Université, CNRS, LIP6
Paris, France
Fatma.Hannou@lip6.fr

Bernd Amann
Sorbonne Université, CNRS, LIP6
Paris, France
Bernd.Amann@lip6.fr

Mohamed-Amine Baazizi
Sorbonne Université, CNRS, LIP6
Paris, France
Mohamed-Amine.Baazizi@lip6.fr

ABSTRACT

Information incompleteness is a major data quality issue which is amplified by the increasing amount of data collected from unreliable sources. Assessing the completeness of data is crucial for determining the quality of the data itself, but also for verifying the validity of query answers over incomplete data. While there exists an important amount of work on modeling data completeness, deriving this completeness information has not received much attention. In this work, we tackle the issue of efficiently describing and inferring knowledge about data completeness w.r.t. to a complete *reference data set* and study the use of a *pattern algebra* for summarizing the completeness and validity of query answers. We describe an implementation and experiments with a real-world dataset to validate the effectiveness and the efficiency of our approach.

PVLDB Reference Format:
Fatma-Zohra HANNOU, Bernd AMANN and Mohamed-Amine BAAZIZI. Explaining Query Answer Completeness and Correctness with Minimal Pattern Covers. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Information incompleteness is a major data quality issue that is exacerbated by the growing number of applications, collecting data from distributed, open, and unreliable environments. Sensor networks and data integration are significant examples in which data incompleteness naturally arises due to hardware or software failures, data incompatibility, missing data access authorizations etc. In all these situations, querying and analyzing data can lead to deriving partial or incorrect answers. Extensive effort has been devoted to representing and querying incomplete databases [11, 8, 3, 7, 13, 16]. The common characteristics of these approaches is the use of some intensional or extensional information about completeness for deciding whether a query returns complete answers and, in some cases, for annotating the query answers with some completeness meta-data. Despite these efforts, reasoning about data completeness remains tricky due to the complexity of exhaustively representing and deriving information about available and missing data in large datasets.

In many situations, datasets and query results are explicitly or implicitly depend on some complete reference (or master) datasets to describe their expected full extent. For instance, sensor databases are usually construed within a spatio-temporal reference delimiting the coverage of the captured data. It is also believed according to [9] that 80 % of enterprises maintain master data with their analytical databases (customers informations, product) . In other data-centric applications, a reference is defined by domain experts during database design and updated when necessary. Finally, it may also sometimes be useful to use an existing table or query result as a reference for deriving a comprehensive representation about available and missing information in some specific context.

Representing information completeness. To understand the importance of using reference datasets for assessing data completeness, consider the database in Table 2 which depicts an example of a sensor database. The table **Energy** reports on daily energy measurements for some locations specified by `floor` (`fl`) and `room` (`ro`). This database is endowed with a spatial reference **MAP** describing all locations in some building and a calendar **CAL** indicating the expected temporal coverage (Table 1). Observe that both reference data sets are not necessarily validated master data but might have been built by an expert for a specific analysis task.

Table 1: Reference tables

MAP

fl	ro
f1	r1
f1	r2
f2	r1

CAL

we
w1
w2

×

da
Mon
Tue

For various reasons, the current database misses some values that are pinpointed in grayscale in the **Energy** table. Assume that an analyst wants to gain a full knowledge about the segments of the data that are available or missing. To facilitate her understanding of the data, the analyst would like a *summarized* version of the completeness information and may opt for a *pattern-based* representation like the one presented in Table 3. This figure shows two tables $\mathbf{P_E}$ and

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

Table 2: Data table

Energy	fl	ro	we	da	kWh
t_0	f1	r1	w1	Mon	10
t_1	f1	r1	w1	Tue	12
t_2	f1	r1	w2	Mon	10
m_0	f1	r1	w2	Tue	$\perp$
t_3	f1	r2	w1	Mon	8
t_4	f1	r2	w1	Tue	10
m_1	f1	r2	w2	Mon	$\perp$
m_2	f1	r2	w2	Tue	$\perp$
t_5	f2	r1	w1	Mon	12
t_6	f2	r1	w1	Tue	7
t_7	f2	r1	w2	Mon	8
t_8	f2	r1	w2	Tue	8

$\overline{\mathbf{P_E}}$ capturing the available and the missing information of table **Energy** respectively. More exactly, table $\mathbf{P_E}$ contains pattern tuples that capture all partitions which are complete w.r.t. the reference. For instance, pattern tuple p_0 indicates that all measurements pertaining to week $w1$ are available, whatever the values of `floor, room` or `day` are. Pattern p_3 in table $\overline{\mathbf{P_E}}$ reports that no measure can be found for `room` $r2$ and `week` $w2$. This representation is *compact* as it only reports on the largest possible partitions that are complete (resp. missing) in the data. It is also *covering* as it reports on every possible maximal complete (resp. missing) partition of the data. Without this presentation, the analyst might have to issue several queries without any guarantee of deriving an exhaustive information about completeness that the pattern representation offers in a rather natural manner.

Table 3: Completeness pattern tables of Energy

$\mathbf{P_E}$	fl	ro	we	da
p_0	*	*	w1	*
p_1	f2	*	*	*
p_2	f1	r1	*	Mon

$\overline{\mathbf{P_E}}$	fl	ro	we	da
p_3	*	r2	w2	*
p_4	f1	r1	*	Tue

Analyzing information completeness. Pattern tables can become very large due to randomness of missing information and may not be easily analyzed by hand. Querying pattern tables with SQL turns out to be a convenient means for extracting and reasoning about data completeness. In our example, if an analyst wants to identify the floors where all measures are available, she could issue the following query on $\mathbf{P_E}$ and notice that, since p_1 is the only pattern satisfying the predicate of Q_0, the only floor satisfying her criteria is $f2$.

Q_0: **select** fl **from** $\mathbf{P_E}$
where ro='*' **and** we='*' **and** da='*'

Querying pattern tables have another interesting application when it comes to extracting completeness information of query answers obtained from incomplete input tables. One solution consists in evaluating a query over the data and in extracting afterwards the completeness of the query answer from the corresponding reference dataset. However, extracting completeness patterns is costly and it often is more effective to derive the completeness patterns of query answers directly from the patterns of the input data. To illustrate this idea, consider Q_1 which retrieves all measures referring to week $w2$:

Q_1: **select** * **from Energy where** we = 'w2' ;

By evaluating Q_1 over $\mathbf{P_E}$ and $\overline{\mathbf{P_E}}$ respectively, we obtain Table $Q_1(\mathbf{P_E})$ and table $Q_1(\overline{\mathbf{P_E}})$: Q_1 produces a complete answer for floor $f2$ while the first tuple in $Q_1(\overline{\mathbf{P_E}})$ indicates that no answer is returned for room $r2$ during week $w2$:

$Q_1(\mathbf{P_E})$

fl	ro	we	da
f2	*	*	*
f1	r1	*	Mon

$Q_1(\overline{\mathbf{P_E}})$

fl	ro	we	da
*	r2	w2	*
f1	r1	w2	Tue

The pattern completeness model can also play a crucial role for validating the correctness of aggregation queries answers. When such queries are applied on incomplete data, the values resulting from aggregating incomplete partitions are simply incorrect and there is means to notify this fact to the user. To illustrate the role of the pattern model in detecting potential problems with aggregation queries, consider Q_2 which sums the energy consumption over all `day` values.

Q_2: **select** fl , ro , we, **sum**(kWh) **as** kWh
from Energy
group by fl , ro , we

This query returns both valid and non-valid answers produced by complete and incomplete partitions respectively. Incomplete partitions producing incorrect results can easily be identified by patterns for which the value of the attribute `day` is a constant instead of $*$. These "incompleteness" patterns are separated from the correct partitions in $Q_2(\mathbf{P_E})$ and $Q_2(\overline{\mathbf{P_E}})$.

$Q_2(\mathbf{P_E})$

fl	ro	we
*	*	w1
f2	*	*
f1	r1	*

$Q_2(\overline{\mathbf{P_E}})$

fl	ro	we
*	r2	w2
f1	r1	*

Annotating query results. Completeness pattern tables also can be used for rewriting aggregation queries to automatically annotate the produced results. For example, Table 4 shows the annotated result of query Q_2 where completeness information is directly extracted from P_2 and $\bar{P}_2$.

Table 4: Annotated Query Answer

Result Q_2	fl	ro	we	kWh	annot
	f1	r1	w1	22	ok
	f1	r1	w2	10	incorr
	f1	r2	w1	18	ok
	f2	r1	w1	19	ok
	f2	r1	w2	16	ok

This result can be obtained by rewriting Q_2 into a union of two queries separating the correct and incorrect answers. For example, for generating the subset of correct results generated by complete partitions, we use the following query rewriting for Q_2 :

$Q_{complete}$:

```
select fl, ro, we,
       sum(kWh) as kWh, 'ok' as annot
  from Energy d natural join P p
 where (d.fl = p.fl or p.fl = '*')
   and (d.ro = p.ro or p.ro = '*')
   and (d.we = p.we or p.we = '*')
   and p.da = '*'
 group by fl, ro, we
```

The partition of incorrect and incorrect results can be generated by changing the last filter into p.da != '*'. In a similar way, it is straightforward to to define a SQL query which generates the set of missing answers by joining the reference data set with the missing pattern table.

The last use-case shows another benefit of completeness patters when joining data tables. Consider that an analyst wants to exploit the result of Q_2 to derive the consumption per square meter. To do so, she join the result of Q_2, which has been materialized in the table **EnergyWeek**, with the table **Surface**, which contains the surface of room $r1$ on both floors.

```
Q3: select fl, ro, we, kwh/M2 as kWh_m2
    from EnergyWeek natural join Surface;
```

In this case, the completeness pattern table $Q_3(\mathbf{P_E}, \mathbf{P_S})$ of Q_3 can be generated by joining the pattern table $Q_2(\mathbf{P_E})$ and the pattern table $\mathbf{P_S}$ of table **Surface**.

$\mathbf{P_S}$

fl	ro
*	r1

$Q_3(\mathbf{P_E}, \mathbf{P_S})$

fl	ro	we
*	r1	w1
*	r1	w1

Contributions. In this article we are interested in reasoning with relative information completeness in general and in analyzing query answers over incomplete datasets. This setting is of interest for many practical situations where data analysts need to assess the quality of complex queries. Our main contributions are the following:

- a new data completeness model based on the notion of *minimal completeness pattern cover* for summarizing relative completeness information ;
- a new sound and complete pattern algebra extending the relational algebra with two fundamental operators, allowing for generating and transforming completeness pattern covers ;
- an implementation and an experimental evaluation on a real-world sensor dataset, on top of a standard relational DBMS.

Paper Outline. The rest of the article is structured as follows. Section 2 discusses related work. Section 3 introduces the pattern model as well as the notions of completeness and correctness of SQL queries. The pattern algebra and some applications of pattern queries are presented in Section 4. Section 5 describes our solution for processing and optimize pattern queries using standard relational database technology and presents two algorithms for generating pattern tables. The experimentations presented in Section 6 evaluate our approach on a real-world sensor dataset.

2. RELATED WORK

Modeling Information Completeness. Information completeness is a major data quality issue that received attention in several contexts [12, 1]. In the database context, data completeness usually addresses the question of query answer completeness under two different settings: the Closed-World-Assumption (CWA) which considers that the database contains all tuples while some of them might have *null* values [6], and the open-world assumption (OWA) which considers, in addition to the possibility of attributes with *null* values, tuples which are missing in the database.

Under the OWA setting, some line of work considers using reference database DB_C which can be either *virtual* or *materialized* to capture the full extent of data. A standard way for representing the completeness of a database DB w.r.t. a *virtual* database DB_C consists in defining complete information as views over DB_C [11, 8, 13]. Deciding whether a query is complete then relies on determining whether this query can be rewritten into queries that can be answered on DB_C using the defined views.

Our work is more reminiscent to the *relative information completeness* line of work using *materialized* reference (master) datasets [3]. In this work, given a database DB and a master database DB_C, deciding whether DB is *complete* for a query Q resorts to finding a set of containment constraints V of the form $q(DB) \subseteq p(DB_C)$ where q is a query on DB and p is a projection on DB_C. The complexity bounds obtained for different languages used for expressing queries and containment constraints demonstrate the difficulty of reasoning about information completeness [3]. Our pattern tables can be considered as exhaustive sets (disjunctions) of conjunctive containment constraints. Our goal within this setting is not only to decide if the answer is complete, but also to compute all containment constraints (patterns) satisfied by the query answer.

[16] introduces *m-tables* (inspired from *c-tables* [6]) for representing completeness information and an algebra over m-tables for annotating query answers with certainty information. Our completeness patterns can be assimilated to extended tuples in the m-tables model, by substituting m by $*$. This theoretical model is more expressive than our pattern based model, but also has more practical issues than our model which can efficiently be implemented by using standard relational database technology.

Pattern-based Completeness Models and Algebras. The seminal work of [11] suggests a model based on meta tuples that describe data integrity (completeness and correctness) constraints. Meta relations are similar to our pattern tables, where meta tuples are used to define available, valid and invalid data and to encode logical views over virtual complete and correct data tables. Query completeness checks if there exists a rewriting of the query using only complete views. Another idea of this early work is the definition of an algebra that manipulates meta tuples for producing sound (but not complete) sets of meta tuples satisfied by an input query.

More recently, [13] presents an approach which consists in associating completeness patterns to data tables and an algebra for querying patterns to produce query answer completeness information. From this work we adopt the approach of using patterns and of defining an algebra to ma-

nipulate these patterns. In [13] completeness patterns describe the extent of data completeness as views over a *virtual* complete database, whereas we suppose that completeness of a data table or query answer is automatically assessed w.r.t. a *materialized* reference table. This introduces an additional practical and semantic dimension for analyzing quality issues of data and query results related to information incompleteness.

The work in [7] analyzes different types of partial result anomalies engendered by data incompleteness. The data completeness model distinguishes between cardinality (incomplete, phantom, indeterminate) and correctness (credible and non-credible) anomalies at different granularity levels (input, operator, column, partition). The authors also study how these anomalies are propagated within a query plan. We follow the same approach regarding completeness propagation using operators at the granularity of partition (down to individual tuples). We derive raw completeness informations from reference data whereas [7] derives completeness information from observed physical access anomalies.

Query Result Correctness. In our setting, correctness does not deal with the validity of data tuples w.r.t. logical constraints as in [11, 8], but is more related to the concept of *summarizability* [10]. The notion of summarizability was first introduced by [14] in the context of statistical databases, where it refers to the correct computation of aggregate values with a coarser level of detail from aggregate values with a finer level of detail. One of the summarizability conditions defined in [14] is *completeness* which checks if all elements in a cluster (coarser level) exist and are attached to some cluster. In our setting, this mainly corresponds to the constraint that the partitions (clusters) generated by the `group-by` clause of an analytical query are complete. As we we will show in Section 4.2, our pattern model and algebra also allows us to identify and filter incomplete partitions.

Provenance and explanations. Many existing work deals with deriving explanations for missing answers [5, 4] or with answering why-not questions [17, 2]. These works assume the data to be complete and focus on understanding the behaviour of queries rather than on evaluating the impact of incomplete data on the completeness and the validity of queries with aggregation.

3. DATA MODEL

3.1 Reference tables and completeness

The main extension of our data model with respect to the relational data model is the possibility to define *reference tables* for representing completeness constraints over *data tables*.

Definition 1. Let D and R be two relational tables such that D contains all attributes A of table R. Table R is called a *reference table* for *data table* D with *reference attributes* A and the pair $T = (D, R)$ is called a *constrained table*.

Example 1. The Cartesian product $\mathbf{MAP} \times \mathbf{CAL}$ of tables **MAP** and **CAL** in Table 1 is a reference table of **Energy** with reference attributes $A = \{fl, ro, we, da\}$.

Observe that any table $S(\underline{A}, M)$ with key A and *with null values for attribute* M can be decomposed into a constrained table $\Delta(S) = (D, R)$ where measure table $D \subseteq S$ contains all tuples in S *without null values* and $R = \pi_A(S)$ contains all key values in S. Similarly, we can build from any constrained table $T = (D, R)$ a relational table $\Gamma(T) = R \ltimes D$ with *null* values such that $\Delta(\Gamma(T)) = T$.

Definition 2. A constrained table $T = (D, R)$ with reference attributes A is *complete* iff $R \subseteq \pi_A(D)$.

Example 2. The constrained table T=(**Energy**, R) in Table 2 is not complete for $R = \mathbf{MAP} \times \mathbf{CAL}$ but it is complete for R=$\sigma_{we='w1' \wedge ro='r1'}(\mathbf{MAP} \times \mathbf{CAL})$

3.2 Minimal pattern covers

In this section we introduce the notion of minimal completeness pattern cover as a comprehensive description of all complete and empty data partitions in a constrained table.

Definition 3. Let $A = \{a_1, a_2, ..., a_n\}$ be a set of reference attributes where the domain of each attribute is extended by a distinguished value $*$ called wildcard. A *(completeness) pattern* $p = [a_1 : v_1, a_2 : v_2, ..., a_n : v_n]$ over A is a tuple which assigns to each reference attribute $a_i \in A$ a value $v_i \in dom(a_i) \cup \{*\}$ in the extended domain of a_i. A set of completeness patterns $P(A) = \{p_1, p_2, \ldots, p_k\}$ over a set of reference attributes A is called a *pattern table*.

In the following we will denote by $[*]$ the *wildcard pattern* where all attributes are assigned to wildcards. Observe that a pattern table might contain only data tuples, i.e. pattern tuples without any wildcards. Completeness patterns are part of a generalization/specialization hierarchy defined as follows.

Definition 4. A pattern p_1 *generalizes* a pattern p_2 if p_1 can be obtained from p_2 by replacing zero or more constants by wildcards. Inversely, p_1 *specializes* p_2 if p_1 can be obtained from p_2 by replacing zero or more wildcards by constants.

We can show that this generalization/specialization hierarchy forms a semi-lattice with the wildcard pattern as top-element and data tuples as bottom elements.

Definition 5. The *instance* $I(p, S)$ of a pattern p in some table S is the *subset* of tuples $t \in S$ which are specializations of p.

The instance $I(p, S)$ of a pattern $p = [a_1 : v_1, a_2 : v_2, ..., a_n : v_n]$ in some table S can be computed by a relational selection $I(p, S) = \sigma_{cond}(S)$ with filtering condition $cond = \bigwedge(a_i = p.a_i \vee p.a_i = *)$. It is then easy to show the following properties of pattern instances:

- $I([*], S) = S$;
- $I(p, I(p, S)) = I(p, S)$;
- $S \subseteq S' \Rightarrow I(p, S) \subseteq I(p, S')$.

The notion of instance can naturally be extended from patterns to pattern tables P and constrained tables $T = (D, R)$: $I(P, S) = \bigcup_{p \in P} I(p, S)$ and $I(p, T) = (I(p, D), I(p, R))$. The following definition relates constrained tables to pattern tables:

Definition 6. A constrained table $T = (D, R)$ *satisfies* a completeness pattern p, denoted by $T \models p$, if $I(p, R) \subseteq I(p, D)$. A constrained table T *satisfies* a completeness pattern table P if T satisfies all patterns in P.

It is easy to show that a constrained table T is *complete* if it satisfies wildcard pattern $[*]$.

Definition 7. A pattern p_2 *subsumes* a pattern p_1, denoted by $p_1 \sqsubseteq p_2$, if *for all* constrained tables T: $T \models p_2 \Rightarrow T \models p_1$.

PROPOSITION 1. *$p_1 \sqsubseteq p_2$ if and only if p_1 is a specialization of p_2. (see proof in the Appendix)*

In the following, we define several properties and relationships connecting pattern tables to constrained tables which are necessary to define the final notion of *minimal pattern cover*.

Definition 8. A pattern table P *covers* a constrained table T iff *for all patterns* p satisfied by T there exists a pattern $p' \in P$ subsuming p.

Example 3. Pattern table $\mathbf{P_E}$ in Table 3 covers the constrained table $T = (\mathbf{Energy}, \mathbf{MAP} \times \mathbf{CAL})$. When replacing $p_0 = [*, *, w_1, *]$ by two patterns $p_a = [f_1, *, w_1, *]$ and $p_b = [f_2, *, w_1, *]$ this is not true anymore, since pattern $p_0 = [*, *, w_1, *]$ is satisfied by T but not subsumed by any pattern in $P - \{p_0\} \cup \{p_a, p_b\}$.

Observe that a pattern table P covering a constrained table T is not necessarily satisfied by T. In particular, any pattern table containing the universal pattern covers all constrained tables T.

Definition 9. A pattern table P *strictly covers* a constrained table T if P covers T and $P \not\models T$.

Definition 10. A pattern table P is *reduced* if there exists no pair of distinct patterns $p \in P$ and $p' \in P$ such that p is a generalization of p'.

PROPOSITION 2. *For each constrained table T, there exists a unique reduced strict cover $P^*(T)$ called the* minimal pattern cover *of T. (see proof in the Appendix)*

Example 4. Pattern table $\mathbf{P_E}$ in Table 2 is the minimal pattern cover of constrained table $T = (\mathbf{Energy}, \mathbf{MAP} \times \mathbf{CAL})$.

3.3 Reasoning with minimal covers

Completeness patterns define data partitions and minimal pattern covers can be used for reasoning about the completeness of these partitions. For example, by the definition of minimal cover, we can show for all constrained tables $T = (D, R)$ where $R \neq \emptyset$ and all patterns $p \in P^*(T)$ in the minimal cover of T that:

- the instance $I(p, D)$ is *complete* and *not empty*.
- the instances $I(p', D)$ of all specializations p' of p are complete (but might be empty) and
- the instances $I(p', D)$ of all generalizations $p' \neq p$ of p are incomplete and not empty.

Similarly, let $\overline{T} = (\overline{D}, R)$ denote the *complement* of T where $\overline{D}$ contains all tuples "missing" in D w.r.t. R and $P^*(\overline{T})$ be the minimal cover of $\overline{T}$. Then as before, we can show for all $T = (D, R)$ where $R \neq \emptyset$ and all patterns $p \in P^*(\overline{T})$ in the minimal cover of $\overline{T}$:

- the instance $I(p, D)$ is *incomplete* and *empty*.
- the instances $I(p', D)$ of all specializations p' of p are empty (but might be complete) and
- the instances $I(p', D)$ of all generalization $p' \neq p$ of p are incomplete and not empty.

We introduce the following attributes for characterizing some completeness pattern p:
- E : $I(p, D)$ is empty;
- N : $I(p, D)$ is not empty;
- C : $I(p, D)$ is complete;
- I : $I(p, D)$ is incomplete;

For example, a pattern p is an E-pattern, if its partition is empty. Then, an IE-pattern denotes a partition which is incomplete and empty, *i.e.* missing.

Example 5. Consider a minimal cover P and its complement $\overline{P}$ given in Table 5. Figure 1 is a tree representation

Table 5: A pattern table and its complement

P

A	B	C
a2	*	*
a1	b1	*
*	b1	c1
a1	b3	c2

$\overline{P}$

A	B	C
a4	*	*
a1	b2	*
a3	*	c2
*	*	c4

of patterns where each node at some level i in the tree corresponds to a pattern with i constant attributes. The wildcard pattern $[*]$ is the root, the first level corresponds to all patterns $[a_i, *, *]$, $[*, b_j, *]$ and $[*, *, c_k]$ with one attribute etc. The leftmost node at the second level denotes pattern $[a1, b1, *] \in P$. All patterns in P are labeled CN (in blue) and all patterns in $\overline{P}$ are labeled IE (missing, in red). All ancestors of both kind of patterns nodes are N-patterns. Pattern $[a1, *, *]$ is incomplete (I), pattern $[a2, b1, *]$ is C (complete) and $[a4, b1, *]$ is IE, *i.e.* missing.

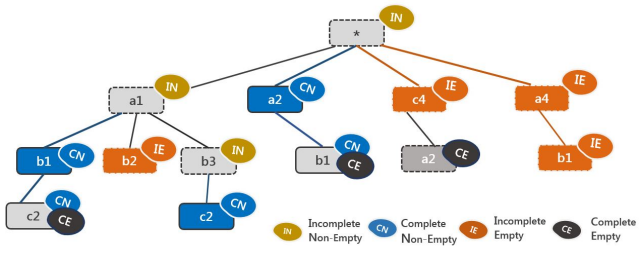

Figure 1: Labeled completeness pattern lattice.

4. COMPLETENESS PATTERN ALGEBRA

4.1 Pattern queries and algebra

Let $T = (D, R)$ be a constrained table and Q be a relational query which can be applied to D and R. Our goal is to define a set of operators which allow us to compute the minimal cover $P^*(T')$ of the result of $T' = Q(T) = (Q(D), Q(R))$. Suppose that there exists an operator $\triangleright$ computing the minimal cover of some constrained table T. Then it is possible to obtain the minimal cover $P^*(T')$ of T' by applying $\triangleright$: $P^*(T') = \triangleright(Q(D), Q(R))$ (see red dashed lines in Figure 2). An alternative way is to rewrite $Q(D)$ into a new query $\hat{Q}(P, R)$ over a strict (not necessarily minimal) cover $P(T)$ of constrained table T such that $\hat{Q}$ takes as input the couple (P, R) and produces the new couple $(P^*(T'), Q(R))$ (see blue solid line in Figure 2).

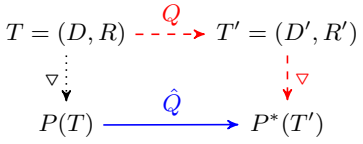

Figure 2: Pattern queries

In the following we will show that it is possible to extend the relational algebra RA with two operators *unfold* ($\triangleleft$) and *fold* ($\triangleright$) for transforming strictly covering (but not necessarily reduced) pattern tables and to use this extended algebra $RA_{ext} = RA \cup \{\triangleright, \triangleleft\}$ to define a new *pattern algebra* RA_{patt} over pattern tables.

Definition 11. The *unfold* operator $\triangleleft_A(P, R)$ generates for a given pattern table P and reference table R an *equivalent* pattern table $P' \equiv_R P$ where all values of attributes $a_i \in A$ are constant values.

The unfolding $\triangleleft_A(P, R)$ of a pattern table P on some attribute set A w.r.t. its reference table R can be defined by the following relational algebra expression:

$$\triangleleft_A(P, R) = \pi_{R.A, P.\neg A}(P \bowtie_{\theta_\triangleleft} R) \quad (1)$$

where $\theta_\triangleleft = \bigwedge_{a_j \in A}(P.a_j = * \vee P.a_j = R.a_j)$ for all attributes in A and $\pi_{R.A,P.\neg A}$ denotes the projection on attributes A of R and on all attributes of P except A.

Example 6. Consider the following pattern table P and the reference of Table 1. Unfolding a minimal pattern table does not preserve minimality. For example, the third pattern subsumes the second one in the result.

P

fl	ro	we	da
*	*	w1	*
f2	*	*	*
f1	r1	*	Mon

$\triangleleft_{\{fl\}}(P, R)$

fl	ro	we	da
f1	*	w1	*
f2	*	w1	*
f2	*	*	*
f1	r1	*	Mon

Operator fold $\triangleright_{a_i}$ is the inverse operator of $\triangleleft_{a_i}$ and *generalizes*, when possible, all subsets S of patterns $p \in S$ which are equal for all attributes except for attribute a_i into a single pattern $p_{a_i:*}$ with a wildcard value for attribute $a_i = *$:

Definition 12. The *fold* operator $\triangleright_{a_i}(P, R)$ generates for a given pattern table P and reference table R an *equivalent* pattern table $P' \equiv_R P$ where there exists no pattern $p_{a_i:*}$ and subset $S \subseteq P'$ of specializations p of $p_{a_i:*}$ where $p_{a_i:*} =' *'$ and $p_{a_i:*}$ is equivalent to S: $\nexists p_{a_i:*}, S \subseteq P' : p_{a_i:*} = * \wedge \{p_{a_i:*}\} \equiv_R S$.

To compute $\triangleright_A(P, R)$ we first compute the set $M = R - I(P, R) = R - D$ of all reference tuples missing in D[1]. Then, let $p_{a_i:*}$ denote the pattern obtained from p by replacing the constant value v_i of attribute a_i in p by a wildcard $*$. The semi-join expression $\overline{G}(a_i) = \sigma_{a_i \neq *}(P) \ltimes_{\theta_\triangleright} M$ where $\theta_\triangleright = \bigwedge_{i \neq j}(P.a_j = * \vee P.a_j = M.a_j)$ returns all patterns p in P which *cannot* be generalized on a_i: condition $\theta_\triangleright$ is true for all patterns $p \in P$ where the $p_{a_i:*}$ is incomplete (its instance in M is not empty). Then $G(a_i) = \sigma_{a_i \neq *}(P) - \overline{G}(a_i)$ returns the set of patterns p where $p_{a_i:*}$ is complete and we can define the folding operator as follows:

$$\triangleright_{a_i}(P, R) = \sigma_{a_i = *}(P) \cup \overline{G}(a_i) \cup \{[a_i : *]\} \times \pi_{\neg a_i}(G(a_i))$$

Observe that if there is no pattern $p \in P$ where a_i has a constant value, then $\overline{G}(a_i)$ and $G(a_i)$ are empty and $\triangleright_{a_i}(P, R) = P$.

Example 7. Consider the pattern table P' below.

P'

fl	ro	we	da
f2	*	*	*
f1	r1	*	Mon
*	r1	w1	*
*	r2	w1	*

Observe that

- $\sigma_{ro=*}(P') = \{[f2, *, *, *]\}$,
- $\overline{G}(ro) = \{[f1, r1, *, Mon]\}$ and
- $G(ro) = \{[*, r1, w1, *], [*, r2, w1, *]\}$ and thus $\{[a_i : *]\} \times \pi_{\neg a_i}(G(a_i)) = \{[*, *, w1, *]\}$.

Therefore, $\triangleright_{ro}(P', R)$ returns $\mathbf{P_E}$ in Table 3.

As for unfold, the fold operation is associative and can be generalized on a set of attributes $A = \{a_1, a_2, ..., a_n\}$:

$$\triangleright_A(P, R) = \begin{cases} P & \text{for } A = \emptyset \\ \bigcup_{a_i \in A_h}(\triangleright_{a_i}(\triangleright_{A - a_i}(P, R), R)) & \text{otherwise} \end{cases} \quad (2)$$

In the following, $\triangleleft(P, R)$ (unfold all) and $\triangleright(P, R)$ (fold all) will denote the unfold and fold operations over *all* reference attributes in P (and R). It is easy to show that $\triangleright(\triangleleft(P, R), R) = P$ and $\triangleleft(\triangleright(D, R), R) = D$

Using the extended relational algebra RA_{ext}, we can now define a *pattern algebra* RA_{patt} which consists in defining for each relational operator $op \in = \{\sigma, \pi, \bowtie, \cup, \cap, -\}$ its counterpart $\hat{op} \in \{\hat{\sigma}, \hat{\pi}, \hat{\bowtie}, \hat{\cup}, \hat{\cap}, \hat{-}\}$. Let P and P' be two strict covers of constrained tables $T = (D, R)$ and $T' = (D', R')$. Then we define the following pattern algebra $RA_{patt} = \{\hat{\sigma}, \hat{\pi}, \hat{\bowtie}, \hat{\cup}, \hat{\cap}, \hat{-}\}$ where each operator $\hat{op}$ is defined by using its relational counterpart op and operators $\triangleright$ and $\triangleleft$:

$$\hat{op}(P) = \triangleright(op(\triangleleft(P, R)), op(R)) \quad (3)$$

$$P \,\hat{op}\, P' = \triangleright(\triangleleft(P, R)\, op\, \triangleleft(P', R'), R\, op\, R') \quad (4)$$

[1]Remind that since P is a strict cover $I(P, R) \subseteq D$.

THEOREM 3. *RA_{patt} is sound and complete.*

PROOF. *We show that for all relational operators $op \in \{\sigma, \pi, \bowtie, \cup, \cap, -\}$, constrained tables $T = (D, R)$ and $T' = (D', R')$ with strict covers P and P' respectively, equations 5 and 6 are true:*

$$\hat{op}(P) = \triangleright(op(D), op(R)) \quad (5)$$

$$P \,\hat{op}\, P' = \triangleright(D \,op\, D', R \,op\, R') \quad (6)$$

For proving soundness and completeness we use the two equalities $P = \triangleright(D, R)$ and $D = \triangleleft(P, R)$;
Soundness:

$$\hat{op}(P) = \triangleright(op(\triangleleft(P, R)), op(R)) = \triangleright(op(D), op(R))$$

$$P \,\hat{op}\, P' = \triangleright(\triangleleft(P, R) \,op\, \triangleleft(P', R'), R \,op\, R')$$
$$= \triangleright(D \,op\, D', R \,op\, R')$$

The first equality in both equations is obtained by definitions 3 and 4, respectively.
Completeness:

$$\triangleright(op(D), op(R)) = \triangleright(op(\triangleleft(P, R)), op(R)) = \hat{op}(P) \quad (7)$$

$$\triangleright(D \,op\, D', R \,op\, R') = \triangleright(\triangleleft(P, R) \,op\, \triangleleft(P', R'), R \,op\, R') \quad (8)$$

$$= P \,\hat{op}\, P' \quad (9)$$

The last equality in both equations is obtained by definitions 3 and 4, respectively. □

Safe projection. Using the extended relational algebra, we also can define a specific operator called *safe projection* which filters all complete partitions before projection. Let A_π denote the attributes removed by some projection. Then $\theta_\pi = \bigwedge_{a_i \in A_\pi}(a_i = *)$ filters all patterns which are incomplete for attributes A_π. The *safe projection* operator $\hat{\pi}^*$ first folds all patterns over the attributes which are projected out and filters all incomplete dimensions before projection. This guarantees that the result only contains patterns corresponding to partitions which were complete w.r.t. the removed attributes:

$$\hat{\pi}^*_{\neg A_\pi}(P, R) = (\pi_{\neg A_\pi}(\sigma_{\theta_\pi}(\triangleright_{A_\pi}(P, R))), \pi_{\neg A_\pi}(R)) \quad (10)$$

Observe also that if P is already a minimal cover, projection produces a minimal cover without requiring a final fold operation.

4.2 Pattern query examples

The extended relational algebra and the pattern algebra can be used to define queries on pattern tables. Let $T = (D, R)$ be a constrained table with a strict cover P over some reference attributes A.

Generating pattern and dimension tables. By definition, the *complete* unfolding $\triangleleft(P, R)$ of P generates table D. On the other hand, the *complete* folding $\triangleright(P, R)$ of P generates the *minimal cover* of T. Observe that we also can obtain the same minimal cover by a complete folding $\triangleright(D, R)$ of D.

Querying pattern tables. As mentioned in Section 3.3, it is possible to check the completeness and emptiness of partitions in T using only information available in the minimal cover P and its complement $\overline{P}$. It is easy to show that simple selection over pattern tables is sufficient to check if a given pattern p exists in some pattern table P or is a specialization/generalization of a pattern $p' \in P$. For example pattern $[a5, *, *]$ is incomplete and not empty (IN) or complete and empty (CE) since query $\sigma_{A='a5'}(P \cup \overline{P})$ is empty (see Section 3.3).

Identifying complete partitions. Consider, for example the SQL query with reference table $R(fl, ro, we, da)$:

```
select fl, we, avg(khw) from D
 group by fl, we
```

Let P be a strict cover of constrained table $T = (D, R)$. The completeness patterns of the result can be computed by the projection $\hat{\pi}_{fl,we}(P)$ whereas the safe projection presented in Section 4.1 $\hat{\pi}^*_{fl,we}(P)$ retrieves the patterns of all partitions with correct result.

Identify missing information in joins. Folding can be used to analyze completeness issues in join queries and to identify missing information in input data tables. For example, consider a natural join $T \bowtie S$ between two data tables T and S sharing a set of attributes A. Then, $\triangleright(\pi_A(T), \pi_A(S))$ returns a compact representation of all tuples in T that can be joined with S whereas $\triangleright(\pi_A(T) - \pi_A(S), \pi_A(T))$ returns a compact representation of the tuples in S that are *missing* with respect to T in order to produce a complete result.

5. PATTERN QUERY PROCESSING

5.1 Pattern query optimization and execution

As shown in Section 4.1 unfolding $\triangleleft$ can directly be translated into the relational algebra, whereas folding $\triangleright$ over a set of attributes needs recursion which is not expressible in relational algebra (see Section 5.3 for implementations of $\triangleright$). Based on these observations, it is possible to rewrite any pattern query *without folding* into a relational query over pattern tables and reference tables. We will illustrate this by two examples with selection and projection.

Example 8. Let $T = (D, R)$ be a constrained table strictly covered by a pattern table P. Let $\sigma_{\theta_\sigma}(D)$ be a selection with a filtering predicate θ_σ using only reference attributes. The following pattern selection generates a minimal cover for the result of Q:

$$\hat{\sigma}_{\theta_\sigma}(P, R) = \triangleright(\sigma_{\theta_\sigma}(\triangleleft(P, R)), \sigma_{\theta_\sigma}(R)) \quad (11)$$

Unfolding is necessary to check the existence of tuples in pattern instances. For example, in order to check if a pattern $p = (a_1 : v_1, \ldots, a_i : *, \ldots, a_n : v_n)$ satisfies a filtering condition $a_i{=}c_i$, p must be unfolded on attribute a_i. As shown before, the final fold operation cannot be translated into relational algebra without recursion. However the subexpression composed of a selection and an unfold can be translated into the relational algebra and optimized using standard relational query rewriting. Starting from the filtering expression $Q = \sigma_{\theta_\sigma}(\triangleleft(P, R))$ we can apply following rewriting steps to obtain a more optimal expression in relational algebra:

- replace $\triangleleft(P, R)$ by $\triangleleft_A(P, R)$ unfolding P only over the filtering attributes $A = \{a_i, ..., a_k\}$ used in the filtering condition θ_σ;

- replace $\triangleleft_A(P, R)$ by its relational definition and push the selection condition θ_σ into the unfold rewriting:

$$Q = \pi_{R.A,P.\neg A}(P \bowtie_{\theta_\triangleleft} (\sigma_{\theta_\sigma}(R)))$$

where $\theta_\triangleleft = \bigwedge_{a_j \in A}(P.a_j = * \vee P.a_j = R.a_j)$

Finally, if the condition is a conjunction of equality predicates $a_i = c_i$, all pattern attributes $a_i \in A$ can safely be replaced by $a_i : *$ and join can be replaced by semi-join:

$$Q = (\{[A : *]\} \times (\pi_{P.\neg A}(P \ltimes_{\theta_\triangleleft} (\sigma_{\theta_\sigma}(R))), \sigma_{\theta_\sigma}(R)))$$

The final query expression can directly be translated into SQL.

Example 9. Consider the following more complex query expression over the same constrained table (D, R):

$$Q = \pi_{fl,ro,we,da}(\sigma_{fl='f1'}(D))$$

Let P be a strict (not necessarily minimal) cover of D and $R = \mathbf{MAP} \times \mathbf{CAL}$. By applying similar rewriting steps as before, we can obtain the following pattern query which generates a strict cover of the constrained query result $(Q(D), Q(R))$:

$$\{fl : *\} \times \pi_{P.ro,P.we,P.da}(P \bowtie_{cond} (\sigma_{fl='f1'}(\mathbf{MAP})) \quad (12)$$

The corresponding SQL query is the following:

```
select '*' as fl, p.ro, p.we, p.da
from P join MAP on
    (P.fl='*' or P*.fl=MAP.fl)
  and (P.ro='*' or P.ro=MAP.ro)
where MAP.fl='f1'
```

Observe that the SQL query only refers to table **MAP** for unfolding attribute fl. Reference table **CAL** is *independent* of **MAP** and can be ignored. This point is discussed in more detail in the following section.

5.2 Independent reference tables

Fold ($\triangleright$) and unfold ($\triangleleft$) comprise costly joins with reference tables. In many real world settings, reference tables $R = R_1 \times R_2 \times ... \times R_n$ are defined by the Cartesian product of independent reference tables R_i corresponding to spatial, temporal and other dimensions. These reference tables R_i are obviously much smaller than the generated reference table R and introduce optimization opportunities for reducing unfolding/folding costs.

Definition 13. Let P be a pattern table with a reference table $R = R_1 \times R_2 \times ... \times R_n$. The unfolding of a pattern table P on some attribute set $A = A_1 \cup A_2 \cup ... \cup A_k$ where A_i are non-empty subsets of attributes of sub-table R_i (*wlg.* unfolding is done over the first k reference tables R_i) is defined as follows:

$$\triangleleft_A(P, R) = \pi_{R_i.A_i,P.\neg A}(P \bowtie_{\theta^1_\triangleleft} R_1 \bowtie_{\theta^2_\triangleleft} R_2 \bowtie_{\theta^3_\triangleleft} ... \bowtie_{\theta^k_\triangleleft} R_k)) \quad (13)$$

where $\theta^i_\triangleleft = \bigwedge_{a_j \in A_i}(P.a_j = * \vee P.a_j = R_i.a_j)$ and $\pi_{R_i.A_i,P.\neg A}$ denotes the projection on attributes A_i of R_i and on all attributes of P except A.

Observe that $\triangleleft_A$ only joins with reference tables R_i which contain at least one attribute $a \in A$.

Definition 14. The folding of a pattern table P on some attribute a_i of a reference table R_j is defined as in Definition 12, except that the missing tuples can directly be computed from the reference table R_j without considering the other tables: $M = R_j - \pi_{A_j}(D)$.

Then, similarly to unfold, the fold operator $\triangleright_A$ only needs to access reference tables R_j which contain at least one attribute $a_i \in A$.

PROPOSITION 4. *If all attribute domains are independent and the input pattern tables are minimal covers, selection with equality, projection and Cartesian product can be expressed using the relational algebra (without $\triangleleft$) and generate minimal covers.*

PROOF. *The proof directly follows from the definitions of selection with equality, projection and Cartesian product.* □

5.3 Folding algorithms

As shown in Section 4.1 the fold operator can be implemented by an iterative evaluation of standard SQL queries including joins with reference tables (Equation 2). This section will present two optimized folding algorithms. The first algorithm *FoldData* computes minimal covers for *data* tables and the second algorithm *FoldPatterns* directly folds *pattern* tables into minimal covers without a preliminary unfolding step.

5.3.1 Folding data

Algorithm *FoldData* (Algorithm 1) computes for a given constrained table $T = (D, R)$ a strict cover $P^*(T)$ following a set of attributes A. If A is the set of all attributes in T, *FoldData* produces the minimal cover of T. The algorithm explores the data table by searching for complete partitions. It starts from the most general pattern *i.e.* wildcard pattern $[*]$ (level 0) and explores *top-down* and *breadth-first* the pattern subsumption lattice L_D generated by the active attribute domains in the data table D. Each level l corresponds to all patterns p with l constants. For checking if some pattern p is satisfied by D, the algorithm compares the cardinality of p in D and R using standard SQL queries (we assume that the projection of D on the reference attributes is included in R). After each level, all specializations of the found complete patterns p are by definition also complete and the tuples covered by p can be pruned from D before executing the next level. Algorithm *FoldData* uses the following functions:

- $powerSet(A, N)$ produces all subsets of A of cardinality N.
- $patterns(A, D)$ produces for a set of attributes A, all patterns $\pi_A(D) \times \{[*]\}$
- $checkComp(p, D, R)$ checks if $I(p, D) = I(p, R)$
- $prune(P, D)$ deletes from D all tuples satisfied by patterns $p \in P$.

Observe that operations *checkComp* and *patterns* can be implemented by standard SQL queries. In particular, *patterns* is a simple projection on D and *checkComp* can be implemented by comparing the result of two *count*-queries on D and R (we suppose that $D \subseteq R$). In the worst case,

Algorithm 1: Algorithm *FoldData*

Data: constrained table $T = (D, R)$, attribute set A
Result: minimal cover $P^*(T)$

```
1 P := ∅ ; for level := 0 to |A| do
2 |   X := ∅ ;
3 |   for B ∈ powerSet(A, level) do
4 |   |   for p ∈ patterns(B, D) do
5 |   |   |   if checkComp(p, D, R) then
6 |   |   |   |   P := P ∪ {p} ; X := X ∪ {p} ;
7 |   prune(X, D) ;
8 return P
```

FoldData explores (almost) the whole pattern lattice L_D that is generated by all attribute/value combinations in the data table. The number of patterns $size(L_D)$ of L_D depends on the active attribute domains in the data table D and is exponential in the number of attributes: $size(L_D) = \sum_{i=1}^{n}(C_n^i) * D_n$ where n is the number of attributes, and D_i is the maximum size of the Cartesian product of the active domain of i attributes in the data table. The size of the reference table influences the cost of checking pattern satisfaction. Such a worst case scenario corresponds to particular case of random missing data which generates large pattern tables without pruning opportunities. As we show in our experiments, real-world data generally follows more regular incompleteness schemes which increase the compression rate and folding performance.

5.3.2 Folding pattern tables

Algorithm *FoldData* operates exclusively on data tables and cannot be used to fold *pattern* tables without a preliminary complete unfold. This unfolding obviously loses the compression of pattern tables, in particular for pattern tables with a high compactness ratios.

A pattern table P is not minimal for two main reasons. First, it might not be reduced, *i.e.* it contains two patterns p_1 and p_2 such that $p_1 \sqsubset p_2$. Second, it might not be a cover, *i.e.* there might exist a subset of patterns $S \subseteq P$ which could be merged into a single equivalent pattern $p \notin P$. For example [f1,r1,w1,$*$] and [f1,r2,w1,$*$] from P can be merged into [f1,$*$,w1,$*$] $\notin P$.

Algorithm 2 treats these issues separately. The *merge* step (lines 1 to 11) first solves the second issue and checks if the instance $I(S, R)$ of a subset $S \subseteq P$ is equivalent to the instance $I(p, R)$ of a pattern $p \notin P$. The algorithm explores the pattern lattice *bottom-up* starting from the most specialized pattern (at the lowest level) and by recursively merging sets S of patterns which differ only on the constant of one attribute. As soon as S can be merged into one pattern p, the latter is added to P (it might be merged with a higher level pattern at the next iteration). Notice that merged patterns are removed only after all level merges are performed (line 11), because one pattern can take part in several pattern merges. For example, [f1,r1,w1,$*$] can merge first with [f1,r2,w1,$*$] to generate [f1,$*$,w1,$*$], and merge a second time with [f2,r1,w1,$*$] to produce [$*$,r1,w1,$*$]).

The first outer loop performs the merge and uses the following functions:

- $getPatt(P, level)$ returns all patterns $p \in P$ with $level$ constant attributes.
- $isGen(p1, p2)$ checks if $p1$ is a generalization of $p2$ (Definition 4)
- $gen(p, a)$ generalizes pattern p by replacing the constant attribute a by a wildcard.
- $getConstAttrs(p)$ finds all constant attributes of pattern p.
- $checkComp(X, p, T, R)$ checks if the instance of pattern set X in data table T is equal to the instance of pattern p in R.
- $getSimPatt(P, p, a)$ returns all patterns in P which differ from p by a different constant value for attribute a.

To deal with the first issue, Algorithm *FoldPatterns reduces* P by removing all remaining patterns $p_1 \in P$ which specialize another pattern $p_2 \in P$. This can be done by a simple auto-join on P (lines 12 to 17).

Algorithm 2: Algorithm *FoldPatterns*

Data: pattern table P, reference table R, data table T, attribute set A
Result: minimal cover $P^*(I(P, R))$

```
1  for level := |A| to 0 do
2  |   S_level = ∅
3  |   for p ∈ getPatt(P, level) do
4  |   |   for a ∈ getConstAttrs(p) do
5  |   |   |   p_{a_i:*} := gen(p, a)
6  |   |   |   S := getSimPatt(P, p, a)
7  |   |   |   if p_{a_i:*} ∉ P then
8  |   |   |   |   if checkComp(S, p_{a_i:*}, T, R) then
9  |   |   |   |   |   P := P ∪ {p_{a_i:*}}
10 |   |   |   |   |   S_level := S_level ∪ S
11 |   P := P − S_level
12 for level1 := 0 to |A| do
13 |   for p1 ∈ getPatt(P, level1) do
14 |   |   for level2 := level1 + 1 to |A| do
15 |   |   |   for p2 ∈ getPatt(P, level2) do
16 |   |   |   |   if isGen(p1, p2) then
17 |   |   |   |   |   P := P − {p2}
18 return P
```

In the worst case, the reduce step generates $O(|P|^2)$ generalization tests. Similar to the top-down algorithm *FoldData*, the size $size(L_P)$ of the pattern lattice L_P explored by the merge step can be estimated by $size(L_P) = \sum_{i=1}^{n}(C_n^i) * D_i$ where n is the number of attributes, and D_i is the maximum size of the Cartesian product of the active domain of i attributes in the *pattern* table. Since the size D_i over the pattern table P is in general much smaller than the size D_i over the data set D, computing the minimal cover for P (without unfold) is in general much more efficient than computing the minimal cover on the data set D. This observation is confirmed by our experiments (see Section 6)

PROPOSITION 5. *Algorithm FoldPatterns is correct.*

PROOF. *We can show that a pattern p can only be generated if there exists an attribute a and a subset of patterns $S \subseteq P$ such that p generalizes all patterns in S on attribute a and S is equivalent to p. Then by following a recursive*

bottom-up strategy we guarantee that all possible generalizations are tested, which preserves the pattern covers equivalence. The reduce step guarantees minimality, by browsing the pattern lattice and filtering all possible specializations. □

6. EXPERIMENTATION

The goal of our experiments are manifold: (i) testifying on the effectiveness of patterns in compactly representing completeness, (ii) analyzing the performance of the pattern derivation algorithms (Algorithms 1 and 2) and (iii) studying the efficiency impact of the pattern algebra.

We ran our experiments on a standard Linux machine equipped with a 2.4 GhZ dual core CPU, 8GB of RAM and 350 GB of standard storage. The algorithms are implemented in Python 2.6 whereas data and patterns were managed in PostgresSQL [15] and accessed using the psycopg2+ library of Python. We did not define any indexes to accelerate filters and joins.

6.1 Datasets

We use both a *real-world* and a *synthetic* dataset. The real-world dataset corresponds to sensor measurements of different kinds (electricity, heating,...) collected during one year at our University campus. This dataset features both spatial and temporal incompleteness since not at all parts of the campus are covered and many of the sensors operate erratically. The synthetic dataset is generated from the real one by introducing more randomness for the purpose of studying the impact of data distribution on pattern compactness.

We restrict on measures pertaining to temperatures collected in 12 out of 96 buildings and refer to this data with **Temp**. We build two reference tables with different spatial coverage and identical temporal span. The first reference, noted R_{All}, includes all spatial locations of the campus regardless of the existence of temperature sensors. The second reference, noted R_{Temp}, restricts on localities equipped with a *temperature* sensor, that is, localities present in **Temp**. The schema of the data and the reference tables together with their sizes are reported in 6.

Temp($\underline{building, floor, room, year, month, day, hour}, value$)

Loc($\underline{building, floor, room}$) **Cal**($\underline{year, month, day, hour}$)

$Sch(R_{All}) = Sch(R_{Temp}) = Sch(\mathbf{Loc}) \cup Sch(\mathbf{Cal})$

Table 6: Size of reference tables R_{all} and R_{Temp}

variant x	$\lvert Loc_x\rvert$	$\lvert Cal_x\rvert$	$\lvert R_x\rvert = \lvert Loc_x\rvert \times \lvert Cal_x\rvert$
all	10,757	8,760	94,231,320
Temp	2,810	8,760	24,615,600

Intuitively, the choice of the reference has an impact on the derivation of completeness patterns in terms of efficiency and effectiveness. We start by investigating the effectiveness by studying the variation of the compaction ratio when varying the size of the data and of its associated reference. To do so, we build two smaller data tables by restricting **Temp** *spatially*, by selecting only the measures of one building (e.g. Building 25), and *temporally*, by keeping the measures pertaining to one month out of 12 (e.g. January). The resulting tables are respectively noted with **T_OneBlg** and **T_OneMon**. Their cardinalities are reported in Table 7 together with their completeness ratio $CR(ds, R) = |ds|/|R_x|$ w.r.t. their references R_x. We denote by R^{ds}_{All} and R^{ds}_{Temp} the reference tables obtained by using the same spatial or temporal restriction of the dataset ds. For example, $R^{T\_OneBlg}_{All} = \sigma_{building='25'}(R_{All})$.

Table 7: Sizes and completeness ratio.

dataset ds	$\lvert ds\rvert$	$CR(ds, R^{ds}_{all})$	$CR(ds, R^{ds}_{Temp})$
Temp	1,321,686	1.4%	5.36%
T_OneBlg	341,640	21.43%	21.43%
T_OneMon	88,536	1.4%	4.23%

As expected, the closer data is to its reference, the better is the completeness ratio. We observe that the spatial restriction allows for achieving the highest completeness ratio (21.43%).

6.2 Pattern table generation

We perform a preliminary experiment to verify the effectiveness of patterns in terms of compactness and the efficiency of Algorithm *FoldData*. The *compactness* of a pattern table P is defined by the ratio $|P|/|D|$ between the size of P and the size of its data table. We consider all combinations of datasets and references and report the results in Table 8.

Table 8: Pattern derivation: preliminary results.

			Execution Time (sec)	
dataset ds	$\lvert P\rvert$	Comp.	$P(ds, R_{all})$	$P(ds, R_{Temp})$
Temp	11,269	8.5×10^{-3}	36,620	5,983
T_OneBlg	39	1.1×10^{-4}	45	45
T_OneMon	119	1.3×10^{-3}	90	75

Observe that using a more precise reference (spatial restriction) leads to decreasing execution time for **Temp** and **T_OneMon** due to reduced completeness check time. The results also confirm that the reference choice impacts the dataset completeness and is a determinant factor for the execution time.

Variability analysis. Since the compactness of a pattern table plays a major role in the performance of the query result explanation process, we investigate the problem of determining which dataset features lead to high compactness. We examine three features : (i) distribution of completeness, (ii) dataset completeness and (iii) dataset size. We use in this experiment both the real-world dataset and the synthetic one which is obtained by randomly inserting or deleting measures starting from real-world dataset.

In both cases, we generate a series of datasets, by increasing and decreasing the completeness ratio. Starting from datasets with a fixed completeness ratio, we build two series of datasets: one obtained by successively inserting tuples from the reference until reaching full completeness, another one obtained by successively deleting its tuples until reaching emptiness. The insertion and deletions follow two strategies: i) a sequential strategy which selects the (inserted or deleted) tuples using their spatial and temporal domain order preserving the original data distribution and which we call *real distribution*, and ii) a random strategy which picks

these tuples in a random fashion, we call *simulated distribution*.

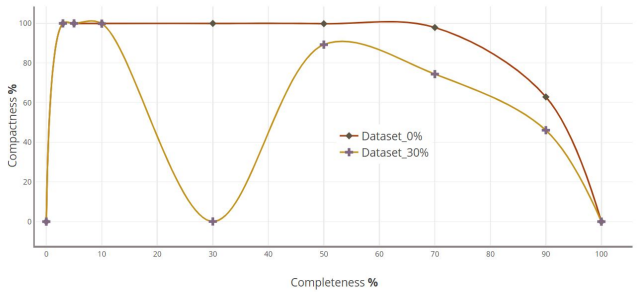

Figure 3: Synthetic datasets : Data missing randomly

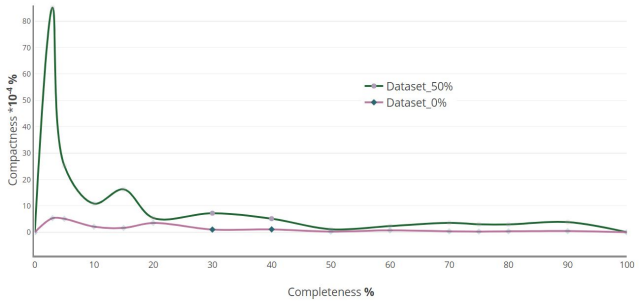

Figure 4: Real datasets : missing data following sensor failures

Figures 3 and 4 depict the evolution of compactness w.r.t. completeness for each distribution and each of the synthetic dataset. In the synthetic datasets (Figures 3), the compactness of a random dataset with 30% completeness evolves symmetrically in both directions (insertion and deletion): successive insertions/deletions generate/remove tuples which give raise to new patterns. At some point, these insertions/deletions will cause the merging of fine-grained patterns to coarser-grained ones increasing the compactness ratio to achieve maximum compactness at both extremities. In the real datasets we observe the same trend with a lower amplitude for a dataset with 50% initial completeness: insertions lead to a faster completion of the partial partitions (thanks to order sensitive updates) and thus to faster derivation of coarser patterns without deriving all their subsumed patterns.

Finally, notice that the completeness distribution is a direct factor impacting compactness. Different completeness ratios lead to variable pattern numbers, the more a dataset is complete, the fewer the patterns.

Performance. In the following experiment we evaluate the performance of algorithm *FoldData*. From the original dataset Temp, we derived 30 datasets grouped into three categories, each with approximately the same completeness rate, but different dataset sizes. Figure 5 shows the running time of *FoldData* for all datasets according to the number of generated patterns. Categories are represented by points of different colors (*orange* = 15%, *violet* = 10% and *green* = 3% completeness rate). Notice that execution time is not impacted by the data completeness but grows exponentially with the number of generated patterns.

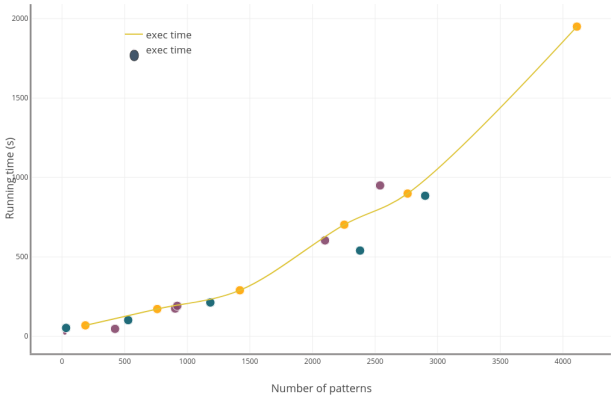

Figure 5: *FoldData* performance

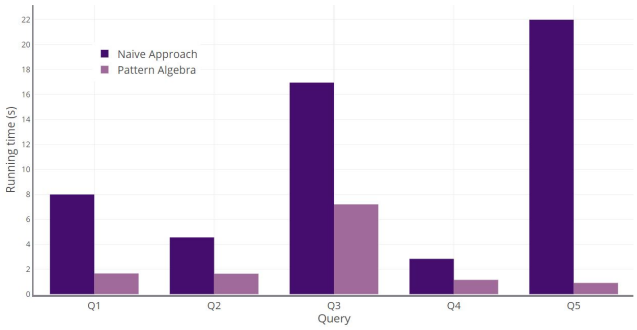

Figure 6: Pattern algebra performance

6.3 Pattern Query Processing

The following experiment measures the efficiency of processing pattern queries for producing minimal covers for queries over constrained tables. We compare the pattern-based query plans (blue solid path in Figure 2) using the techniques described in Section 5 by comparing it with the "naive" strategy of computing the minimal cover from the results of the query applied to the data and reference tables (red dashed path in Figure 2). We tested both approaches on the queries below and report the result in Figure 6.

Q_1:	$\sigma_{b=2223}(Temp)$	Q_2:	$\sigma_{b=2223 \wedge f=1}(Temp)$
Q_3:	$\sigma_{b=2223 \wedge (m=11 \vee m=12)}(Temp)$	Q_4:	$\pi_{b,f,r,y,m,d}(Temp)$
Q_5:	$\pi_{f,r,m,d}(Temp)$	Q_6:	$Temp \bowtie_b Elec$

Assessing the completeness of queries using pattern algebra outperforms the naive approach for all of the tested queries. For Q_1 and Q_2 efficiency is obtained by exploiting the fact that both queries use only the spatial reference (reference independence). For Q_3, where the gain is less important, partial unfolding over both reference tables incurs some overhead while remaining reasonably low. Queries Q_4 and Q_5 need no unfolding which explains their performance gain with respect to the naive solution. For Q_5, pattern-based evaluation is much more efficient because of the pattern cover compactness ($11K$ tuples). Note also that this

particular pattern query doesn't need unfolding, in contrast with Q_4. We observe the same trend with join query Q_6 (not shown in the figure) which takes 31 seconds to be executed and where the pattern algebra query only takes 7.9 seconds. The pattern join implies a partial unfold on attribute b with a small number of values (96b) and generates a small number of patterns which have to be refolded.

Folding pattern query results. The last experiments set aims at digging deeper in the efficiency of pattern queries by analyzing the overhead of $FoldPatterns$. We consider different pattern table sizes,to minimize. Running the $FoldPatterns$ algorithm produces pattern minimal covers with variable compactness values (see Table 9). We report the $FoldPatterns$ phase execution time while keeping track of the exact number of merge and reduce operations (see section 5.3).

As expected, the running time grows with the number of patterns to minimize and merging patterns is much more expensive than reducing patterns which is a purely syntactic operation.

Table 9: Pattern Fold algorithm performances

P. size	P_{min}.size	Compac.	time	**m**erges /**r**educes
106	22	20.75%	0.29s	7 $\boldsymbol{m}$
238	32	13.44%	0.32s	9$\boldsymbol{m}$+ 79 $\boldsymbol{r}$
570	30	5.2%	0.38s	45$\boldsymbol{m}$
992	864	87%	0.47s	6 $\boldsymbol{m}$ + 32 $\boldsymbol{r}$
10961	3921	35.77%	1.33s	6 $\boldsymbol{m}$ + 7040 $\boldsymbol{r}$
11285	11178	99.01%	0.35s	107 $\boldsymbol{r}$
12054	11440	94.90%	6.59s	38 $\boldsymbol{m}$ + 158 $\boldsymbol{r}$

7. CONCLUSION

In this paper, we presented a pattern-based approach for representing and summarizing relative completeness information. We proposed a formal model and characterized a powerful reasoning mechanism for inferring and analyzing exhaustive sets of completeness statements about data and query answers. We validated our approach experimentally and confirm the efficiency of the pattern algebra and its usefulness in evaluating query completeness and correctness.

Extending the model with statistical information about data completeness is a challenging future direction. A natural extension under study is the use of a map-reduce platform like Apache Spark [18] to compute minimal pattern covers and implement the pattern algebra. Developing an interactive tool for reasoning on patterns is an interesting application that we are considering.

APPENDIX

A. PROOFS

LEMMA 6. $p_1 \sqsubseteq p_2 \Rightarrow \forall S : I(p_1, S) \subseteq I(p_2, S)$

PROOF. *We show that if there exists a table S where $I(p_1, S) \not\subseteq I(p_2, S)$, then $p_1 \not\sqsubseteq p_2$. For showing $p_1 \not\sqsubseteq p_2$, we define a constrained table $T = (D, R)$ such that $I(p_2, R) \subseteq I(p_2, D)$ and $I(p_1, R) \not\subseteq I(p_1, D)$. Let $R = S$ and $D = I(p_2, R)$. Then, $I(p_2, D) = I(p_2, I(p_2, R)) = I(p_2, R)$ (by idempotency). Now we have to show that $I(p_1, R) \not\subseteq I(p_1, D)$. Based on the initial assumption $I(p_1, S) \not\subseteq I(p_2, S)$ and $S = R$ we conclude $I(p_1, R) \not\subseteq I(p_2, R)$ and it is sufficient to show that $I(p_1, D) \subseteq I(p_2, R)$: $I(p_1, D) = I(p_1, I(p_2, R)) \subseteq I(p_2, R) = I(p_2, R)$.* □

PROOF OF PROPOSITION 1. We first prove that if $p_1 \sqsubseteq p_2$ then p_1 is a specialization of p_2. Suppose that p_1 is *not* a specialization of p_2, *i.e.* there exists no mapping from p_1 to p_2 such that p_2 can be obtained from p_1 by replacing one or more constants by a wildcard. In other terms, there exists an attribute a_i such that $p_2.a_i = c$ is a constant and $p_1.a_i \neq p_2.a_i$. This is equivalent to the statement that $cond(p_1)$ contains a condition $p_1.a_i = c$ which is not contained in $cond(p_2)$. Then it is easy to define a table $S = \{t\}$ where t satisfies p_1 but not p_2 which leads to $I(p_1, S) \not\subseteq I(p_2, S)$. On the other hand, by proposition 6, we know that $p_1 \sqsubseteq p_2$ implies $\forall S : I(p_1, S) \subseteq I(p_2, S)$ (contradiction).

We now show that if p_1 is a specialization of p_2, then $p_1 \sqsubseteq p_2$. If p_1 is a specialization of p_2, then for all S, $I(p_1, S) = I(p_1, I(p_2, S))$ (then filtering condition of p_1 is subsumed by the filtering condition of p_2). Then, if $I(p_2, R) \subseteq I(p_2, D)$ we know by monotonicity of I that $I(p_1, I(p_2, R)) \subseteq I(p_1, I(p_2, D))$ which is equivalent to $I(p_1, R) \subseteq I(p_1, D)$. □

PROOF OF PROPOSITION 2. By contradiction using the notion of cover and subsumption. Suppose that there exist two minimal strict covers $P^*(T)_1$ and $P^*(T)_2$. Then there exists a pattern $p_1 \in P^*(T)_1 - P^*(T)_2$ and a pattern $p_2 \in P^*(T)_2 - P^*(T)_1$ such that $p_1 \sqsubseteq p_2$ (otherwise $P^*(T)_2$ would not be a cover). Since $p_1 \neq p_2$ and by Proposition 1, we can conclude that $p_1 \sqsubset p_2$. By Definition 8 there must exist a third pattern $p_1' \in P^*(T)_1$ such that $p_2 \sqsubseteq p_1'$ (otherwise $P^*(T)_1$ would not be a cover). Then, we obtain $p_1 \sqsubset p_2 \sqsubseteq p_1'$ where p_1 and p_1' are two distinct patterns in $P^*(T)_1$ and p_1' subsumes p_1. This is in contradiction with the claim that $P^*(T)_1$ is a minimal cover. □

B. REFERENCES

[1] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques (Data-Centric Systems and Applications)*. Springer, 2006.

[2] N. Bidoit, M. Herschel, and A. Tzompanaki. Efficient computation of polynomial explanations of why-not questions. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 713–722. ACM, 2015.

[3] W. Fan and F. Geerts. Relative Information Completeness. *ACM Trans. Database Syst.*, 35(4):27:1–27:44, Oct. 2010.

[4] M. Herschel and M. A. Hernández. Explaining missing answers to spjua queries. *Proceedings of the VLDB Endowment*, 3(1-2):185–196, 2010.

[5] M. Herschel, M. A. Hernández, and W.-C. Tan. Artemis: A system for analyzing missing answers. *Proceedings of the VLDB Endowment*, 2(2):1550–1553, 2009.

[6] T. Imieliński and W. Lipski. Incomplete information in relational databases. In *Readings in Artificial Intelligence and Databases*, pages 342–360. Elsevier, 1988.

[7] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. In *International Conference on Management of Data, SIGMOD*, pages 1275–1286, Snowbird, USA, June 2014.

[8] A. Y. Levy. Obtaining Complete Answers from Incomplete Databases. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 402–412, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[9] D. Loshin. *Master data management*. Morgan Kaufmann, 2010.

[10] J.-N. Mazón, J. Lechtenbörger, and J. Trujillo. A survey on summarizability issues in multidimensional modeling. *Data & Knowledge Engineering*, 68(12):1452–1469, 2009.

[11] A. Motro. Integrity = Validity + Completeness. *ACM Trans. Database Syst.*, 14(4):480–502, Dec. 1989.

[12] L. L. Pipino, Y. W. Lee, and R. Y. Wang. Data Quality Assessment. *Commun. ACM*, 45(4):211–218, Apr. 2002.

[13] S. Razniewski, F. Korn, W. Nutt, and D. Srivastava. Identifying the extent of completeness of query answers over partially complete databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 561–576, Melbourne, Victoria, Australia, May 31 - June 4 2015.

[14] A. Shoshani. OLAP and statistical databases: Similarities and differences. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 185–196. ACM, 1997.

[15] M. Stonebraker and L. A. Rowe. The design of postgres. *SIGMOD Rec.*, 15(2):340–355, June 1986.

[16] B. Sundarmurthy, P. Koutris, W. Lang, J. F. Naughton, and V. Tannen. m-tables: Representing missing data. In *20th International Conference on Database Theory, ICDT*, pages 21:1–21:20, Venice, Italy, March 2017.

[17] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 15–26. ACM, 2010.

[18] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.