Identifying the Extent of Completeness of Query Answers over Partially Complete Databases

Simon Razniewski
Free University of Bozen-Bolzano
razniewski@inf.unibz.it

Flip Korn
Google Research
flip@google.com

Werner Nutt
Free University of Bozen-Bolzano
nutt@inf.unibz.it

Divesh Srivastava
AT&T Labs-Research
divesh@research.att.com

ABSTRACT
In many applications including loosely coupled cloud databases, collaborative editing and network monitoring, data from multiple sources or authors is regularly used for query answering. For reasons such as system failures, unspecified subjects or network issues, data may be temporarily unavailable or generally nonexistent. Hence, not all data needed for query answering may be available.

In this paper, we propose a natural class of completeness patterns, expressed by selections on database tables, to specify complete parts of database tables. We then show how to modify the operators of relational algebra so that they manipulate these completeness patterns to compute completeness patterns pertaining to query answers. Our proposed algebra is computationally sound and complete with respect to the information that the patterns provide. We show that stronger completeness patterns can be obtained by considering not only the schema but also the database instance and we extend the algebra to take into account this additional information.

We develop novel techniques to efficiently implement the computation of completeness patterns on query answers and demonstrate their scalability on real data.

1. INTRODUCTION
In many applications, data from multiple sources (tables, sites, authors, measurement systems, etc.) is used in query answering. Examples are loosely coupled cloud databases such as AzureDB [17], [16], crowd-sourced collaborative projects such as Wikipedia and warehouses collecting monitoring data such as Darkstar [3].

In all these scenarios, data may be incomplete. In the first case, network delays or the failure of data nodes may render data to be temporarily unavailable. In the second case, the open nature makes it possible that anyone can enter nearly any information, no matter how partial the entered information is. In the third case, partial integration of source information or operational failures of reporting systems may lead to the data warehouse having only partially complete information.

In these scenarios, the following mechanisms have been proposed: For distributed cloud databases, Lang et al. [17], [16] traced where issues due to incompleteness arise and are propagated through a query plan. For Wikipedia, a template was introduced that allows to label tables as complete. For data streams, it was proposed to deliver data only after so-called punctuations have arrived, that is, checkpoints that allow to conclude that all data up to a certain timepoint has been received [20],[12].

There are some obvious differences: for cloud databases, incompleteness occurs mainly due to delays and failures, while for Wikipedia the main cause is insufficient or biased author knowledge and for network databases it often arises from having multiple heterogeneous source feeds that need to be integrated. As a result, the languages used to specify the completeness are different. While the first approach uses units of computation (shards, columns, tables), the second uses natural-language statements and the third uses selection conditions on data streams.

Despite these differences, in common between all three scenarios is that it is aimed to specify which parts of a database are complete. In all three scenarios, knowledge about which parts of the query results are complete can help users decide whether the presented results meet their expectations, that is, whether their information needs can be satisfied with the available data. If this is not the case, users might try to consult other data sources, use historical data/statistical approaches to estimate missing data, or reformulate their queries.

In this paper we show how completeness information for parts of a database can be used to annotate query answers with information about which parts of the query answers are complete (or which aggregates are correct). Such information is valuable to users to understand the completeness and reliability of query answers. In particular, we provide methods to compute completeness information for query answers. The setting discussed here differs from the one commonly called incomplete databases (for a recent survey, see e.g. [19]). There, one focuses on the meaning and consequences of null values, that is, individual cell values in known records that are missing or incomplete. In contrast, we investigate the impact on query results of entire database records that are missing.

Network Data Warehouse Use Case. We consider a simplified database D_{most} of a network provider, for illustrative purposes. The database schema contains the following three tables:

- Warnings(day, week, ID, message)
- Maintenance(ID, responsible, reason)
- Teams(name, specialization)

The first table stores the warnings received from network elements, identified by an ID, together with a time stamp. The second stores the set of network elements that are currently in maintenance, together with the responsible teams and the reasons for
maintenance. The third stores the maintenance teams together with their specialization. We sometimes refer to these tables using only the first letter of their name.

A sample database using this schema is depicted in Table 1. Each table contains in addition to the actual records (rows with r) also a set of completeness patterns (rows with p).

Suppose that warnings are loaded daily from an operational system. Suppose the current date is Thursday, week 2. The reports from last week (week 1) have all been loaded, and from this week, so far the reports for Monday and Wednesday have been loaded, while not all the ones for Tuesday have arrived yet. Then the completeness of the data for week 1, and for Monday and Wednesday of week 2, are expressed by the completeness patterns (*, 1, *, *), (Mon, 2, *, *), (Wed, 2, *, *), which appear as records p1 to p3 in Table 1. Intuitively, these patterns assert that all possible records that describe real-world facts and that match these patterns (with "*" being a wildcard), are present in the database.

Suppose also that we have information about all network elements currently in maintenance with teams A, B and C, because their local systems automatically export data to the central database, while others do not. This information is represented by the completeness patterns p4 to p6 in Table 1.

We also know all maintenance teams in the company, with their specializations. This is pattern p7.

Suppose now we are interested in getting all warnings in week 2 for all elements that are in maintenance with a hardware team being responsible. This information can be obtained via the following SQL query $Q_{w=2}$:

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Warnings W} \\
\text{JOIN} & \quad \text{Maintenance M ON W.ID=M.ID} \\
\text{JOIN} & \quad \text{Teams T ON M.responsible=T.name} \\
\text{WHERE} & \quad W.week=2 \\
\text{AND} & \quad T.specialization='\text{hardware}'
\end{align*}
\]

This query can be expressed by different algebraic expressions, for instance the following:

\[
\sigma_{\text{week}=2}(W) \bowtie \delta_{\text{ID}=M.ID} (M \bowtie_{\text{M.responsible}=T.name} \sigma_{\text{spec}=\text{\text{\textquoteleft}}\text{hw}\text{\textquoteright}}(T)).
\]

When evaluating this expression over the database $D_{\text{maint}}$ we get the result shown in the first three rows in Table 2.

But can we also use the completeness information that we have about the database to learn about the completeness of the query result?

Let us look at a particular example, namely the first selection operation in the expression shown above, that is, the selection $\sigma_{\text{week}=2}(W)$. The pattern $p1 = (*, 1, *, *)$ does not say anything about the result of this selection because it talks about completeness of week 1 only.

The pattern $p2$, however, which is $(\text{Mon}, 2, *, *)$, tells us something about the result. Since we are selecting data only from week 2, and $p2$ says that all data from Monday week 2 is there, we can infer that the result will be complete for all data on Monday. Thus, we can infer that the pattern $(\text{Mon}, *, *, *)$ for week 2 holds. The same holds for the pattern $p3$. We summarize this information in Table 3.

We can proceed similarly for the selection by $\text{spec} = \text{\textquoteleft}\text{hw}\text{\textquoteright}$ on $\text{Teams}$, which preserves the pattern $(*, *)$. Subsequently joining the completeness patterns analogously to the base data, with the wildcard matching any constants, we arrive at the conclusion that the query result is complete for teams A, B and C on Monday and Wednesday. This can be represented using completeness patterns as shown in Table 4. Note that these conclusions have been drawn by applying algebraic operations only to the patterns $p1$ to $p7$.

Taking into account the interplay of query and state of the database, more can be concluded: Inspecting the database, one observes that teams A and B are the only known hardware teams, and that due to the completeness of the $\text{Teams}$ table, no other hardware teams can exist. Since the query asks only for hardware teams, it is therefore possible to summarize the six completeness patterns for the teams A, B and C into just two patterns where A, B and C have been replaced by a *. The result can be seen in Table 5. We will later refer to this summarization as pattern promotion.

An end user would therefore know that on Monday and Wednesday, the retrieved warnings for tw37 and tw83 are really the only ones that occurred. In contrast, for Tuesday, the user can see that a warning for tw37 occurred, but there is no assurance that this is the only warning on that day.
<table>
<thead>
<tr>
<th>W.day</th>
<th>W.week</th>
<th>W.ID</th>
<th>W.message</th>
<th>M.ID</th>
<th>M.responsible</th>
<th>M.reason</th>
<th>T.name</th>
<th>T.specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed</td>
<td></td>
<td>tw37</td>
<td>overheated</td>
<td>tw37</td>
<td>A</td>
<td>disk failure</td>
<td>A</td>
<td>hardware</td>
</tr>
<tr>
<td>Mon</td>
<td></td>
<td>tw83</td>
<td>high voltage</td>
<td>tw83</td>
<td>B</td>
<td>unknown</td>
<td>B</td>
<td>hardware</td>
</tr>
<tr>
<td>Tue</td>
<td></td>
<td>tw83</td>
<td>auto restart</td>
<td>tw83</td>
<td>B</td>
<td>unknown</td>
<td>B</td>
<td>hardware</td>
</tr>
<tr>
<td>Mon</td>
<td></td>
<td>*</td>
<td>*</td>
<td>A</td>
<td>*</td>
<td>A</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Mon</td>
<td></td>
<td>*</td>
<td>*</td>
<td>A</td>
<td>*</td>
<td>A</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Mon</td>
<td></td>
<td>*</td>
<td>*</td>
<td>C</td>
<td>*</td>
<td>C</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Wed</td>
<td></td>
<td>*</td>
<td>*</td>
<td>A</td>
<td>*</td>
<td>A</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Wed</td>
<td></td>
<td>*</td>
<td>*</td>
<td>B</td>
<td>*</td>
<td>B</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Wed</td>
<td></td>
<td>*</td>
<td>*</td>
<td>C</td>
<td>*</td>
<td>C</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Result of the query $Q_{hw}$ over the database $D_{maint}$ annotated with completeness information.

<table>
<thead>
<tr>
<th>name</th>
<th>country</th>
<th>City</th>
<th>state</th>
<th>county</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>USA</td>
<td>Virginia</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Germany</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Ukraine</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Bulgaria</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>USA</td>
<td>New York</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>UK</td>
<td>Carmarthenshire</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>USA</td>
<td>West Virginia</td>
<td>Hampshire County</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Czech</td>
<td>Moravian-Silesia</td>
<td>Novy Jicin</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Completeness statements for cities on Wikipedia

Previous approaches (see Section 2) would only be able to say that the query $Q_{hw}$ is not complete over the database $D_{maint}$.

Wikipedia Use Case. Wikipedia is an extreme example of a crowd-sourced data source, as a vast number of authors are contributing to articles. As a consequence, completeness of the extracted data is a big issue. To specify completeness of of data in various contexts such as enumerated lists, authors use either natural language statements, e.g. "This is a complete list of cities in the Commonwealth of Virginia in the United States" or a template to mark lists as complete. Currently, ~500 pages use the template and ~5000 pages use natural language expressions to state completeness.

Suppose we want to count the number of cities in certain countries. In the spirit of Google Fusion Tables [13] and DBpedia [3], we can scrape the data from Wikipedia to obtain lists of cities in various countries. But what quality guarantees do we find for this data? Table 1 shows the completeness statements that we found for cities on Wikipedia. If we were e.g. to count the number of cities in Bulgaria based on the Wikipedia data, then we can use the guarantee given by some author that Wikipedia really contains all those cities. In contrast, for France there is no completeness statement (and indeed the relevant article does not include cities with small populations).

Contributions. In the rest of this paper, we discuss how the intuitive conclusions above can be formalized and automated. The contributions are the following:

1. We introduce a natural class of completeness patterns that correspond to selections on database tables and can be expressed in the same schema as the normal data.
2. We define the pattern algebra that, given completeness patterns as input, provides metadata as output to identify the extent of completeness of the query answer over a partially complete database. This metadata is computed similarly to the way the query answer itself is computed and can be executed almost as efficiently.
3. We extend this algebra to additionally take into account the database instance, thus allowing one to compute more general completeness patterns (i.e., stronger assertions), but potentially requiring computation time exponential in the size of the data.
4. We develop techniques to efficiently implement the algebras presented and show their scalability on real data. Our experiments show that considering the database instance does indeed lead to stronger completeness assertions while avoiding exponential blow-up in practice.

Outline. This paper is structured as follows. In Section 2 we discuss related work, in Section 3 we formalize partially complete databases and completeness patterns, in Section 4 we present an algebra for completeness patterns, in Section 5 we extend this algebra to consider the state of the database instance, and discuss practical aspects of our framework in Section 6.

2. RELATED WORK

The classical paradigm for databases is the closed-world assumption [24], which asserts that the information in the database is complete. For data integration and on the Semantic Web, often the open-world assumption is preferred [1], where nothing is assumed about the completeness of the database. Open and closed-world semantics were first discussed by Reiter in [24], where he formalized that negation as failure [6] corresponds to standard negation under the closed world assumption.

Scenarios as the one discussed in our motivating example, however, require a middle ground. In such situations, parts of the database known to be complete are considered closed-world, while the incomplete parts are open-world. This intermediate paradigm, called the partial (or, local [7]) closed-world assumption, has not received as much attention so far.

The first formalization of this setting was by Motro [22], who showed how to derive the completeness of a query answer provided statements about the completeness of other query answers. Subsequently, Levy extended this idea by introducing table completeness (TC) statements [18], which are similar to tuple-generating dependencies, and showed how to derive query answer complete-
ness from them. The completeness patterns used in this paper are exactly the intersection of the statements introduced by Motro and Levy. While Levy’s statements allow to express the completeness of a table in terms of other, incomplete tables, Motro’s statements assume that all tables used in a statement are complete. The intersection of these two is therefore the set of all statements that does not use additional tables.

Other work, such as Denecker et al. [9], Cortes-Calabuig et al. [7] and Razniewski and Nutt [23], looked into reasoning about query completeness using the database instance. However, they used more expressive languages for their completeness descriptions and/or queries and, thus, suffer from high complexity (coNP-data complexity or \(\Pi^p_2\)-query complexity). More importantly, the previous work only studied the decision problem of whether the query results are fully complete.

Fan and Geerts [10, 11] investigated a related problem of completeness assessment in the presence of a database instance. However, they make the unrealistic assumption that a master (“upper-bound”) database instance is given to be able to determine which tuples are missing.

Lang et al. [17] address the problem of characterizing partial results computed by query plans when query results are produced even when some data may be unavailable. Their key contributions are: (1) a taxonomy for classifying query result semantics based on two properties of the partial result: cardinality and correctness; according to cardinality, the result may be missing some tuples or may have extra tuples; according to correctness, the result may be credible or non-credible; (2) an identification of four models, with different granularities of analysis: query, operator, column and partition; and (3) a discussion of how the partial result semantics can be propagated through various operators in a query plan. A recent paper by Lang et al. [16] additionally allows users to control the cost of a query returning partial results by defining the sorts of partial results that are acceptable. In terms of the classification by Lang et al., we explore the cardinality dimension (completeness/incompleteness), for the (finest) partition granularity, in much greater technical depth. Our key contributions, which distinguish us from [17], are: (1) a natural class of completeness patterns that correspond to selections on database tables; (2) pattern algebras that allow us to formally reason over completeness patterns and the database instance as input, to compute completeness patterns on the query output - in particular, our techniques can use the database instance to infer stronger completeness patterns on query results compared to the techniques described in [17]; and (3) techniques to efficiently implement the algebras presented, and an empirical evaluation of their scalability in practice. Thus, the overlap with [17] is only conceptual: the technical contributions of the two papers are complementary.

Punctuations, checkpoints that allow to conclude that all data up to a certain timepoint has been received, are used in data streams as a way to assert completeness in a time interval [20] [13]. While they are most often used to assert temporal completeness, they have a similar flavor to our completeness patterns. However, existing work has not studied how to use punctuations to determine which portions of query answers are complete.

Data cleaning, the process of detecting and correcting errors, inconsistencies, duplicates and other problems related to data correctness, has received considerable attention in the past [14, 4, 27]. Regarding incompleteness, extensive work has been done on statistical approaches to missing value imputation [25, 2]. The difference to our problem is that in missing value imputation, one is already aware of the location of the incompleteness.

Finally, we note that our setting is different from what in the literature is commonly called incomplete databases (for a recent survey, see e.g. [19]). While this setting allows also for missing records, in most of this work, implications on query answers of null values in existing records is studied, such as providing certain answers, where query answers over all possible completions of an incomplete database are provided, and possible answers, where answers over at least one such completion are provided [1]. In contrast, we investigate the problem of how missing records in the database lead to missing records in the query answer, and if so, from which parts of the query answer.

### 3. DATABASE AND QUERY COMPLETENESS PATTERNS

In this section we review database terminology and notation and introduce the framework for completeness assessment.

We first introduce databases and the query language we are using in this paper, then formalize completeness patterns, the semantics of completeness patterns, and the notion of entailed completeness patterns for query answers.

#### 3.1 Databases and Queries

A relation schema is a sequence of attributes. A database schema is a set of table names, each with a relation schema. If table \( R \) has attributes \( A_1, \ldots, A_n \), we say that \( n \) is the arity of \( R \). We often write a tuple \( t \) for \( R \) as \( t = (d_1, \ldots, d_n) \), where \( d_1, \ldots, d_n \) are constants. We interchangeably refer to \( d_i \), the value of \( t \) for attribute \( A_i \), as \( t[A_i] \) or simply \( t[i] \). Given a fixed schema, a database instance \( D \) stores for each table name a finite bag (= multiset) of tuples for that table. The bag of tuples in \( D \) for table \( R \) is denoted as \( R(D) \). If schema and instance are clear from the context, we simply speak of a database.

An example of a database is \( D_{\text{example}} \) as shown in Table 1 while ignoring the rows with \( p_i \).

As queries we consider those that can be expressed using the relational algebra operations select, project and (equi)join (SPJ), which correspond to single-block queries in SQL. To ease the exposition, we do not consider renaming in our presentation.

As in SQL, we assume bag semantics for both databases and queries, that is, both can contain the same row multiple times. When discussing generic properties of a query, we use the letter \( Q \). We use the letter \( E \) if the specific structure of the expression is important. With \( Q(D) \) and \( E(D) \) we denote their evaluation over a

<table>
<thead>
<tr>
<th>W.day</th>
<th>W.week</th>
<th>W.ID</th>
<th>W.message</th>
<th>M.ID</th>
<th>M.responsible</th>
<th>M.reason</th>
<th>T.name</th>
<th>T.specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed</td>
<td>2</td>
<td>tw37</td>
<td>overheated</td>
<td>tw37</td>
<td>A</td>
<td>disk failure</td>
<td>A</td>
<td>hardware</td>
</tr>
<tr>
<td>Mon</td>
<td>2</td>
<td>tw83</td>
<td>high voltage</td>
<td>tw83</td>
<td>B</td>
<td>unknown</td>
<td>B</td>
<td>hardware</td>
</tr>
<tr>
<td>Tue</td>
<td>2</td>
<td>tw83</td>
<td>auto restart</td>
<td>tw83</td>
<td>B</td>
<td>unknown</td>
<td>B</td>
<td>hardware</td>
</tr>
<tr>
<td>Mon</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Wed</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 5: Result of the query \( Q_{\text{hw}} \) over the database \( D_{\text{example}} \) with summarized completeness information.
3.2 Completeness Patterns

We now want to formalize the meaning of completeness patterns such as rows $p_1$ to $p_2$ in Table\textsuperscript{1} and the bottom six rows in Table\textsuperscript{3}. A completeness pattern is a tuple composed of constants or the wildcard symbol “$*$”. As patterns like those in Table\textsuperscript{1} are expressed over database tables, we call them base completeness patterns. As the patterns in Table\textsuperscript{3} refer to a query result, we call them query completeness patterns.

**Definition 1 (Base Completeness Pattern).** A base completeness pattern for a table $R$ is a pair $(p, R)$ where $p$ is a completeness pattern with the same arity as $R$.

The relation $R$ is dropped if it is clear from the context.

**Example 1.** Assume we want to say that the Maintenance table is complete for all elements maintained by team A. Then this can be expressed by the base completeness pattern $(+, A, +)$, for Maintenance, which is the pattern $p_A$ in Table\textsuperscript{1}.

A set of completeness patterns for a table can be stored in a corresponding metadata table, which has the same attributes as the data table. In the following, we assume that in a database each table has an associated metadata table.

Intuitively, a base completeness pattern says that all rows of a certain kind, namely those subsumed by the pattern, are present in the table and no rows of this form are missing. In this way, they create a middle ground between closed and open-world assumption.

A pattern $p_1$ subsumes a pattern $p_2$ if for every position $i$ it holds that $p_1[i] = p_2[i]$ or $p_1[i] = *$. In other words, $p_1$ and $p_2$ are the same except that $p_1$ can contain additional $*$-values. A pattern can also subsume a data tuple, since tuples are a special case of patterns.

Consider a database $D$. A second database $D'$, with the same schema, contains $D$, written $D \subseteq D'$, if $R(D) \subseteq R(D')$ for all relations $R$ in the schema. Intuitively, a database $D'$ containing $D$ corresponds to a possible state of the real world, about which $D$ has only limited information.

We say that $D'$ satisfies the completeness pattern $(p, R)$ wrt. $D$ if $D \subseteq D'$ and all tuples in $R(D')$ that are subsumed by $p$ are also in $R(D)$. While in general, we do not know what is all the data that holds in reality, completeness patterns restrict the possible complete databases to the satisfying ones. Although completeness patterns in general still allow infinitely many possible states of the real world, all these possible states contain the same information in the parts asserted to be complete.

**Definition 2 (Candidate Set).** Let $D$ be a database and $\mathcal{P}$ be a set of base completeness patterns that are annotated with their respective relations. A database $D'$ with the same schema is called a candidate completion for $D$ and $\mathcal{P}$ if (1) $D \subseteq D'$ and (2) $D'$ satisfies all patterns in $\mathcal{P}$ wrt. $D$.

The set of all candidate completions is called the candidate set, written Card\textsuperscript{P}($D$).

**Example 2.** Consider again the database $D_{\text{maint}}$ from Table\textsuperscript{1}.

Because of pattern $p_3$, which expresses that all warnings from Wednesday of week 2 are already in $D_{\text{maint}}$, a database $D'$ that contains the fact Warning(Wed, 2, tw37, highvoltage) would not be a candidate.

On the other hand, a database $D'$ that additionally contains the fact Warning(Tue, 2, tw37, highvoltage) would be a candidate, since it is not subsumed by any pattern for Warnings.

We remark that base completeness patterns capture exactly the intersection of the statements defined in \cite{22} and \cite{18}, as they do not contain any join conditions, but are purely selections on tables.

We chose the class of completeness patterns because, in contrast to the previously used statements, base completeness patterns can be expressed as rows in the same schema as the data.

We now define completeness patterns for queries, such as the bottom six rows in Table\textsuperscript{3}. Completeness patterns for queries are patterns paired with query expressions, for parts of which they assert completeness.

For a query $Q$ with result schema $A_1, \ldots, A_n$ we use the shorthand “$attr(Q) = p$” to denote the condition “$A_1 = p_1 \land \cdots \land A_n = p_n$”, where a conjunct “$A_i = *$” simplifies to true.

**Definition 3 (Query Completeness Pattern).** A query completeness pattern for a query $Q$ is a pair $(p, Q)$, where $p$ is a pattern that conforms to the result schema of $Q$. The associated query of $(p, Q)$ is $Q_p := \sigma_{attr(Q)=p}(Q)$.

A database $D'$ satisfies $(p, Q)$ wrt. $D$, if $Q_p(D') = Q_p(D)$.

Intuitively, a completion $D' \supseteq D$ satisfies $(p, Q)$ if all answer tuples $t \in Q(D')$ that match $p$ are already present in $Q(D)$.

Note that a tuple $t \in R(D')$ is subsumed by a pattern $p$ iff $t \in \sigma_{attr(Q)=p}(R(D'))$. Thus, $D'$ satisfies the base pattern $(p, R)$ wrt. $D$ iff $\sigma_{attr(Q)=p}(R(D')) \subseteq R(D)$. In other words, satisfaction of base patterns is a special case of query patterns if we view base relations as queries that return the content of a database.

Completeness patterns can be compared wrt. generality. We say that $p_1$ is more general than $p_2$ if every tuple subsumed by $p_2$ is also subsumed by $p_1$. It is straightforward to see that $p_1$ is more general than $p_2$ if $p_1$ subsumes $p_2$.

For example, the pattern $(+, A, +)$ for the Maintenance table subsumes $(+, A, \text{unknown})$. At the same time, every tuple subsumed by the second pattern is also subsumed by the first.

Let $D$ be a set of patterns over the same schema. Clearly, subsumption is a partial order on $D$. A pattern $p_0 \in D$ is maximal in $D$ if there is no pattern in $D$ that strictly subsumes $p_0$. Clearly, every element in $D$ is subsumed by some element that is maximal in $D$.

We say that $D$ is minimal (wrt. subsumption) if no element is strictly subsumed by another element. We are interested in minimal sets of patterns because they do not contain any redundant patterns. It is straightforward to see that $D$ is minimal if and only if all its elements are maximal in $D$. Thus, to minimize a set $P$ of patterns, one has to eliminate from $P$ all non-maximal elements.

3.3 Problem Statement

We are investigating how to compute query completeness patterns (Def\textsuperscript{3}) from base completeness patterns (Def\textsuperscript{1}) so that the results satisfy the following entailment property.

**Definition 4 (Entailment).** Let $D$ be a database and $\mathcal{P}$ be a set of base completeness patterns for the schema of $D$. We say that $\mathcal{P}$ entails a pattern $(p, Q)$ (or $\mathcal{P}$ entails $p$ for $Q$) wrt. $D$, if all candidate completions in Card\textsuperscript{P}($D$) satisfy $p$ for $Q$.

As discussed in Section\textsuperscript{1} all patterns in Table\textsuperscript{3} are entailed for $Q_{\text{low}}$ by the base patterns in $D_{\text{maint}}$ wrt. $D_{\text{maint}}$.

It is easy to show that the problem to decide, given some $D$, $\mathcal{P}$, $Q$, and $p$, whether $\mathcal{P}$ entails $(p, Q)$ wrt. $D$, is coNP-complete in combined complexity and polynomial in data complexity (that is, fixing $(p, Q)$ and letting $D$ and $\mathcal{P}$ vary).

The problem that we investigate in the rest of this paper is, given a database $D$, base completeness patterns $\mathcal{P}$, and query $Q$, how to compute the set of all patterns $p$ entailed for $Q$, or a minimal representation of that set.
4. PATTERN ALGEBRA

In Section 4 we have seen how query completeness patterns can be computed from base completeness patterns using algebra operations. In this section we formalize an algebra for completeness computations analogous to the SPJ fragment of relational algebra, which in turn captures the single-block SQL queries, and present experiments regarding the expected size of completeness patterns and pattern minimization techniques.

4.1 Algebra Definition

The SPJ fragment of relational algebra with equality can be formalized using four operations: Selection by constant (σ_{A=d}(R)), projection (π_{A}(R)), selection by attribute equality (σ_{A=B}(R)) and the equijoin (R₁ \bowtie_{A-B} R₂), where R₁, R₁ and R₂ stand for generic data relations, not necessarily base relations, A and B for attributes, and d for a constant.

Note that to ease the presentation, the projection π_{A}(R) used here is different from the classical projection. Instead of mentioning the surviving attributes, it mentions exactly one attribute that is projected out. For a given relation schema, classical projection can be modeled straightforwardly by this one, and vice versa.

In this section we define for each of the above operators an equivalent one operating on relations P of completeness metadata. We make the distinction between data operators and metadata operators (such as \bowtie) to the latter.

Decision support queries often include aggregation. Aggregation can be applied on top of the abovementioned operations. We discuss completeness calculation for aggregate queries in Appendix [3].

Next, we define each of the four operations of the pattern algebra. We suppose that R is a relation with the same schema as \tilde{R}, but while \tilde{R} contains data tuples, P contains metadata, namely patterns describing the complete parts of R.

4.1.1 Selection by Constant \tilde{σ}_{A=d}(P)

For the selection \tilde{σ}_{A=d}(R), intuitively, only those patterns in P for R give information about the output that have a wildcard or the value d at position A. Other patterns are irrelevant.

Example 3. Consider the selection σ_{\text{week}=2}(\text{Warnings}) in the introductory example. There are three completeness patterns for the table Warnings, namely \((*, 1, *, *)\), \((\text{Mon}, 2, *, *)\) and \((\text{Wed}, 2, *, *)\). For the result of the selection, the first pattern is irrelevant, as it talks about week 1. The second and the third pattern are relevant, since they talk about week 2. Furthermore, since the result may only contain records with \text{week}=2, we can generalize the last two patterns by replacing “2” with “*”. Thus, the resulting metadata table looks as shown in Table 2.

Let p[A]=d denote the replacement of the value for attribute A in p with d. Then, for a metadata relation P, the selection for \tilde{σ}_{A=d}(P) is defined as follows:

\[
\tilde{σ}_{A=d}(P) = \{ p \in P \mid p[A] = d \} \cup \{ p[A]=* \mid p[A] = d, p \in P \}.
\]

In summary, patterns with A = d survive the selection unchanged, while patterns with A = d survive but get generalized.

4.1.2 Projection \tilde{π}_{-A}(P)

Remember that we consider an atomic projection that projects away exactly one attribute.

Example 4. Consider the metadata part of the table Warnings, with the patterns \((+, 1, *, *)\), \((\text{Mon}, 2, *, *)\) and \((\text{Wed}, 2, *, *)\). If we project out the day attribute by \tilde{π}_{-\text{day}}(\text{Warnings}), the projection \((1, *, *)\) of the first pattern still holds over the resulting table. For week 2, however, we cannot assert completeness, since e.g. records from Tuesday could be missing.

Generally, the projections of all completeness patterns with A = * continue to hold over the projection of the data relation.

Formally, the result of the projection \tilde{π}_{-A}(P) over a metadata relation P is defined as:

\[
\tilde{π}_{-A}(P) = \{ \tilde{σ}_A(p) \mid p \in P, p[A] = * \}.
\]

4.1.3 Selection by Attribute Equality \tilde{σ}_{A=B}(P)

For a selection \tilde{σ}_{A=B}, those patterns say something about the result that subsume tuples with identical values for A and B, that is, those patterns, that either have the same value for A and B, or have a wildcard at at least one of the two positions.

Example 5. Suppose that \((d₁, d₁, e₁)\), \((d₂, *, e₂)\) and \((*, e₃)\) are patterns for the table \(R(A, B, C)\). Over the result of the selection \tilde{σ}_{A=B}(R), all three completeness patterns continue to hold.

There are two particularities however: First, some generalized patterns also hold over the data selection \(\tilde{σ}_{A-B}(R)\), all three completeness patterns continue to hold.

Second, for patterns involving one wildcard and one constant, such as \((d₂, *, e₂)\), also the symmetric version holds, where the constant and the wildcard are swapped. In this case, this yields the pattern \((*, d₂, e₂)\). While the two patterns are semantically equivalent over the results of a selection, it is important that both be present, because possible subsequent projection operations that project out A would lead to a loss of the first pattern, projections projecting out B would lose the latter pattern.

For a pattern p, let \(p[A] \leftrightarrow B\) denote the operation of swapping the values at positions A and B. Then, the result of the operation \tilde{σ}_{A=B}(P) is formally defined as follows:

\[
\tilde{σ}_{A=B}(P) = \{ p \in P \mid p[A] = \ast \text{ or } p[B] = \ast \} \cup \{ p[A] \leftrightarrow B \mid p \in P, p[A] = \ast \text{ or } p[B] = \ast \} \cup \{ p[A]=d \mid p \in P, p[A] = p[B] \} \cup \{ p[B]=d \mid p \in P, p[A] = p[B] \}.
\]

Example 6. Consider again the patterns \((d₁, d₁, e₁)\), \((d₂, *, e₂)\) and \((*, e₃)\) from above. The result of the selection \tilde{σ}_{A=B}(R) then satisfies exactly the patterns \((d₁, *, e₁)\), \((*, d₁, e₁)\), \((d₂, *, e₂)\), \((*, d₂, e₂)\) and \((*, e₃)\).

Note that in Table 3, for ease of presentation, we have omitted the generalizations and the symmetric tuples.
4.1.4 Equijoin $P \bowtie_{P.A=P'.A'} P'$

Completeness patterns behave in equijoins similar to records in regular equijoins, with the difference that the wildcard symbol * joins with any constant. As in relational algebra, equijoins are equivalent to a combination of a cartesian product of the completeness patterns for $R$ and $R'$ and a subsequent selection.

**Example 7.** Suppose we want to compute the metadata join $P_{\text{ maint}} \bowtie_{\text{name} \bowtie \text{name}} \sigma_{\text{spec} = \text{"hw"}} (P_{\text{ teams}})$, where the $P_{\text{ maint}}$ and $P_{\text{ teams}}$ denote the metadata of the tables Maintenance and Teams. The join corresponds to a subexpression of the algebraic expression in (1) that computes the query $Q_{\text{ hw}}$. Then, we first compute a normal join between the patterns in the two tables, and afterwards introduce the symmetric versions for those patterns that involve one wildcard and one constant at the positions name and responsible. The input and output of this calculation are shown in Table 3.

Formally, we define the result of a join $P \bowtie_{P.A=P'.A'} P'$ as:

$$\delta_{P.A=P'.A'}^P (P \times P').$$

Note that one need not actually compute a cartesian product, as the selection conditions can be pushed directly into the join, thus yielding a union of four smaller joins.

**Aggregation.** Decision support queries often contain aggregations such as COUNT, SUM, MIN or MAX. A discussion on how the pattern algebra can be extended to these aggregate functions is contained in Appendix A.

**Soundness and Completeness.** In logic, an inference system is called sound, if all conclusions produced by the system are valid. An inference system is called complete, if the inference system produces all valid inferences.

We show below that the pattern algebra is a sound inference system, and that it is also complete, if the state of the database is ignored. To avoid confusion, we always write “computationally complete” when referring to the completeness of the algebra.

Intuitively, soundness means that whenever we have completeness patterns for an expression $E$ and apply one of the algebra operations to $E$, the resulting expression also satisfies the completeness patterns calculated by the analogous metadata operation.

Let $D$ be a database, $P$ a set of base patterns for the schema of $D$, $Q$ a query and $p$ a pattern for the schema of $Q$. We write $\mathcal{P}, D \models (p, Q)$ to express that for every completion $D^\prime \in \text{Cand}^P (P)$ it holds that $D^\prime$ satisfies $(p, Q)$ wrt. $D$. In other words, if $D^\prime$ is a completion of $D$ that satisfies the completeness assertions in $\mathcal{P}$ about the base tables of $D$, then the $p$-part of $Q(D)$ is the same as the $p$-part of $Q(D^\prime)$. The latter has been expressed formally in Def. 3 as $Q_p(D) = Q_p(D^\prime)$.

We write $\mathcal{P}, D \models (P, Q)$ in case $\mathcal{P}, D \models (p, Q)$ for every $p \in P$. For queries in relational algebra analogous definitions apply.

**Proposition 5 (Soundness).** Let $D$ be a database, $\mathcal{P}$ be a collection of base patterns for the schema of $D$, and let $P$ and $P'$ be sets of completeness patterns for algebra expressions $E$ and $E'$ respectively. Furthermore, let op be an operation among $\{\pi_{-A}, \sigma_{A \bowtie d}, \sigma_{A = A'} \bowtie_{A = A'} \}$. Then:

1. $\mathcal{P}, D \models (P, E) \implies \mathcal{P}, D \models (\text{op}(P), \text{op}(E))$.
2. $\mathcal{P}, D \models (P, E) \land (P', E') \implies \mathcal{P}, D \models (P \bowtie_{A = A'} P', E \bowtie_{A = A'} E')$.

**Proof.** See Appendix A.

Intuitively, the proposition says that if the patterns in $P$ are valid for $E(D)$, then those computed by $\text{op}(P)$ are valid for $\text{op}(E(D))$. Since the base patterns are valid for the base relations, we can conclude soundness for arbitrary expressions by induction.

Before proving the completeness of the pattern algebra, we have to differentiate patterns into meaningful and meaningless patterns.

**Example 8.** Consider the selection $\sigma_{\text{spec} = \text{"hw"}}$ over the table Teams(name, specialization). Over the result of this selection, the pattern $(*, \text{ software})$ can never subsume a record, as there can never be software teams in a table that was selected as only containing hardware teams.

We say that a query completeness pattern $(p, E)$ is satisfiable if there exists a database $D$ such that $E(D)$ contains at least one record that is subsumed by $p$.

Clearly, the pattern algebra cannot compute all entailed patterns, since there may be infinitely many. Instead, it will already be enough if it computes all satisfiable ones, and among those, only the maximally general ones. To make this formal, we need some notation.

Given a set of base completeness patterns $\mathcal{P}$, and a query pattern $(p, E)$, we write $\mathcal{P}, D \models (p, E)$ if for all database instances $D$ it holds that $\mathcal{P}, D \models (p, E)$.

If $E$ is an SPJ-algebra expression, then $\tilde{E}$ is the corresponding expression in the pattern algebra, obtained by replacing every operator in $E$ by its $\bowtie$-counterpart. Then $\tilde{E}(P)$ is the set of patterns produced by evaluating $\tilde{E}$ over $P$.

**Proposition 6 (Completeness Without Instance).** Let $\mathcal{P}$ be a set of base completeness patterns, $E$ an SPJ-algebra expression and $(p, E)$ a satisfiable query completeness pattern. Then $\mathcal{P}, D \models (p, E) \implies p \preceq \tilde{E}(P)$.

**Proof.** See Appendix A.

The proposition states that if a satisfiable query completeness pattern $(p, E)$ is a logical consequence of $\mathcal{P}$, then we can evaluate the corresponding pattern algebra expression $\tilde{E}$ over $\mathcal{P}$ and find a $p'$ in the result that is at least as general as $p$.

A corollary of soundness and completeness is that for any two equivalent relational algebra expressions, completeness patterns computed by the pattern algebra are equivalent.

One can show as another property of the pattern algebra that it never introduces redundancies, that is, if the completeness patterns for the base are minimal, then also the pattern sets computed by any algebra expression are minimal.

This means that for an expression $E$ and a minimal $\mathcal{P}$ as input, the output $\tilde{E}(P)$ is identical for all equivalent expressions and thus does not depend on the specific expression.

4.2 Experiments Using Wikipedia Data

In all experiments that follow, the implementations were done in Java, using a PostgreSQL database, and executed on a machine with 2.4 Ghz and 4 GB RAM.
To show the feasibility of our approach, we used a semiautomated scraping technique to identify 21 completeness statements from Wikipedia for tables about cities, countries and schools. We also extracted 10k schools and 200 countries from DBpedia, and 55k cities from OpenGeoDB and geodatasource.com. Note that DBpedia does not extract all the information in Wikipedia.

We then created seven join-queries over these tables and compared the time needed for query evaluation with the time needed for completeness calculation. The median query execution time was 2290 ms, while the median completeness calculation time was 543 ms, which implies that the completeness calculation took 23% of the time of the query calculation. Considering averages this number is even much better at only 2.4%, but this number may be confusing because one query took 175 seconds to execute, while completeness calculation took never longer than 1 second.

The query for which completeness calculation took longest compared with query evaluation was the query:

```
SELECT * FROM country, city
WHERE country.capital=city.name
```

For this query, execution took 30 ms while completeness calculation took 797 ms. The query for which the completeness calculation took shortest compared with the query evaluation was the query:

```
SELECT * FROM city, school
WHERE city.state=school.state
```

For this query, execution took 175 seconds while completeness calculation took 421 ms.

Both results can be explained by the size of the query results. While the first query is highly selective and produces only 278 records, the second produces 3 million records. In contrast, the number of completeness patterns computed for query results was between 9 and 100 (median 46) for the seven queries, and the time needed for their calculation exhibited a much lower variance with a range between 397 and 991 ms.

### 4.3 Number of Patterns

To understand the possible overhead of computing query completeness patterns, it is necessary to understand how large sets of completeness patterns for base tables may actually be in applications. We synthetically created completeness patterns to measure the number of patterns wrt. the size of the data. The main conclusion is that over real data, the number of completeness patterns remains reasonably small compared with the normal data.

**Test Case Generation.** We believe that a typical situation may be one where most parts of a dataset are complete, and only a few records are missing. We also assume that in completeness patterns only attributes with a relatively low number of distinct values are used, which naturally correspond to the dimension tables in data warehouses.

After identifying such a set of attributes, we proceeded in our experiments as follows:

- We loaded a dataset.
- We assumed the most general pattern \( \{*,*,\ldots,*\} \) to hold over the loaded dataset.
- We then repeatedly selected one record from the data set, which we dropped. Whenever some of the completeness patterns asserted for the dataset would subsume this record, they would cease to be true, so we dropped those patterns, and tried to add instead all possible most general specializations that continued to hold over the dataset.

For instance, if from the original Teams table we dropped a hardware team, the \( \{*,*,\ldots,*\} \) pattern would not be correct any more. Instead, we would insert specializations that only assert completeness for software and networking teams.

We also experimented with two strategies for dropping records. In the first, the dropped records were selected purely at random. However, in reality, causes for missing data may lead to systematic data loss. We therefore investigated whether dropping related records leads to a different structure of completeness patterns than dropping random records.

**Sample Cases.** We first did experiments on a network element table from a network provider. The table has 64 attributes and 7660 records. Among those 64 attributes, we manually identified six attributes that qualified as dimension attributes and seemed plausible for completeness patterns: region_name (6), technology (3), vendor (7), technology_capability_type (6), sector (13), and state (53), where the numbers in parentheses report the number of distinct values present in the data. Multiplying these numbers, we find a total of 1,185,408 possible different completeness patterns. In the data, only 1,158 possible different completeness patterns were present. Also, the frequency of these combinations was exponentially distributed.

As comparison we used the lineitem table from the TPC-H benchmark (scale factor 1), where we manually selected seven attributes which had few distinct values, leading to 460,800 possible completeness patterns. Of these, in the dataset of 6 million records, 73,419 combinations (1.22%) were present.

Skewed data generators for TPC-H appear in the literature [5] [8], but the tools are not publicly available. Also the more recent TPC-DS benchmark does not provide realistic data, as skew appears there only in a few fields such as “names”, and fields are not correlated.
Experiments. We first dropped random records from our network element table. It turned out, that the number of completeness patterns converged after 300 dropped records at around 1000 completeness patterns. The likely explanation is that due to the few combinations in the real data set and due to the correlations, most dropped records have the same values as previously dropped records and therefore do not lead to further specialization of patterns. A graph is shown in Fig. 1.

We tried the same method on the TPC-H dataset, where the number of completeness patterns did not show any convergence behavior (same figure). A likely reason is that there are no correlations in the TPC-H dataset.

Finally, we tested dropping related records. As indicator for relatedness we used the names of the network elements, because these names are not purely random but often carry some semantics. We dropped elements sharing three different prefixes of their names (Cnu, Dxu, Clu). It turned out that in all cases, the number of completeness patterns needed to describe the remaining complete data converged more quickly and was smaller. The results are shown in Fig. 2. Note that curves increase whenever some violated pattern can be specialized, and decrease if violated patterns cannot be specialized any more.

Conclusions. We draw three conclusions from these experiments: First, we see that the number of completeness patterns is relatively small compared to tuples in the data. Second, we see that when data loss is due to systematic reasons, the number of completeness patterns may be even smaller. Third, we see that this only applies to data sets which show skew and correlation, while for data sets that do not have these characteristics, the number of completeness patterns may be much higher.

4.4 Minimization of Sets of Completeness Patterns

Completeness patterns can subsume each other. Redundancies or replications could be reasons why that happens in practice, e.g. when first team A reports that data for itself is complete, and later a department head reports that data for all teams is complete. Redundancies can also occur due to additional inferences drawn when considering the state of the database instance, as will be shown in Section 5.

A naive approach to minimization of sets of completeness patterns is pairwise comparison, however, already for 10k patterns, the runtime of this approach was more than 100 seconds, and the complexity is quadratic.

Since the experiments in the previous section showed that 1000 completeness patterns is a realistic input size, and pattern sets can grow quadratically in joins, we consider the minimization of sets of patterns a crucial operation.

There are two aspects to minimization that allow for optimization:

1. The approach how patterns are processed. We investigated three approaches, which we call all-at-once, incremental, and sorted incremental.
2. The choice of the data structure. We analyzed naive sets, hash tables, path indexes and discrimination trees, that latter two being data structures borrowed from the area of theorem proving, where a related problem of pattern minimization occurs.

We discuss each of these in detail below. As approaches and data structures are orthogonal, we use numbers 1 to 3 for referring to the approaches, and letters A to D for referring to the data structures.

Approaches. The first and most straightforward approach to minimization is (1) all-at-once processing. In this method, all patterns are loaded into memory, then for each pattern it is checked whether it is subsumed by some other pattern (see Subsection 3.2). Naively, this requires quadratic time, but data structures like hash tables, path indexes or discrimination trees can speed up this check.

Another approach is (2) incremental processing. Here, each pattern is loaded separately, and subsequently compared with the maximal patterns among the ones that already have been loaded. The comparison requires both to check whether the pattern is subsumed by one of the already loaded patterns (subsumption checking), and, if that is not the case, to find the patterns among the already loaded ones that are subsumed by the new pattern (supersumption retrieval).

An extension of incremental processing is (3) sorted incremental processing. When completeness patterns are sorted in decreasing order wrt. the number of wildcards they contain, then no later pattern can entail an earlier pattern, because subsumption requires that the subsuming pattern has more wildcards than the subsumed pattern (except for duplicate patterns). Therefore, incremental sorted processing does not require supersumption retrieval.

The advantages of the incremental approaches are that, given that the set of maximal patterns is sufficiently smaller than the input, these methods require less memory.
Data Structures. Lists (A) are the most basic data structure, which we use as baseline. Using lists, a single subsumption check or supersumption retrieval requires linear time w.r.t. the number of patterns, thus leading to a quadratic algorithm.

Hash tables (B) can be used to achieve faster subsumption checks: Given a pattern, checking whether it is subsumed by some stored pattern can be done by explicitly generating all generalizations of that pattern, and checking whether any of those is stored in the hash table. However, the number of generalizations of a pattern is exponential in the number of constants in the pattern, as each way of replacing some constants with wildcards yields a generalization.

In the AI community, similar problems occur when managing sets of terms used in theorem provers. Besides the subsumption checks and superset retrieval, theorem provers also need to be able to find instantiations and unifiable patterns. Also, theorem provers need to deal with terms that involve repeated variables and function symbols. There are two classical techniques for term indexing: discrimination trees and path indexing [21].

Path indexes (C) are similar to inverted indexes. The idea is to create a list of occurrences for each symbol (constants and wildcard) at each position. To find subsumed and superseded patterns, relevant lists then have to be intersected or merged. Especially the intersection is expensive, leading to a poor performance of the method in our experiments.

Discrimination Trees (D) are essentially tries over completeness patterns, which treat the wildcard value in the same way as other constants. An example of a discrimination tree is shown in Fig. 3. Given a pattern \( \alpha \), subsumption checking is done by a search from the root, which at any node \( n \) at level \( i \) always continues searching in branches labelled with \( \ast \), and if \( \alpha[i] = d \), also searches branches labelled with \( d \). Thus, subsumption checking requires search with a branching factor of at most 2. Supersumption retrieval, on the other hand, considers the branch labelled with \( d \), if \( \alpha[i] = d \), and if \( \alpha[i] = \ast \), then all branches. Thus, the branching factor is only limited by the branching factor of the discrimination tree.

Experiments. In the experiments, we used random subsets of one million completeness patterns generated as the result of a cartesian product between two tables with 1000 patterns each.

In Fig. 4 we see the runtime comparison. It turns out that the all-at-once approach is faster than the two other approaches, and that for this approach, discrimination trees (D1) are 25% faster than hashing (B1). Pairwise comparison (A1) and path indexes (C2) turn out to be totally unusable.

In Fig. 5 we compare different approaches in combination with discrimination trees and hash tables regarding the space needed. Here, the sorting approaches (B3 and D3) perform clearly best, while the all-at-once approaches, as expected, consume an amount of space that is linear in the input size. The fact that the sorted approaches consume even less space for 600k patterns than for 400k is consistent with calculations that show that the number of maximal patterns decreases when using more than 300k random patterns out of the one million patterns.

Conclusions. We conclude that regarding space, approaches that sort the input first perform best, while regarding runtime, approaches that process all patterns at once perform best. While regarding space there is no difference between the best discrimination tree and the best hashing method, regarding runtime, discrimination trees are consistently faster than hashing. The advantage of discrimination trees is likely to increase when more attributes (here twelve) are used for completeness patterns.

5. MAKING THE PATTERN ALGEBRA INSTANCE-AWARE

So far we have computed completeness patterns for queries only from the patterns that hold for base relations. However, as seen in the motivating example, additional inferences are possible when taking into account also the tuples in the database instance. In this section we argue that such an extension of the pattern algebra brings about benefits, show that instance awareness may be expensive in theory, and evaluate the efficiency of the extended algebra on a test case.

5.1 Join Extension

In the motivating example we saw that patterns in a join result
can sometimes be summarized to more general patterns. Let us look at that case again in more detail.

**Example 9.** Consider again the join

\[ M \bowtie_{\text{first} = T, \text{name}} \sigma_{\text{spec} = \ast}(\text{where}(T)) \]

whose computation is shown in Table 6. We see that the only completeness pattern for \( \sigma_{\text{spec} = \ast}(\text{where}(T)) \) is \((\ast, \ast)\). We also see that the only records in \( T \) that match this pattern are \((A, \text{hardware})\) and \((B, \text{hardware})\). This allows us to conclude that all records appearing in the join result must have either the value \( A \) or \( B \) for the attributes \( M.\text{resp} \) and \( T.\text{name} \). We also see that the metadata for \( M \) contains two patterns \((\ast, \ast, \ast, \ast)\) and \((\ast, \ast, \ast, \ast)\), which contain exactly the values \( A, B \) in the second position, the join position, and which are identical in the other positions.

We therefore conclude that these two patterns cover all possible values that can appear in the join result, and that the result patterns \((\ast, \ast, \ast, \ast)\) and \((\ast, \ast, \ast, \ast)\) can therefore be promoted to the more general pattern \((\ast, \ast, \ast, \ast)\) for the join result. As a consequence, we know that the entire results of the join is complete.

The preceding example illustrates that promotion of statements gives some benefits. First, it allows one to identify larger parts of the query result as complete. Moreover, the new pattern subsumes all the promoted patterns and thus leads to a more compact representation. More general statements with \( \ast \) in more attribute positions can also survive more projection operations, since a pattern \( p \) over \( R \) only gives rise to an element in \( \pi_{A^\prime}(R) \) if \( p[A] = \ast \). As discussed in Appendix B aggregation bears some similarity with projection and thus more general statements allow one also to recognize more results of an aggregate query as correct.

Next, we show with an extended example how based on this idea one can define a general promotion technique for joins. Consider two tables \( R(A, B, C), R'(A', B') \) as shown below, and the join query \( R \bowtie_{A=A'} R' \).

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( c )</td>
<td>\ast</td>
</tr>
<tr>
<td>( b )</td>
<td>\ast</td>
<td>( d )</td>
</tr>
</tbody>
</table>

\[ P \bowtie_{A=A'} P' \]

Clearly, all three patterns of \( R \) join with the pattern of \( R' \), because the latter has a “\( \ast \)” in the join position. The result records of the join all have the constant \( a \) or \( b \) in the join positions because \( R \) does so. We show how we can infer more general completeness patterns if we take into account the records in \( R' \).

Let us consider first the pattern \( p \) over \( R' \). The only records in \( R' \) that match \( p \) are \((a, g)\) and \((b, g)\), and since \( p \) is a completeness pattern, these are all such records that will ever find in \( R' \). Thus, the only values that can possibly occur for attribute \( A' \) in a match of \( p \) in \( R' \), the “allowable domain” for \( A' \) with respect to \( p' \), so to speak, are \( a \) and \( b \). We can retrieve these values from the instance \( D \) with the project-select query \( \pi_{A'}(\sigma_{p'=g}(R'(D))) \) where \( R'(D) \) denotes the instance of \( D \) in \( R' \).

Now, we move to the patterns for \( R \). Suppose, the pattern \( p_1 = (\ast, c, d) \) would hold over \( R \). Then we could conclude that the concatenation \( p_1 \cdot p = (\ast, c, d, \ast, g) \) holds over the join \( R \bowtie_{A=A'} R' \). In fact, \( p_1 \cdot p \) does hold over the join. Since \( p_1 \) holds over \( R \), also the specialization \((a, c, d)\) holds, and since \( p_2 \) holds over \( R \), also the specialization \((b, c, d)\) holds. Thus, both \((a, c, d, \ast, g)\) and \((b, c, d, \ast, g)\) hold over the join. However, as seen above, \( a \) and \( b \) are the only possible values for \( A' \) in \( R' \), and thus, they are also the only possible values for \( A \) in the join \( R \bowtie_{A=A'} R' \). Consequently, \((\ast, c, d, \ast, g)\) holds, which has the “\( \ast \)” in \( A \) that, intuitively, has been transferred from \( A' \). We say that \((\ast, c, d, \ast, g)\) has been obtained by “promoting” the statements \( p_1 \) and \( p_2 \) using \( p' \).

**Allowable Domain:** Let us discuss which difficulties can arise in applying this technique. We make a promotion attempt for every pattern \( p' \) over \( R' \) that has a “\( \ast \)” in the join position \( A' \). To this end, we construct a project-select query out of \( p' \) that retrieves the allowable domain \( \Delta_{A'}^{p'} \) for \( A' \) with respect to \( p' \), which consists of all \( A'\)-values of records in \( R' \) that match \( p' \). In the example above, we had \( \Delta_{A'}^{p'} = \{ a, b \} \). This step is easy.

**Choice Sets:** Having identified the allowable domain, we look for all possible choice sets of patterns over \( R \), that is, sets that contain for each value in \( \Delta_{A'}^{p'} \) one statement with that value for the join attribute \( A \). In the example above, \( \{ (a, c, \ast), (b, \ast, d) \} \) was a choice set, and \( \{ (a, c, d), (b, \ast, d) \} \) is another one. A potential difficulty is that there may be exponentially many choice sets.

Then we check for every choice set whether, together with the initial pattern \( p' \) over \( R' \), it can be promoted. To this end, we replace the \( A \)-values in the records of the choice set with a “\( \ast \)” and check whether the resulting pattern set is unifiable, that is, whether for every attribute position there is at most one constant that occurs in that position in the set. In the example above we obtain \( \{ (\ast, c, \ast), (\ast, \ast, d) \} \) and \( \{ (\ast, c, d), (\ast, \ast, d) \} \), which clearly are unifiable.

**Unification:** If the check is positive, we compute the unifier, that is, the most general pattern that matches every pattern in the set. The unifier has a constant in those positions for which there is a (single) constant in the set and has a “\( \ast \)” in the other ones. Thus, the unifiers in our example are \( (\ast, c, d) \) and \( (\ast, \ast, c, d) \). Checking unifiability and computing unifiers is conceptually easy.

The promoted statements are then obtained by concatenating the unifiers of the unifiable choice sets with the initial pattern \( p' \). Thus, in the example, the promoted statements are \( (\ast, c, d, \ast, g) \) and \( (\ast, \ast, c, d, \ast, g) \).

So far, we have discussed how to use statements over \( R' \) to promote statements over \( R \). Of course, this can also be done in the reverse direction, from \( R \) to \( R' \).

To incorporate promotion into the pattern algebra, we introduce the join operator

\[ P \bowtie_{A=A'} P' \]

that first computes \( P \bowtie_{A=A'} P' \) and then adds all patterns that can be obtained by promotion in either direction. The algebra with this join operator is the instance-aware algebra.

**Soundness and Completeness.** As for the pattern algebra, the computational properties of the extended algebra are of interest.

It is easy to show that the instance-aware algebra is sound. Regarding the computational completeness, we conjecture that the algebra is complete for inferences wrt the state of the database, for all queries in which no attribute is reused in joins (e.g. the query \( Q_{bm} \) is in this class). As we show however in Appendix D for general queries the pattern algebra with promotion is still not complete.

### 5.2 Runtime and Space Effects of Promotion

Promotion is possibly expensive, because for an initial pattern \( p' \) over \( R' \) we iterate over all choice sets of patterns over \( R \), to perform the unifiability check and possibly compute a promoted statement. As the number of subsets of a set and therefore the number of choice sets can be exponential, this iteration may be expensive. Furthermore, as each unifiable choice set gives rise to a
new completeness pattern, the output of promotion may also have exponential size.

We discuss here possible optimizations, and present experimental results regarding the runtime and space effects of promotion.

**Implementation of Promotion.** Given an initial pattern \( p' \) over \( R' \), promotion considers all choice sets of patterns over \( R \) that contain one pattern \( p \) with \( |p| = d \) for every \( d \in \Delta_{p'} \).

It is natural to implement the procedure to find all such choice sets \( S \) by first splitting \( P \), the set of patterns over \( R \), into so-called \( A \)-sets, that is, one set for each value appearing at position \( A \). Given \( \Delta_{p'} \), choice sets \( S \) to test for unification are then all sets which are composed by choosing exactly one pattern from each \( A \)-set whose value appears in \( \Delta_{p'} \).

There exist a few optimizations to this procedure:

- **Trivial failure:** If one \( A \)-set required by \( \Delta_{p'} \) is empty, promotion is impossible.
- **Pruning:** Instead of choosing one pattern from each \( A \)-set and then testing for unification, one can test unification “on-the-go”, whenever choosing one more pattern. Thus, if e.g. \( \Delta_{p'} \) has size 5 but already the first two chosen patterns are not unifiable, there is no need to choose patterns from the other \( A \)-sets.
- **Subsumption detection:** Similarly to pruning, if an intermediate unifier is already more specific than a pattern computed for the result, any patterns retrieved using this unifier will be redundant and therefore the search can be pruned.
- **Due to the previous optimizations, the iteration order of the \( A \)-sets can play a role.** As common in search problems, it turned out that starting with the smallest sets gave the best results.

In the following experiments, these optimizations reduced the number of sets to be tested by 40% to 99%.

**Experiments.** When using pattern promotion, two questions arise: first, how the runtime is affected by iterating through possibly exponentially many sets, and second, what effect promotion has on the number of patterns in the output of a join.

We evaluated both aspects in two typical scenarios. The first scenario is a join of a fact table with a dimension table, which is considered to be complete. The second is a join between two partially complete fact tables. For both experiments, we used the network element table from Section 4.3 and in the latter case performed a selfjoin.

The results for the join of the fact table with the dimension table were that the median runtime of the join was between 91 and 661 milliseconds for various join attributes, opposed to a table scan taking 37 seconds. Nevertheless, for the two attributes with the highest cardinality, 5% and 10% of the runs took longer than 30 seconds and were considered as timeouts.

For the join of the two partially complete fact tables, we observed a quadratic growth of the runtime wrt. the number of input patterns, similarly as one would expect it from normal joins.

For both kinds of join, the promoted patterns, instead of leading to a bigger output, lead to a smaller output, because they subsumed other statements in the output. For the join of the two partially complete tables, this reduction was especially significant, causing a reduction between 80% and 95%. In no case promotion lead to an increased output size.

More details on the experiments and graphs are shown in Appendix C.

6. **DISCUSSION**

In this section we discuss aspects concerning the source of completeness patterns and implementational aspects.

**Source of Completeness Patterns.** We have already discussed the source of completeness information in some scenarios. To generalize those observations, we believe that in domains where data is loaded automatically, such as stream processing [12] or feed management [20], completeness patterns should also be generated in an automated way by the loading system.

On the other hand, in applications where data is created by humans, completeness patterns cannot be automated but should be part of the workflow used to enter the data. Concretely, web interfaces that allow to enter data could e.g. contain fields that allow to specify that all data of a certain kind was entered.

**Storage.** A first question is how to store completeness patterns in a RDBMS. The conceptually straightforward solution is to introduce one metadata table for each table. For storing the wildcard value (“*”), one either has to use new data types that extend previous data types by allowing the wildcard value, or, for data types such as string, one may use string escaping techniques.

**Placement of Reasoner.** The next question is the placement of the completeness reasoner. It could either be placed within the DBMS itself, e.g. as plugin, or it could be placed outside as an independent component. While the pattern algebra works only on the schema level, the subsequently introduced promotion operator needs access to the database content, and would therefore benefit from fast access to the database content. The advantage of placing the reasoner outside the DBMS would be that the reasoner could work independently from a specific database.

**Plan Generation and Execution.** One way to do completeness reasoning is to attach it to the normal query evaluation. While the query result is computed in the database, for each algebraic operation applied to the data, in parallel, we execute the corresponding operation on the metadata. This may however lead to sub-optimal computation of completeness. Because the metadata can be very different from the normal data in size and distribution, the optimal plan for query computation may not be the optimal plan for completeness calculation.

A proper implementation might therefore benefit from having its own cost model to use in query plan optimization. We leave this as future work.

7. **CONCLUSION**

In this paper we have discussed that in some applications databases should be considered partly under closed-world and partly under open-world semantics, and showed how that can be formalized using completeness patterns. We then investigated the problem of computing completeness patterns for query results over such databases.

We presented the pattern algebra, which operates purely at the schema level and can be executed analogously to relational algebra, and then discussed extensions to make use of inferences possible due to the database instance. We highlighted implementational challenges regarding the minimization of sets of completeness patterns and the computation of promoted completeness patterns.

An interesting extension would be to take into account database constraints such as keys, foreign keys, inclusion or functional dependencies.
8. REFERENCES


APPENDIX

A. PROPERTIES OF THE PATTERN ALGEBRA

In this section we give the proofs for the soundness of the pattern algebra and for completeness of the pattern algebra when neglecting the state of the database instance.

PROPOSITION 5 (SOUNDNESS). Let \( D \) be a database, \( \mathcal{P} \) be a collection of base patterns for the schema of \( D \), and let \( P_1 \) and \( P_2 \) be sets of completeness patterns for algebra expressions \( E_1 \) and \( E_2 \), respectively. Furthermore, let \( \sigma \) be an operation among \( \{\pi, \sigma, \sigma_{\text{at}}, \sigma_{\text{op}}\} \). Then:

1. \( \mathcal{P}, D \models (P, E) \) implies \( \mathcal{P}, D \models (\sigma(P), \sigma(E)) \).
2. \( \mathcal{P}, D \models (P_1, E_1) \land (P_2, E_2) \) implies \( \mathcal{P}, D \models (P_1 \land P_2, E_1 \land E_2) \).

PROOF. 1. \( \sigma = \pi_{\text{-A}} \) (Selection by constant): This operation is correct for two reasons: (1) because patterns with \( A = * \) clearly continue to hold when some tuples are dropped; (2) because the result of a query \( \sigma_{\text{at}}(E) \) cannot contain any tuples \( t \) with values \( t[A] \neq d \), and therefore, whenever a pattern \( p \) with \( p[A] = d \) is in the metadata, the same pattern can also be generalized by replacing \( d \) at position \( A \) with *.

2. \( \sigma = \pi_{\text{-D}} \) (Projection): This operation is correct because wildcards cover all possible values and may therefore be projected out.

3. \( \sigma = \sigma_{\text{-B}} \) (Selection by attribute equality): This operation is correct, because the query result table can only contain tuples \( t \) with \( t[A] = t[B] \), and thus, completeness patterns \((d, d, \ldots)\) and \((*, d, \ldots)\) are equivalent (assuming wlog that \( A, B \) are the first two attributes).

2. \( \sigma \) (Equijoin): This operation is correct, because, as for selection by attribute equality, the result can only contain tuples satisfying symmetry, and thus, the various patterns \((d, d, \ldots), (d, *, \ldots)\) and \((*, d, \ldots)\) produced by the variable selection on top of the cartesian are equivalent.

PROPOSITION 6 (COMPLETENESS WITHOUT INSTANCE). Let \( \mathcal{P} \) be a set of base completeness patterns, \( E \) an SPJ-algebra expression and \((p, E)\) a satisfiable query completeness pattern. Then \( \mathcal{P} \models (p, E) \) implies \( p \preceq E(\mathcal{P}) \).

PROOF. We prove the claim by induction over the height of the expression \( E \).

Base case: The height of the expression \( E \) is one, that is, the expression is exactly some base table \( R \).

Then, the completeness patterns for \( E \) are exactly the completeness patterns of the form \((p, R)\) in \( \mathcal{P} \), and thus the algebra is complete.
A. AGGREGATION

Decision-support queries often include aggregation. For instance, it may be interesting to count the number of cities in a certain country, to sum up populations, or to obtain the country with the largest population.

The pattern algebra can naturally be extended to aggregation operators. Because aggregation corresponds to a form of negation, as Lang et al. observed [17], incompleteness of the database may lead to incorrect records in query answers.

Consider for instance a query for the number of cities in each country. As shown in Table 4, Wikipedia is complete for all cities in Bulgaria, but not for all cities in France. So if in reality there were 700 cities in France, but Wikipedia contains only 200, then the query answer will not only be incomplete (it misses the record (France, 700)), but also incorrect (it contains the record (France, 200)).

Fortunately, completeness patterns as derived by the pattern algebra also guarantee correctness. Whenever cities in France are complete, this implies that the number of cities in France is also correct.

In our framework, one can extend the pattern algebra by additional operators for simple aggregation operations, that project away all attributes not mentioned in group-by statements.

For instance, the query

```
SELECT country, COUNT(*)
FROM City
GROUP BY country
```

could be computed using a count operator on top of \( \pi_{\text{country}}(\text{City}) \).

The count operator would extend completeness patterns by a wildcard for the count value: Given the statements in Table 4, the query for the number of cities per country would satisfy the following patterns: (Germany, *), (Ukraine, *), (Bulgaria, *). Extensions for the aggregate functions SUM, MIN and MAX are analogous.

C. EXPERIMENTS FOR PATTERN PROMOTION

In the following, we give details about the evaluation of the runtime and the space growth due to promotion in use cases. We use the same network element table as in Section 4.2 and focus on two kinds of joins:

The first and in a data warehouse most straightforward case is the join of a fact table with a dimension table. It is reasonable to assume that dimension tables are complete. We therefore model this case by joining the table from our use case, augmented with completeness patterns using the method presented in Section 4.3, with a unary second table that contains a random subset of all values appearing for the join attribute in the use case table (these values determine \( Val \) in the algorithm), and where the second table is asserted to be complete.

The second case is the join between two fact tables, which are both only partially complete. We simulate this by a selfjoin between two versions of our use case table.

Join with a Dimension Table. The first experiment was on measuring the number of sets \( S \) that had to be checked for unifiability by the promotion algorithm. As input we used 1000 completeness patterns. Table 7 shows the result of these experiments measuring the number of sets \( S \) to be tested naively, the calculations are between two tables that are partially complete by a selfjoin of the same network element table as in Section 4.3, and focus on two kinds of joins:

The second case is the join between two fact tables, which are both only partially complete. We simulate this by a selfjoin between two versions of our use case table.
Table 7: Runtime analysis of a join of a table with a complete dimension table. Each time, 1000 completeness patterns are used.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Number of distinct attribute values</th>
<th>Size of minimized output (avg)</th>
<th>Promoted statements in minimized output (avg)</th>
<th>Size of S value (avg)</th>
<th>Runtime in ms (median)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor</td>
<td>7</td>
<td>515</td>
<td>45</td>
<td>1.910^7</td>
<td>270</td>
</tr>
<tr>
<td>region</td>
<td>6</td>
<td>751</td>
<td>39</td>
<td>7.910^7</td>
<td>521</td>
</tr>
<tr>
<td>technology</td>
<td>3</td>
<td>801</td>
<td>22</td>
<td>3.610^7</td>
<td>241</td>
</tr>
<tr>
<td>tech capability</td>
<td>6</td>
<td>202</td>
<td>2</td>
<td>1</td>
<td>130</td>
</tr>
<tr>
<td>sector</td>
<td>13</td>
<td>806</td>
<td>22</td>
<td>1.610^8</td>
<td>661 (+4% timeouts)</td>
</tr>
<tr>
<td>state</td>
<td>13</td>
<td>929</td>
<td>7</td>
<td>2.110^8</td>
<td>91 (+25% timeouts)</td>
</tr>
</tbody>
</table>

Table 8: Growth of completeness patterns in a selfjoin. Notably, the number of patterns in the output does not increase due to promotion but even decreases.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Statements in input</th>
<th>Size of nonminimized output</th>
<th>Size of minimized output</th>
<th>Promoted statements in minimized output</th>
<th>Saved statements per promoted statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor</td>
<td>100</td>
<td>9,792</td>
<td>930</td>
<td>232</td>
<td>38</td>
</tr>
<tr>
<td>region</td>
<td>100</td>
<td>9,499</td>
<td>1031</td>
<td>224</td>
<td>38</td>
</tr>
<tr>
<td>technology</td>
<td>100</td>
<td>10,216</td>
<td>523</td>
<td>236</td>
<td>41</td>
</tr>
<tr>
<td>tech capability</td>
<td>100</td>
<td>10,168</td>
<td>420</td>
<td>168</td>
<td>58</td>
</tr>
<tr>
<td>sector</td>
<td>100</td>
<td>6,945</td>
<td>932</td>
<td>103</td>
<td>58</td>
</tr>
<tr>
<td>state</td>
<td>100</td>
<td>2,118</td>
<td>467</td>
<td>21</td>
<td>79</td>
</tr>
</tbody>
</table>

One can also observe a high variation between different attributes. For tech capability for instance, the minimized output contains only 61 patterns. The reason is that the attribute tech capability has only three different values, so whenever only two values are present, the chance is high that for those two values completeness patterns exist. Out of the 100 patterns that could be promoted (50 on each side of the join), 61 got promoted, and these 61 patterns then subsume all other patterns that resulted from the regular join. For the state attribute on the other hand, there are 53 values, so any random subset still has a large cardinality and it is unlikely that one finds in only 50 patterns enough patterns that together are unifiable for promotion. So as only 17 patterns get promoted, the promoted patterns subsume only few patterns from the regular join and therefore the minimized result has the highest cardinality compared with the other join attributes.

We draw three conclusions: First, the runtime in a join of fact tables grows quadratically in the number of patterns for the input tables (Fig. 6). Second, the runtime grows polynomially wrt. the number of attributes used (Fig. 7). Third, promotion leads to a significant reduction of the number of patterns in the output (Table 8).

D. TOWARDS ALGEBRAIC COMPLETENESS

One may wonder whether the pattern algebra extended by the promotion operator is computationally complete. Unfortunately, as shown below, this is still not the case. In this section, we discuss a further extension to the pattern algebra, which introduces additional unsatisfiable patterns in intermediate results, which may be needed for computational completeness.

Recall from Example 8 that completeness patterns can be unsatisfiable. The example was about the fact that after a selection by “specialization = hardware”, the resulting table is certainly complete for software teams.

While one might consider such patterns as unintuitive and irrelevant, the following example shows that not producing these patterns may lead to missing relevant conclusions in subsequent operations:

Example 10. Consider $M \bowtie T \bowtie M \bowtie T$. The patterns $(\ast, C, \ast, \ast)$ and $(\ast, \ast, \ast, C, \ast)$ seem meaningless, because, due to the completeness pattern $(\ast, \ast)$ for $\sigma_{\text{specialization}=T}$ we know that A and B are the only hardware teams, and therefore, the join result cannot contain any rows for team C. We call the patterns for C therefore zombie patterns.

Can we just drop these meaningless patterns? The contrary is the case as we can benefit from the introduction of more zombie patterns.

As we know that no results for team D are possible, we can also introduce the zombie patterns $(\ast, D, \ast, \ast, \ast)$ and $(\ast, \ast, \ast, D, \ast)$. Now suppose we are computing a join with the following table:

**Best teams:**

```plaintext
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>1000</td>
</tr>
<tr>
<td>Software</td>
<td>1000</td>
</tr>
<tr>
<td>D</td>
<td>500</td>
</tr>
<tr>
<td>C</td>
<td>1000</td>
</tr>
<tr>
<td>D</td>
<td>500</td>
</tr>
</tbody>
</table>
```
Zombie patterns are explicit assertions of completeness that cannot occur in results may be needed to ensure computational completeness. We conclude from this example that information about values that cannot occur in results may be needed to ensure computational completeness.

In this section we discuss how zombie patterns can help deduce more general and thus stronger completeness patterns, how they can be represented, and what is the experimental impact of generating them.

**Zombie Patterns.** Zombie patterns are explicit assertions of completeness for values that currently not appear in the result of an expression, either due to algebra operations or due to the state of the database instance.

Naively, that is, without further domain information, this approach may require one to assert one zombie pattern for every value in the active domain of the database (set of all constants appearing in the database). Therefore, zombie pattern generation is only feasible for attributes with known domain (e.g. month or state).

**D.1 Adding Zombie Patterns**

Zombie patterns naturally arise in two operations: Selection by constant and join. In the first case, the introduction is instance-independent, in the second case not.

Consider first a selection by constant \(\sigma_A = d(P)\), where the attribute \(A\) has the domain \(\text{dom}(A)\). Then, we add the zombie patterns

\[
\{(c,\ast,\ldots,\ast) \mid c \in \text{dom}(A), c \neq d\},
\]

to the result of the selection, where we assumed without loss of generality that \(A\) is the first attribute of \(P\).

For a join, the added patterns depend on the database instance. Consider the join \(E_1 \bowtie_{A=B} E_2\) of expression \(E_1\) and \(E_2\) whose complete parts are described by the pattern sets \(P_1\) and \(P_2\). We define the zombie patterns similarly to the promotion result using a function \(\text{addZombies}\), which, too, is called twice with interchanged arguments.

\[
\text{addZombies}(E_1, P_1, A, P_2, B, D)
\]

\[
\text{result} := \emptyset;
\]

\[
\text{for each } p \text{ in } P_1 \text{ with } p[A] = \ast
\]

\[
\text{for each } d \text{ in } \text{dom}(A) \setminus \pi_A(E_1(D))
\]

\[
\text{result} := \text{result} \cup \{ p[A/d] \ast,\ldots,\ast \};
\]

\[
\text{return result};
\]

Intuitively, given a completeness pattern with \(\ast\) for the join attribute on one side of the join, we insert all values that are not covered by this pattern into the join position, and extend the pattern with \((\ast,\ldots,\ast)\) on the other side, because no such rows can occur at all due to the completeness pattern.

How many patterns get created depends on the size of the domains and on the database instance. In the next section, we report insights from our use case.

Whether the zombie algebra is computationally complete is an open question to date, because it is not clear whether the extensions discussed above introduce all possible kinds of zombie patterns that could be needed in the algebra.

**D.2 Impact of Zombie Patterns**

The introduction of zombie patterns gives rise to two questions: The first is how big is the overhead when computing zombie patterns. The second is how often further promotion is possible when computing zombie patterns. We look into both questions next.

**Overhead due to Zombie Patterns.** The number of zombie patterns generated by selection operations is always the number of possible values of that attribute minus one. In a join, the number of zombie patterns can be varying. We therefore tested both a self-join and a join with a complete dimension table, both set up as discussed in Section 5.2.

<table>
<thead>
<tr>
<th>Join attribute</th>
<th>Number of distinct attribute values</th>
<th>Zombie statements before minimization</th>
<th>Zombie statements after minimization</th>
<th>Size of minimized output</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor</td>
<td>7</td>
<td>6,833</td>
<td>380</td>
<td>515</td>
</tr>
<tr>
<td>region</td>
<td>6</td>
<td>5,841</td>
<td>577</td>
<td>751</td>
</tr>
<tr>
<td>technology</td>
<td>3</td>
<td>2,843</td>
<td>516</td>
<td>801</td>
</tr>
<tr>
<td>tech_capabil-</td>
<td>6</td>
<td>7,001</td>
<td>101</td>
<td>202</td>
</tr>
<tr>
<td>ity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sector</td>
<td>13</td>
<td>12,835</td>
<td>510</td>
<td>806</td>
</tr>
<tr>
<td>state</td>
<td>53</td>
<td>53,870</td>
<td>733</td>
<td>929</td>
</tr>
</tbody>
</table>

Table 9: Number of zombie patterns in a join of a fact table with 1000 patterns with a complete dimension table, before and after minimization.

Table 9 contains rows reporting the number of zombie patterns generated in a join of a fact table with 1000 patterns with a complete dimension table, measured before and after removing subsumed patterns from the result. Before minimization there is a clear correlation with the number of attribute values. After minimization, the number of zombie patterns is more uniform around 66%.

In a self-join experiment with 100 completeness patterns and 500 random tuples in the database (note that the less data, the more zombie patterns are created), one third of the resulting completeness patterns were zombie patterns.

As minimization of sets of completeness patterns takes a considerable share of the runtime of completeness calculation, the addition of zombie statements also increases the runtime. Both in the self-join and in the join with a complete dimension table, the introduction of zombies led to an average runtime increase by 250%.

**Additional Inferences Due to Zombie Patterns.** Zombie patterns in intermediate join results may be required for promotion in subsequent joins. We therefore computed a three-way-join, where once we computed zombie patterns in the intermediate result and once not. It turned out that out of 200 runs with random sets of 70 completeness patterns for each of the three tables, only in 2 runs there were additional patterns in the result when computing zombie patterns in the intermediate result. In these two runs, once 4% (10 patterns) and once 7% (1 pattern) were additionally computed. Over all runs, this amounts to 0.08% additionally computed patterns. Note that, even though this numbers suggest that zombie patterns have little impact, completeness patterns can have very different generality and a few more computed patterns can still be important in some applications.