# Keys and Conditional Key Discovery for Data Linking

Pablo Mollá Chárlez

February 8, 2025

## Contents

## 1 Keys

A key is a set of properties that uniquely identifies every instance in the data.

### 1.1 Monotonicity & Minimality

- Key Monotonicity
  - **Definition:** If a set of properties $K$ is a key, then any superset of $K$ is also a key.
  - Explanation: Once a set of properties uniquely identifies every row (i.e., no two rows share the same values for all those properties), adding more properties cannot break that uniqueness—it can only maintain or further refine it.
- Minimal Key
  - **Definition:** A key is minimal if removing any one property from it causes it to stop being a key.
  - Explanation: This means no proper subset of that set of properties can serve as a key. Minimal keys are also sometimes called candidate keys because they are the smallest sets of properties that maintain uniqueness.

### 1.2 Multi-Valued Properties & Incomplete Descriptions

- Multi-Valued Properties
  - **Definition:** Properties that can take multiple values for a single entity (e.g., an actor who has performed in multiple movies).
  - Explanation: In order to compare these multiple values, either we proceed with value difference which refers to comparing individual values within a multi-valued property or, by set difference which typically refers to comparing the sets of values as a whole.
- **Incomplete Descriptions**
  - Optimistic Approach:
    - **Definition:** Any empty or missing value is considered different from all other existing values.
    - Implication: This maximizes the likelihood that a record with a missing value is still "unique" if all its other known values match another record's values.
  - Pessimistic Approach:
    - **Definition:** Any empty or missing value is considered possibly identical to existing values.
    - Implication: This minimizes the likelihood of treating a record with missing information as unique and instead assumes missing data might match existing known values elsewhere.

### 1.3 Types of Keys in Knowledge Graphs

1. S-Key
   - **Definition:** A set of properties $K$ is an S-Key if every row in the dataset has a unique combination of those properties under the following rules:
   - For set-valued properties, two sets are considered the same if they share at least one common element.
   - Empty values are considered different from existing ones (i.e., the optimistic approach to missing data).
2. F-Key:
   - **Definition:** A set of properties $K$ is an F-Key: if every row in the dataset has a unique combination of those properties under the following rules:

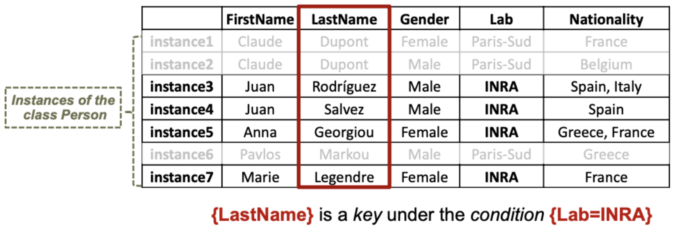
- For set-valued properties, two sets are considered the same if they are exactly identical (i.e., have the same elements).
- Empty values are considered possibly identical to existing ones (i.e., the optimisitc approach to missing data).

3. SF-**Key:**

   - **Definition:** A set of properties $K$ is an SF-Key if every row in the dataset has a unique combination of those properties under the following rules:
   - For set-valued properties, two sets are considered the same if they are exactly identical
   - Empty values are considered different from existing ones (i.e., the optimistic approach to missing data).

## 1.4 Conditional Keys Symeonidou et al. 2017

A conditional key is a set of properties that uniquely identifies all instances of a class only if those instances satisfy a certain condition (a set of property-value pairs). For instance, {LastName} can be a key for "Person" instances, but only if {Lab = INRA}. For people affiliated with "INRA," last name alone is sufficient to uniquely identify them.


Instances of the class Person

| | FirstName | LastName | Gender | Lab | Nationality |
|---|---|---|---|---|---|
| instance1 | Claude | Dupont | Female | Paris-Sud | France |
| instance2 | Claude | Dupont | Male | Paris-Sud | Belgium |
| instance3 | Juan | Rodríguez | Male | INRA | Spain, Italy |
| instance4 | Juan | Salvez | Male | INRA | Spain |
| instance5 | Anna | Georgiou | Female | INRA | Greece, France |
| instance6 | Pavlos | Markou | Male | Paris-Sud | Greece |
| instance7 | Marie | Legendre | Female | INRA | France |

**{LastName}** is a *key* under the *condition* **{Lab=INRA}**

Figure 1: Conditional Key

### 1.4.1 Quality Measures

We can distinguish mainly 2 measures:

- Support: The number of instances that both satisfy the condition (e.g., {Lab = INRA}) and instantiate (have values for) the key part $\implies$ Support = 4.
- Coverage: The ratio of that support to the total number of instances in the class. Formally, Coverage = $\frac{\text{Support}}{\#\text{AllInstances}}$. In the example would be Coverage = $\frac{4}{7}$.

These measures help ensure that the discovered conditional keys are both widely applicable (high coverage) and relevant to a significant subset of data (high support).

### 1.4.2 VICKEY: Mining Conditional Keys Efficiently

In section **Different Approaches: KD2R, SAKEY and VICKEY**, you can have a bit more of information about VICKEY.

- **Goal:** Given a dataset (with all instances of some class), a minimum support, and a minimum coverage, discover all minimal conditional keys whose support and coverage exceed those thresholds.
- **Challenge:** The search space can explode exponentially (roughly $\mathcal{O}(|V|^{|P|})$, where $|V|$ is the number of objects and $|P|$ is the number of properties).
- **Leveraging Non-keys:** Conditional keys can often be derived by starting from combinations that are known not to be keys (non-keys) and then adding property-value conditions that fix collisions.

- **Graph Exploration:** VICKEY systematically enumerates conditions.
    1. Start with single-property conditions $\{p = a\}$.
    2. Refine to two-property conditions $\{p_1 = a_1 \wedge p_2 = a_2\}$, etc.
    3. Continue until all minimal conditional keys are identified.

By using these steps and filtering out large parts of the search space via known non-keys, VICKEY manages to mine conditional keys efficiently, ensuring the discovered keys are both minimal and meet user-defined thresholds for support and coverage.

## 2 Key Discovery Approaches

### 2.1 SF-Keys: Keys and Pseudo-Keys Detection

#### 2.1.1 Bottom-Up Approach

A **bottom-up approach** for discovering keys starts with checking single properties and moves on to pairs, triplets, etc., until it either finds a key or exhausts all property combinations. The rationale is:

1. **Single-property check:** Test if $P1$ alone is a key (i.e., it distinguishes all entities uniquely). If it is, then by key monotonicity, any superset containing $P1$ (like $P1P2$, $P1P3$, etc.) must also be a key.
2. **Increasing combination size:** If no single property is a key, proceed to all pairs of properties ($P1P2$, $P1P3$, etc.). Any pair that is found to be a key again triggers the same monotonicity principle, meaning all supersets of that pair (e.g., $P1P2P3$) will also be keys.
3. **Continue until you find the minimal keys (those whose proper subsets are not keys)**. If necessary, check triplets or even the entire set of properties.

#### 2.1.2 Key Verification: Keys and Pseudo-Keys

To verify whether a set of properties $K$ is truly a key, we typically:

- **Dataset Partition:** Partition the dataset based on the values of $K$. For each combination of values in $K$, gather all instances that share those values into a partition.
- **Check uniqueness:** if every partition has exactly one instance, then $K$ is a key (no two entities have the same values for all properties in $K$).

For SF-pseudo keys (or almost keys), the same process applies, but you allow a small number of partitions to contain multiple instances - i.e., a few exceptions are tolerated. These pseudo keys can be valuable in real-world data settings where small inconsistencies or errors are common.

#### 2.1.3 Quality Measures for (Pseudo-)Keys Atencia et al. 2012

Even if a set of properties is a key, it may only describe a fraction of the data or may not discriminate well in certain cases. Two common quality measures are:

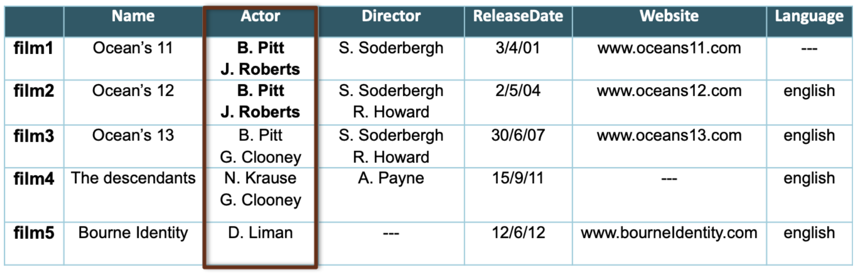
1. Support:
$$\text{support}(P) = \frac{\#\,\text{instances described by } P}{\#\,\text{all instances}}$$
   This indicates the coverage of the property set—how many entities actually have values for those properties (versus missing data).

2. Discriminability:

$$\text{dis}(P) = \frac{\#\,\text{singleton partitions}}{\#\,\text{partitions}}$$

After creating partitions by grouping entities with the same values of $P$, discriminability checks the fraction of those partitions that are singletons (contain exactly one instance). A value of 1.0 would indicate a perfect key for the subset of data that actually has values for $P$.

Together, these measures help assess both how widely applicable a (pseudo-)key is (support) and how effectively it distinguishes instances (discriminability).



| | Name | Actor | Director | ReleaseDate | Website | Language |
|---|---|---|---|---|---|---|
| **film1** | Ocean's 11 | **B. Pitt** **J. Roberts** | S. Soderbergh | 3/4/01 | www.oceans11.com | --- |
| **film2** | Ocean's 12 | **B. Pitt** **J. Roberts** | S. Soderbergh R. Howard | 2/5/04 | www.oceans12.com | english |
| **film3** | Ocean's 13 | B. Pitt G. Clooney | S. Soderbergh R. Howard | 30/6/07 | www.oceans13.com | english |
| **film4** | The descendants | N. Krause G. Clooney | A. Payne | 15/9/11 | --- | english |
| **film5** | Bourne Identity | D. Liman | --- | 12/6/12 | www.bourneIdentity.com | english |

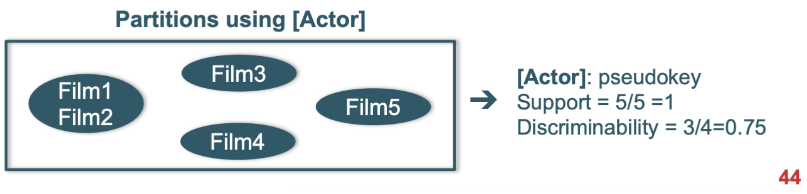


Figure 2: Support and Discriminality

In the example, support $= \frac{5}{5} = 1$ means that every film (all 5) actually has Actors specified —so the "Actor" property covers all instances (100% coverage). If one film did not have any actor specified (empty value), then only 4 of the 5 films would be "described" by the **Actor property**, resulting in a support of 4/5. Besides, discriminability $= \frac{3}{4} = 0.75$ comes from the fact that the "Actor" property creates 4 partitions overall $(\{film1, film2\}, \{film3\}, \{film4\}, \{film5\})$, and in 3 of those partitions there is only a single film (**film3**, **film4**, **film5**). Hence 3 singletons out of 4 total partitions yields 0.75.

## 2.2 F-Keys: Detection

### 2.2.1 Bottom-Up Approach

A bottom-up approach for F-keys works just like before: you first test each single property to see if it is an F-key (i.e., if it uniquely identifies every instance by exact-match of values); if not, you move on to property pairs, then triplets, and so on. Once a set of properties is found to be an F-key, key monotonicity guarantees all supersets are also F-keys.

### 2.2.2 Quality Measures Soru et al. 2015

We distinguish 2 main quality measures for F-Keys:

- Discriminability($P$): the number of instances that $P$ can uniquely distinguish.
- Score$(P) = \frac{\text{Discriminability}(P)}{\#\text{instances}}$.
  - If Score$(P) = 1$, then $P$ is a perfect key (it differentiates every instance).
  - If Score$(P) < 1$, then $P$ is only a pseudo-key (some duplicates remain).

## 2.3 S-Keys: Detection

### 2.3.1 Key Discovery Is Complex

Among several reasons, key discovery is complex due to:

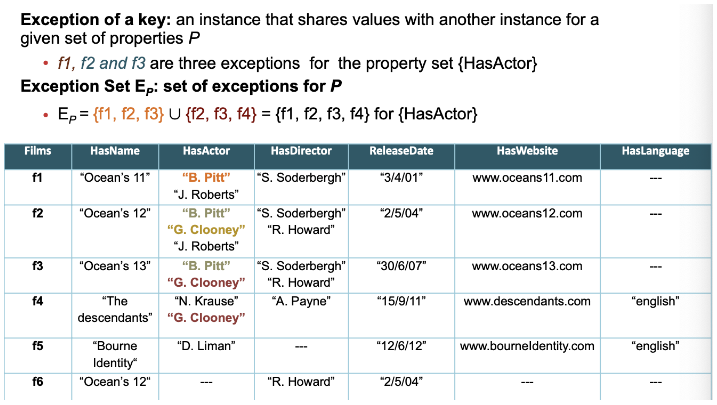
- **Exponential Search Space:** Checking all combinations of $n$ properties means exploring up to $2^n$ subsets.
- **Data Scans:** For each combination, you need to verify uniqueness (or almost-uniqueness) by scanning every instance—costly for large or incomplete datasets.
- **Efficient Filtering & Pruning:** Techniques like key monotonicity (once a set is known to be a key, all its supersets are also keys) and non-key monotonicity (if a set is not a key, all its subsets are also not keys) **help cut down the search space**.

### 2.3.2 S-Keys vs. Other Types of Keys

Recall that an S-Key uses a "set-overlap" comparison for multi-valued properties and treats missing values optimistically (empty values are different from existing ones). This is often trickier than standard (F-Key) discovery because you need to handle partial overlaps of sets rather than exact matches.

### 2.3.3 Different Approaches: KD2R, SAKEY and VICKEY

- **KD2R (Pernelle et al.)** Focuses on data linking by discovering minimal keys automatically in RDF or similar structured data. It uses ontology mappings plus a bottom-up key discovery approach on each dataset ($D_1, D_2$). Only minimal keys discovered are then merged or compared to improve linkage between the two datasets. $\boxed{\text{Pernelle et al. 2015 (53)}}$
- **SAKEY (Scalable Almost Key Discovery)** Addresses incomplete and erroneous data in large datasets. Discovers almost keys (a set of properties fails to be a key for only a small number of exceptions). The algorithm often starts by identifying non-keys (property sets that definitely do not provide uniqueness) and prunes them before searching for near-unique sets. This approach is well-suited for real-world RDF data with lots of missing or noisy values.



**Exception of a key:** an instance that shares values with another instance for a given set of properties *P*

- *f1, f2 and f3* are three exceptions for the property set {HasActor}

**Exception Set $E_P$: set of exceptions for *P***

- $E_P$ = {f1, f2, f3} ∪ {f2, f3, f4} = {f1, f2, f3, f4} for {HasActor}

| Films | HasName | HasActor | HasDirector | ReleaseDate | HasWebsite | HasLanguage |
|---|---|---|---|---|---|---|
| f1 | "Ocean's 11" | **"B. Pitt"** "J. Roberts" | "S. Soderbergh" | "3/4/01" | www.oceans11.com | --- |
| f2 | "Ocean's 12" | **"B. Pitt"** **"G. Clooney"** "J. Roberts" | "S. Soderbergh" "R. Howard" | "2/5/04" | www.oceans12.com | --- |
| f3 | "Ocean's 13" | **"B. Pitt"** **"G. Clooney"** | "S. Soderbergh" "R. Howard" | "30/6/07" | www.oceans13.com | --- |
| f4 | "The descendants" | "N. Krause" **"G. Clooney"** | "A. Payne" | "15/9/11" | www.descendants.com | "english" |
| f5 | "Bourne Identity" | "D. Liman" | --- | "12/6/12" | www.bourneIdentity.com | "english" |
| f6 | "Ocean's 12" | --- | "R. Howard" | "2/5/04" | --- | --- |

Figure 3: Number of Exceptions

- **VICKEY** Another system for key discovery (or partial key discovery), typically employing advanced indexing or partitioning strategies to handle large-scale or heterogeneous data.

### 2.3.4 "Non-Key Discovery First" Strategy $\boxed{\text{Symeonidou et al. 14}}$

Many algorithms begin by finding collisions - combinations of properties that definitely do not distinguish all instances. These identified "non-keys" prune large portions of the search space, because their supersets or subsets can be inferred quickly:

- Subsets of a non-key remain non-keys (they cannot suddenly become unique by dropping properties).
- Supersets might still be tested for uniqueness or near-uniqueness, but the presence of collisions guides the algorithm to refine or skip certain paths.
- Skip Irrelevant Combinations: If a property is incomplete (missing values for all or most instances) or clearly refers to another class (e.g., a property only used in a different domain), we can avoid exploring that property in your key-discovery search. Hence, combinations of properties that are obviously irrelevant—missing data, wrong class—need not be explored.
- Identify Potential n-non Keys When scanning the data, any set of properties that possibly yields $\geq n$ collisions is a candidate for being an n-non key.

An n-non key is simply a set of properties that fails to be unique for at least n instances (i.e., it has at least n "exceptions"). Formally, if $|EP|$ denotes the number of exceptions for property-set $P$, then $P$ is an n-non key if $|EP| \geq n$. By identifying all the maximal n-non keys (the largest property-sets that still have $\geq n$ exceptions), one can deduce all the minimal $(n-1)$-almost keys, because those property-sets would now have fewer $(\leq n-1)$ exceptions. This technique helps prune the search space in almost-key discovery.

Given the following dataset, we can drop singletons to find the non-key properties:

| **Property** | **Exceptions (before removing singletons)** |
|---|---|
| **HasActor** | $\{\{f1, f2\}, \{f1, f2, f3\}, \{f2, f3, f4\}, \{f4\}, \{f5\}\}$ |
| **HasDirector** | $\{\{f1, f2, f3\}, \{f2, f3, f6\}, \{f4\}\}$ |
| **ReleaseDate** | $\{\{f1\}, \{f2, f6\}, \{f3\}, \{f4\}, \{f5\}\}$ |
| **HasName** | $\{\{f1\}, \{f2, f6\}, \{f3\}, \{f4\}, \{f5\}\}$ |
| **HasLanguage** | $\{\{f4, f5\}\}$ |
| **HasWebsite** | $\{\{f1\}, \{f2\}, \{f3\}, \{f4\}, \{f5\}, \{f6\}\}$ |

Table 1: Partitions after removing singletons

| **Property** | **Exceptions (after removing singletons)** |
|---|---|
| **HasActor** | $\{\{f1, f2\}, \{f1, f2, f3\}, \{f2, f3, f4\}\} \implies$ Non-Key |
| **HasDirector** | $\{\{f1, f2, f3\}, \{f2, f3, f6\}\} \implies$ Non-Key |
| **ReleaseDate** | $\{\{f2, f6\}\} \implies$ Non-Key |
| **HasName** | $\{\{f2, f6\}\} \implies$ Non-Key |
| **HasLanguage** | $\{\{f4, f5\}\} \implies$ Non-Key |
| **HasWebsite** | $\{\} \implies$ Single Key |

Table 2: Partitions after removing singletons

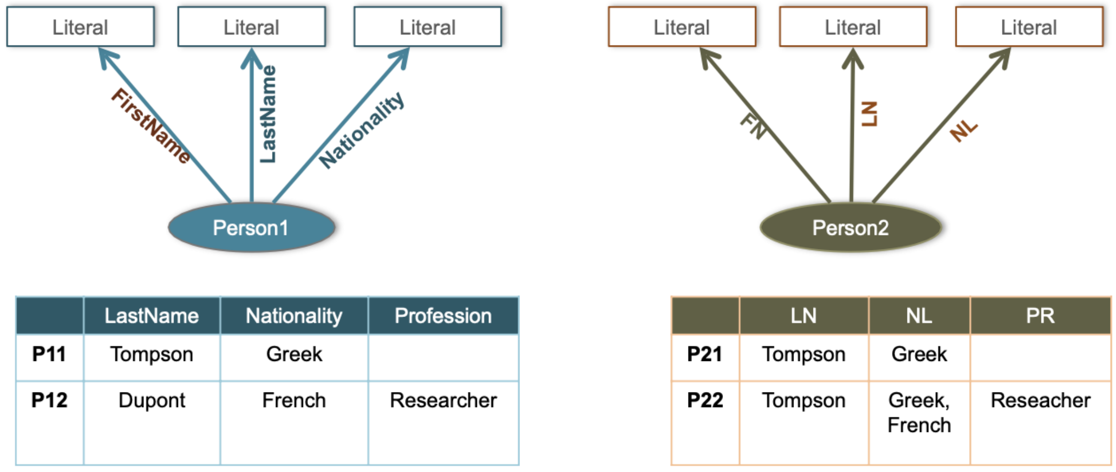
## 3 Key Limitations

- **No universal guarantee:** Some datasets simply have no keys at all.
- **Overly generic:** When keys exist, they usually apply to all instances of a class in the dataset, which might not reflect nuanced or local constraints.
- **Context dependence:** Real-world data often has conditions (e.g., "in German universities") under which a property set acts like a key, but not elsewhere.
- **Limited flexibility:** If a property is almost unique rather than fully unique, standard keys cannot capture this partial uniqueness.

## 4 LINKKEY: Data Interlinking

Data interlinking via LINKKEY is about discovering property-pair sets that uniquely link instances from two different classes—each class coming from different ontologies or datasets.

- **Setting:** We have two ontologies (**Ontology1**, **Ontology2**) and two corresponding datasets (Data of classA, Data of classB). Each dataset describes "similar" entities but using potentially different property names or schemas.
- **LinkKey Concept:** A LinkKey is a maximal set of property pairs $\langle p_1, p_2 \rangle$ such that, whenever two individuals (one from **classA**, one from **classB**) match on all those property pairs, they refer to the same real-world entity. "Maximal" means you cannot add more property pairs without losing that linking guarantee.
- **Example:** In **Dataset1** (Person1), you have LastName, Nationality, etc. In **Dataset2** (Person2), these might appear as LN, NL, etc. A discovered linkkey might be $\{\langle \text{LastName}, \text{LN} \rangle, \langle \text{Nationality}, \text{NL} \rangle\}$. If those properties match pairwise, it signals the same person across datasets.



| | LastName | Nationality | Profession |
|---|---|---|---|
| P11 | Tompson | Greek | |
| P12 | Dupont | French | Researcher |

| | LN | NL | PR |
|---|---|---|---|
| P21 | Tompson | Greek | |
| P22 | Tompson | Greek, French | Reseacher |

Figure 4: Linkkey

- **Role in Data Interlinking:** By automatically discovering LinkKeys, you can align (or "interlink") records across two datasets even when they use different vocabularies/ontologies. This forms the basis for entity reconciliation (finding duplicates) and data integration, enabling cross-dataset queries.

Hence, LinkKey discovery helps unify records from heterogeneous sources, ensuring that if two instances match on the LinkKey properties, they can be treated as the same entity.

# 1 Rule Mining

In rule mining for knowledge graphs, we automatically discover logical statements—often called rules or integrity constraints -that capture regularities in the data. Below are four common types of rules/constraints, each illustrated by a natural-language example:

- Horn Rule: A Horn rule has the form Body $\implies$ Head, meaning whenever the conditions in the "Body" are satisfied, the "Head" must also be true. For instance, "Married people live in the same city." Formally, if $x$ and $y$ are married, then $x$ and $y$ share a city.
- Functional Dependency: A functional dependency states that one property (or set of properties) uniquely determines another property. In relational databases, we often write this as $X \rightarrow Y$. For example, "A person has only one birth date." In other words, knowing the identity of the person uniquely determines that person's birth date.
- Inclusion Dependency: An inclusion dependency enforces that all instances falling under a certain property or relationship must also fit into a specified category. For instance, "Every subject of the predicate **marriedTo** is of type **Person**." Formally, if an entity is used in **marriedTo**, it must be included in the class **Person**.
- Key Constraint: A key constraint says that one or more properties constitute a unique identifier for entities, or that matching on these properties implies the entities are the same. For example, "Two authors of the same publication with the same name are the same person." Here, the combination of $\{\text{publication}, \text{name}\}$ effectively behaves as a key, identifying a unique individual.

By learning and enforcing such rules or constraints, one can improve the consistency, quality, and expressiveness of knowledge graphs.

## 1.1 Challenges

We can face the followingchallenges when searching or discovering rules in knowledge graphs:

1. **Generality vs. Specificity** Finding rules that are neither too broad (over-generalizing and capturing false statements) nor too narrow (missing key relationships) can be difficult.
2. **Evaluating the Rules** Determining how "good" or "correct" a rule is often tricky—especially when ground truth or labeled data is limited.
3. **Incomplete Knowledge Bases** Many KBs have missing facts, so a rule might appear false when the evidence is just unrecorded. Distinguishing genuine errors from missing information is a major hurdle.
4. **Vast Search Space** There are potentially countless candidate rules, and exhaustively exploring them all is impractical.
5. **Rule-Language Expressivity** The complexity of rules allowed (e.g., Horn clauses, existential quantifiers) further expands the search space and adds computational overhead.
6. **Smart Heuristics** Given these constraints, efficient heuristics and pruning strategies are critical for discovering high-quality rules without wasting time on unpromising candidates.

## 1.2 Techniques/Tools to Address Challenges

Conjunctive queries and Datalog rewritings don't by themselves solve the rule-mining challenges—rather, they are tools or techniques that can **help address** them by extracting patterns that can then be used or generalized into rules. For example:

- **Handling a huge search space:** Conjunctive queries (or their Datalog rewritings) let you systematically enumerate patterns (joins, conditions) in a more structured way, pruning unpromising candidates more easily.

- **Dealing with incomplete data:** A query-based approach can pinpoint exactly where data is missing or contradictory, and thus help differentiate "no match found" from "fact not present" in the knowledge base.
- **Ensuring the right level of specificity:** Queries can be refined (or rewritten in Datalog) to capture increasingly specific patterns, enabling a "drill-down" to discover rules with appropriate granularity.

So, rather than being complete solutions to the challenges, they serve as methodological steps or paths for organizing, pruning, and checking the rule or pattern search. For instance, conjunctive queries using SPARQL can be used to find patterns. By specifying a pattern like

```
SELECT ?y ?z
WHERE {
    ?x marriedTo ?y.
    ?x livedIn ?z.
}
```

we identify all matching triples in the graph. Repeated occurrences of the same pattern can point to a potential rule (e.g., "If $?x$ is married to $?y$, then $?x$ and $?y$ share a location"). On the other hand, datalog represents the same pattern as a logical rule, such as

$$\text{marriedTo}(x, y),\ \text{livedIn}(x, z)\ \Rightarrow\ R_1(y, z).$$

This logical form allows easy composition with other rules, unification of variables, and automated reasoning (e.g., checking for contradictions or inferring additional facts).

## 2 Horn Rules

A Horn rule is a logical rule of the form:

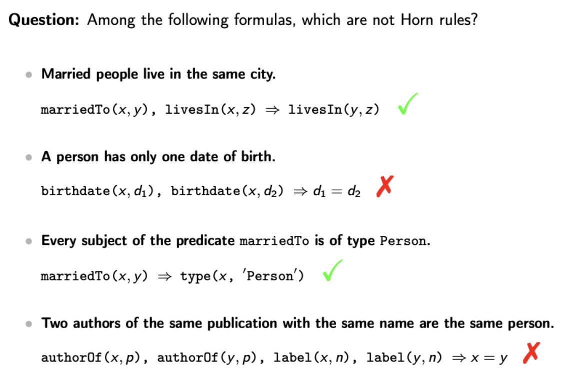
$$A_1 \wedge A_2 \wedge \cdots \wedge A_n\ \Rightarrow\ B,$$

where $A_i$ and $B$ are positive literals (like marriedTo$(x, y)$), and the variables are universally quantified—they apply to all individuals in the knowledge base. For instance, a horn rule might be:

$$\text{marriedTo}(x, y),\ \text{livedIn}(x, z)\ \Rightarrow\ \text{livedIn}(y, z).$$

This states that for any $x$, $y$, and $z$, if $x$ is married to $y$ and $x$ lived in $z$, then $y$ also lived in $z$. In logical notation, the knowledge graph $K$ plus the rule $R$ entails the fact $p$:

$$K\ \wedge\ R\ \vDash\ p.$$



**Question:** Among the following formulas, which are not Horn rules?

- **Married people live in the same city.**
  `marriedTo`$(x, y)$, `livesIn`$(x, z)$ $\Rightarrow$ `livesIn`$(y, z)$ ✓
- **A person has only one date of birth.**
  `birthdate`$(x, d_1)$, `birthdate`$(x, d_2)$ $\Rightarrow d_1 = d_2$ ✗
- **Every subject of the predicate `marriedTo` is of type `Person`.**
  `marriedTo`$(x, y)$ $\Rightarrow$ `type`$(x,$ 'Person'$)$ ✓
- **Two authors of the same publication with the same name are the same person.**
  `authorOf`$(x, p)$, `authorOf`$(y, p)$, `label`$(x, n)$, `label`$(y, n)$ $\Rightarrow x = y$ ✗

Figure 1: Horn Rules Examples

### 2.1 Horn Rule Properties

Systems like **AMIE** and **AnyBURL** mine horn rules from knowledge graphs. They focus on positive Horn rules - where both the premise (the body) and the conclusion (the head) consist only of positive atoms. To reduce the enormous search space, they often impose additional constraints such as:

- **Connectedness:** A rule is connected if all atoms (premise plus conclusion) share variables in such a way that you can travel from any variable to any other via those shared variables. For instance,

$$\text{knows}(x,y),\ \text{livesIn}(y,z)\ \Rightarrow\ \text{friendOf}(x,z).$$

  Here, $x \to y \to z$ forms a single chain: we can "connect" all variables $x, y, z$. On the contrary,

$$\text{knows}(x,y),\ \text{likes}(u,v)\ \Rightarrow\ \text{friendOf}(x,z).$$

  The atom likes$(u,v)$ is disjoint from the other atoms; there is no path of shared variables bridging $(x,y)$ and $(u,v)$.

- **Closedness:** A rule is closed if (1) every variable appears in at least two atoms, and (2) variables used in the conclusion also appear in the premise (this property is called **safely**). For example, a closed example could be:

$$\text{marriedTo}(x,y),\ \text{livesIn}(x,z)\ \Rightarrow\ \text{livesIn}(y,z).$$

  - $x$ appears in marriedTo$(x,y)$ and livesIn$(x,z)$.
  - $y$ appears in marriedTo$(x,y)$ and livesIn$(y,z)$.
  - $z$ appears in livesIn$(x,z)$ and livesIn$(y,z)$.

  Because all three variables show up at least twice—and $z$ doesn't magically appear only in the conclusion - this rule is closed. On the contrary,

$$\text{knows}(x,y)\ \Rightarrow\ \text{friendOf}(x,z).$$

  Here, the variable $z$ appears only in the conclusion (friendOf$(x,z)$) and never in the premise. That makes it not closed (and also not safe for many logic engines).

- **Query Containment:** Horn rules can be "extended" by adding more atoms in the premise (the body). For example,

$$R1 : marriedTo(x,y),\ livedIn(x,z)\ \Rightarrow\ livedIn(y,z)$$

$$R2 : marriedTo(x,y),\ livedIn(x,z),\ type(x,\text{'Homemaker'})\ \Rightarrow\ livedIn(y,z).$$

  Here, $R2$ is contained in $R1$, meaning any instance satisfying $R2$ also satisfies $R1$. Formally, $R2 \subseteq R1$.

- **Rule Specialization:** Extending a rule by adding atoms to the body or by replacing variables with constants is called specialization. As a rule becomes more specialized, it applies to fewer (more specific) cases-so its support typically goes down, while its precision may go up. For instance,

$$R3 :\ marriedTo(x,y),\ livedIn(x,\text{'Paris'})\ \Rightarrow\ livedIn(y,\text{'Paris'})$$

  This only triggers if $x$ lives in Paris, restricting the scope of $R1$. We can view rules in a lattice structure where general rules sit at the top and specific rules appear lower down. Moving **from top to bottom** corresponds to specialization—adding more conditions or constraints (this procedurea and structure is called Rule Generation Lattice)

### 2.2 Rule Evaluation: Quality Measures

- Support (Relevance)

$$\text{support}(R) = |\{p : (K \wedge R \vDash p) \wedge p \in K\}|$$

  Measures how many facts in $K$ the rule correctly predicts, and it decreases as the rule becomes more specialized (fewer cases match).

- Confidence (Accuracy)

$$\text{confidence}(R) = \frac{\text{support}(R)}{\text{support}(R) + |cex(R)|},$$

  Where cex$(R)$ is the set of counterexamples-cases that the rule predicts but are actually false. Confidence goes up when the rule has fewer counterexamples relative to the number of correct predictions.

#### 2.2.1 The Problem of Counterexamples

Many knowledge graphs are incomplete: they rarely store explicit negative facts. Therefore, systems often rely on assumptions like:

- Closed World Assumption (CWA): anything not stated is false.
- Open World Assumption (OWA): anything not stated is unknown.
- Partial Completeness Assumption (PCA): a middle ground (used by AMIE) to infer negative facts more carefully.

### 2.3 Overview of Rule Discovery Methods

Both **top-down** and **bottom-up** search strategies are part of Inductive Logic Programming (ILP). ILP is a broad framework that can be instantiated in different ways:

- **Top-down ILP** (e.g., AMIE, RUDIK) starts with very general rules and specializes them by adding conditions.
- **Bottom-up ILP** (e.g., GOLEM, AnyBURL) begins with specific examples or "clauses" and generalizes them step by step.

So these two approaches are both part of the ILP family—just different ways of navigating the search space for logic rules. Together, these techniques (query containment, specialization, support/confidence measures, etc.) build on the Horn rule framework and the earlier constraints (connectedness, closedness), providing a structured way to navigate and prune the huge space of possible rules.

# 1 Rule Evaluation Metrics

## 1.1 Support

Support of a rule $R$ indicates how many facts in the knowledge base $K$ the rule correctly accounts for. Formally:

$$\text{support}(R) = \Big|\{ p \mid (K \wedge R \vDash p) \wedge p \in K \}\Big|.$$

As rules become more specialized (adding more conditions in the premise), support often decreases because fewer facts match. In practice, a rule with very low support may be discarded (its applicability is minimal). Support addresses the **relevance of the rule** (how many facts it covers).

## 1.2 Support of rule in sets

The coverage of a rule $R$ in a given set $S$ (Support of the rule in set $S$) is defined as:

$$\text{support(R,S)} = |p : (K \wedge R \vDash p) \wedge p \in S|$$

## 1.3 Confidence

Confidence of a rule $R$ measures the proportion of the rule's predictions that are actually true in $K$:

$$\text{confidence}(R) = \frac{\text{support}(R)}{\text{support}(R) + |\,\text{cex}(R)\,|},$$

where $\text{cex}(R)$ is the set of counterexamples-predictions the rule makes that are not found in $K$. Confidence represents the **accuracy of the rule** (fraction of correct predictions). Because knowledge graphs typically lack explicit negative facts, we need assumptions to infer counterexamples:

- Closed World Assumption (CWA): Anything not stated is considered false. If the rule predicts a fact not in $K$, that prediction counts as a counterexample.

$$\text{cex\_CWA} = |\{(x,y) \in B : (x,y) \notin H\}| \quad \text{where} \quad R : B \longrightarrow H$$

- Open World Assumption (OWA): Anything not stated is unknown. A missing fact is not necessarily a counterexample; it could still be true but not yet recorded.

$$\text{cex\_OWA} = |B \cap \neg H| \quad \text{where} \quad R : B \longrightarrow H$$

- Partial Completeness Assumption (PCA): If any relationship $r$ is known for a subject $s$, then all such $r$-relationships for $s$ are expected to be recorded. This assumption tries to balance the extremes of CWA and OWA, treating some parts of the knowledge base as "closed" while leaving others "open."

$$\text{cex\_PCA} = |\{(x,y) \in B : x \in H \wedge (x,y) \notin H\}| \quad \text{where} \quad R : B \longrightarrow H$$

## 1.4 Head Coverage

Head coverage for a rule $B \Rightarrow r(x,y)$ captures what fraction of the known facts for the predicate $r$ (in the knowledge base $K$) is "explained" by that rule. Formally:

$$\text{hc}(B \Rightarrow r(x,y)) = \frac{\text{support}(B \Rightarrow r(x,y))}{|\{(x,y) \mid r(x,y) \in K\}|}.$$

Where $\text{support}(B \Rightarrow r(x,y))$ is the number of true pairs $(x,y)$ for which $B$ holds and $r(x,y)$ is actually in $K$, and $|\{(x,y) \mid r(x,y) \in K\}|$ is the total count of true instances of $r$ in $K$.

Suppose we have a knowledge base about people and their living locations, and we are examining the rule:

$$\texttt{marriedTo}(x,y),\ \texttt{livedIn}(x,z) \Rightarrow \texttt{livedIn}(y,z).$$

In $K$, there are 10 known facts of the form `livedIn`$(y, z)$. By checking the premises (`marriedTo`$(x, y)$ and `livedIn`$(x, z)$), we find that the rule correctly predicts 4 of those 10 facts. In other words, the rule's support is 4 (it matches 4 existing "livedIn" facts). So the head coverage is:

$$\text{hc} = \frac{\text{support}}{\text{total known facts for } r} = \frac{4}{10} = 0.4.$$

Thus, 40% of the `livedIn`$(y, z)$ relationships in the KB are "covered" (i.e., explained) by that rule.

### 1.5 Weight of a Rule

The weight of a rule $w(R)$ is a composite metric that tries to balance how well $R$ covers the true facts in the knowledge graph (akin to recall) and how many non-facts or potential negatives it predicts (related to precision). One way to define it is:

$$w(R) = \alpha\Big(1 - \frac{\text{support}(R, \mathcal{G})}{|\mathcal{G}|}\Big) + \beta\Big(\frac{\text{support}(R, \mathcal{V})}{|\mathcal{V}|}\Big),$$

where

- $\mathcal{G}$ is the set of grounded facts (true statements) in the knowledge graph (finite).
- $\mathcal{V}$ is a set representing potential negative or unobserved facts (which can be extremely large). We can also use $|U(R, \mathcal{V})|$, $U(R, \mathcal{V}) \subseteq \mathcal{V}$ is chosen so that $\text{support}(R, \mathcal{V}) \subseteq U(R, \mathcal{V})$, ensuring proper normalization.
- $\text{support}(R, \mathcal{G})$ counts how many true facts $R$ covers, and $\text{support}(R, \mathcal{V})$ counts how many "negative" or unobserved facts $R$ covers.
- $\alpha, \beta$ let you tune how much each term contributes.

In words, the first term $\big(1 - \frac{\text{support}(R, \mathcal{G})}{|\mathcal{G}|}\big)$ decreases when $R$ correctly covers more real facts (increasing recall), whereas the second term $\frac{\text{support}(R, \mathcal{V})}{|\mathcal{V}|}$ increases if $R$ also covers many negative or unobserved facts (worsening precision).

For instance, suppose that $\mathcal{G}$ has 100 known true facts relevant to $R$. The rule covers 30 of them: $\text{support}(R, \mathcal{G}) = 30$. Moreover, $\mathcal{V}$ has 300 potential negative facts; the rule covers 45 of those: $\text{support}(R, \mathcal{V}) = 45$. We choose $\alpha = 0.5$ and $\beta = 0.5$. We would get:

$$w(R) = 0.5\Big(1 - \frac{30}{100}\Big) + 0.5\Big(\frac{45}{300}\Big) = 0.5\big(0.70\big) + 0.5\big(0.15\big) = 0.35 + 0.075 = 0.425.$$

A lower weight in this scheme would typically mean the rule either fails to cover enough true facts or covers too many negatives-hence is less desirable.

## 2 SHACL: Shape-Based Constraint Language

SHACL (Shapes Constraint Language) is a **W3C Recommendation (2017)** that provides a standard schema language for RDF. It allows one to define validation constraints on classes, nodes, properties, and literals by expressing them directly in RDF. The validation process takes two inputs-a data graph (the RDF dataset to be validated) and a shape graph (the RDF-based definitions of the constraints)-and produces a validation report. A shape graph can contain multiple shapes, each described by RDF triples using the SHACL vocabulary (sh:). Shapes come in two forms: **Node Shapes**, which apply constraints directly to a node, and **Property Shapes**, which specify constraints on the nodes reached via a particular property path. The goal of SHACL is thus to enforce and verify that RDF data conforms to the intended structure and semantics.

### 2.1 SHACL Constraints

Table 1: SHACL Constraint Forms

| **Constraint** | **Shape Type** | **Description** |
|---|---|---|
| **Node Kind and Datatype Constraints** | | |
| `sh:class c` | Node / Property | Target nodes must be instances of class `c`. Multiple `sh:class` values are conjunctive. |
| `sh:datatype t` | Node / Property | Target nodes must be literals of datatype `t`; at most one `sh:datatype`. |
| `sh:nodeKind k` | Node / Property | At most one `sh:nodeKind` declaration for the shape. |
| **Numeric Constraints** | | |
| `sh:minInclusive / sh:minExclusive /` | Node / Property | All target nodes must satisfy the numeric comparison (often combined with `sh:datatype xsd:integer` or other numeric types). |
| `sh:maxInclusive / sh:maxExclusive` | | |
| **String Length Constraints** | | |
| `sh:minLength n; sh:maxLength m` | Node / Property | Target nodes must be literals or IRIs. One `sh:minLength` and one `sh:maxLength` allowed. |
| **Regular Expression Constraints** | | |
| `sh:pattern regex; sh:flags flag` | Node / Property | Target nodes must match the given `regex`; `flag` often includes `"i"` (case-insensitive). |
| **Language Tag Constraints** | | |
| `sh:languageIn (lang_list)` | Node / Property | All target nodes must have a language tag in the given list. |
| `sh:uniqueLang true` | Property shape | Ensures at most one value per language tag among value nodes. |
| **Cardinality Constraints** | | |
| `sh:maxCount n; sh:minCount m` | Property | On `sh:path p`, bounds the number of value nodes from `p`. `n` and `m` must be `xsd:integer`. |
| **Fixed Value Constraints** | | |
| `sh:hasValue v` | Property | All values from the path must be equal to `v`. |
| `sh:in (v1 v2 ... vn)` | Property | All value nodes must be one of `v1, v2, ..., vn`. |
| **Property Pair Constraints** | | |
| `sh:equals r` | Property | Values from `p` must match values from `r` exactly. |
| `sh:disjoint r` | Property | Values from `p` must not overlap values from `r`. |
| `sh:lessThan r` | Property | `max`(Vp) < `min`(Vr). |
| `sh:lessThanOrEquals r` | Property | `max`(Vp) ≤ `min`(Vr). |
| **Logical Operators** | | |
| `sh:not [shape]` | Node / Property | Negates the constraints defined in the nested shape. |
| `sh:and (shapes...)` | Node / Property | Target node must satisfy all listed shapes. |
| `sh:or (shapes...)` | Node / Property | Target node must satisfy at least one listed shape. |
| `sh:xone (shapes...)` | Node / Property | Target node must satisfy exactly one listed shape. |

### 2.2 SHACK's Validation Steps

1. Identify Target Nodes. Determine which nodes the constraint applies to (e.g., via **sh:targetClass**, **sh:targetSubject** or **sh:targetObjectsOf**).
2. Define the Constraint Condition in SPARQL. Typically a SELECT query that returns any nodes violating the rule.
3. Check for Violations. If the query returns any results, the graph fails to conform. Otherwise, it passes.

This SPARQL-based method is quite flexible—constraints can incorporate arbitrary graph patterns and even aggregations. Additional Constraints Expressible in SHACL include:

- Unary Inclusion Dependencies: Restricting which nodes can appear as property values.
- (Conditional) Inclusion Dependencies (IND/CIND): Ensuring that values in one property subset must appear in another, potentially under certain conditions.
- Relation Functionality: Enforcing a property can have at most one value (i.e., functional).
- Inverse Relation Functionality: Ensuring that the property's inverse is functional (each value node points back to at most one subject).